UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

GOOGLE LLC,
Petitioner,

v.

SINGULAR COMPUTING LLC,
Patent Owner.

Case No. TBD
Patent No. 11,169,775

DECLARATION OF DR. MARY K. BOLIN

Google Exhibit 1005
Google v. Singular

I, Mary K. Bolin, declare as follows:

1. I am Emeritus Professor at the University of Nebraska, Lincoln Libraries. I have held the following positions: Professor and Catalog and Metadata Librarian, Digital Initiatives and Special Collections Department, University of Nebraska—Lincoln Libraries (2015–2020); Professor and Chair, Discovery and Resource Management and Coordinator of Collection Development, University of Nebraska, Lincoln Libraries (2004–2015); Lecturer, San José State University School of Information: Lecturer (2008–present); Head of the Technical Services Department, University of Idaho Library (1993–2003); promoted to Professor in 1999); Head of the Cataloging Department, University of Idaho Library (1986–1993); promoted to Associate Professor and tenured (1991); Head of Records Maintenance Section, Cataloging Department, University of Georgia Libraries (1981–1986).

2. Over the course of my work as a librarian, professor, researcher, student research supervisor, and author of numerous scholarly papers and other publications, I have had extensive experience with cataloging records and online library management systems built around Machine-Readable Cataloging (MARC) standards. I also have substantial experience in authenticating printed documents and establishing the date when they were accessible to ordinarily skilled researchers.

3. Exhibit B to this declaration is a copy of my CV.

4. I have been retained by Wolf, Greenfield, & Sacks, P.C., counsel for Petitioner Google LLC, to assess the authenticity and dates of public availability of the documents discussed in this declaration. For this service, I am being paid my usual hourly fee of $150.00 per hour, plus actual expenses. My compensation is not dependent in any way upon any outcome in the above-captioned proceeding.

I. SUMMARY OF DOCUMENTS

5. The documents that I discuss in this declaration are the following:

- **Document 1**. Schmidt et al., "KPROC - An Instruction Systolic Architecture for Parallel Prefix Applications," Parallel and Distributed Computing Practices Vol. 3, No. 2, pp. 163–178 (2000) (hereafter the "Schmidt article");
- **Document 2**. Schimmler et al., "A bit-serial floating-point unit for a massively parallel system on a chip," *Parallel Algorithms and Applications*, Vol. 19, No. 2-3, pp. 79-95 (2004) (hereafter the "Schimmler-2004 article");
- **Document 3**. Schmidt et al., "A Parallel Accelerator Architecture for Multimedia Video Compression," in Euro-Par'99, Lecture Notes in Computer Science 1685, Springer Verlag, 1999, pp. 950–960 (hereafter the "Schmidt-1999 article");

- **Document 4**. Schimmler et. al., "The Instruction Systolic Array – Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers, *High Performance Computing and Networking*, Lecture Notes in Computer Science 797, Springer Verlag, 1994, pp. 487–488 (hereafter the "Schimmler-1994 article");
- **Document 5**. Schimmler et. al., "The Instruction Systolic Array in Image Processing Applications," *Vision Systems: Sensors, Sensor systems, and Components*, Europto 96, pp. 136–144 (hereafter the "Schimmler-1996 article");
- **Document 6.** Yeh, et. al., "Parallax: An architecture for real-time physics," the 34th International Symposium on Computer Architecture (ISCA), 2007, pp. 232-243 (hereafter the "Yeh-ISCA07 article");
- **Document 7.** Yeh, et. al., "The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration," 40th IEEE/ACM International Symposium on Microarchitecture (MICRO'07), 2007, pp. 394-406 (hereafter the "Yeh-Micro07 article");
- **Document 8**. Burger, et. al., "Composable Multicore Chips," in Multicore Processors and Systems, pp. 73-109 (Springer-Verlag 2009) (hereafter the "Burger article");

- **Document 9.** Thornton et al., "Tradeoff analysis of integer multiplier circuits implemented in FPGAs," 1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM 1999), pp. 301-304 (1999) (hereafter the "Thornton article");
- **Document 10.** Oh et. al., "A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor," IEEE Journal of Solid-State Circuits, vol. 41, no. 4, pp. 759-771 (April 2006) (hereafter the "Oh article");
- **Document 11.** S. Borkar, "Thousand Core Chips A Technology Perspective," Proceedings of the 44th ACM/IEEE Design Automation Conference, pp. 746-749 (2007) (hereafter the "Borkar article");
- **Document 12.** Stackhouse et. al., "A 65 nm 2-Billion Transistor Quad-Core Itanium Processor" Solid-State Circuits," 2008 IEEE International Solid-State Circuits Conference – Digest of Technical Papers, pp. 92-93, 598 (2009) (hereafter the "Stackhouse article");
- **Document 13.** Bradley, et. al., "Exploiting Loop-Level Parallelism for SIMD Arrays Using OpenMP," *in* Vol. 4935 of the Lecture Notes in Computer Science, pp. 89-199 (Springer-Verlag 2008) (hereafter the "Bradley article");

- **Document 14.** Roma et al., "In the development and evaluation of specialized processors for computing high-order 2-D image moments in real-time," in *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pp. 170-179 (2000) (hereafter the "Roma article");
- **Document 15.** Lokhmotov, et. al., "Revisiting SIMD Programming," *in* Vol. 5234 of the Lecture Notes in Computer Science, pp. 32-46 (Springer-Verlag 2008) (hereafter the "Lokhmotov article").

II. METHODOLOGY

6. I searched for each of these documents (1) in subscription databases (large repositories of reliable scholarly content from well-known publisher) such as *IEEE Xplore* and the *ACM Digital Library*, and (2) in *Google Scholar* (an open access database that aggregates content from subscription sites as well as other freely available repositories of scholarly and research data). I searched for the conference proceedings where these papers appear in the Online Computer Library Center ("OCLC") database. I also searched the library catalogs of institutions such as the University of Nebraska—Lincoln and other large research libraries. My goal was to find a record in one of these databases that contained a date indicating when the work in question was publicly accessible.

7. *Use of MARC Records*. The "Machine Readable Cataloging" standard, also known as "MARC," is an industry-wide standard method of organizing library catalog information. The Library of Congress first developed MARC in the 1960s. A MARC-compatible library is one that has a catalog consisting of individual MARC records for each of its works. Today, MARC is the primary communication protocol for the transfer and storage of bibliographic metadata in libraries.

8. OCLC is a database of MARC records used by libraries worldwide. When an OCLC-participating institution acquires a work, it creates a MARC record for this work in its computer catalog system, and OCLC automatically supplies the date of creation. The MARC record creation date reflects the date on which the item was first acquired and catalogued in MARC by the library; or in some cases the MARC record creation date is shortly after the date on which the item was first acquired by the library and placed in its internal catalogue. Once the MARC record is created by a cataloger at an OCLC-participating member institution, it is instantaneously available to any OCLC-participating members, and, therefore available to the public.

9. A MARC record comprises several fields, each of which contains specific data about the work. Each field is identified by a standardized, unique, three-digit code corresponding to the type of data that follows. For example, a

work's title is recorded in field 245, the primary author of the work is recorded in field 100, an item's International Standard Book Number ("ISBN") is recorded in field 020, an item's Library of Congress call number is recorded in field 050, and the publication date is recorded in field 260 or 264 under the subfield "c." If a work is a periodical, then its publication frequency is recorded in field 310, and the publication dates (*e.g.*, the first and last publication) are recorded in field 362, which is also referred to as the enumeration/chronology field. The library that created the record is recorded in field 040 in subfield "a" with a unique library code. When viewing the MARC record online via OCLC's Connexion database, hovering over this code with the mouse reveals the name of the library. I used this method of "mousing over" the library codes in the OCLC database to identify the originating library for all of the MARC records in this declaration, as well as the "display holdings" command to see a list of libraries that own the item. In the MARC records in the OCLC Connexion database, MARC field 040 contains the OCLC symbols of institutions transcribing, creating, and modifying the record.

10. OCLC was created "to establish, maintain and operate a computerized library network and to promote the evolution of library use, of libraries themselves, and of librarianship, and to provide processes and products for the benefit of library users and libraries, including such objectives as increasing availability of library resources to individual library patrons and reducing the rate of rise of

library per-unit costs, all for the fundamental public purpose of furthering ease of access to and use of the ever-expanding body of worldwide scientific, literary and educational knowledge and information."[1] Among other services, OCLC and its members are responsible for maintaining the WorldCat database (http://www.worldcat.org/), which is used by independent and institutional libraries throughout the world. OCLC is the international cooperative to which libraries catalog records and others pay to access MARC records. The MARC records referenced in this declaration, extracted from the OCLC database, were generated in the ordinary course of business. Further, I personally located and downloaded each of these records associated with the documents on which my declaration focuses.

11. OCLC provides its members online access to MARC records through its OCLC Connexion database. When an OCLC member institution acquires a work, it creates a MARC record for this work in its computer catalog system in the ordinary course of its business. MARC records created at the Library of Congress are batch-loaded into the OCLC database through a subscription to MARC

---

[1] Third Article, Amended Articles of Incorporation of OCLC Online Computer Library Center, Incorporated (available at http://www.oclc.org/en-US/membership/councils.html).

Distribution Services daily or weekly. Once the MARC record is created by a cataloger at an OCLC member institution or is tape-loaded from the Library of Congress, it is then made available to any other OCLC members online, and, therefore, made available to the public.

12. When an OCLC member institution creates a new record, OCLC automatically supplies the date of creation for that record. As noted before, the MARC record creation date reflects the date on which the item was first acquired or cataloged by the institution; or in some cases the MARC record creation date is shortly after the date on which the item was first acquired and placed in the institution's internal catalogue.

13. The date of creation for the MARC record appears in the fixed field (008), characters 00 through 05. Field 005 is a 16-digit field that records the Date characters yyyymmdd including 4 for the year, 2 for the month, and 2 for the day. The Date entered on file never changes. These dates are automatically supplied by the OCLC system and cannot be changed by the user. When the MARC record is created in OCLC, it is available throughout the world to other libraries and in WorldCat, the public version of OCLC Connexion. Thus, the date of creation of the MARC record by a cataloger at an OCLC member institution reflects when the underlying item is accessible to the public.

14. For each of Documents 1-15, I included a search of OCLC, www.oclc.org, which is a large shared database used by libraries around the world to create and exchange cataloging data, using MARC format, which is a universal communications format for that data.

15. For those documents for which there is a record in OCLC, I determined when the record was created and which libraries own the item. I then visited the website of the library or libraries that own the item, searched their library catalog, and determined when the item was added to the library's collection.

16. I am familiar with many online catalogs, databases, and search engines. In preparing for this declaration, I used authoritative information systems such as the above-mentioned WorldCat (https://www.worldcat.org), the online catalog of the Library of Congress (https://catalog.loc.gov), shared research databases such as ResearchGate and CiteSeerX, the online catalog of a number of large research libraries such as the University of Nebraska—Lincoln, IEEE Xplore, ACM Digital Library, and Google Scholar search for records. These records are identified and discussed in this declaration.

17. *ACM Digital Library*. The ACM Digital Library is a digital database of research publications and other information resources produced by the Association for Computing Machinery (ACM), which is a well-known, reputable

publisher of scientific books and academic journals. The ACM Digital Library is a well-known and widely used resource for researchers, educators, and professionals in the field of computer science and related disciplines. The ACM Digital Library provides access to a wide range of content, including journal articles, conference papers, technical reports, and other research documents. It also includes several other resources, such as conference proceedings, videos, and standards published by ACM. With the ACM Digital Library, users can search for and access a wealth of information on a wide range of topics in computer science and related fields.

18. *IEEE Xplore*. IEEE Xplore is a digital database that provides access to a vast collection of research publications and standards published by the Institute of Electrical and Electronics Engineers (IEEE), which is a well-known, reputable publisher of scientific books and academic journals. IEEE Xplore is a well-known and widely used resource for researchers, educators, and professionals in the fields of electrical engineering, computer science, and related disciplines. With IEEE Xplore, users can search for and access articles, conference papers, and other research documents, as well as standards and other technical information, published by the IEEE. The IEEE Xplore database is a comprehensive source of scholarly literature in the fields of electrical engineering, computer science, and other related fields.

III. SCOPE OF MY DECLARATION

19. I am not an attorney and will not offer opinions on the law. I am, however, rendering my expert opinion on the authenticity of the documents referenced herein and when and how each of these documents was disseminated or otherwise made available to the extent that any member of the public interested in the subject matter, exercising reasonable diligence, could have located the documents by the dates I indicate below.

20. I understand from counsel that a printed publication qualifies as publicly accessible as of the date it was disseminated or otherwise made available such that any member of the public interested in the subject matter could locate it through the exercise of ordinary diligence.

21. While I understand that the determination of public accessibility under the foregoing standard rests on a case-by-case analysis of the facts particular to an individual publication, I also understand that a printed publication is rendered "publicly accessible" if it is cataloged and indexed by a library such that a member of the public interested in the subject matter a could locate it exercising reasonable diligence (i.e., I understand that cataloging and indexing by a library in a manner that permits a member of the public interested in the subject matter to locate the publication is sufficient, though there are other ways that a printed publication may qualify as publicly accessible, such as being indexed and catalogued in an online

database). One manner of sufficient indexing is indexing according to subject matter category. I understand that the cataloging and indexing by a single library of a single instance of a particular printed publication is sufficient, even if the single library is in a foreign country. I also understand that the cataloging and indexing of information that would guide a person interested in the relevant subject matter to the printed publication, such as the cataloging and indexing of an abstract for the printed publication, is sufficient to render the printed publication publicly accessible.

22. I understand that a document may also be publicly accessible if it is catalogues and indexed in an online digital database such that a person interested in the relevant subject matter could locate the document in the database exercising reasonable diligence. For example, as I discussed in paragraphs 17-18 above, in my opinion the ACM Digital Library and IEEE Xplore are examples of digital databases that are well-regarded among the academic community and considered to be accurate and trustworthy sources of information regarding the scientific and academic publications catalogued in those databases.

23. I understand that routine business practices, such as general library cataloging and indexing practices, can be used to establish an approximate date on which a printed publication became publicly accessible. I further understand that the date on which digital databases, like IEEE Xplore and ACM Digital Library,

indicate a document was published and/or added to the database can be used to establish a date on which the document became publicly accessible.

24. I am told by counsel that the subject matter of this proceeding relates generally to the fields of computing. I have been informed by counsel that a "person of ordinary skill in the art at the time of the invention" (POSA) is a hypothetical person who is presumed to be aware of all pertinent prior art and the conventional wisdom in the art, and is a person of ordinary creativity. I am told by counsel that persons of ordinary skill in this subject matter or art would have had a bachelor's degree in Electrical Engineering, Computer Engineering, Applied Mathematics, or the equivalent, and two years of academic or industry experience in computer architecture. More education could substitute for experience, and vice versa.

25. It is my opinion that such a person would have been engaged in research, learning through study and practice in their field, and possibly through formal instruction, the bibliographic resources relevant to his or her research. In the late-2000s such a person would have had access to a vast array of long-established print and online resources providing indexing information, abstracts, and full-text services for publications in the field of computing.

26. Based on my experience working in research libraries with researchers having the qualifications described above, or even lesser qualifications,

it is my opinion that such researchers would have been able to locate the material discussed herein on their own or with the assistance of a research librarian, with relative ease using the tools and resources described herein.

IV. AUTHENTICATION AND PUBLIC ACCESSIBILITY OF DOCUMENTS

A. Document 1 ("Schmidt")

27. Below are my opinions on the public accessibility of Schmidt et al., "KPROC – An Instruction Systolic Architecture for Parallel Prefix Applications," *Parallel and Distributed Computing Practices* Vol. 3, No. 2, pp. 163–178 (2000) (hereafter the "Schmidt article") and the authenticity of EX1007 attached hereto.

1. Authentication of EX1007

28. EX1007 is a scan of the (1) inside cover of the journal; (2) the Table of Contents (TOC); and (3) the Schmidt article from the print copy of Volume 3, Number 2 of the journal *Parallel and Distributed Computing Practices* (PDCP) that I found within the custody of the University of Hong Kong (HKU) library, and from where I requested and obtained the copy in EX1007.

29. EX1007 appears to be a true and correct copy in a condition that creates no suspicion about its authenticity, particularly given that it was obtained from the HKU library from its own collection at my request. The TOC includes the journal name ("*Parallel and Distributed Computing Practices*"), issue identification and issue date. Also, the text of the article is complete and the page

numbers for the Schmidt article (pp. 163-177) match the TOC. The first page of the Schmidt article (pp. 163) shows the correct issue information, bears a copyright notice from the publisher ("© 2000 Nova Science Publishers, Inc.") and an international standard serial number (ISSN) of "1097-2803" that matches the HKU library catalog information and the MARC records for this journal, as discussed in further detail below.

2. Public Accessibility of the Schmidt Article

30. The inside cover and the TOC in EX1007 show that (1) this issue of *Parallel and Distributed Computing Practices* had an issue date of June 2000; (2) contains the Schmidt article on pages 163-177; and (3) was published by Nova Science Publishers, Inc. (Huntington NY) in 2000. Nova Science Publishers was and still is a well-known, reputable publisher of scientific books and academic journals. According to its website in the 2000-2002 timeframe,[2] Nova Science Publishers was publishing "about 275 books per year and 30 journals." The website maintained the company catalog[3] and provided links to a wide range of journals available, including the journal *Parallel and Distributed Computing*

---

[2] https://web.archive.org/web/20011204191802/http:/www.nexusworld.com/nova/

[3] https://web.archive.org/web/20011217202938/http://www.nexusworld.com/nova/catalog.html

Practices [4] (with links to further information about this journal).[5] Nova Science's website also provided contact information for the publisher and a "How to Order" tab and a "search to order" tool for obtaining the Nova Science's offerings, which tells me that interested persons could find and order Nova Science's offering from their website, which is corroborated by the subscription information on the inside cover of the journal.[2,3,4]

31. Based on this information, it is my opinion that the Schmidt article was published in the June 2000 issue (i.e., Volume 3, Number 2) of the journal *Parallel and Distributed Computing Practices* and was accessible by the public from Nova Science Publishers in June of 2000 or shortly thereafter in the year 2000, and that a person interested in and ordinarily skilled in the relevant subject matter could have located the Schmidt article through the exercise of ordinary diligence (e.g., via the Nova Science Publishers website).

---

[4] https://web.archive.org/web/20011221140039/http://www.nexusworld.com/nova/journals.html

[5] https://web.archive.org/web/20010418221324/http://www.nexusworld.com/nova/jparallel.html

32. Additionally, I located the Schmidt article via HKU's online search tool FIND@HKUL.[6] Attachment 1a attached hereto shows the HKU library catalog entry for the journal *Parallel and Distributed Computing Practices* retrieved from FIND@HKUL listing a creation date of 1998 for this journal and showing that the "Main Library" has available a "Bound issue" of "Volume 3 (2000)" in which the Schmidt article was published. Print copies of this journal are located in "Main Serials" under classification number "S 004.35 P22 D." "S" indicates that this is a serial (a journal) and 004 is the Dewey Decimal classification for "data processing and computer science" with a further refining of the topic represented by .35 and P22 representing the title of the work. EX1007 is a scanned copy of the above-noted portions of this bound issue obtained from the HKU library.

33. Furthermore, Attachment 1b is the MARC record for the journal *Parallel and Distributed Computing Practices* that I retrieved from the OCLC bibliographic database. As previously noted, the library that created the record is recorded in field 040 with a unique library code. For Attachment 1b, that library

[6] https://julac-hku.primo.exlibrisgroup.com/discovery/search?vid=852JULAC_HKU:HKU

code is “NSD,” which means that the MARC record for this serial was cataloged as part of the National Serials Data Program at the Library of Congress.

34. As can be seen in the “Entered” field (“19971209”) in the MARC record for this exhibit, a cataloger at the Library of Congress created OCLC record number 38060488 on December 9, 1997. The MARC record demonstrates that the journal *Parallel and Distributed Computing Practices* began with the Volume 1, Number 1 issue dated March 1998 and ended with the Volume 5, Number 4 issue dated December 2002, and that this journal was a printed “Quarterly” by its publisher Nova Science Publishers, corroborating the information in EX1007.

35. This MARC record indicates 8 holdings for this title. Attachment 1c is the list of institutions the OCLC database indicates are “Holding[]” copies of the PDCP journal, including the HKU library (OCLC Symbol “HUA”) from which EX1007 was obtained, the National Science Library of Canada (OCLC Symbol “cai”) (whose MARC record I discuss further below), Stanford University (OCLC symbol “stf”), the Library of Congress (“DLC”), Dickinson College (“DKC”), and the University of Wisconsin System (OCLC symbol “GZM”), which are institutions from which this journal would be accessible. The MARC records for this journal (Attachments 1b-1c) also corroborate that the journal *Parallel and Distributed Computing Practices* was published by Nova Science Publishers and

that issues of the journal were acquired and added to multiple library collections and would have been accessible to the public in the June 2000 timeframe.

36. Attachment 1d is the National Science Library of Canada's (NSLC's) MARC record for the journal *Parallel and Distributed Computing Practices.* Record 005 in Attachment 1d shows that the journal was loaded into the NSLC catalog on June 30, 1998 and that the NSLC holds all volumes of this journal.

37. As noted in Attachment 1b, the 2002 Volume 5, Number 4 issue of this journal was the last to be published by Nova Science Publishers. This journal subsequently changed its name to *Scalable Computing: Practice and Experience* (SCPE). The SCPE website[7] maintains an archive of past issues,[8] including full PDFs of the online issues from 2005 and after and catalogs information on the print issues from March 1998-December 2002 discussed above, including information on the Volume 3, Number 2 issue in which the Schmidt article was published.[9] The SCPE archive lists the publication date for the Volume 3, Number 2 issue of *Parallel and Distributed Computing Practices* as June 18, 2000, corroborating the June 2000 issue date shown in EX1007. As previously noted, the

---

[7] https://www.scpe.org/index.php/scpe

[8] https://www.scpe.org/index.php/scpe/issue/archive

[9] https://www.scpe.org/index.php/scpe/issue/view/41

journal *Parallel and Distributed Computing Practices* was a quarterly (Attachment 1b), and the SCPE archive indicates that each quarterly issue for the years between 1998 and 2002 (i.e., Volumes 1-5) was published on the 18th of March, June, September and December for Numbers 1-4, respectively.

38. Based on the above information and on finding a print copy in the HKU library, it is my opinion that the Schmidt article was published and publicly accessible and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian, by June 18, 2000 or shortly thereafter in the year 2000 (and remained publicly accessible in the subsequent years) from the publisher (as noted previously), from the HKU library (e.g., via the online search tool maintained by HKU as least as early as 1999 and throughout the time period between the publication of the Schmidt article and the 2009-2010 timeframe[10]) from which EX1007 was obtained, or from another of the libraries that subscribed to and received the journal *Parallel and Distributed Computing Practices*.

B. Document 2 ("Schimmler-2004")

39. Following are my opinions with respect to the public accessibility of Schimmler et al., "A bit-serial floating-point unit for a massively parallel system

[10] https://web.archive.org/web/20000407195152/http://library.hku.hk/

on a chip," *Parallel Algorithms and Applications*, Vol. 19, No. 2-3, pp. 79-95 (2004) (hereafter the "Schimmler-2004 article"), and the authenticity of EX1008 attached hereto.

1. Authentication of EX1008

40. EX1008 is a copy of (1) the stamped cover of Volume 19, Numbers 2-3 of the journal *Parallel Algorithms and Applications*; (2) the stamped TOC; and (3) the Schimmler-2004 article contained therein (pages 79-95) that was found within the custody of the library at the University of Nebraska at Omaha, and from where I requested and obtained the copy in EX1008.

41. EX1008 appears to be a true and correct copy in a condition that creates no suspicion about its authenticity, particularly given that a stamped copy was obtained from the University of Nebraska library from its own collection. Specifically, the cover of the journal lists the title (*Parallel Algorithms and Applications*), ISSN ("1063-7192"), issue information ("Volume 19, Number 2-3 (June-September)), publisher ("Taylor & Francis"), a library sticker showing the classification number and issue date, and the cover was stamped by the library at the University of Nebraska on September 7, 2004. The TOC was also stamped by the library on September 7, 2004. Additionally, the first page of the Schimmler-2004 article includes the journal information and publisher logo (header), and the ISSN and a 2004 copyright notice from the publisher (footer) matching the

information on the cover and the TOC. The text of the article is complete, the page numbers are correct and there are no missing pages.

2. Public Accessibility of the Schimmler-2004 Article

42. Based on finding a stamped copy of the June-September issue (i.e., Volume 19, Numbers 2-3) of *Parallel Algorithms and Applications* containing the Schimmler-2004 article in the University of Nebraska library, it is my opinion that the Schimmler-2004 article was published and accessible by the public at least by September 7, 2004, the date the University of Nebraska library stamped its own copy of the journal.

43. Additionally, the MARC record for the *Parallel Algorithms and Applications* journal attached hereto as Attachment 2a was created December 16, 2002 and shows that the journal was in print from "vol. 1, no. 1, published in 1993" though "vol. 19 in 2004" and that there are 1018 holdings for this journal. Attachment 2a also shows that the MARC record for *Parallel Algorithms and Applications* journal was entered on December 16, 2002 by the division of OCLC that creates MARC records that libraries subscribe to and download (OCLC symbol "F#A"), which is often done for packages of e-books and electronic journals that were publicly accessible. Attachment 2e is the list of institutions the OCLC database indicates are "Holding[]" copies of the *Parallel Algorithms and Applications* journal, including the University of Nebraska—Lincoln library

(OCLC symbol “LDL”) and numerous other libraries such as the University of Arizona (OCLC symbol “AZU”), University of California, Berkeley (OCLC symbol “CUY”), Massachusetts Institute of Technology (OCLC symbol “MYG”), North Carolina State University (OCLC symbol “NRC”), Ohio State University (OCLC symbol “OSU”), and Penn State University (OCLC symbol “UPM”), which are institutions from which this journal would be accessible.

44. The Schimmler-2004 article was also published online at least by August 6, 2006. Attachment 2b is the search results from Taylor and Francis online for the Schimmler-2004 article which list the online publication date of August 6, 2006. Attachment 2c is the PDF of the Schimmler-2004 article downloaded from Taylor & Francis online and Attachment 2d is a screenshot of the PDF properties for the Schimmler-2004 article from Taylor and Francis Online showing a PDF creation date of November 17, 2004, corroborating that the version available from Taylor & Francis is not a later version of the Schimmler-2004 article.

45. To the best of my knowledge based on my review, the electronic copy of the Schimmler-2004 article downloaded from Taylor and Francis Online includes the same content as EX1008. The pagination, journal citation, publisher, ISSN and copyright notice match between the two copies. Based on this information, it is my opinion that the Schimmler-2004 article was published and publicly accessible and could have been located by any member of the public

interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian, either from the publisher or from a library, such as the University of Nebraska library from which EX1008 was obtained, by September 7, 2004 or shortly thereafter in the year 2004 (and in the subsequent years), and/or could have obtained an electronic copy from Taylor and Francis online by August 6, 2006 (and anytime thereafter).

C. Document 3 ("Schmidt-1999")

46. Below are my opinions with respect to the public accessibility of Schmidt et al., "A Parallel Accelerator Architecture for Multimedia Video Compression," in Euro-Par'99, Lecture Notes in Computer Science 1685, Springer Verlag, 1999, pp. 950–960 (hereafter the "Schmidt-1999 article"), and the authenticity of EX1009 attached hereto.

1. Authentication of EX1009

47. EX1009 is a copy of (1) the cover of Lecture Notes in Computer Science 1685, Euro-Par'99 Parallel Processing Conference Proceedings; (2) title and copyright pages; (3) Table of Contents; and (4) the Schmidt-1999 article contained therein (pages 950-960) that was found in the custody of the library of the University of Wisconsin at Madison, and from where I requested and obtained the copy in EX1009.

48. EX1009 appears to be a true and correct copy in a condition that creates no suspicion about its authenticity, particularly because the copy was obtained from the University of Wisconsin library from its own collection. Also, the cover includes logos of the proceedings, sponsors and publisher, the title and copyright pages include publisher logos and copyright notice ("© Springer-Verlag Berlin Heidelberg 1999" and "Printed in Germany"), and the first page of the Schmidt-1999 article (p. 950) includes the correct citation to the proceeding and the copyright notice. The article is complete, has no missing pages and its pagination matches the Table of Contents.

2. Public Accessibility of the Schmidt-1999 Article

49. Based on finding a copy of the Euro-Par'99 Parallel Processing Conference Proceedings in the University of Wisconsin library, that the copyright notice on the title page says it was "Printed in Germany" in 1999 and the August 31-September 3, 1999 conference dates on the cover and title pages, it is my opinion that the Schmidt-1999 article published and was accessible by the public at least within months of the conference dates and no later than the end of 1999.

50. Additionally, the MARC record for the Euro-Par'99 Parallel Processing Conference Proceedings in Attachment 3a attached hereto was created by the Library of Congress (DLC) on August 5, 1999 (which is consistent with the online publication date of August 6, 1999 on the Springer website shown in

Attachment 3b discussed below)[11] and shows 153 holdings for this proceeding. Attachment 3e is the list of institutions the OCLC database indicates are "Holding[]" copies of this proceeding, including the University of Wisconsin library (OCLC symbol "GZM") and numerous other libraries such as UCLA (OCLC symbol "CLU"), the University of Illinois (OCLC symbol "UIU"), the University of Kentucky (OCLC symbol "KUK"), and the University of Missouri, Columbia (OCLC symbol "MUU") which are institutions from which this journal would be accessible.

51. Further, the publisher Springer-Verlag (hereafter "Springer") was and is a well-known and respected publisher (founded in 1842) of academic journals and books and it is my opinion that Schmidt-1999 would have been available by late 1999 (and in subsequent years) from the publisher. Springer also provides an online search tool ("SpringerLink")[12] for providing researchers with access to millions of academic publications.

52. Attachment 3b attached hereto is the search result I obtained for the Schmidt-1999 article showing a "first online" date of January 1, 1999 and an online publication date of August 6, 1999. The search result page provides a link to

---

[11] https://link.springer.com/chapter/10.1007/3-540-48311-X_133

[12] https://link.springer.com/

download a copy of the Schmidt-1999 article (which I downloaded and which is attached hereto as Attachment 3c), which electronically downloadable PDF has a creation date of July 23, 1999 and the same pagination as the Schmidt-1999 article and appears to be the electronic version of the print copy published in the proceedings. This information corroborates my opinion that the Schmidt-1999 article was published and accessible by the public from multiple sources, including online, in the year 1999 and remained publicly available thereafter.

53. Based on this information, it is my opinion that the Schmidt-1999 article was published and publicly accessible and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian, either from the publisher (Springer-Verlag) or from a library, such as the Library of Congress or the University of Wisconsin from which I obtained a copy, by August 5, 1999 and at the latest the end of 1999 (and at any time thereafter in subsequent years).

D. Document 4 ("Schimmler-1994")

54. Following are my opinions with respect to the public accessibility of Schimmler et. al., "The Instruction Systolic Array – Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers," Lecture Notes in Computer Science 797, High Performance Computing and Networking,

International Conference and Exhibition Proceedings, Springer Verlag, 1994, pp. 487–488 (hereafter the "Schimmler-1994 article"), and the authenticity of EX1010.

1. Authentication of the Schimmler-1994 Article

55. EX1010 is a copy of (1) the cover of Lecture Notes in Computer Science 785, High-Performance Computing and Networking (HPCN) International Conference; (2) title page and a stamped copyright page; (3) Table of Contents; and (4) the Schimmler-1994 article contained therein (pages 487-488) that was found in the custody of the library of the University of Wisconsin at Madison, and from where I requested and obtained the copy in EX1010.

56. EX1010 appears to be a true and correct copy in a condition that creates no suspicion about its authenticity, particularly because the copy was obtained from the University of Wisconsin library from its own collection (and the copyright page was stamped). Also, the cover includes logos of the proceedings, sponsors and publisher, and the title and copyright pages include publisher logos and copyright notice ("© Springer-Verlag Berlin Heidelberg 1994" and "Printed in Germany"). The Schimmler-1994 article is complete, has no missing pages and its pagination matches the Table of Contents.

2. Public Accessibility of the Schimmler-1994 Article

57. Based on finding a copy of the HPCN '94 Proceeding in the University of Wisconsin library, that the copyright notice on the title page says it

was “Printed in Germany” in 1994 and the April 18-20, 1994 date of the international conference, it is my opinion that the Schimmler-1994 article was published and was accessible by the public in 1994. Attachment 4d attached hereto is the search result I obtained from the ACM digital library listing the publication date for Schimmler-1994 as April 18, 1994, corroborating the date on the title page of EX1010.

58. Additionally, the MARC record for the HPCN ’94 Proceeding attached hereto as Attachment 4a was created on June 30, 1994 by the University of Louisiana at Lafayette (OCLC symbol “LWA”) and shows 17 holdings for this proceeding at the time I retrieved that MARC record. Attachment 4c is the list of institutions the OCLC database indicates are “Holding[]” copies of this proceeding, including the University of Wisconsin library (OCLC symbol “GZM”) from which EX1010 was obtained and multiple other libraries such as the University of California Santa Barbara (OCLC symbol “CUT”), Georgia Tech University (OCLC symbol “GAT”), and the University of Pennsylvania (OCLC symbol “PAU”) which are institutions from which this journal would be accessible.

59. As noted previously (¶ 51 above), the publisher Springer-Verlag (hereafter “Springer”) was and is a well-known and respected publisher (founded in 1842) of academic journals and books and it is my opinion that the Schimmler-1994 article would have been available in 1994 and in subsequent years from the

publisher. Springer also provides an online search tool ("SpringerLink")[13] for providing researchers with access to millions of academic publications.

60. Attachment 4b attached hereto is the search result I obtained from SpringerLink for the Schimmler-1994 article showing a publication date of "first online" date of January 1, 2005 and an online publication date of May 26, 2005. Based on this information, it is my opinion that the Schimmler-1994 article was published and publicly accessible and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian, either from the publisher (Springer-Verlag) or from a library, such as the University of Louisiana at Lafayette and University of Wisconsin from which I obtained a copy, as early as April 18, 1994 and at least by June 20, 1994, and would have continued to be available thereafter, and further would have been accessible online for by May 26, 2005.

E. Document 5 ("Schimmler-1996")

61. Below are my opinions with respect to the authenticity and public accessibility of Schimmler et al., "The Instruction Systolic Array in Image Processing Applications," Vision Systems: Sensors, Sensor systems, and

[13] https://link.springer.com/

Components, Europto 96, O. Loffeld ed.: SPIE 2784, 1996, pp. 136–144 (hereafter the "Schimmler-1996 article") and the authenticity of EX1011.

1. Authentication of the Schimmler-1996 Article

62. EX1011 is a stamped copy of Vision Systems: Sensors, Sensor systems, and Components, Europto 96, published by SPIE in Volume 2784 that was found within the custody of the library at the University of Wisconsin at Madison, and from where I requested and obtained the copy in EX1011.

63. EX1011 appears to be a true and correct copy in a condition that creates no suspicion about its authenticity, particularly given that a stamped copy was obtained from the University of Wisconsin library from its own collection. Specifically, the cover lists the proceeding title (Vision Systems: Sensors, Sensor Systems, and Components), the publisher ("SPIE"), proceeding date ("10-12 June 1996"), a library sticker showing the classification number and issue date, and was stamped by the library at the University of Wisconsin on September 27, 1996. The text of the Schimmler-1996 article is complete, the page numbers are correct and match the Table of Contents, and each page includes the correction publisher and volume information (SPIE Vol. 2784).

2. Public Accessibility of the Schimmler-1996 Article

64. Based on finding stamped copy of the proceeding containing the Schimmler-1996 article in the University of Wisconsin library, it is my opinion

that the Schimmler-1996 article was published and publicly accessible and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence at least by September 17, 1996, the date the University of Wisconsin library stamped its own copy of the journal, or shortly thereafter in the year 1996.

65. Additionally, the MARC record for the Vision Systems: Sensors, Sensor systems, and Components attached hereto as Attachment 5a was created May 16, 1996 by the United States Air Force, Institute of Technology, Wright-Patterson Air Force Base, Ohio (OCLC symbol "OWpIT") and shows 68 holdings for this journal. Attachment 5c is the list of institutions the OCLC database indicates are "Holding[]" copies of this journal, including the University of Wisconsin (OCLC symbol "GZM") where EX1011 was obtained and numerous other libraries such as Georgia Tech University (OCLC symbol "GAT"), University of Maryland (OCLC symbol "UMC"), the Massachusetts Institute of Technology (OCLC symbol "MYG"), and the Linda Hall Library (OCLC symbol "LHL") which are institutions from which this journal would be accessible. Further, SPIE was and is a well-known international technical society and respected publisher of engineering and scientific journals and proceedings, and it is my opinion that the Schimmler-1996 article would have available at least by September 17, 1996 from the publisher.

66. SPIE also offers a Digital Library[14] that lists a publication date of August 21, 1996 for this title,[15] corroborating my opinion that the Schimmler-1996 article was published at least by September 17, 1996 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian, either from the publisher (SPIE) or from a library, such as the University of Wisconsin library from which I obtained a copy or from other libraries holding this title.

F. Document 6 ("Yeh-ISCA07")

67. Document 6 is Yeh, et al., "Parallax: An architecture for real-time physics," Computer Architecture *News*, Vol. 35, No. 2, pp. 232-243 (2007) (hereafter the "Yeh-ISCA07 article"). Below are my opinions on the public accessibility of the Yeh-ISCA07 article.

---

[14] https://www.spiedigitallibrary.org

[15] https://www.spiedigitallibrary.org/conference-proceedings-of-spie/2784/0000/Instruction-systolic-array-in-image-processing-applications/10.1117/12.248524.short?SSO=1

1. Authentication of EX1012

68. EX1012 is a copy of (1) the cover page and Table of Contents of Volume 35, Number 2 of Computer Architecture *News* dated May 2000, which was published in connection with the Proceedings of the 34th International Symposium on Computer Architecture (ISCA'07) and (2) the Yeh-ISCA07 article, published in this issue of Computer Architecture *News*. A print copy of this issue of Computer Architecture *News* was found within the custody of the University of Wisconsin's library. EX1012 is a true and correct copy of Yeh-ISCA07 that I requested and obtained from the University of Wisconsin's library. As the first page of EX1012 shows, the issue of Computer Architecture *News* containing the Yeh-ISCA07 article was published in 2007 by the Association for Computing Machinery ("ACM"), which was in the 2007 timeframe (and still is) a well-known, reputable compiler and publisher of scientific and technical publications. *See* ¶ 17 above.

69. EX1012 appears to be a true and correct copy of Yeh-ISCA07 in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete; no pages are missing and the page numbers for the Yeh-ISCA07 article (pp. 232-243) match the Table of Contents from the copy of the issue of Computer Architecture *News* in EX1012 and publication data for the Yeh-ISCA07 article available online and in records in other sources discussed below.

Additionally, the first page of the Yeh-ISCA07 article (p. 232) includes a copyright notice from the publisher ("Copyright 2007 ACM").

70. Attached hereto as Attachment 6a is a true and correct copy of the results page from the ACM Digital Library for my search of the Yeh-ISCA07 article in that database. As shown on the first page of Attachment 6a, the ACM Digital Library indicates that the Yeh-ISCA07 article was published on June 9, 2007 in connection with ISCA '07: Proceedings of the 34th annual symposium on Computer architecture, with the article on pages 232-242, which is consistent with the symposium and journal pages indicated in EX1012.

71. Attached hereto as Attachment 6b is a true and correct copy of the PDF of the Yeh-ISCA07 article that I retrieved from the ACM Digital Library. Attachment 6b appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete and no pages are missing. The first page of the Yeh-ISCA07 article (p. 232) indicates it was published in connection with "ISCA'07, June 9-13, 2007" and bears a "Copyright 2007 ACM," which are consistent with Attachment 6a and EX1012. The PDF I obtained from the ACM Digital Library (Attachment 6b) also matches the body of the Yeh-ISCA07 article in EX1012.

72. Attached hereto as Attachment 6c is a true and correct copy of a screenshot of the PDF properties for Attachment 6b that I retrieved from the ACM Digital Library. Attachment 6c shows a PDF creation date of April 7, 2007.

2. Public Accessibility of the Yeh-ISCA07 article

73. Based on the fact that Document 6 (EX1012) (1) was found at the University of Wisconsin's library in a print copy bearing a 2007 copyright date in a journal indicating it was published by the ACM in May 2007, (2) is retrievable from the ACM Digital Library, with a publication date indicated as June 9, 2007, and (3) the PDF properties of the PDF retrievable from the ACM Digital Library's shows a creation date of April 7, 2007, it is my opinion that the Yeh-ISCA07 article was publicly accessible by June 2007 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

G. Document 7 ("Yeh-Micro07")

74. Document 7 is Yeh, et al., "The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration," Proceedings of The 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07), pp. 394-406 (2007) (hereafter the "Yeh-Micro07 article"). Below are my opinions with respect to the authenticity and public accessibility of the Yeh-Micro07 article.

1. Authentication of EX1013

75. EX1013 is a copy of (1) the cover page and Table of Contents of the Proceedings of The 40th Annual IEEE/ACM International Symposium on Microarchitecture International Symposium on Computer Architecture on December 1-5, 2007, in Chicago, Illinois (the "IEEE Microarchitecture Proceedings") and (2) the Yeh-Micro07 article, published in this journal. A print copy of this issue of the IEEE Microarchitecture Proceedings with Yeh-Micro07 was found within the custody of the Linda Hall Library in Kansas City, MO. EX1013 is a true and correct copy of Yeh-Micro07 that I requested and obtained from the Linda Hall Library. As the first page of EX1013 shows, the issue of the IEEE Microarchitecture Proceedings containing the Yeh-Micro07 article was published in 2007 by the IEEE, which was in the 2007 timeframe (and still is) a well-known, reputable compiler and publisher of scientific and technical publications. *See* ¶ 18 above (discussing IEEE and IEEE Xplore).

76. Exhibit F appears to be a true and correct copy of Yeh-Micro07 in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete; no pages are missing and the page numbers for the Yeh-Micro07 article (pp. 394-406) match the Table of Contents from the copy of the issue of IEEE Microarchitecture Proceedings in EX1013 and is consistent with the publication data for the Yeh-Micro07 article available online and in records in

other sources discussed below. Additionally, the first page of the Yeh-Micro07 article (p. 406) includes a copyright notice from the publisher ("© 2007 IEEE").

77. Attached hereto as Attachment 7a is a true and correct copy of the results page from IEEE Xplore for my search of the Yeh-Micro07 article in that database. *See* ¶ 18 above (discussing IEEE and IEEE Xplore). As shown on page 2 of Attachment 7a, IEEE Xplore indicates that the Yeh-Micro07 article was "Published in" the "40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO07), in connection with a December 1-5, 2007 conference in Chicago, Illinois, and was added to IEEE Xplore on December 26, 2007, which is consistent with the information found in Exhibit 6. The PDF of Yeh-Micro07 that I obtained from IEEE Xplore (Attachment 7a) matches the Yeh-Micro07 article in Exhibit 6.

78. Attached hereto as Attachment 7b is a true and correct copy of the PDF of the Yeh-Micro07 article that I retrieved from IEEE Xplore. Attachment 7b appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete and no pages are missing. The first page of the Yeh-ISCA07 article (p. 394) indicates it was published in connection with the "40th IEEE/ACM International Symposium on Microarchitectures" and bears a copyright "© 2007 IEEE," which are consistent

with Attachment 7a and EX1013. The PDF I obtained from IEEE Xplore (Attachment 7b) also matches the body of the Yeh-Micro07 article in Exhibit 6.

79. Attached hereto as Attachment 7c is a true and correct copy of a screenshot of the PDF properties for Attachment 7b. Attachment 7c shows a PDF creation date of October 11, 2007 and a modified date of October 25, 2022, which is the date I retrieved Attachment 7a from IEEE Xplore.

2. Public Accessibility of the Yeh-Micro07 article

80. Based on the fact that Document 7 (EX1013) (1) was found at the Linda Hall library in a print copy bearing a 2007 copyright date in a journal indicating it was published by the IEEE in 2007, (2) is retrievable from IEEE Xplore, and added to IEEE Xplore on December 26, 2007, and (3) the PDF properties of the PDF retrievable from IEEE Xplore shows a creation date of October 11, 2007, it is my opinion that the Yeh-Micro07 article was publicly accessible by December 2007 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

H. Document 8 ("Burger")

81. Document 8 is Burger, et. al., "Composable Multicore Chips," *in* Multicore Processors and Systems, pp. 73-109 (Springer-Verlag 2009) (hereafter

the "Burger article"). Below are my opinions with respect to the authenticity and public accessibility of the Burger article.

1. Authentication of Document 8

82. Attached hereto as Attachment 8a is a true and correct copy of the MARC record for Burger article that I retrieved from the WorldCat bibliographic database. As shown on page 1 of Attachment 8a, a MARC entry for the Burger article was entered on June 18, 2009. This MARC record shows 34 other holdings for the Burger article. Attachment 8b is the list of institutions the OCLC database indicates are "Holding[]" copies of the book Multicore Processors and Systems, which contains the Burger article, including the California Institute of Technology (OCLC Symbol "CIT"), the Library of Congress ("DLC"), and Cornell University (OCLC Symbol "COO"), which are institutions from which Multicore Processors and Systems would be accessible.

83. Attached hereto as Attachment 8c is a true and correct copy of the results page from my search of the Burger article in the SpringerLink database. *See* ¶ 120 below (discussing the SpringerLink database). Page 1 of Attachment 8c indicates that the Burger article was published on pages 73-109 of the book entitled "Multicore Processors and Systems" and was "First Online: 01 January 2009." As shown on page 14 of Attachment 16b, SpringerLink indicates a 2009 copyright: "©

2009 Springer-Verlag US." *See* ¶ 51 above (discussing the publisher Springer-Verlag).

84. Attached hereto as Attachment 8d is a true and copy of (1) the cover of Multicore Processors and System; (2) the TOC; and (3) the Burger article contained therein (pages 77-109) that was found within the PDF of the copy of Multicore Processors and System that I obtained from Springer Link. Attachment 8d appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the cover of the journal lists the title (*Multicore Processor and Systems*) and publisher ("Springer"). Page 4 of Attachment 8d identifies (1) a "Library of Congress Control Number: 2009930936," which indicates the journal was registered with the Library of Congress in 2009, (2) ISSN, ISBN, and e-ISBN numbers, and (3) bears a copyright "© Springer Science+Business Media, LLC 2009." The text of the Burger article on pages 73-109 is complete, the page numbers match the TOC, and there are no missing pages.

2. Public Accessibility of the Burger article

85. Based on the facts that (1) there is a MARC record in WorldCat with a creation date of June 18, 2009 for the Burger article, (2) SpringerLink indicates that the Burger article was first available online on January 1, 2009, and (3) a PDF of the book that contains the Burger article is retrievable from SpringerLink and bearing a 2009 copyright date, it is my opinion that the Burger article was indexed,

searchable, and publicly accessible by January 1, 2009, but at least no later than June 18, 2009, and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

I. Document 9 ("Thornton")

86. Document 9 is Thornton et al., "Tradeoff analysis of integer multiplier circuits implemented in FPGAs," 1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM 1999), pp. 301-304 (1999) (hereafter the "Thornton article"). Below are my opinions with respect to the authenticity and public accessibility of the Thornton article.

1. Authentication of Document 9

87. Attached hereto as Attachment 9a is a true and correct copy of the results page from IEEE Xplore for my search of the Thornton article in that database. As shown on page 2 of Attachment 9a, IEEE Xplore indicates that the Thornton article was "Published in" the "1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing," in connection with an August 22-24, 1999 conference in Victoria, BC and was added to IEEE Xplore on August 6, 2002. *See* ¶ 18 above (discussing IEEE and IEEE Xplore).

88. Attached hereto as Attachment 9b is a true and correct copy of the PDF of the Thornton article that I retrieved from IEEE Xplore. Attachment 9b

appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete and no pages are missing. The first page of the Thornton article (p. 301) includes a copyright notice from the publisher ("© 1999 IEEE"), which was in the 1999-2002 timeframe (and still is) a well-known, reputable complier and publisher of scientific and technical publications.

89. Attached hereto as Attachment 9c is a true and correct copy of a screenshot of the PDF properties for Attachment 9b. Attachment 9c – the screen capture of the PDF properties of Attachment 9b – shows a creation date of March 22, 2004 and a modified date of November 18, 2022, which is the date I retrieved Attachment 9b from IEEE Xplore.

2. Public Accessibility of the Thornton article

90. Based on the fact that Attachment 9b (1) is retrievable from IEEE Xplore, and added to IEEE Xplore on August 6, 2002, and (2) the PDF properties of the PDF retrievable from IEEE Xplore shows a creation date of March 22, 2004, it is my opinion that the Thornton article was publicly accessible by August 2002 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

J. Document 10 ("Oh")

91. Document 10 is Oh et al., "A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor," IEEE Journal of Solid-State Circuits, vol. 41, no. 4, pp. 759-771 (April 2006) (hereafter the "Oh article"). Below are my opinions with respect to the authenticity and public accessibility of the Oh article.

1. Authentication of Document 10

92. Attached hereto as Attachment 10a is a true and correct copy of the results page from IEEE Xplore for my search of the Oh article in that database. As shown on pages 1-2 of Attachment 10a, IEEE Xplore indicates that the Oh article was "Published in" the "Digest of Technical Papers, 2005 Symposium on VLSI Circuits, 2005," in connection with a June 16-18, 2005 conference, and was added to IEEE Xplore on July 25, 2005. Page 2 of Attachment 10a further provide a "Print ISBN" number and indicates that the publisher was the IEEE, which was in the 2005 timeframe (and still is) a well-known, reputable complier and publisher of scientific and technical publications. *See* ¶ 18 above (discussing IEEE and IEEE Xplore).

93. Attached hereto as Attachment 10b is a true and correct copy of the PDF of the Oh article that I retrieved from IEEE Xplore. Attachment 10b appears to be a true and correct copy in a condition that creates no suspicion about its

authenticity. Specifically, the text of the article is complete and no pages are missing. The first page of the Oh article (p. 24) indicates it was published in connection with the 2005 Symposium on VLSI Circuits Digest of Technical Papers, which is consistent with Attachment 10a.

94. Attached hereto as Attachment 10c is a true and correct copy of a screenshot of the PDF properties for Attachment 10b. Attachment 10c shows a creation date of May 19, 2005 and a modified date of November 18, 2022, which is the date I retrieved Attachment 10b from IEEE Xplore.

2. Public Accessibility of the Oh article

95. Based on the fact that Attachment 10b (1) is retrievable from IEEE Xplore, and was added to IEEE Xplore on July 25, 2005, and (2) the PDF properties of the PDF retrievable from IEEE Xplore shows a creation date of May 19, 2005, it is my opinion that the Oh article was publicly accessible by July 2005 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

K. Document 11 ("Borkar")

96. Document 11 is S. Borkar, "Thousand Core Chips A Technology Perspective," Proceedings of the 44th ACM/IEEE Design Automation Conference,

pp. 746-749 (2007) (hereafter the "Borkar article"). Below are my opinions with respect to the authenticity and public accessibility of the Borkar article.

1. Authentication of Document 11

97. Attached hereto as Attachment 11a is a true and correct copy of the results page from the ACM Digital Library for my search of the Borkar article in that database. As shown on pages 1-2 of Attachment 11a, the ACM Digital Library indicates that the Borkar article was "Published" June 4, 2007, in connection with a June 2007 conference. *See* ¶ 18 above (discussing the ACM and ACM Digital Library).

98. Attached hereto as Attachment 11b is a true and correct copy of the PDF of the Borkar article that I retrieved from the ACM Digital Library. Attachment 11b appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete and no pages are missing. The first page of the Borkar article (p. 716) indicates it was published in connection with DAC 2007 in 2007 and bears a copyright from the publisher ACM: "Copyright 2007 ACM," which is consistent with Attachment 11a.

99. Attached hereto as Attachment 11c is a true and correct copy of a screenshot of the PDF properties for Attachment 11b. Attachment 11c shows a

creation date of May 2, 2007 and a modified date of December 4, 2022, which is the date I retrieved Attachment 11b from the ACM Digital Library.

2. Public Accessibility of the Borkar article

100. Based on the fact that Attachment 11b (1) is retrievable from the ACM Digital Library, which lists a publication date of June 4, 2007, and (2) the PDF properties of the PDF retrievable from the ACM Digital Library shows a creation date of May 2, 2007, it is my opinion that the Borkar article was publicly accessible by June 2007 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

L. Document 12 ("Stackhouse")

101. Document 12 is Stackhouse et. al., "A 65 nm 2-Billion Transistor Quad-Core Itanium Processor" Solid-State Circuits," 2008 IEEE International Solid-State Circuits Conference – Digest of Technical Papers, pp. 92-93, 598 (2009) (hereafter the "Stackhouse article"). Below are my opinions with respect to the authenticity and public accessibility of the Stackhouse article.

1. Authentication of Document 12

102. Attached hereto as Attachment 12a is a true and correct copy of the results page from IEEE Xplore for my search of the Stackhouse article in that database. As shown on pages 1-2 of Attachment 12a, IEEE Xplore indicates that

the Stackhouse article was "Published in" the "2008 IEEE International Solid-State Circuits Conference – Digest of Technical Papers," in connection with a February 3-7, 2008 conference and was added to IEEE Xplore on January 9, 2009. Page 2 of Attachment 12a further indicates that the publisher was the IEEE, which was in the 2008-2009 timeframe (and still is) a well-known, reputable complier and publisher of scientific and technical publications. *See* ¶ 18 above (discussing IEEE and IEEE Xplore).

103. Attached hereto as Attachment 12b is a true and correct copy of the PDF of the Stackhouse article that I retrieved from IEEE Xplore. Attachment 12b appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete and no pages are missing. The first page of the Stackhouse article (p. 92) indicates it was published in connection with the 2008 IEEE International Solid-State Circuits Conference and bears a 2008 copyright ("©2008 IEEE"), which are consistent with Attachment 12a.

104. Attached hereto as Attachment 12c is a true and correct copy of a screenshot of the PDF properties for Attachment 12b. Attachment 12c shows a creation date of February 24, 2008 and a modified date of November 18, 2022, which is the date I retrieved Attachment 12b from IEEE Xplore.

2. Public Accessibility of the Stackhouse article

105. Based on the fact that Attachment 12b (1) is retrievable from IEEE Xplore, and was added to IEEE Xplore on January 9, 2009 and (2) the PDF properties of the PDF retrievable from IEEE Xplore shows a creation date of February 24, 2008, it is my opinion that the Stackhouse article was publicly accessible at least as of January 9, 2009 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

M. Document 13 ("Bradley")

106. Document 13 is Bradley, et; al., "Exploiting Loop-Level Parallelism for SIMD Arrays Using OpenMP," *in* Vol. 4935 of the Lecture Notes in Computer Science ("LNCS"), pp. 89-199 (Springer-Verlag 2008) in connection with International Workshop on OpenMP, IWOMP 2007 Beijing, China, June 3-7, 2007 (hereafter "the Bradley article"). Below are my opinions with respect to the authenticity and public accessibility of the Bradley article.

1. Authentication of Document 13

107. Attached hereto as Attachment 13a is a true and correct copy of the MARC record for the Bradley article that I retrieved from the WorldCat bibliographic database. As shown on page 1 of Attachment 13a, a MARC entry for the journal containing the Bradley article was entered on November 11, 2008. This

MARC record shows 361 other holdings for the journal containing the Bradley article. The title of the Bradley article – "Exploiting Loop-Level Parnellism for SIMD Arrays Using Open MP – is listed in the MARC entry 505, which is used to indicate the titles of separate works or parts of an item (i.e., the separate articles complied in Volume 4935 of LNCS). Attachment 13f is the list of institutions the OCLC database indicates are "Holding[]" copies of the Proceedings of the 3rd International Workshop on OpenMP, IWOMP 2007, Beijing China, June 3-7, 2007, which is indicated in MARC record entry 245 in Attachment 13a as the proceedings associated with the Bradley article. As shown in Attachment 13f, publications of this proceeding are available from numerous libraries, including North Carolina State University (OCLC symbol "NRC") and Technishe Universitat Hamburg, Germany (OCLC symbol "h9h"), which are institutions from which Vol. 4935 of the Lecture Notes in Computer Science would be accessible.

108. Attached hereto as Attachment 13b is a true and correct copy of the results page from my search of the Bradley article in the Springer Link database, which I discussed above in connection with the Bradley article. Page 3 of Attachment 13b indicates that the Bradley article was published on pages 89-100 of the printed journal. As shown on page 1 of Attachment 13b, SpringerLink indicates a 2008 copyright: "© 2008." Pages 7-8 of Attachment 13c identifies (1)

an eBook ISBN, Series ISSN, Softcover ISBN, and Series E-ISSN numbers and (2) "Copyright Information": "Springer-Verlag Berlin Heidelberg 2008."

109. Attached hereto as Attachment 13c is a true and copy of the Bradley article that I retrieved from the Springer Link database. Attachment 13c appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. The text of the Bradley article on pages 89-100 is complete, the page numbers match the page numbers indicated in Attachment 13b, and there are no missing pages. The front page of Attachment 13c also bears a 2008 copyright of the publisher: "© Springer-Verlag Berlin Heidelberg 2008." *See* ¶ 51 above (discussing the publisher Springer-Verlag); ¶ 120 below (discussing the SpringerLink database). Page 1 of Attachment 16b indicates that the Burger article was published on pages 77-109 of the book entitled "Multicore Processors and Systems" and was "First Online: 01 January 2009." As shown on page 14 of Attachment 16b, SpringerLink indicates a 2009 copyright: "© 2009 Springer-Verlag US."

110. Attached hereto as Attachment 13d is a true and correct copy of a screenshot of the PDF properties for a PDF of the entirety of Volume 4935 of LNCS that I retrieved from SpringerLink, which contains the Bradley article. Attachment 13d shows that the PDF of Volume 4935 of LNCS that I retrieved

from SpringerLink has a creation date of July 7, 2008 and a modified date of July 8, 2008.

111. Attached hereto as Attachment 13e is a true and correct copy of a screenshot of the PDF properties for a PDF of the Bradley article itself that I obtained from Springer Link. Attachment 13e shows a creation date of April 1, 2008 and a modified date of July 8, 2008.

2. Public Accessibility of the Bradley article

112. Based on the facts that (1) there is a MARC record in WorldCat with a creation date of November 11, 2008 for the Bradley article, (2) a PDF of the Bradley article is retrievable form SpringerLink with a creation date of April 1, 2008, and bearing a 2008 copyright date, and (3) a PDF of the journal that contains the Bradley article is retrievable from SpringerLink, with a creation date of July 7, 2008 and bearing a 2008 copyright date, it is my opinion that the Bradley article was indexed, searchable, and publicly accessible at least as of November 11, 2008 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

N. Document 14 (“Roma”)

113. Document 14 is Roma et al., “In the development and evaluation of specialized processors for computing high-order 2-D image moments in real-time,”

in *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*, pp. 170-179 (2000) (hereafter the "Roma article"). Below are my opinions with respect to the authenticity and public accessibility of the Roma article.

1. Authentication of Document 14

114. Attached hereto as Attachment 14a is a true and correct copy of the results page from IEEE Xplore for my search of the Roma article in that database. As shown on pages 1-2 of Attachment 14a, IEEE Xplore indicates that the Roma article was "Published in" the "Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception," in connection with a September 11-13, 2000 conference, and was added to IEEE Xplore on August 6, 2002. Page 2 of Attachment 14a further provide a "Print ISBN" number and indicates that the publisher was the IEEE, which was in the 2000-2002 timeframe (and still is) a well-known, reputable complier and publisher of scientific and technical publications. *See* ¶ 18 above (discussing IEEE and IEEE Xplore).

115. Attached hereto as Attachment 14b is a true and correct copy of the PDF of the Roma article that I retrieved from IEEE Xplore. Attachment 14b appears to be a true and correct copy in a condition that creates no suspicion about its authenticity. Specifically, the text of the article is complete and no pages are

missing. The first page of the Roma article (p. 170) indicates it was published in by the IEEE because it bears a copyright "© 2000 IEEE."

116. Attached hereto as Attachment 14c is a true and correct copy of a screenshot of the PDF properties for Attachment 14. Attachment 14c shows a creation date of March 1, 2004 and a modified date of December 5, 2022, which is the date I retrieved Attachment 15b from IEEE Xplore.

2. Public Accessibility of the Roma article

117. Based on the fact that Attachment 14b (1) is retrievable from IEEE Xplore, and added to IEEE Xplore on August 6, 2002 and bears a year 2000 copyright by the IEEE and (2) the PDF properties of the PDF retrievable from IEEE Xplore shows a creation date of March 1, 2004, it is my opinion that the Roma article was publicly accessible by August 6, 2002, but at least no later than March 1, 2004, and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

O. Document 15 ("Lokhmotov")

118. Document 15 is Lokhmotov, et; al., "Revisiting SIMD Programming," *in* Vol. 5234 of the Lecture Notes in Computer Science ("LNCS"), pp. 32-46 (Springer-Verlag 2008) in connection with the 20th International Workshop, LPCP 2007, Urbana, IL, USA, October 2007 (hereafter the "Lokhmotov article"). Below

are my opinions with respect to the authenticity and public accessibility of the Lokhmotov article.

1. Authentication of Document 15

119. Attached hereto as Attachment 15a is a true and correct copy of the MARC record for Lokhmotov article that I retrieved from the WorldCat bibliographic database. As shown on page 1 of Attachment 15a, a MARC entry for the Lokhmotov article was entered on August 12, 2008. This MARC record shows 24 other holdings for the Lokhmotov article. Attachment 15e is the list of institutions the OCLC database indicates are "Holding[]" copies of selected papers from the Languages and Compilers for Parallel Computing : 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007 which is indicated in MARC record entry 245 in Attachment 15a as the conference associated with the Lokhmotov article. As shown in Attachment 15e, the selected papers from this conference that included the Lokhmotov article are available from numerous libraries, including University of New Hampshire (OCLC symbol "NHM"), Rutgers University (OCLC symbol "NJR"), and Cornell University (OCLC symbol "COO"), which are institutions from which Vol. 5234 of the Lecture Notes in Computer Science would be accessible.

120. Attached hereto as Attachment 15b is a true and correct copy of the results page from my search of the Lokhmotov article in the Springer Link

database. Springer-Verlag was in the 2007-2008 timeframe (and still is) a well-known, reputable compiler and publisher of scientific and technical publications. *See* ¶ 51 above (discussing the publisher Springer-Verlag). SpringerLink is a "comprehensive online collection of scientific, technological and medical journals, books and reference works" that "offers electronic and printed literature from Springer-Verlag" and other reputable publishers, like Urban and Vogel, Steinkopff, and Birkhäuser."[16] Page 1 of Attachment 15b indicates that the Lokhmotov article was published on pages 32-46 of the printed journal. As shown on page 7 of Attachment 15b, SpringerLink indicates a 2008 copyright: "© 2008 Springer-Verlag Berlin Heidelberg." As shown on page 8 of Attachment 15b, LNCS Volume 5234 had a "Print ISBN" of "978-3-540-85260-5."

121. Attached hereto as Attachment 15c is a true and copy of (1) the cover of Volume 5234 of the LNCS; (2) the TOC; and (3) the Lohkmotov article contained therein (pages 32-46) that was found within the PDF of Volume 5234 of the LNCS that I obtained from Springer Link. Attachment 15c appears to be a true and correct copy in a condition that creates no suspicion about its authenticity.

---

[16] https://www.springer.com/gp/help/about-springerlink/18548#:~:text=SpringerLink%20is%20the%20world%27s%20most,journals%2C%20books%20and%20reference%20works.

Specifically, the cover of the journal lists the title (*Languages and Compilers for Parallel Computing*), the issue information ("LNCS 5234"), and publisher ("Springer"). Page 4 of Attachment 15c identifies (1) a "Library of Congress Control Number: 2008932585," which indicates the journal was registered with the Library of Congress in 2008, (2) ISSN, ISBN-10, and ISBN-13 numbers, and (3) bears a copyright "© Springer-Verlag Berlin Heidelberg 2008." The text of the Lokhmotov article on pages 32-46 is complete, the page numbers match the TOC, and there are no missing pages. The ISBN-13 number in Attachment 15c matches the "Print ISBN" number on page 7 of Attachment 15b.

122. Attached hereto as Attachment 15d is a true and correct copy of a screenshot of the PDF properties for a PDF of the entirety of Volume 5234 of LNCS that I retrieved from SpringerLink, which contains the Lokhmotov article. Attachment 15d shows that the PDF of Volume 5234 of LNCS that I retrieved from SpringerLink has a creation date of August 4, 2008 and a modified date of August 12, 2008. Because that PDF was voluminous and contained several articles in addition to the Lohkmotov article, Attachment 15c is a PDF that I created from the PDF of the full journal that I retrieved from SpringerLink, comprising the cover page through TOC and the Lokhmotov article.

2. Public Accessibility of the Lokhmotov article

123. Based on the facts that (1) there is a MARC record in WorldCat with a creation date of August 12, 2008 for the Lokhmotov article and (2) a PDF of the journal that contains the Lokhmotov article is retrievable from SpringerLink, with a creation date of August 4, 2008 and bearing a 2008 copyright date, it is my opinion that the Lokhmotov article was indexed, searchable, and publicly accessible at least as of August 12, 2008 and could have been located by any member of the public interested in the subject matter by exercising reasonable diligence, on their own or with the assistance of a research librarian.

V. CONCLUSION

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements are made with the knowledge that willful false statements and the like are punishable by fine, imprisonment, or both, under Section 1001 of Title 18 of the U.S. Code. I state under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

Executed this December 11, 2022, in Lincoln, Nebraska.



Mary K. Bolin

ATTACHMENT 1A

JOURNAL

Parallel and distributed computing practices : PDCP.

Commack, N.Y. : Nova Science Publishers 1998

Available at **Main Library** Main Serials (S 004.35 P22 D) >

TOP

SEND TO

SEARCH INSIDE

GET IT

DETAILS

LINKS

VIRTUAL BROWSE

Send to

QR | READING LIST | EXPORT BIBTEX | CITATION | PERMALINK | PRINT | E-MAIL | ENDNOTE DESKTOP | ENDNOTE WEB

Search inside

Search for articles within this journal

Article title or keyword

Get It

To request, please Sign in

BACK TO LOCATIONS

LOCATION ITEMS

Main Library

Available , Main Serials ; S 004.35 P22 D

Summary holdings: v.2 (1999)-v.3 (2000) v.5 (2002)

LOCATE

Item in place (0 requests) v.5 2002
Loanable

Item in place (0 requests) v.3 2000
Loanable
Description: Bound Issue
Location: Main Library Main Serials S 004.35 P22 D
Barcode: X19726060

Item in place (0 requests) v.2 1999
Loanable

Details

Title	Parallel and distributed computing practices : PDCP.
Subjects	Parallel processing (Electronic computers) >
	Electronic data processing -- Distributed processing >
Other title	PDCP
Publisher	Nova Science Publishers
Format	v. : ill. ; 28 cm..
Identifier	ISSN : 1097-2803
Creation Date	1998
Language	English

Links

Virtual Browse

ATTACHMENT 1B

WorldCat: Parallel and distributed computing practices

OCLC	38060488	**No holdings in LDL - 8 other holdings**									
Continuing Resources		**Rec Stat**	c	**Entered**	19971209		**Replaced**	20210927102744.5			
Type	a	**ELvl**		**Srce**		**GPub**		**Ctrl**		**Lang**	eng
BLvl	s	**Form**		**Conf**	0	**Freq**	q	**MRec**		**Ctry**	nyu
S/L	0	**Orig**		**EntW**		**Regl**	r	**Alph**	a		
Desc	a	**SrTp**	p	**Cont**		**DtSt**	d	**Dates**	1998,2002		

010		sn 97001181
040		NSD ‡b eng ‡c NSD ‡d DLC ‡d WAU ‡d OCLCQ ‡d OCLCF ‡d CUS ‡d OCLCQ
012		‡l 1 ‡k 1
022	0_	1097-2803 ‡l 1097-2803 ‡2 1
037		‡b Nova Science Publishers, 6080 Jericho Turnpike, Ste. 207, Commack, NY 11725
042		pcc ‡a nsdp
050	00	QA76.54 ‡b .P368
082	10	005 ‡2 13
090		
049		LDLL
222	_0	Parallel and distributed computing practices
245	00	Parallel and distributed computing practices : ‡b PDCP.
246	30	PDCP
260		Commack, NY : ‡b Nova Science Publishers
300		v. : ‡b ill. ; ‡c 29 cm.
310		Quarterly
362	1_	Begain with vol. 1, no. 1 (Mar. 1998) ; ceased with vol. 5, no. 4 (Dec. 2002).
588		Description based on: Vol. 5, no. 1 (Jan. 2002); title from cover.
588		Latest issue consulted: Vol. 5, no. 2 (Apr. 2003).
650	_0	Real-time data processing ‡v Periodicals.
650	_7	Real-time data processing. ‡2 fast ‡0 (OCoLC)fst01091219
655	_7	Periodicals. ‡2 fast ‡0 (OCoLC)fst01411641
776	08	‡i Online version: ‡t Parallel and distributed computing practices ‡x 1097-2803 ‡w (OCoLC)608172397
785	00	‡t Scalable computing ‡x 1895-1767 ‡w (DLC) 2010252198 ‡w (OCoLC)70901537
850		DLC
856	41	‡u http://bibpurl.oclc.org/web/22890 ‡u http://www.scpe.org/ ‡7 0
029	1_	AU@ ‡b 000028435688

Action Status Delete Holdings _ Export _ Label _ Produce _ Update Holdings _ Validate _

ATTACHMENT 1C

All locations : No holdings in LDL - 8 other holdings

Parallel and distributed computing practices : PDCP. Commack, NY : Nova Science Publishers v. : ill. ; 29 cm.
OCLC:38060488

Holdings Information

STATE	LOCATIONS
CA	stf
DC	DLC
PA	DKC
WI	GZM
ON	cai
HK	H2K HUA
ZA	OL$

CLOSE WINDOW

ATTACHMENT 1D

Government of Canada Gouvernement du Canada

MARC display

Clear this record | Return to record | View/export saved records

```
LEADER 00000cas  2200000 a 4500
005    19980603073452.0
008    971209c19989999nyuqr p       0    0eng d
010    sn 97001181
012    |l1
022    1097-2803
035    (OCoLC)ocm38060488
037    |bNova Science Publishers, 6080 Jericho Turnpike, Ste. 207,
       Commack, NY 11725
040    NSD|cNSD|dCaOON
042    msc
049    CAIA
090    QA76.9.D5 P222
245 00 Parallel and distributed computing practices :|bPDCP.
246 13 PDCP
260    Commack, NY :|bNova Science Publishers,|cc1998-
300    v. :|bill. ;|c29 cm.
310    Quarterly
362 0  Vol. 1, no. 1 (Mar. 1998)-
500    Title from cover.
590    n-mf,ces c-ag
650  0 Electronic data processing|xDistributed processing
       |vPeriodicals.
650  0 Parallel processing (Electronic computers)|vPeriodicals.
710 2  Nova Science Publishers.
```

Location	Call number & Local holdings	Status

NRC Ottawa Ser	QA76.9.D5 P222	v. 1-4; 1998-2001.	In library use only

Date modified: 2022-09-29

ATTACHMENT 2A

WorldCat: Parallel algorithms and applications.

OCLC	51232058	**LDL Holdings - 1018 other holdings**									
Continuing Resources		**Rec Stat**	c	**Entered**	20021216	**Replaced**	20200716172314.6				
Type	a	**ELvl**		**Srce**	c	**GPub**		**Ctrl**		**Lang**	eng
BLvl	s	**Form**	o	**Conf**	0	**Freq**		**MRec**		**Ctry**	enk
S/L	0	**Orig**	s	**EntW**		**Regl**	x	**Alph**	a		
Desc	a	**SrTp**	p	**Cont**		**DtSt**	d	**Dates**	1993,2004		

007 c $b r $d m $e n $f u
010 2005249173
040 F#A $b eng $c F#A $d OCLCQ $d ZYU $d OCL $d MYG $d OCLCQ $d OCLCF $d OCLCQ $d WT2 $d OCLCQ $d AU@ $d OCLCQ $d SXB
016 7_ 2171287 -6 $2 DE-600
019 1058422557
022 $y 1063-7192
037 $b Taylor & Francis, Inc., 11 New Fetter Lane, London EC4P 4EE, UK
042 pcc
050 14 QA76.9.A43 $b P347
082 04 005.1
090
049 LDLL
130 0_ Parallel algorithms and applications (Online)
245 10 Parallel algorithms and applications.
260 [London] : $b Gordon and Breach
310 Irregular
336 text $b txt $2 rdacontent
337 computer $b c $2 rdamedia
338 online resource $b cr $2 rdacarrier
362 1_ Print began with vol. 1, no. 1, published in 1993; ceased with: vol. 19 in 2004.
530 Also issued in print.
538 Mode of access: World Wide Web.
588 0_ Vol. 17, no. 3 (2001); title from journal information screen (FirstSearch, viewed Dec. 13, 2002).
588 1_ Vol. 19, no. 1 (2004) (FirstSearch, viewed Oct. 27, 2005).
650 _0 Computer algorithms $v Periodicals.
650 _0 Parallel computers $v Periodicals.
650 _7 Algorithms. $2 ebps
650 _7 Parallel Computation. $2 ebps
650 _7 Computer algorithms. $2 fast $0 (OCoLC)fst00872010
650 _7 Parallel computers. $2 fast $0 (OCoLC)fst01052922
655 _0 Electronic journals.
655 _7 Periodicals. $2 fast $0 (OCoLC)fst01411641
776 1_ $t Parallel algorithms and applications $x 1063-7192 $w (DLC) 97661034 $w (OCoLC)26123823
785 00 $t International journal of parallel, emergent and distributed systems (Online) $x 1744-5779 $w (DLC) 2005249172 $w (OCoLC)57439120
856 40 $u http://firstsearch.oclc.org $z Address for accessing the journal using authorization number and password through OCLC FirstSearch Electronic Collections Online
856 40 $u http://firstsearch.oclc.org/journal=1063-7192;screen=info;ECOIP $z Address for accessing the journal from an authorized IP address through OCLC FirstSearch Electronic Collections Online
856 40 $u http://informaworld.com/openurl?genre=journal&issn=1744-5760 $z On website of later title
029 1_ AU@ $b 000028645546
029 1_ AU@ $b 000044938311
029 1_ AU@ $b 000045726210

029 1_ NZ1 $b 12399019
029 1_ OCLCS $b 6774
029 1_ OCLCS $b OP-24434

006 fields for Computer Files Computer File						
Audn	**Form** o	**File** d	**GPub**			
Action Status	Delete Holdings _	Export _	Label _	Produce _	Update Holdings _	Validate _

ATTACHMENT 2B

Taylor & Francis Online

Log in | Register

On Monday 5 December, 05:00-23:00 GMT, we'll be making some site updates on Taylor & Francis Online. You'll still be able to search, browse and read our articles, where access rights already apply. Registration, purchasing, activation of tokens, eprints and other features of Your Account will be unavailable during this scheduled work.

Home ▶ All Journals ▶ Parallel Algorithms and Applications ▶ List of Issues ▶ Volume 19, Issue 2-3 ▶ A bit-serial floating-point unit for a m ....

Parallel Algorithms and Applications ›
Volume 19, 2004 - Issue 2-3: Parallel and Distributed Algorithms

Full access

75 Views | 0 CrossRef citations to date | 0 Altmetric

Listen

Original Articles

A bit-serial floating-point unit for a massively parallel system on a chip

Manfred Schimmler *, **Bertil Schmidt** & Hans-Werner Lang ‡

Pages 79-95 | Received 23 Oct 2003, Published online: 06 Aug 2006

Download citation https://doi-org.libproxy.unl.edu/10.1080/10637190410001725454

Full Article Figures & data References Citations Metrics

Reprints & Permissions View PDF

Abstract

This paper presents the design of a new bit-serial floating-point unit (FPU). It has been developed for the processors of the instruction systolic array (ISA) parallel computer model. In contrast to conventional bit-parallel FPUs the bit-serial approach requires a different data format. Our FPU uses an IEEE compliant internal floating-

In this article

Keywords: Bit-serial floating point units Massively parallel processors Systolic arrays Floating-point unit

Introduction

Modern VLSI technology allows us to integrate massively parallel systems on a single chip. The area limitations for the processors of such systems require small but efficient computational units ,,. In particular, it may motivate the choice of a bit-serial data organization of the individual processors. The fine-grained processor arrays are used in a variety of applications where identical operations have to be performed on large data sets such as signal, image and video processing ,, cryptography , biological sequence analysis , scientific computing , visualization , and medical imaging . Implementations of relevant numerical and signal processing algorithms often require floating arithmetic in order to achieve adequate accuracy. Thus, the integration of efficient floating-point arithmetic is essential to these processors.

Motivated by the design of an instruction systolic array (ISA) with 1024 processors on one chip , an FPU has been designed that meets these requirements. Its bit-serial structure allows for a fine-grained pipelined implementation on a minimal area of silicon. The design has been guided by experiences with a lower-integrated predecessor version of the processor array, which is available as an add-on board for a PC .

In the early days of massively parallel machines the individual processors had no integrated FPUs . But when MFlops evolved as one of the most important measures for supercomputer performance, the designers added commercial bit-parallel floating-point processors as coprocessors to small PE-clusters of their massively parallel

not allow fitting both the FPUs and the PEs on the same die. But with the advances in semiconductor technology in the last decade, designers tended to increase the computational capabilities of the individual processors rather than designing parallel systems of floating-point processors.

More recent work has focused on FPU design for FPGAs. In Ref. a serial multiplier is combined with a parallel adder in order to achieve high performance with a limited number of CLBs. A similar approach is taken in Ref. , in order to investigate the resource utilization for the FPU in modern FPGAs. In Ref. a technique of parameterizing the FPU-design by the desired precision is discussed. Another adaptive approach has recently been taken in Ref. . A generator tool is presented that designs an FPU for FPGA implementation dependent on the input of performance requirements. Even though the performance results are impressive, all these designs are currently not suitable for a massively parallel processor array, since they are not optimized with respect to area and power consumption.

The rest of this paper is organized as follows. The second section describes the ISA principle. An overview of the individual ISA processor architecture is given in the third section. The design methods for implementing efficient floating-point operations on bit-serial hardware are described in the fourth section. The fifth section presents the corresponding execution units. The performance of several floating-point operations is given in the sixth section. The seventh section concludes the paper.

Principle of the ISA

The ISA , is a mesh-connected processor grid, where the processors are controlled by three streams of control information: instructions, row selectors and column selectors (see Fig. 1). The instructions are input in the upper left corner of the processor array, and from there they move step by step in horizontal and vertical direction through the

active during each clock cycle. In clock cycle k processor $(i-1,j)$ and $(i,j-1)$ execute the instruction that has been executed by processor (i, j) in clock cycle k. The selectors also move systolically through the array: the row selectors move horizontally from left to right; the column selectors move vertically from top to bottom. Selectors mask the execution of the instructions within the processors, i.e. an instruction is executed if and only if both selector bits, currently in that processor, are equal to one. Otherwise, a no-operation is executed.

Figure 1 Control flow in an ISA.



Display full size

Every processor has read and write access to its own memory. Besides that, it has a designated *communication register* (*C-register*) that can also be read by the four neighbour processors. Within each clock phase reading access is always performed before writing access. Thus, two adjacent processors can exchange data within a single clock cycle in which both processors overwrite the contents of their own *C*-register with the contents of the *C*-register of its neighbour. This convention avoids read/write

Processor Architecture

The processor architecture is part of a chip design containing an ISA of size 32×32. The main components of the processor architecture are a set of 64 data registers, the communication register, a unit for integer addition, logical operations and conditional instructions, a 16-bit multiplier, and a shifter/adder for floating-point arithmetic (see Fig. 2). The data word length is 16 bits. Data is processed bit-serially, such that the execution of each instruction takes exactly 16 clock cycles. However, there are macro-instructions formed as series of elementary instructions in order to handle data types consisting of more than one machine, word, e.g. double precision floating-point numbers (see "Floating Point Operations" section). After receiving an instruction, the processor stores it in the instruction register, decodes the two operand addresses and the destination address, retrieves the operands from the register file, executes the instruction, writes the result back to the destination register, and passes the instruction to the next processor. Since all this is done bit-serially, it can be pipelined on bit level, such that a new instruction can be fetched and processed every 16 clock cycles. For instance, the first bit of the result is computed already after the second bits of the operands have entered the respective unit.

Figure 2 Block diagram of the processor architecture.

Display full size

The instructions are 3-address instructions (two operand registers—op0 and op1 and destination register—dest) coded in two 32 bits. Several instructions are related to floating-point operations. However, these instructions in general do not make sense on their own, but only within macro-instructions, i.e. a sequence of instructions that realize a certain operation on floating-point numbers. The instruction

computes the sum of the exponents of two floating-point numbers, as required for a multiplication. Similarly, there exist instructions EXPSUB for computing the difference of two exponents and EXPSEL for selecting the greater exponent as the result exponent in the case of an addition of two floating-point numbers. The instructions SHADD (shift and add), shsub (shift and subtract), nrm (normalize), LDMUL (load multiplier), and MULT (multiply) are used in macro-instructions for computing the mantissa of the result. The instruction

double precision numbers. The instructions

support the conversion between the IEEE floating-point format and the internal format, forming the mantissa words and the exponent word of the internal format.

Floating Point Operations

The processor supports and double precision floating-point arithmetic floating-point registers are implemented by groups of 16-bit data registers, and floating-point operations are implemented as instruction macros, i.e. as sequences of elementary instructions.

Internal Floating Point Format

The special cases for IEEE numbers are coded in all bits of the number representation. For example, in order to find out if a number equals infinity the complete mantissa and the complete exponent have to be compared with 0 and −1, respectively. Therefore, a bit-serial compare to infinity (required for every arithmetic operation) would need at least 32 internal clock cycles for single and 64 for double. To solve this problem a conversion into an internal format is performed. The internal representation of a double *X* consists of five 16-bit words Here, *X*[4] contains the exponent of the double, the sign, the kind of representation (absolute value/sign or 2-complement), and information about possible special cases (infinity, zero, NaN=not a number). *X*[0] to *X*[3] contain the significant (mantissa) of the double, extended to 64 bits.

1.
2. Sm: sign of the significant, 0: positive, 1: negative
3. *T*1, *T*0: encoding of special cases: 00: normal number, 01: zero, 10: INIF (infinity), 11: NaN

complement representation

5. Se: sign of exponent (two identical bits for overflow detection)
6. *e*0,...,*e*9: binary representation of the exponent *E* in 2-complement (little endian mode)
7. 64 bit of the significant where 63 is the most significant bit (MSB):
8.

The value of such a double is:

Double numbers can be normalized or not normalized. A positive number and a number in absolute value/sign-representation is normalized iff bits 63, 62, 61 of the significant are 001, i.e. bits 63 and 62 hold the sign and then in bit 61 comes the MSB of the significant. As usual in absolute value/sign representation, the significant is positive and the sign is stored in the sm-bit of the exponent (see *X*[4]). A negative number in 2-complement is normalized iff bits 63, 62, 61 of the significant are 110. Again, bits 63 and 62 form the sign and it follows the most significant digit of the significant represented in 2-complement.

The four 16-bit words of the IEEE format are converted into 5 words of the internal format. The 52-bit mantissa is stored in four words, where in the first word an explicit 1 is introduced (this 1 is implicit in the IEEE representation). The first two 0's serve as two sign bits of the mantissa. Two sign bits are useful for detecting overflows: an overflow after an addition changes the second sign bit and a right shift for normalization is necessary. The exponent is stored in the last 12 bits of a separate word, also with two sign bits. The value 01 of the sign bits indicates *exponent overflow*, whereas 10 indicates *exponent underflow*. In these cases, the number is changed to INF

of the mantissa is irrelevant. Bit 12 is used internally for indicating if the mantissa is in 2-complement representation after an addition or subtraction. The conversion is supported by special instructions of the processor.

Floating Point Addition

An addition of two floating-point numbers is performed in the usual way: the two exponents are compared, the mantissa of the operand with smaller exponent is shifted to the right appropriately, then the mantissas are added. If any operand is negative and not yet in 2-complement representation, its mantissa is complemented before. The bit-serial organization allows on-the-fly computation with one single half adder. If any operand is 0, its mantissa is set to 0. The result mantissa is in 2-complement representation. The result exponent is the larger exponent of the two operands. The tag of the result is computed depending on the tags of the two operands according to Table I.

Table I Computation of result tags in the case of an addition

Download CSV Display Table

The sign of the result is the sign of the result mantissa, except in the case that any operand is INF, then the sign is equal to the sign of this operand. If the result is 0 or NaN, the sign is irrelevant. Since the operands consist of more than one word (e.g. of four mantissa words in the case of double precision floating-point numbers), shifting of a mantissa by more than 15 bits is realized by displaced addressing of the operands (see Fig. 3).

Figure 3 Addition of two 64-bit mantissas.

Display full size

In each case, in order to realize a shift distance of *s*, in the first step A_{-1} and B_{-1+k} are addressed, where B_{-1+k} is loaded into the shifter and delayed by bits. In the next step A_0 and B_{0+k} are addressed. While B_{0+k} is loaded to the shifter, the d bits of B_{-1+k} appear at the output of the shifter and are added to A_0. In the last step A_3 and B_{3+k} are addressed. However, B_{3+k} is an invalid operand, therefore it is cancelled, i.e. set to 0…0 or to 1…1 depending on the sign of the mantissa. If *k* becomes greater than 4, all *B*-operands are cancelled, i.e. set to 0…0. In the case that *B* has the greater exponent, then the difference of the exponents is negative and *A* has to be shifted. The situation is essentially the same as above with the roles of *A* and *B* interchanged.

Normalization

After an addition, in general the result mantissa is not in normalized form, i.e. there are not exactly two leading 0's or 1's. Several cases are possible: there is one leading digit, two leading digits, (this is the normalized case), three through 63 leading digits, or 64 leading digits. In the latter case, the result is considered to be 0, if the leading digits are 0. In all other cases a left shift by −1…62 is necessary so that the mantissa starts with 001 or 110. The exponent has to be corrected by this shift distance *A* counter; the leading digits counter (LDC) is used to count the number of leading digits (−2) of the result produced during an addition/subtraction or multiplication. The LDC also records

control register. But now this value has to be interpreted in a different way, since it is the distance for a left shift instead of a right shift as in the case of an addition. However, this can be done using the same hardware for normalization as for addition.

Floating Point Multiplication

A multiplication of two 64-bit mantissas is performed according to the scheme shown in Fig. 4, where e.g. denotes the 80-bit product of with B_0. The result would be 128 bit long, however, only the most significant 64 bits can be used for the result mantissa *F*. Therefore the above scheme is truncated, in order to reduce the number of 16-bit multiplications. The scheme is shown in Fig. 5.

Figure 4 Multiplication scheme for 4-word operands.



Display full size

Figure 5 Truncated multiplication scheme for 4-word operands.



Display full size

multiplications in the multiplier, which is actually a multiply and add unit. The error introduced by the truncation amounts to at most 3 bits. This seems to be tolerable, because the precision of the internal floating point format is 61 significant bits, whereas the original IEEE format has only 52 bits. If more precise arithmetic is required, the complete multiplications have to be executed. The exponents of the two floating-point numbers are added. The sign of the result is the exclusive-or of the two-operand signs. The tag of the result is computed according to Table II.

Table II Computation of result tags in the case a multiplication

Download CSV Display Table

Floating Point Units

Multiplier for Positive Operands

As explained in "Floating Point Operations" section, bit-serial floating-point multiplication can be implemented by a simple combination of integer addition (for the exponents) and long integer multiplication (for the significants). In this section, we describe design of the multiplier for this purpose.

Its original architecture is shown in Fig. 6. Before starting the multiplication, one operand *x* is loaded into a parallel register (the multiplicand register) with the MSB to least significant bit (LSB) from left to right. This is controlled by a signal *r* being high for one clock cycle and low for the rest of the instruction execution. Now the second operand *y* is input into the multiplier LSB-first. In each clock cycle, *i* one bit y_i of *y* is processed. It is multiplied with *x* by computing the bitwise AND. The result is added to the current sum, producing a new intermediate result. This result is stored in a carry save representation instead of standard binary representation. This means the

representation avoids the propagation of the carry bits for more than one position.

Figure 6 The original multiplier architecture for unsigned integers.



Display full size

The next bit y_{i+1} of y is processed in the following clock cycle. Since the result to be computed in this step has to be weighted by a factor of 2 with respect to the old intermediate result, we have to shift the old result by one position to the right before we add the new product. Propagating the sum bits to the right neighbour and recycling the carry bits to the same full adder that has just produced this carry performs this shift operation. The shift operation outputs one bit at the rightmost full adder. This is the ith bit of the final product counted from the LSB to the MSB.

This multiplier design has been used in the parallel computer Systola 1024 . One interesting property of this architecture is its capability to perform *multiply–add* operations. Each bit operation consists of the addition of a new bit product to the intermediate result and a shift operation. Therefore, the interconnection between adjacent full adders form a shift register (the sum register). Now we can use this shift

operand into the sum register of the multiplier with the LSB first. The first bits of the summand can be shifted in concurrently with the *x* operand. During the multiplication, in every clock cycle, we input one additional bit of the summand while one bit of *y* is processed.

The multiplier is capable of executing two different instructions, corresponding to the two phases for the multiplication. In the first phase, the operand *x* is loaded into the multiplicand register. We call this instruction LDMUL (load multiplicand). During the execution of a LDMUL-instruction, the *y*-input is forced to be 0. If a multiply–add operation is executed the least significant word of the operand *s* is shifted into the sum register during the LDMUL-instruction, too. The resulting bits produced by the first LDMUL-instruction are irrelevant.

The second phase is the multiplication with *y*. The instruction is called MULT (multiply). *y* is consumed bit by bit as described above. In a MULT-instruction, the *x*-input is not interpreted. The *s*-input is used to read the most significant word of the operand *s*. The output produces the least significant word of the result. After this MULT-instruction the most significant word of the result is still stored in the sum- and carry-registers of the multiplier. In order to output this rest of the result we need to execute another LDMUL-instruction. Therefore, the computation of the last example can be coded in three instructions:

Numerous applications in cryptography deal with long operands, e.g. current RSA standards operate with lengths between 1024 and 4096 bits . Such bit lengths cannot be processed in one operation of a standard processor. One multiplication with large operands can be decomposed into a sequence of smaller multiplications and additions.

bit multiplications as follows:

where

In this case, we need four multiplications of half the size and three additions to get one result of full size. If we recursively use this technique, we can perform an $N \times N$-bit multiplication by using an n-bit multiplier. It requires $N/n \cdot N/n$ small multiplications and additions.

The bit serial structure of the above multiplier allows for a simple and efficient execution of long multiplications: only the operand x is chopped into pieces where each piece has the length n of the multiplicand. After loading one of these pieces of x using a LDMUL-instruction the complete factor y is processed by a sequence of MULT-instructions. This takes $N/n+1$ instruction cycles. The result has the length $N+n$. For the complete long multiplication we need to add one such partial product for each piece of x.

To speed up this procedure, we can take advantage of the multiply–add capability of our multiplier. We start with the least significant word X_0 of x. It is multiplied by the complete operand y leading to a result words $RES_{N/n}$ $RES_{N/n-1}$...RES_0. RES_0 is the least significant word of the final result. Afterwards, we proceed with X_1. Using multiply–add, we compute Again we come up with an intermediate result with bits, where the least significant word is part of the final result. In the same way, we continue until we come to the most significant word of x. The result of its multiplication with y delivers the rest of the final result. The following example shows the multiplication of two 2-word operands x and y together with the addition with a third 2-word operand s according to the scheme described above.

1. LDMUL X_{low}, S_{low}, 0; //**first stage**: load x_{low}
2. MULT Y_{low}, S_{high}, RES_0; //compute lower result

4. LDMUL x_{high}, res_1, res_2, //**first stage**: compute 3rd partial result, **second**
5. //**stage**: load x_{high}
6. MULT Y_{low}, res_2 RES_1; //compute 2nd final result
7. MULT Y_{high}, 0, RES_2; //compute 3rd final result
8. LDMUL 0, 0, RES_3; //compute upper final result

Note that there are slots left to add 4-word operand; however, an overflow might occur. The programmer would have to take appropriate precautions.

This example can be extended to longer operands as described above. In general, the multiplication of two *m*-word operands requires $m\cdot(m+1)+1$ instruction cycles.

Extended Multiplier for Negative Operands

The multiplier of the previous section is capable of multiplying two unsigned integers. With only some minor modifications in the instruction set and in the hardware, it is possible to extend it to a signed multiplier where one operand may be negative in 2-complement representation. We make use of a well known property of 2-complement numbers .

PROPOSITION

Let *x* and *y* be *n*-bit numbers in 2-complement representation, i.e. $x, y \in [-2^{n-1}, 2^{n-1}-1]$ and let *x*' and *y*' be the values of *x* and *y* interpreted as unsigned integers, i.e. if ≥0 and if respectively if and if Let and Then, it holds where sign(*x*) and sign(*y*) denotes the sign-bit of *x* and *y*, respectively, i.e. if and if

Proof

We consider the four cases determined by the signs of *x* and *y*.

-
- =

- =

In case (iv) the value 2^{2n} can be omitted, because interpreted as a $2n$-bit number it is equal to 0. □

In the above proposition z' corresponds to the result of the multiplication of x and y with the unsigned multiplier of “Principle of the ISA” section. The terms and denote the correction steps associated with negative operands to compute z from z'. In order to extend the original multiplier to deal with 2-complement numbers these correction terms have to be integrated into the new multiplier architecture.

Let us assume that x is negative. According to the proposition has to be subtracted from the result z'. This is equivalent to adding the 2-complement of y to the most significant word of z' (which has the weight 2^n). In the bit-serial case, the 2-complement can be computed on-line with the multiplication by using a multiplexer and a half adder. For the implementation of the multiplication with negative x, we introduce the instruction MULTSX. MULTSX is basically identical to MULT. Additionally, it sets *xneg* to the sign (stored in the MSB) of x. It is used to select either 0 or $-y$ as additional summand to s. The addition is performed on-line with the input of y and bit-serially delivers at s_{in}. The resulting architecture is shown in Fig. 7.

Figure 7 Multiplier for nonnegative y and arbitrary x.

Display full size

Floating Point Adder

The floating point adder consists of a shifter, an adder, two units for displaced addressing of the operands, two complement units and a logic for cancellation of operands. The shifter is a configurable shift register of variable length between 0 and 15 bits (see Fig. 8).

Figure 8 16-bit shifter.

Display full size

For a floating-point addition, the result of the exponent comparison is stored in a special register, the shift control register (SCR). The highest bit SCR[11] indicates if the exponent difference is negative and determines the shift operand. Bits SCR[0–3] determine the delay of the shifter. If this is a non-negative number *s*, it is complemented by the. XOR-gates and the last 1-bit delay is activated, in order to achieve the required delay of 16−*s*. Bits SCR[4–5] are used for displaced addressing of the shift operand, as explained in the "Floating Point Operations" section. If any of the bits SCR[6–10] is different from SCR[11], the shift distance would be greater than 64, so the shift operand maybe cancelled. A bit-serial adder adds the shifted operand and the other operand. If any operand is negative and not yet in 2-complement representation, it is complemented before the addition.

Performance

addition of the exponents and the multiplication and rounding of the significants. The internal format allows us to detect all exceptions (0, infinity, or NaN) by interpretation of the tag-bits during the addition of the exponents in the first instruction. A truncated multiplication of 32-bit operands requires six multiplication operations: the first loads the least significant half of the first operand into the multiplier. The second multiplies it with the most significant half of the second operand. The third loads the most significant half of the first operand and generates the most significant 16 bits of the intermediate result. In the remaining three instructions, the intermediate result is added to the product of the most significant word of the first operand and the whole second operand. Table III shows detailed instructions for the multiplication.

Table III Multiplication of two single precision numbers with the presented FPU

Download CSV Display Table

The addition is more complicated, since the difference of the exponents provides the shift distance of one operand. It is computed in the first instruction again together with the tags for possible exceptions. Controlled by the shift control register (SCR) the indirect addressing fetches the right fractions of the significants to be added. The leading digit counter (LDC) counts the number of leading 0's in order to detect the final shifting distance for normalization after the addition. Finally, the result is shifted left by this distance, and the exponent of the result is decreased accordingly. Table IV shows details of the addition.

Table IV Addition of two single precision numbers with the proposed FPU

Download CSV Display Table

with we have a performance of one processor of 200/168 million floating-point operations/second or 1.2 MFlops; this amounts to 1.2 GFlops for one chip with 1024 processors or to 20 GFlops for a board with 16 chips. For double precision floating-point numbers, an addition takes 18 instruction cycles, a multiplication takes 16 instruction cycles. The performance of one processor is

with we have a performance of one processor of 200/272 million floating-point operations/second or 0.735 MFlops; this amounts to 735 MFlops for one chip with 1024 processors or to 12 GFlops for a board with 16 chips.

Also more complex floating-point operations can be computed with the presented FPU, e.g. division, square root. Table V shows the instruction count for several floating-point operations with the presented FPU.

Table V Instruction counts for several floating-point operations in single/double precision

Download CSV Display Table

Since instruction count for individual floating operations does not consider format conversion time and communication time within the processor array to fetch the operands, it is often a weak performance measure that does not reflect the behaviour of a complete application. Therefore, we will use the instruction count for two typical floating-point applications on the ISA in our performance evaluation (see Table VI). The two applications are 2-dimensional matrix multiplication and 1-dimensional fast Fourier transform (FFT).

Table VI Floating-point performance of the 2-dimensional matrix multiplication and 1-dimensional FFT on the ISA processor array of size 32 × 32

Matrix Multiplication

The problem is to compute the product of two matrices defined as . We first assume that (thus, fitting into the processor array). Suppose that matrices *A* and *B* have been loaded into the array. Then, the first column of *A*, which is in the first column of the array, is broadcast along the rows and stored intermediately in some single or double register *H*. Similarly, the first row of *B* is broadcast along the columns. Now each processor (*i*, *j*) has elements A_{i1} and B_{1j} available and can compute . The next step is to ringshift *A* by one column to the left in order to get the second column of *A*, elements A_{i2}, into the first column of the array, from where they can be broadcast along the rows. Similarly, *B* is ringshifted along the columns. After *n* such steps, registers *F* contain the product of *A* and *B*. Multiplication of larger matrices can be easily partitioned using 32 × 32 matrix multiplication as a building block.

Each iteration step consists of two floating-point operations per processor (addition and multiplication) and the ringshift/broadcast communication. The two floating-point operations need 21 instructions for single precision and 34 instructions for double precision. Ringshift/broadcast requires 3 instructions per register on the ISA. Communication costs can be halved by performing a 64 × 64 matrix multiplication and assigning 2 × 2 values to each processor. This leads to 30 instructions per iteration step for single precision and 49 instructions per iteration step for double precision, which corresponds to a performance of 0.853 GFlops (single) and 0.522 GFlops (double) receptively. The time required for format conversion can be neglected, since it is only done at the beginning and at the end of the procedure.

FFT

Figure 9 shows the signal flow graph of the FFT of a sequence of *n* points, here for It consists of log(*n*) stages. In each stage, a complex multiplication with some primitive

the n points. The output of stage k is permuted by interchanging the two halves of subsequences of length 2^k before being input to the next stage. Initially, the sequence of the n points is permuted according to the so-called bit reversal permutation.

Figure 9 Signal flow graph of an 8-point FFT.



Display full size

The implementation of the ID-FFT of size on a 32 × 32 ISA assigns two values to each processor. The computation in each stage consists of four (real) multiplications and six (real) additions in each processor. The implementation of the permutation between every two stages involves routing of data, which is performed either within the rows of the ISA (in the case of interchanging the two halves of subsequences of length ≤64) or by interchanging complete rows of the ISA (in the case of interchanging the two halves of longer subsequences). The interchanging of the *C*-register contents of two halves of *m* processors is implemented by *m*/2 ringshift operations.

The ten floating-point operations per stage require 112 instructions for single precision and 172 instructions for double precision. The necessary communication per iteration step can be performed on the average within 92 instruction (single) and 153 instruction (double), respectively. This leads to a performance of 0.613 GFlops (single) and 0.427 GFlops (double) for the processor array of size 32 × 32.

Conclusions and Future Work

In this paper, the design of a FPU has been presented. Its bit-serial nature allows for an area efficient and energy efficient implementation. Because bit-serial operations with the IEEE floating-point number representation are difficult to implement, we use an internal floating-point format that allows fast LSB-first arithmetic. Our FPU is optimally suited for a fine-grained pipelined architecture. That is why it has been used in the processing elements of the KPROC , a parallel system with 1024 processors on a single chip.

Most architectures require fast floating-point computation for any price. In case of a massively parallel system on a single chip, there are other criteria that might be even more important than latency: area and power consumption. In this case, it turns out that the bit serial approach provides significant advantages in comparison to the bit parallel one. Adder and multiplier can be implemented in a more economic way, exceptions can be detected earlier, and rounding and normalization are relatively simple in comparison to the parallel case. The disadvantage of higher number of clock cycles for one operation is compensated by an increased number of processing elements that fit on one chip.

Due to the advances in FPGA technology, it will soon be possible to integrate an array of several hundred simple processors on one single FPGA. An architecture like that could be immediately equipped with floating-point capabilities using the FPU presented in this paper.

References

References

[Google Scholar]

2. Hennessy JL Patterson DA 1996 *Computer Architecture: A Quantitative Approach, Morgan Kaufman Publishers* [Google Scholar]

3. Hillis D 1986 *The Connection Machine, MIT Press* Cambridge, MA [Google Scholar]

4. Kunde M et al. 1988 The instruction systolic array and its relation to other models of parallel computers *Parallel Computing* 7 25 39 [Google Scholar]

5. Lang, HW. 1986. The instruction systolic array, a parallel architecture for VLSI. *Integration, the VLSI Journal*, 4: 65–74. [Google Scholar]

6. Lang HW Maaβ R Schimmler M 1994 The instruction systolic array—implementation of a low-cost parallel architecture as add-on board for personal computers *Proceedings of HPCN'94, LNCS 797, Springer* 487 488 [Google Scholar]

7. Liang J Tessier R Mencer O 2003 Floating point unit generation and evaluation for FPGAs *Proceedings of FCCM'03, IEEE Computer Society* [Google Scholar]

8. Lienhart G Kugel A Männer R 2002 Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulation *Proceedings of FCCM'02, IEEE Computer Society* [Google Scholar]

9. Ligon WB McMillan S Monn G Schoonover K Stivers F Underwood KD 1998 A re-evaluation of the practicality of floating-point operations on FPGAs *Proceedings of FCCM'98, IEEE Computer Society* [Google Scholar]

Society 107 116 [Google Scholar]

11. Meißner M Grimm S Straßer W Packer J Latimer D 2001 Parallel volume rendering on a single chip SIMD architecture *Proceedings of the IEEE Symposium on Parallel and Large Data Visualization and Graphics, IEEE Computer Society* 107 113 [Google Scholar]

12. Menezes AJ van Oorschot PC Vanstone SA 1997 *Handbook of Applied Cryptography, CRC Press* [Google Scholar]

13. Schmidt, B, Schröder, H and Schimmler, M. 2001. Tomographic image reconstruction on the instruction systolic array. *Computing and Informatics*, 20: 27–42. [Google Scholar]

14. Schmidt, B and Schimmler, M. 2000. KPROC—an instruction systolic architecture for parallel prefix applications. *Parallel and Distributed Computing Practices*, 3(2): 163–178. [Google Scholar]

15. Schmidt B Schimmler M Schröder H 1998 Long operand arithmetic on instruction systolic computer architectures and its application to RSA cryptography *Proceedings of Euro-Par'98* LNCS 1470 *Springer* 916 922 [Google Scholar]

16. Schmidt B Schimmler M 1999 A parallel accelerator architecture for multimedia video compression *Proceedings of Euro-Par'99* LNCS 1685 *Springer* 950 960 [Google Scholar]

17. Schmidt B Schröder H Schimmler M 2002 Massively parallel solutions for molecular sequence analysis *Proceedings of IPDPS'02, IEEE Computer Society* [Google Scholar]

19. Sim, LC, Schröder, H, Chian, LC and Leedham, G. 2003. Fast solution of large N × N matrix equations in an MIMD–SIMD hybrid system. *Parallel Computing*, 29(11–12): 1669–1684. [Google Scholar]

20. Taveniku M Ahlander A Jonsson M Svensson B 1998 The VEGA moderately parallel MIMD, moderately parallel SIMD, architectures for high performance array signal processing *Proceedings of IPPS/SPDP'98, IEEE Computer Society* 226 232 [Google Scholar]

21. Thiel, T. 1994. The design of the connection machine. *Design Issues*, 10(1) [Google Scholar]

22. Valafar F Ersoy OK 1995 A parallel implementation of backpropagation neural network on masspar MP-1 *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens* GA November 158–166. [Google Scholar]

Related research

Recommended articles Cited by

parallel parsing from recurrence equations

C. Barsan et al.
International Journal of Computer Mathematics
Published online: 19 Mar 2007

Parallel inference algorithms for the connection method on systolic arrays

International Journal of Computer Mathematics
Published online: 19 Mar 2007

Massively parallel >

J. S. Lucas
Contemporary Physics
Published online: 20 Aug 2006

View more

Information for

Authors

R&D professionals

Editors

Librarians

Societies

Opportunities

Reprints and e-prints

Advertising solutions

Accelerated publication

Corporate access solutions

Open access

Overview

Open journals

Open Select

Dove Medical Press

F1000Research

Help and information

Help and contact

Newsroom

All journals

Books

Keep up to date

Register to receive personalised research and resources by email

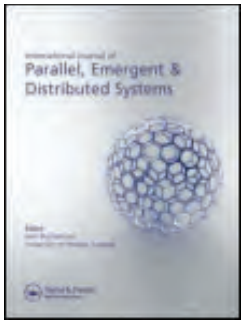
Sign me up

Copyright © 2022 Informa UK Limited Privacy policy Cookies Terms & conditions

Accessibility

Registered in England & Wales No. 3099067
5 Howick Place | London | SW1P 1WG

ATTACHMENT 2C



Parallel Algorithms and Applications

ISSN: 1063-7192 (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/gpaa19

A bit-serial floating-point unit for a massively parallel system on a chip

Manfred Schimmler , Bertil Schmidt & Hans-Werner Lang

To cite this article: Manfred Schimmler , Bertil Schmidt & Hans-Werner Lang (2004) A bit-serial floating-point unit for a massively parallel system on a chip, Parallel Algorithms and Applications, 19:2-3, 79-95, DOI: 10.1080/10637190410001725454

To link to this article: https://doi.org/10.1080/10637190410001725454

Published online: 06 Aug 2006.

Submit your article to this journal

Article views: 75

View related articles

Full Terms & Conditions of access and use can be found at
https://www.tandfonline.com/action/journalInformation?journalCode=gpaa20

Parallel Algorithms and Applications,
VOL. 19 (2–3) June–September 2004, pp. 79–95

Taylor & Francis
Taylor & Francis Group

A BIT-SERIAL FLOATING-POINT UNIT FOR A MASSIVELY PARALLEL SYSTEM ON A CHIP

MANFRED SCHIMMLER[a,*], BERTIL SCHMIDT[b,†] and HANS-WERNER LANG[c,‡]

[a]*Institut für Informatik und Praktische Mathematik, CAU Kiel, 24098 Kiel, Germany;* [b]*School of Computer Engineering, Nanyang Technological University, N4-02a-32 Nanyang Avenue, Singapore, Singapore 639798;* [c]*Fachhochschule Flensburg, 24943 Flensburg, Germany*

(Received 23 October 2003; In final form 8 April 2004)

This paper presents the design of a new bit-serial floating-point unit (FPU). It has been developed for the processors of the instruction systolic array (ISA) parallel computer model. In contrast to conventional bit-parallel FPUs the bit-serial approach requires a different data format. Our FPU uses an IEEE compliant internal floating-point format that allows a fast least significant bit (LSB)-first arithmetic and can be efficiently implemented in hardware.

Keywords: Bit-serial floating point units; Massively parallel processors; Systolic arrays; Floating-point unit

INTRODUCTION

Modern VLSI technology allows us to integrate massively parallel systems on a single chip. The area limitations for the processors of such systems require small but efficient computational units [11,14,20]. In particular, it may motivate the choice of a bit-serial data organization of the individual processors. The fine-grained processor arrays are used in a variety of applications where identical operations have to be performed on large data sets such as signal, image and video processing [16,20], cryptography [15], biological sequence analysis [17], scientific computing [19], visualization [11,18] and medical imaging [13]. Implementations of relevant numerical and signal processing algorithms often require floating arithmetic in order to achieve adequate accuracy. Thus, the integration of efficient floating-point arithmetic is essential to these processors.

Motivated by the design of an instruction systolic array (ISA) with 1024 processors on one chip [14], an FPU has been designed that meets these requirements. Its bit-serial structure allows for a fine-grained pipelined implementation on a minimal area of silicon. The design has been guided by experiences with a lower-integrated predecessor version of the processor array, which is available as an add-on board for a PC [6].

*Tel.: + 49-431-880-4480. Fax: + 49-431-880-4054. E-mail: masch@informatik.uni-kiel.de
†Corresponding author. Tel: + 65-67905194. Fax: + 65-67926559. E-mail: asbschmidt@ntu.edu.sg
‡Tel.: + 49-461-8051235. Fax: + 49-461-8051527. E-mail: lang@fh-flensburg.de

ISSN 1063-7192 print/ISSN 1029-032X online © 2004 Taylor & Francis Ltd
DOI: 10.1080/10637190410001725454

In the early days of massively parallel machines the individual processors had no integrated FPUs [3]. But when MFlops evolved as one of the most important measures for supercomputer performance, the designers added commercial bit-parallel floating-point processors as coprocessors to small PE-clusters of their massively parallel systems [21,22]. This approach was useful as long as the integrated circuit technology did not allow fitting both the PEs and the FPUs on the same die. But with the advances in semiconductor technology in the last decade, designers tended to increase the computational capabilities of the individual processors rather than designing parallel systems of floating-point processors.

More recent work has focused on FPU design for FPGAs. In Ref. [10], a serial multiplier is combined with a parallel adder in order to achieve high performance with a limited number of CLBs. A similar approach is taken in Ref. [9], in order to investigate the resource utilization for the FPU in modern FPGAs. In Ref. [8], a technique of parameterizing the FPU-design by the desired precision is discussed. Another adaptive approach has recently been taken in Ref. [7]. A generator tool is presented that designs an FPU for FPGA implementation dependent on the input of performance requirements. Even though the performance results are impressive, all these designs are currently not suitable for a massively parallel processor array, since they are not optimized with respect to area and power consumption.

The rest of this paper is organized as follows. The second section describes the ISA principle. An overview of the individual ISA processor architecture is given in the third section. The design methods for implementing efficient floating-point operations on bit-serial hardware are described in the fourth section. The fifth section presents the corresponding execution units. The performance of several floating-point operations is given in the sixth section. The seventh section concludes the paper.

PRINCIPLE OF THE ISA

The ISA [4,5] is a mesh-connected processor grid, where the processors are controlled by three streams of control information: instructions, row selectors and column selectors (see Fig. 1). The instructions are input in the upper left corner of the processor array, and from there they move step by step in horizontal and vertical direction through the array. This guarantees that within each diagonal of the array the same instruction is active during each clock cycle. In clock cycle $k + 1$ processor $(i + 1, j)$ and $(i, j + 1)$ execute the instruction that

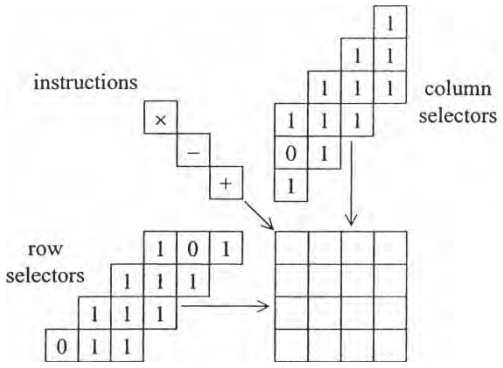


FIGURE 1 Control flow in an ISA.

has been executed by processor (i, j) in clock cycle k. The selectors also move systolically through the array: the row selectors move horizontally from left to right; the column selectors move vertically from top to bottom. Selectors mask the execution of the instructions within the processors, i.e. an instruction is executed if and only if both selector bits, currently in that processor, are equal to one. Otherwise, a no-operation is executed.

Every processor has read and write access to its own memory. Besides that, it has a designated *communication register* (*C-register*) that can also be read by the four neighbour processors. Within each clock phase reading access is always performed before writing access. Thus, two adjacent processors can exchange data within a single clock cycle in which both processors overwrite the contents of their own *C*-register with the contents of the *C*-register of its neighbour. This convention avoids read/write conflicts and also creates the possibility to perform parallel prefix operations efficiently, e.g. row sum.

PROCESSOR ARCHITECTURE

The processor architecture is part of a chip design containing an ISA of size 32 × 32 [14]. The main components of the processor architecture are a set of 64 data registers, the communication register, a unit for integer addition, logical operations and conditional instructions, a 16-bit multiplier, and a shifter/adder for floating-point arithmetic (see Fig. 2).

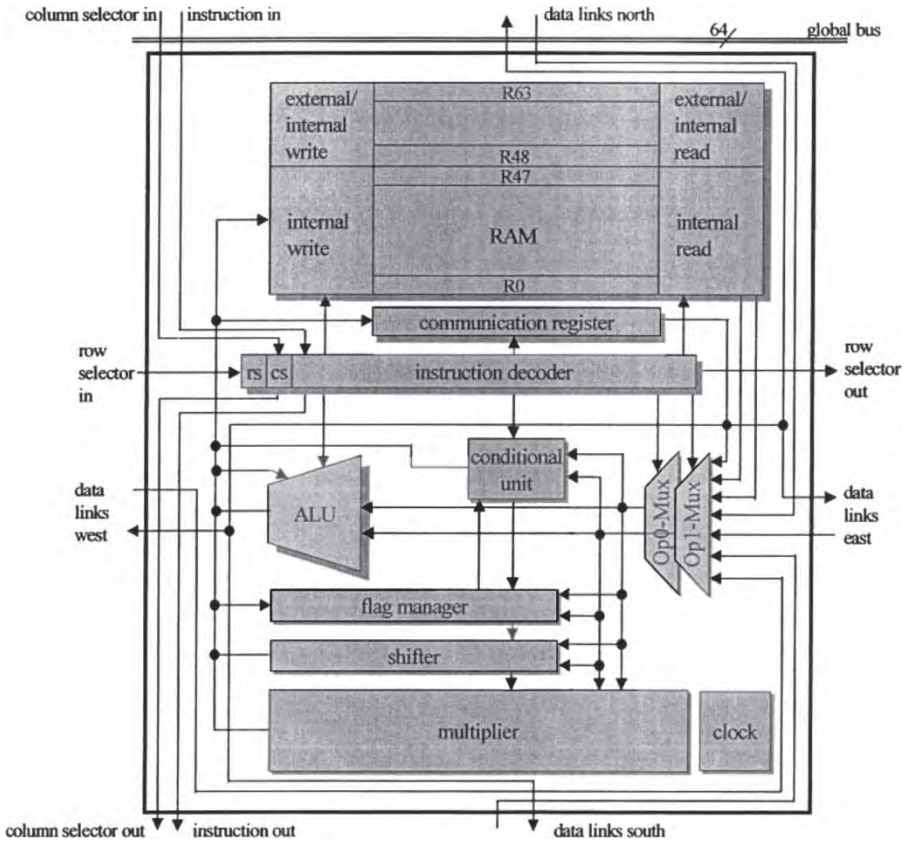


FIGURE 2 Block diagram of the processor architecture.

The data word length is 16 bits. Data is processed bit-serially, such that the execution of each instruction takes exactly 16 clock cycles. However, there are macro-instructions formed as series of elementary instructions in order to handle data types consisting of more than one machine, word, e.g. double precision floating-point numbers (see "Floating Point Operations" section). After receiving an instruction, the processor stores it in the instruction register, decodes the two operand addresses and the destination address, retrieves the operands from the register file, executes the instruction, writes the result back to the destination register, and passes the instruction to the next processor. Since all this is done bit-serially, it can be pipelined on bit level, such that a new instruction can be fetched and processed every 16 clock cycles. For instance, the first bit of the result is computed already after the second bits of the operands have entered the respective unit.

The instructions are 3-address instructions (two operand registers—op0 and op1 and destination register—dest) coded in two 32 bits. Several instructions are related to floating-point operations. However, these instructions in general do not make sense on their own, but only within macro-instructions, i.e. a sequence of instructions that realize a certain operation on floating-point numbers. The instruction

```
expadd  op0, op1, dest;
```

computes the sum of the exponents of two floating-point numbers, as required for a multiplication. Similarly, there exist instructions EXPSUB for computing the difference of two exponents and EXPSEL for selecting the greater exponent as the result exponent in the case of an addition of two floating-point numbers. The instructions SHADD (shift and add), shsub (shift and subtract), nrm (normalize), LDMUL (load multiplier), and MULT (multiply) are used in macro-instructions for computing the mantissa of the result. The instruction

```
mode  length;
```

is used for configuration of the floating-point control for handling single precision or double precision numbers. The instructions

```
mtw  op0, dest;
expw  op0, dest;
```

support the conversion between the IEEE floating-point format [1] and the internal format, forming the mantissa words and the exponent word of the internal format.

FLOATING POINT OPERATIONS

The processor supports and double precision floating-point arithmetic floating-point registers are implemented by groups of 16-bit data registers, and floating-point operations are implemented as instruction macros, i.e. as sequences of elementary instructions.

Internal Floating Point Format

The special cases for IEEE numbers are coded in all bits of the number representation. For example, in order to find out if a number equals infinity the complete mantissa and the complete exponent have to be compared with 0 and -1, respectively.

Therefore, a bit-serial compare to infinity (required for every arithmetic operation) would need at least 32 internal clock cycles for single and 64 for double. To solve this problem a conversion into an internal format is performed. The internal representation of a double X consists of five 16-bit words $X[0], \ldots, X[4]$. Here, $X[4]$ contains the exponent of the double, the sign, the kind of representation (absolute value/sign or 2-complement), and information about possible special cases (infinity, zero, NaN = not a number). $X[0]$ to $X[3]$ contain the significant (mantissa) of the double, extended to 64 bits.

$X[4]$ Exponent:	Sm	T1	T0	R	Se	Se	e9	e8	e7	e6	e5	e4	e3	e2	e1	e0

Sm: sign of the significant, 0: positive, 1: negative

$T1$, $T0$: encoding of special cases: 00: normal number, 01: zero, 10: INIF (infinity), 11: NaN

R: representation of significant: 0: absolute value/sign representation, 1: 2-complement representation

Se: sign of exponent (two identical bits for overflow detection)

$e0,\ldots,e9$: binary representation of the exponent E in 2-complement (little endian mode)

64 bit of the significant $M = (X[3], X[2], X[1], X[0])$, where 63 is the most significant bit (MSB):

$X[3]$ significant:	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48
$X[2]$ significant:	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
$X[1]$ significant:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
$X[0]$ significant:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The value of such a double is:

$0\cdot(-1)^{\text{sm}}$ if $T0 = 1$ and $T1 = 0$;

$\text{INF}\cdot(-1)^{\text{sm}}$ if $T0 = 0$ and $T1 = 1$;

NaN if $T0 = 1$ and $T1 = 1$;

$(-1)^{\text{sm}}\cdot 0\cdot M\cdot 2^{E}$ if $T0 = 0$, $T1 = 0$ and $R = 0$;

$(-1)^{\text{sm}}\cdot 0\cdot M'\cdot 2^{E}$ if $T0 = 0$, $T1 = 0$ and $R = 1$, where M' is the $2-$complement of M;

Double numbers can be normalized or not normalized. A positive number and a number in absolute value/sign-representation is normalized iff bits 63, 62, 61 of the significant are 001, i.e. bits 63 and 62 hold the sign and then in bit 61 comes the MSB of the significant. As usual in absolute value/sign representation, the significant is positive and the sign is stored in the sm-bit of the exponent (see $X[4]$). A negative number in 2-complement is normalized iff bits 63, 62, 61 of the significant are 110. Again, bits 63 and 62 form the sign and it follows the most significant digit of the significant represented in 2-complement.

The four 16-bit words of the IEEE format are converted into 5 words of the internal format. The 52-bit mantissa is stored in four words, where in the first word an explicit 1 is introduced (this 1 is implicit in the IEEE representation). The first two 0's serve as two sign bits of the mantissa. Two sign bits are useful for detecting overflows: an overflow after an addition changes the second sign bit and a right shift for normalization is necessary. The exponent is stored in the last 12 bits of a separate word, also with two sign bits. The value 01 of the sign bits indicates *exponent overflow*, whereas 10 indicates *exponent underflow*. In these cases, the number is changed to INF or 0, respectively. The special values zero, INF and NaN are coded using the tag stored in bit 14 and 13 of the exponent word. In the case of a special value, the actual value of the mantissa is irrelevant. Bit 12 is used internally for indicating if the mantissa is in 2-complement representation after an addition or subtraction. The conversion is supported by special instructions of the processor.

Floating Point Addition

An addition of two floating-point numbers is performed in the usual way: the two exponents are compared, the mantissa of the operand with smaller exponent is shifted to the right appropriately, then the mantissas are added. If any operand is negative and not yet in 2-complement representation, its mantissa is complemented before. The bit-serial organization allows on-the-fly computation with one single half adder. If any operand is 0, its mantissa is set to 0. The result mantissa is in 2-complement representation. The result exponent is the larger exponent of the two operands. The tag of the result is computed depending on the tags of the two operands according to Table I.

The sign of the result is the sign of the result mantissa, except in the case that any operand is INF, then the sign is equal to the sign of this operand. If the result is 0 or NaN, the sign is irrelevant. Since the operands consist of more than one word (e.g. of four mantissa words in the case of double precision floating-point numbers), shifting of a mantissa by more than 15 bits is realized by displaced addressing of the operands (see Fig. 3).

In each case, in order to realize a shift distance of s, in the first step A_{-1} and B_{-1+k} are addressed, where $k = (s \operatorname{div} 16) + 1$. B_{-1+k} is loaded into the shifter and delayed by $d = 16(k - s)$ bits. In the next step A_0 and B_{0+k} are addressed. While B_{0+k} is loaded to the shifter, the d bits of B_{-1+k} appear at the output of the shifter and are added to A_0. In the last step A_3 and B_{3+k} are addressed. However, B_{3+k} is an invalid operand, therefore it is cancelled, i.e. set to 0...0 or to 1...1 depending on the sign of the mantissa. If k becomes greater than 4, all B-operands are cancelled, i.e. set to 0...0. In the case that B has the greater exponent, then the difference of the exponents is negative and A has to be shifted. The situation is essentially the same as above with the roles of A and B interchanged.

TABLE I Computation of result tags in the case of an addition

	X	*0*	*INF*	*NaN*
X	*X**	*X*	INF	NaN
0	*X*	0	INF	NaN
INF	INF	INF	INF[†]	NaN
NaN	NaN	NaN	NaN	NaN

* Except the case that the result mantissa is 0, than the result tag is 0.
[†] Except the case that the signs of the two operands are different, then the result tag is NaN.

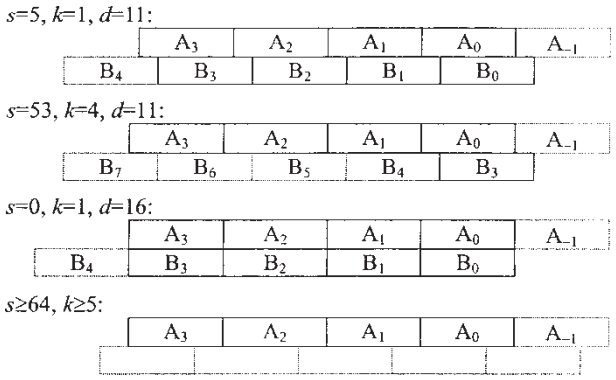


FIGURE 3 Addition of two 64-bit mantissas.

Normalization

After an addition, in general the result mantissa is not in normalized form, i.e. there are not exactly two leading 0's or 1's. Several cases are possible: there is one leading digit, two leading digits, (this is the normalized case), three through 63 leading digits, or 64 leading digits. In the latter case, the result is considered to be 0, if the leading digits are 0. In all other cases a left shift by $-1\ldots62$ is necessary so that the mantissa starts with 001 or 110. The exponent has to be corrected by this shift distance A counter; the leading digits counter (LDC) is used to count the number of leading digits (-2) of the result produced during an addition/subtraction or multiplication. The LDC also records the first bit of the mantissa, indicating the sign of the mantissa, and detects if the mantissa is 0. Before normalization, the value of the LDC is transferred into the shift control register. But now this value has to be interpreted in a different way, since it is the distance for a left shift instead of a right shift as in the case of an addition. However, this can be done using the same hardware for normalization as for addition.

Floating Point Multiplication

A multiplication of two 64-bit mantissas is performed according to the scheme shown in Fig. 4, where e.g. $A_3\,A_2\,A_1\,A_0\,B_0$ denotes the 80-bit product of $A_3\,A_2\,A_1\,A_0$ with B_0. The result would be 128 bit long, however, only the most significant 64 bits can be used for the result mantissa F. Therefore the above scheme is truncated, in order to reduce the number of 16-bit multiplications. The scheme is shown in Fig. 5.

Addition of the intermediate results is performed simultaneously with the multiplications in the multiplier, which is actually a multiply-and-add unit. The error introduced by the truncation amounts to at most 3 bits. This seems to be tolerable, because the precision of the internal floating point format is 61 significant bits, whereas the original IEEE format has only

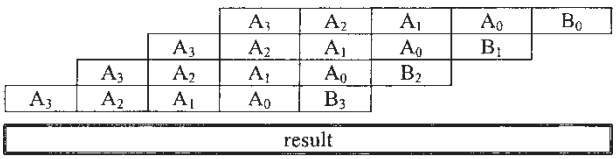


FIGURE 4 Multiplication scheme for 4-word operands.

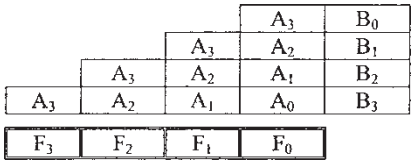


FIGURE 5 Truncated multiplication scheme for 4-word operands.

52 bits. If more precise arithmetic is required, the complete multiplications have to be executed. The exponents of the two floating-point numbers are added. The sign of the result is the exclusive-or of the two-operand signs. The tag of the result is computed according to Table II.

FLOATING POINT UNITS

Multiplier for Positive Operands

As explained in "Floating Point Operations" section, bit-serial floating-point multiplication can be implemented by a simple combination of integer addition (for the exponents) and long integer multiplication (for the significants). In this section, we describe design of the multiplier for this purpose.

Its original architecture is shown in Fig. 6. Before starting the multiplication, one operand x is loaded into a parallel register (the multiplicand register) with the MSB to least significant bit (LSB) from left to right. This is controlled by a signal r being high for one clock cycle and low for the rest of the instruction execution. Now the second operand y is input into the multiplier LSB-first. In each clock cycle, i one bit y_i of y is processed. It is multiplied with x by computing the bitwise AND. The result is added to the current sum, producing a new intermediate result. This result is stored in a carry save representation instead of standard binary representation. This means the intermediate result consists of two binary words: a s-word and a c-word (s for sum and c for carry). The standard binary intermediate result would be $2 \cdot c + s$. This carry save representation avoids the propagation of the carry bits for more than one position.

The next bit y_{i+1} of y is processed in the following clock cycle. Since the result to be computed in this step has to be weighted by a factor of 2 with respect to the old intermediate result, we have to shift the old result by one position to the right before we add the new product. Propagating the sum bits to the right neighbour and recycling the carry bits to the same full adder that has just produced this carry performs this shift operation. The shift operation outputs one bit at the rightmost full adder. This is the ith bit of the final product counted from the LSB to the MSB.

TABLE II Computation of result tags in the case a multiplication

	X	*0*	*INF*	*NaN*
X	*X**	*X*	INF	NaN
0	*X*	0	NaN	NaN
INF	INF	NaN	INF	NaN
NaN	NaN	NaN	NaN	NaN

* Except the case of an exponent underflow or overflow, then the result tag is 0 or INF, respectively.

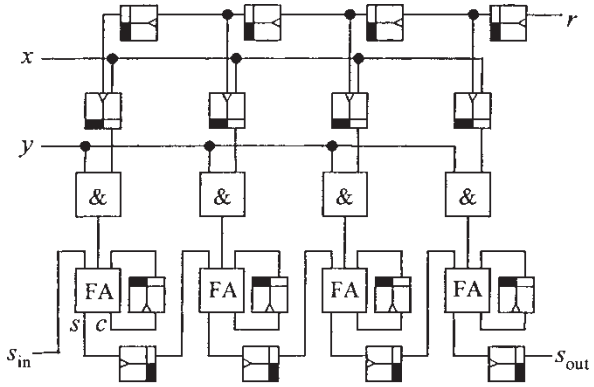


FIGURE 6 The original multiplier architecture for unsigned integers.

This multiplier design has been used in the parallel computer Systola 1024 [5]. One interesting property of this architecture is its capability to perform *multiply–add* operations. Each bit operation consists of the addition of a new bit product to the intermediate result and a shift operation. Therefore, the interconnection between adjacent full adders form a shift register (the sum register). Now we can use this shift register to add an additional operand. The only thing we have to do is to shift the third operand into the sum register of the multiplier with the LSB first. The first bits of the summand can be shifted in concurrently with the x operand. During the multiplication, in every clock cycle, we input one additional bit of the summand while one bit of y is processed.

The multiplier is capable of executing two different instructions, corresponding to the two phases for the multiplication. In the first phase, the operand x is loaded into the multiplicand register. We call this instruction LDMUL (load multiplicand). During the execution of a LDMUL-instruction, the y-input is forced to be 0. If a multiply–add operation is executed the least significant word of the operand s is shifted into the sum register during the LDMUL-instruction, too. The resulting bits produced by the first LDMUL-instruction are irrelevant.

The second phase is the multiplication with y. The instruction is called MULT (multiply). y is consumed bit by bit as described above. In a MULT-instruction, the x-input is not interpreted. The s-input is used to read the most significant word of the operand s. The output produces the least significant word of the result. After this MULT-instruction the most significant word of the result is still stored in the sum- and carry-registers of the multiplier. In order to output this rest of the result we need to execute another LDMUL-instruction. Therefore, the computation of the last example can be coded in three instructions:

$$\text{LDMUL} \quad X, \; S_{\text{low}}, \; \text{don't care};$$

$$\text{MULT} \quad Y, \; S_{\text{high}}, \; \text{RES}_{\text{low}};$$

$$\text{LDMUL} \quad 0, \; 0, \; \text{RES}_{\text{high}};$$

Numerous applications in cryptography deal with long operands, e.g. current RSA standards operate with lengths between 1024 and 4096 bits [12]. Such bit lengths cannot be

processed in one operation of a standard processor. One multiplication with large operands can be decomposed into a sequence of smaller multiplications and additions.

Obviously, an $N \times N$-bit (N even) multiplication $x{\cdot}y$ can be constructed of four $N/2 \times N/2$-bit multiplications as follows:

$$x{\cdot}y = x_{\text{high}}{\cdot}y_{\text{high}}{\cdot}2^N + x_{\text{high}}{\cdot}y_{\text{low}}{\cdot}2^{N/2} + x_{\text{low}}{\cdot}y_{\text{high}}{\cdot}2^{N/2} + x_{\text{low}}{\cdot}y_{\text{low}}$$

where

$$x = x_{\text{high}}{\cdot}2^{N/2} + x_{\text{low}} \text{ and } y = y_{\text{high}}{\cdot}2^{N/2} + y_{\text{low}}$$

In this case, we need four multiplications of half the size and three additions to get one result of full size. If we recursively use this technique, we can perform an $N \times N$-bit multiplication by using an n-bit multiplier. It requires $N/n{\cdot}N/n$ small multiplications and $N^2/n^2 - 1$ additions.

The bit serial structure of the above multiplier allows for a simple and efficient execution of long multiplications: only the operand x is chopped into pieces $X_{N/n-1}, \ldots, X_0$ where each piece has the length n of the multiplicand. After loading one of these pieces of x using a LDMUL-instruction the complete factor y is processed by a sequence of MULT-instructions. This takes $N/n + 1$ instruction cycles. The result has the length $N + n$. For the complete long multiplication we need to add one such partial product for each piece of x.

To speed up this procedure, we can take advantage of the multiply–add capability of our multiplier. We start with the least significant word X_0 of x. It is multiplied by the complete operand y leading to a result words $\text{RES}_{N/n}\text{RES}_{N/n-1}\ldots\text{RES}_0$. RES_0 is the least significant word of the final result. Afterwards, we proceed with X_1. Using multiply–add, we compute $X_1{\cdot}y + \text{RES}_{N/n}\text{RES}_{N/n-1}\ldots\text{RES}_1$. Again we come up with an intermediate result with $N + n$ bits, where the least significant word is part of the final result. In the same way, we continue until we come to the most significant word of x. The result of its multiplication with y delivers the rest of the final result. The following example shows the multiplication of two 2-word operands x and y together with the addition with a third 2-word operand s according to the scheme described above.

```
LDMUL   X_low, S_low, 0;        //first stage: load x_low
MULT    Y_low, S_high, RES_0;   //compute lower result
MULT    Y_high, 0, res_1;       //compute 2nd partial result
LDMUL   X_high, res_1, res_2;   //first stage: compute 3rd partial result; second
                                //stage: load x_high
MULT    Y_low, res_2 RES_1;     //compute 2nd final result
MULT    Y_high, 0, RES_2;       //compute 3rd final result
LDMUL   0, 0, RES_3;            //compute upper final result
```

Note that there are slots left to add 4-word operand; however, an overflow might occur. The programmer would have to take appropriate precautions.

This example can be extended to longer operands as described above. In general, the multiplication of two m-word operands requires $m\cdot(m+1)+1$ instruction cycles.

Extended Multiplier for Negative Operands

The multiplier of the previous section is capable of multiplying two unsigned integers. With only some minor modifications in the instruction set and in the hardware, it is possible to extend it to a signed multiplier where one operand may be negative in 2-complement representation. We make use of a well known property of 2-complement numbers [2].

PROPOSITION Let x and y be n-bit numbers in 2-complement representation, i.e. x, $y \in [-2^{n-1}, 2^{n-1}-1]$ and let x' and y' be the values of x and y interpreted as unsigned integers, i.e. $x' = x$ if ≥ 0 and $x' = 2^n + x$ if $x < 0$ respectively $y = y'$ if $y \geq 0$ and $y' = 2^n + y$ if $y < 0$. Let $z = x\cdot y$ and $z' = x'\cdot y'$. Then, it holds $z = z' - \text{sign}(y)\cdot 2^n\cdot x' - \text{sign}(x)\cdot 2^n\cdot y' + \text{sign}(y)\cdot\text{sign}(x)\cdot 2^{2n}$, where sign($x$) and sign($y$) denotes the sign-bit of x and y, respectively, i.e. sign(x) $= 0$ if $x \geq 0$ and sign(x) $= 1$ if $x < 0$.

Proof We consider the four cases determined by the signs of x and y.

(i) $x \geq 0, y \geq 0 : z = x\cdot y = x'\cdot y' = z'$.
(ii) $x \geq 0, y < 0 : z = x'\cdot(y' - 2^n) = x'\cdot y' - 2^n\cdot x' = z' - 2^n\cdot x'$.
(iii) $x < 0, y \geq 0$: this case is analogue to (ii).
(iv) $x < 0, y < 0 : z = (x' - 2^n)\cdot(y' - 2^n) = x'\cdot y' - 2^n\cdot x' - 2^n\cdot y' + 2^{2n} = z' - 2^n\cdot x - 2^n\cdot y + 2^{2n}$.

In case (iv) the value 2^{2n} can be omitted, because interpreted as a $2n$-bit number it is equal to 0. □

In the above proposition z' corresponds to the result of the multiplication of x and y with the unsigned multiplier of "Principle of the ISA" section. The terms $-\text{sign}(y)2^n\cdot x'$ and $-\text{sign}(x)\cdot 2^n\cdot y'$ denote the correction steps associated with negative operands to compute z from z'. In order to extend the original multiplier to deal with 2-complement numbers these correction terms have to be integrated into the new multiplier architecture.

Let us assume that x is negative. According to the proposition $2^n\cdot y'$ has to be subtracted from the result z'. This is equivalent to adding the 2-complement of y to the most significant word of z' (which has the weight 2^n). In the bit-serial case, the 2-complement can be computed on-line with the multiplication by using a multiplexer and a half adder. For the implementation of the multiplication with negative x, we introduce the instruction MULTSX. MULTSX is basically identical to MULT. Additionally, it sets *xneg* to the sign (stored in the MSB) of x. It is used to select either 0 or $-y$ as additional summand to s. The addition is performed on-line with the input of y and bit-serially delivers $s - y$ at s_{in}. The resulting architecture is shown in Fig. 7.

Floating Point Adder

The floating point adder consists of a shifter, an adder, two units for displaced addressing of the operands, two complement units and a logic for cancellation of operands. The shifter is a configurable shift register of variable length between 0 and 15 bits (see Fig. 8).

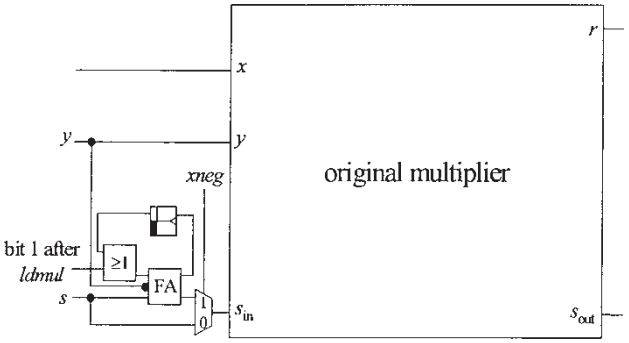


FIGURE 7 Multiplier for nonnegative y and arbitrary x.

For a floating-point addition, the result of the exponent comparison is stored in a special register, the shift control register (SCR). The highest bit SCR[11] indicates if the exponent difference is negative and determines the shift operand. Bits SCR[0–3] determine the delay of the shifter. If this is a non-negative number s, it is complemented by the. XOR-gates and the last 1-bit delay is activated, in order to achieve the required delay of $16 - s$. Bits SCR[4–5] are used for displaced addressing of the shift operand, as explained in the "Floating Point Operations" section. If any of the bits SCR[6–10] is different from SCR[11], the shift distance would be greater than 64, so the shift operand maybe cancelled. A bit-serial adder adds the shifted operand and the other operand. If any operand is negative and not yet in 2-complement representation, it is complemented before the addition.

PERFORMANCE

For single precision floating-point numbers, an addition takes 14 instruction cycles and a multiplication takes 7 instruction cycles. The multiplication consists of the addition of the exponents and the multiplication and rounding of the significants. The internal format allows us to detect all exceptions (0, infinity, or NaN) by interpretation of the tag-bits during

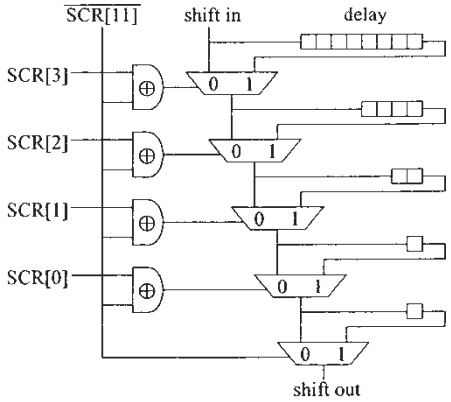


FIGURE 8 16-bit shifter.

TABLE III Multiplication of two single precision numbers with the presented FPU

	Instructions			*Comments*
EXPADD	*B*[2],	*A*[2],	*P*[2];	{add exponents}
LDMUL	0,	*A*[0],	0;	{load *A*[10]}
MULZU	*B*[1],	0,	× [0]:	
LDMUL	× [0],	*A*[1],	× [1];	{load *A*[1]}
MULZU	*B*[0],	*A*[1],	0;	
MULZ	*B*[1],	0,	*P*[0];	
LDMUL	0,	0,	*P*[1]:	

The two operands are stored in registers *A*[0], *A*[l], *A*[2] and *B*[0], *B*[l], *B*[2]. The result is written into the registers *P*[0], *P*[l],*P*[2].

the addition of the exponents in the first instruction. A truncated multiplication of 32-bit operands requires six multiplication operations: the first loads the least significant half of the first operand into the multiplier. The second multiplies it with the most significant half of the second operand. The third loads the most significant half of the first operand and generates the most significant 16 bits of the intermediate result. In the remaining three instructions, the intermediate result is added to the product of the most significant word of the first operand and the whole second operand. Table III shows detailed instructions for the multiplication.

The addition is more complicated, since the difference of the exponents provides the shift distance of one operand. It is computed in the first instruction again together with the tags for possible exceptions. Controlled by the shift control register (SCR) the indirect addressing fetches the right fractions of the significants to be added. The leading digit counter (LDC) counts the number of leading 0's in order to detect the final shifting distance for normalization after the addition. Finally, the result is shifted left by this distance, and the exponent of the result is decreased accordingly. Table IV shows details of the addition.

TABLE IV Addition of two single precision numbers with the proposed FPU

	Instructions			*Comments*
EXPSUB	*A*[2],	*B*[2],	*T*[2];	{compare exponents}
SETNF	*T*[2],	SCR;		{store difference of exponents in SCR register}
NOOP				{wait until SCR is available}
SHADDU	0,	0,	0;	{initialize shifter with 0′sand start adder}
SHADDU	*A*[3],	*B*[3],	0;	{one adders will be modified, the appropriate operand will be shifted and added}
SHADDZ	*A*[0],	*B*[0],	*T*[0];	{start LDC and add}
SHADD	*A*[1],	*B*[1],	*T*[1];	{add}
EXPSE10	*A*[2],	*B*[2],	*T*[2];	{select the greater exponent}
EXPSUB	*A*[2],	LDC,	0;	{wait for LDC and adapt exponent:}
EXPSUB	*A*[2],	LDC,	*B*[2];	{create exponent of result}
OR	0,	LDC,	SCR;	{initialize SCR}
NRMU	*A*[3],	0;		{start normalize}
NRM	*A*[0]	*B*[0];		{normalize}
NRM	*A*[1],	*B*[1];		{normalize}

The two operands are stored in registers *A*[0], *A*[l], *A*[2] and *B*[0], *B*[1], *B*[2]. The result is written into the registers *B*[0], *B*[1], *B*[2].

The performance of one processor is

$$2 \text{ operations/21 instruction cycles} \times 1 \text{ instruction cycle/16 clock cycles}$$
$$\times f \text{ clock cycles/second}$$
$$= f/168 \text{ operations/second}$$

with $f = 200$ MHz, we have a performance of one processor of 200/168 million floating-point operations/second or 1.2 MFlops; this amounts to 1.2 GFlops for one chip with 1024 processors or to 20 GFlops for a board with 16 chips. For double precision floating-point numbers, an addition takes 18 instruction cycles, a multiplication takes 16 instruction cycles. The performance of one processor is

$$2 \text{ operations/34 instruction cycles} \times 1 \text{ instruction cycle/16 clockvcycles}$$
$$\times f \text{ clock cycles/second}$$
$$= f/272 \text{ operations/second}$$

with $f = 200$ MHz, we have a performance of one processor of 200/272 million floating-point operations/second or 0.735 MFlops; this amounts to 735 MFlops for one chip with 1024 processors or to 12 GFlops for a board with 16 chips.

Also more complex floating-point operations can be computed with the presented FPU, e.g. division, square root. Table V shows the instruction count for several floating-point operations with the presented FPU.

Since instruction count for individual floating operations does not consider format conversion time and communication time within the processor array to fetch the operands, it is often a weak performance measure that does not reflect the behaviour of a complete application. Therefore, we will use the instruction count for two typical floating-point applications on the ISA in our performance evaluation (see Table VI). The two applications are 2-dimensional matrix multiplication and 1-dimensional fast Fourier transform (FFT).

Matrix Multiplication

The problem is to compute the product of two $n \times n$ matrices $F = A \cdot B$, defined as $F_{ij} = \sum_{k=1,\ldots,n} A_{ik} \cdot B_{kj}$. We first assume that $n = 32$ (thus, fitting into the processor array). Suppose that matrices A and B have been loaded into the array. Then, the first column of A, which is in the first column of the array, is broadcast along the rows and stored intermediately in some single or double register H. Similarly, the first row of B is broadcast along the columns. Now each processor (i,j) has elements A_{i1} and B_{1j} available and can compute

TABLE V Instruction counts for several floating-point operations in single/double precision

Operation	*Single*	*Double*
Format conversion	6	8
Addition/Subtraction	14	18
Multiplication	7	16
Division	52	101
Square root	92	138

TABLE VI Floating-point performance of the 2-dimensional matrix multiplication and 1-dimensional FFT on the ISA processor array of size 32 × 32

Application	*Single (Efficiency)*	*Double (Efficiency)*
Matrix Multiplication (64 × 64)	0.853 GFlops (71%)	0.522 GFlops (71%)
FFT (2048-points)	0.613 GFlops (51%)	0.427 GFlops (58%)

Efficiency indicates the achieved performance with respect to the theoretical peak performance, i.e. 1.2 GFlops (single) and 0.735 GFlops (double).

$F_{ij} := F_{ij} + A_{il} \cdot B_{lj}$. The next step is to ringshift A by one column to the left in order to get the second column of A, elements A_{i2}, into the first column of the array, from where they can be broadcast along the rows. Similarly, B is ringshifted along the columns. After n such steps, registers F contain the product of A and B. Multiplication of larger matrices can be easily partitioned using 32 × 32 matrix multiplication as a building block.

Each iteration step consists of two floating-point operations per processor (addition and multiplication) and the ringshift/broadcast communication. The two floating-point operations need 21 instructions for single precision and 34 instructions for double precision. Ringshift/broadcast requires 3 instructions per register on the ISA. Communication costs can be halved by performing a 64 × 64 matrix multiplication and assigning 2 × 2 values to each processor. This leads to 30 instructions per iteration step for single precision and 49 instructions per iteration step for double precision, which corresponds to a performance of 0.853 GFlops (single) and 0.522 GFlops (double) receptively. The time required for format conversion can be neglected, since it is only done at the beginning and at the end of the procedure.

FFT

Figure 9 shows the signal flow graph of the FFT of a sequence of n points, here for $n = 8$. It consists of $\log(n)$ stages. In each stage, a complex multiplication with some primitive root of unity, a complex addition, and a complex subtraction is performed at each of the n points. The output of stage k is permuted by interchanging the two halves of subsequences of length

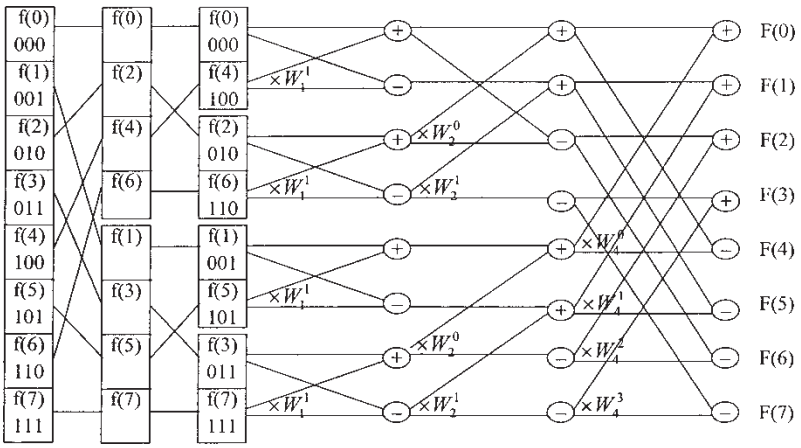


FIGURE 9 Signal flow graph of an 8-point FFT.

2^k before being input to the next stage. Initially, the sequence of the n points is permuted according to the so-called bit reversal permutation.

The implementation of the ID-FFT of size $n = 2048$ on a 32×32 ISA assigns two values to each processor. The computation in each stage consists of four (real) multiplications and six (real) additions in each processor. The implementation of the permutation between every two stages involves routing of data, which is performed either within the rows of the ISA (in the case of interchanging the two halves of subsequences of length ≤ 64) or by interchanging complete rows of the ISA (in the case of interchanging the two halves of longer subsequences). The interchanging of the *C*-register contents of two halves of m processors is implemented by $m/2$ ringshift operations.

The ten floating-point operations per stage require 112 instructions for single precision and 172 instructions for double precision. The necessary communication per iteration step can be performed on the average within 92 instruction (single) and 153 instruction (double), respectively. This leads to a performance of 0.613 GFlops (single) and 0.427 GFlops (double) for the processor array of size 32×32.

CONCLUSIONS AND FUTURE WORK

In this paper, the design of a FPU has been presented. Its bit-serial nature allows for an area efficient and energy efficient implementation. Because bit-serial operations with the IEEE floating-point number representation are difficult to implement, we use an internal floating-point format that allows fast LSB-first arithmetic. Our FPU is optimally suited for a fine-grained pipelined architecture. That is why it has been used in the processing elements of the KPROC [6], a parallel system with 1024 processors on a single chip.

Most architectures require fast floating-point computation for any price. In case of a massively parallel system on a single chip, there are other criteria that might be even more important than latency: area and power consumption. In this case, it turns out that the bit serial approach provides significant advantages in comparison to the bit parallel one. Adder and multiplier can be implemented in a more economic way, exceptions can be detected earlier, and rounding and normalization are relatively simple in comparison to the parallel case. The disadvantage of higher number of clock cycles for one operation is compensated by an increased number of processing elements that fit on one chip.

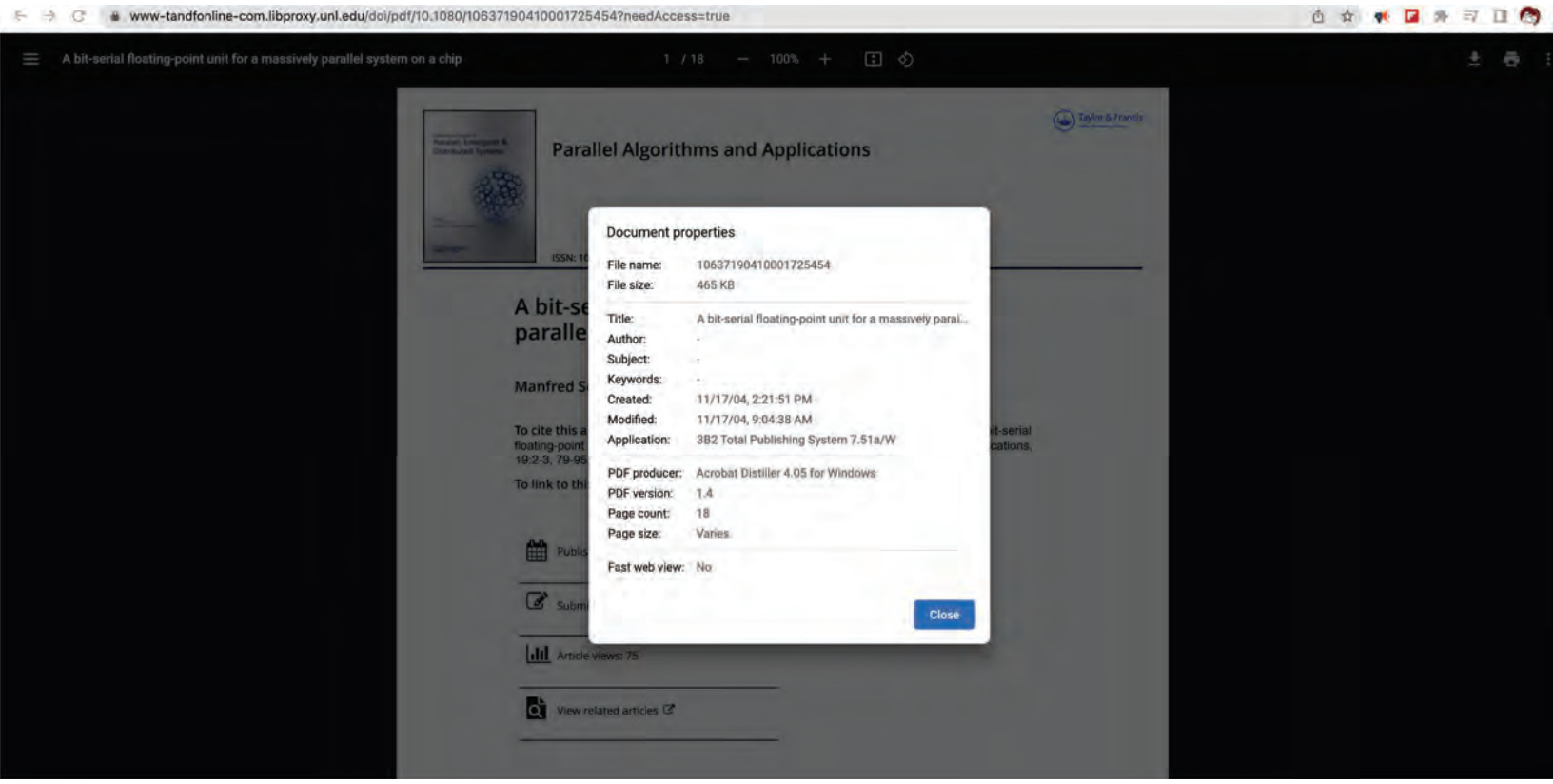
Due to the advances in FPGA technology, it will soon be possible to integrate an array of several hundred simple processors on one single FPGA. An architecture like that could be immediately equipped with floating-point capabilities using the FPU presented in this paper.

References

[1] (1985) *IEEE standard for binary floating point arithmetic*, ANSI IEEE 754-1985, New York.
[2] Hennessy, J.L. and Patterson, D.A. (1996) Computer Architecture: A Quantitative Approach (Morgan Kaufman Publishers).
[3] Hillis, D. (1986) The Connection Machine (MIT Press, Cambridge, MA).
[4] Kunde, M., *et al.* (1988) "The instruction systolic array and its relation to other models of parallel computers", *Parallel Computing* **7**, 25–39.
[5] Lang, H.W. (1986) "The instruction systolic array, a parallel architecture for VLSI", *Integration, the VLSI Journal* **4**, 65–74.

[6] Lang, H.W., Maaβ, R. and Schimmler, M. (1994) "The instruction systolic array—implementation of a low-cost parallel architecture as add-on board for personal computers", Proceedings of HPCN'94, LNCS 797 (Springer), pp 487–488.
[7] Liang, J., Tessier, R. and Mencer, O. (2003) "Floating point unit generation and evaluation for FPGAs", Proceedings of FCCM'03 (IEEE Computer Society).
[8] Lienhart, G., Kugel, A. and Männer, R. (2002) "Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulation", Proceedings of FCCM'02 (IEEE Computer Society).
[9] Ligon, W.B., McMillan, S., Monn, G., Schoonover, K., Stivers, F. and Underwood, K.D. (1998) "A re-evaluation of the practicality of floating-point operations on FPGAs", Proceedings of FCCM'98 (IEEE Computer Society).
[10] Louca, L., Cook, T.A. and Johnson, W.H. (1996) "Implementation of IEEE single precision floating point addition and multiplication on FPGAs", Proceedings of FCCM'96 (IEEE Computer Society), pp 107–116.
[11] Meiβner, M., Grimm, S., Straβer, W., Packer, J. and Latimer, D. (2001) "Parallel volume rendering on a single chip SIMD architecture", Proceedings of the IEEE Symposium on Parallel and Large Data Visualization and Graphics (IEEE Computer Society), pp 107–113.
[12] Menezes, A.J., van Oorschot, P.C. and Vanstone, S.A. (1997) Handbook of Applied Cryptography (CRC Press).
[13] Schmidt, B., Schröder, H. and Schimmler, M. (2001) "Tomographic image reconstruction on the instruction systolic array", *Computing and Informatics* **20**, 27–42.
[14] Schmidt, B. and Schimmler, M. (2000) "KPROC—an instruction systolic architecture for parallel prefix applications", *Parallel and Distributed Computing Practices* **3**(2), 163–178.
[15] Schmidt, B., Schimmler, M. and Schröder, H. (1998) "Long operand arithmetic on instruction systolic computer architectures and its application to RSA cryptography", Proceedings of Euro-Par'98, LNCS 1470 (Springer), pp 916–922.
[16] Schmidt, B. and Schimmler, M. (1999) "A parallel accelerator architecture for multimedia video compression", Proceedings of Euro-Par'99, LNCS 1685 (Springer), pp 950–960.
[17] Schmidt, B., Schröder, H. and Schimmler, M. (2002) "Massively parallel solutions for molecular sequence analysis", Proceedings of IPDPS'02 (IEEE Computer Society).
[18] Sim, L.C. and Schröder, H. (2003) "MIMD–SIMD hybrid system: towards a new low-cost parallel system", *Parallel Computing* **29**(1), 21–36.
[19] Sim, L.C., Schröder, H., Chian, L.C. and Leedham, G. (2003) "Fast solution of large N × N matrix equations in an MIMD–SIMD hybrid system", *Parallel Computing* **29**(11–12), 1669–1684.
[20] Taveniku, M., Ahlander, A., Jonsson, M. and Svensson, B. (1998) "The VEGA moderately parallel MIMD, moderately parallel SIMD, architectures for high performance array signal processing", Proceedings of IPPS/SPDP'98 (IEEE Computer Society), pp 226–232.
[21] Thiel, T. (1994) "The design of the connection machine", *Design Issues* **10**(1).
[22] Valafar, F. and Ersoy, O.K. (1995) A parallel implementation of backpropagation neural network on masspar MP-1, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, Athens, GA, November, 158–166.

ATTACHMENT 2D


www-tandfonline-com.libproxy.unl.edu/doi/pdf/10.1080/10637190410001725454?needAccess=true
A bit-serial floating-point unit for a massively parallel system on a chip
1 / 18
100%
Parallel Algorithms and Applications
Taylor & Francis
Document properties
File name: 10637190410001725454
File size: 465 KB
Title: A bit-serial floating-point unit for a massively parai...
Author: -
Subject: -
Keywords: -
Created: 11/17/04, 2:21:51 PM
Modified: 11/17/04, 9:04:38 AM
Application: 3B2 Total Publishing System 7.51a/W
PDF producer: Acrobat Distiller 4.05 for Windows
PDF version: 1.4
Page count: 18
Page size: Varies
Fast web view: No
Close
Article views: 75
View related articles

ATTACHMENT 2E

All locations : LDL Holdings - 1018 other holdings

Parallel algorithms and applications (Online) Parallel algorithms and applications. [London] : Gordon and Breach
OCLC:51232058

Holdings Information

STATE	LOCATIONS
NE	BCN CKH KRS LDL NBC NBD NBH nbo NBP NBU NCU NEM NFM OCA STM UNM
AK	UAF
AL	ABB ABC ABF ACM ADA ALM AMN AMU ASH HU9 mvu MWR MWT MWW TUS
AR	AHS AKC AKE AKH AKO AKP AKU
AZ	A3E AZK AZN AZO AZU
CA	bcz C5U C7M CAP CBC CBF CCH cci CCM CCT CEC CFI CFT CGN chc CIS CJG cluer CLV CMT CNO CPA CPE CPO CPT CRU cruer csm CSO CUH cuier CUN cuner cuser cuter CUV cuver CUY cuyer cuzer CWS DAC f@n GGB HDC HN# ivx jax LBQ merer MXL PPI rst saa sab SAD sai SBCRU sbvca sj2 SJG smx sq9 stf SXQ TCB u$c wc1 weima whccd yu0 ZCW
CO	CBS coa COD COH con COU COV CYB DVP DVZ NIY UCX
CT	CTB CTW CZL FAU GJC HRM SHU UCH UCW
DC	BID dccfp DCL DGU DGW dhu GWM NIE
DE	DLM WMG
EU	mnbet
FL	AVMAR DBB F08 FBO FCM FDS FGA FGM FHM fje FLT fmc FSA FSC FTU FUG FUT FWA fwr FYQ SOT
GA	A1A a1t AUU bhb ce3 GBC GBT GE7 GE8 GOU GPM GYD o9t TLE ZID ZJA
HI	HGP
IA	IMV IOE IOI IOK IOL IOT IWA jid PWT
ID	IDCWI N#Z OIP ONB RIC WK0 zft ZXQ
IL	ccndg CGU iaa IAO IAS IAV IAX IAY IBC IBE IBP IBT IBZ ICA ICE ICJ ICL ICN ICO ICS ICW IDB IDG IDJ IGB IHT ILK ILPML JAN JNA JYJ QX7
IN	FW8 HV6 I3U IBS idu IEC IFC IGC IGR IHC IHH IIB III IME IMI IMN IMS IMX INA IND IPL ISN ISS ITC ITS ITU IUE IUL IUP IVU IVV IWC IXT p4i XBE XND XUD XUN
KS	av1 HKA KFB KFH KFP KFU KHT KKB KKN KKP KKX KSB
KY	KAG KBC KCA KCC KEU KHN KLD KLF KLG KLH KLM KML KOC KPS KSL kts KYS LWC
LA	J5Z lsh LUU
MA	AZM BDR CKM FTB MTH mtw MYG NDA NED NKB NKF NKT nsb ULN ULY WEX WHE WNE WPG WQC WXW
MD	AAL CUC GOC HCF MRB MSM MSU SSC TSC UMC WTY
ME	C7T CBY MEU umg
MI	EEI EES EVV EXN eyd eyg EYL EYM EZS EZU R2A
MN	CKK DML LNT MAC mha MNA MND MNG MNJ MNK MNM mnsbu MNU MUO NCA NDMLM SPB UTN
MO	CS2 CVN MKN MOP MOV MUU mz4 UMK UMR WTU XII
MS	MFM tgc
MT	MTG MZD MZJ MZK
NC	BEN esm EWF HQP hzv jcrlt KQW KQY KZG KZK NCX NDD NEH NEO NFS NGU NGW NMC NOC NQA NRC NSC nug NVS NZW
ND	NDI NDV
NH	FPR HC7 HF9 KNM NHM PEX PSM SAC
NJ	ANJ BER CAL CMO CSE DRU FCE FDU GGN NJJ NJK NJM NNF NNJ NNO STH STL STP
NM	a4a enmrb ENR FEH IQU iru MQM n5m NMH NMT PUV S2J t4s ugc ULA VL#
NV	nevst TUHNV
NY	a8c COO CTX GGM journ NAM NYG RVA RVE SYB UVT VDB VJA VKC VSI VVB VVH VVJ VVN VVS VVV VXE VXO VXX VYA VYE VYF VYL VYM VZL vzn VZQ VZS VZU xme XQL XQM XTA YBM YJJ YKC YMA YRM YSY YWM ZBL ZBM zcy ZGM ZHL ZHM ZIH ZMU ZNT ZSJ ZVM ZWC ZXC

OH	ANTCH autda BGU cdjad CSU cul DKWMS f#a HUC ND$ NDOSU oclcs ocp OCWMS opt OSU OUN SCT SCW tawms TRN TS225 TS263 ts265 TS282 ts296 tsfil wmsok wmsol woo XAV ZMY
OK	HLO LUO oka OKB OKC OKJ OKN OKR OKT OKU OKY OSZ OUF OUH OUK ouv OVV OVY PS1 SEO
OR	OIT OLP OOY OPU OQP ORX OWS WOS
PA	ALL CHE cmz CRC EVI HFC IMM KZS LAD LBB LFM LVC MAN MOR PB3 PBE PCT PGM pit PJU PTL PVW ROB sbpct SQP TEU UPM wfn WSB
PR	PBR prcay UPTPR
RI	BRB CCW PJW PRC RBN RCM RIU RWB SLV
SC	SBI SBM SCN SEA SFM SGP SPW
SD	DST MIT NOS PRN SDA SDB SDF SDW SMT
TN	tbc TEB tlc TLU TMK TNF TSU TTN TUU TWS
TX	ACNLC AHH bwz CBD CJT cqc DTM EDX HTC IAU ICU IDA IEA IET IHS IUA IWU IYU ktx LET LOR n$v NDT7P NTX POA rl2 SA7 SAJ SAT snc SNM SSP STX SUR SWT TAG TAMCT TAMSA TJF TLS TLT TMI TOM TP# tph TUT TWB TWC TXA TXI TXJ TXT txx TZP ZH7 zxx
UT	UUA UUO UUS
VA	did IPSYS MMV P$H PYY PZC PZL PZO PZT QMC RNKHE TWP VA@ vaevc VAQ VBA VBC vcb VCN VGM VHS VLY VMB VMI VNN VNS VOD VPI VRA VRC VRM VWM
VT	CQG NRU SI# SMD
WA	BV1 esr H5I HNW L@W OEM OPV OXF SJI WUG XFF Z84 Z88
WI	GZB GZC GZM GZS GZU WCD WEZ WII WIO WMO WNL WOM WQT wsd WZJ ZR1
WV	CWV WV3 wvd WVF WVH WVO WVQ WVS WVU WVX
WY	WYU
AE	A7U AEBUD aehbu aepsu UAE ZAQ
AM	AMAUA
AR	U@S
AU	ANZT2 atcdl ATCSE atioc atutq atxas atxja atxnh atxtm aukpl aulsa aumlt auntl auptl AUT auteg auten b2w C7F ld1 lf1 lh1 ll0 LM1 ls0 lzt ntttn YQU zp1 atmet
AW	awuoa
BE	BECOE pwq
BG	BGAUB
BW	BWBIU
AB	cn4x3 cn6eg CNMTR cnnor CNNWL CNPOR cnspo cntab cntbc cocuf gprcl L6B UAB
BC	R7R rct SFB ubc VT2
MB	BBW BCW cn3ha CNCMU JG0 UAT
NB	ubf UNBCA
NL	QE2
NS	a6t akecl AVCLL burcl cumcl HALCL intcl kgscl LUNCL MARCO PICCL SHLBL snk STACL trucl
ON	CNTCS CNTRU lhu OTZ Q2U rdc YOU
QC	B6U b6usb CDN E5P e5psb LGG lvb Q3A Q3C Q3CSB Q3H Q3R q3rsb Q3T Q3TSB RBA U3Q U9S UQ1 UQBSB WMSMB
SK	REB UOH
YT	CN4WZ
CH	f3n
CL	CLUSH rsflo
CN	sja T3U
CW	nauna
CZ	cvk
DE	defhm
DO	drpuc
EC	eclae
ES	RIV S9M spudb
FR	frbey

GB	AUD bo@ BUB cud EQF erf eun eux h@u HS0 LDS ldsns lip lnd lun rdf S2H SXB U3W U@M uhl UK5EZ uk7lj ukbru ukbtc ukbth ukehc ukime UKITR UKOBT UKOBU uktcc yws
GR	geq
HK	HUA
IL	I8H i9w s8b
IQ	iqsul
IT	ITJCU
JM	U@J
KG	kguca
KR	ugk
KW	kuwai
LB	LEAUB
MT	mtaum
MX	unamx
MY	mymas mypen mypmp mytml myump myuth upu
NA	W4K W6@
NL	GRU KOB NEHVU NLFNH NLHHG nlhrd QGJ QGK TSEPM VU@
NZ	NZLIP nzmds UV0 UX0 UX1 Z5A
OM	auq
PH	phust
SE	sehis swesh
SG	sinlb
TH	t5n THCMU THHCU thmsk TULIB
TR	T9K
TT	trutt
TW	NTU t8z WMSRB
VN	vnvgu
ZA	CQ$ OD$ OG# OG$ OH# OI@ OL$ OM@ P4A socms U4V Y#N Y$E Y$R Y7X Y8N Y9N Y@Y Z5F Z7E Z@L ZASPU

CLOSE WINDOW

ATTACHMENT 3A

WorldCat: Euro-Par'99 parallel processing

OCLC	42290131			**No holdings in LDL - 153 other holdings**							
Books	**Rec Stat**	c		**Entered**	19990805		**Replaced**	20220530225832.5			
Type	a	**ELvl**	4	**Srce**			**Audn**		**Ctrl**	**Lang**	eng
BLvl	m	**Form**		**Conf**	1		**Biog**		**MRec**	**Ctry**	gw
		Cont	b	**GPub**			**LitF**	0	**Indx**	1	
Desc	a	**Ills**	a	**Fest**	0		**DtSt**	s	**Dates**	1999,	

010		99040195
040		DLC $b eng $c DLC $d C#P $d OHX $d UBA $d C$Q $d LVB $d OCLCQ $d NLGGC $d BTCTA $d YDXCP $d IG# $d ZWZ $d GW5XE $d OCLCQ $d OCLCF $d DEBSZ $d OCLCQ $d QE2 $d BDX $d OCLCO $d OCLCQ $d OCL $d OCLCO $d CAUOI $d OCLCQ $d UWO $d YOU $d OCLCQ $d IL4J6 $d OCLCO
015		99,N34,0417 $2 dnb
016	7_	957067461 $2 DE-101
019		456631805 $a 960152027 $a 1049759646 $a 1173119531
020		3540664432 $q (alk. paper)
020		9783540664437 $q (alk. paper)
042		pcc
050	00	QA76.58 $b .I553 1999
055	_3	QA75 $b .L38 no.1685
072	_7	QA $2 lcco
082	00	004/.35 $2 21
084		54.51 $2 bcl
090		
049		LDLL
111	2_	International EURO-PAR Conference $n (5th : $d 1999 : $c Toulouse, France)
245	10	Euro-Par'99 parallel processing : $b 5th International Euro-Par Conference, Toulouse, France, August 31-September 3, 1999 : proceedings / $c Patrick Amestoy [and others] (eds.).
260		Berlin ; $a New York : $b Springer, $c 1999.
300		xxxii, 1503 pages : $b illustrations ; $c 24 cm
336		text $b txt $2 rdacontent
337		unmediated $b n $2 rdamedia
338		volume $b nc $2 rdacarrier
490	1_	Lecture notes in computer science, $x 0302-9743 ; $v 1685
504		Includes bibliographical references and index.
530		Also available via the World Wide Web.
650	_0	Parallel processing (Electronic computers) $v Congresses.
650	_0	Parallel processing (Electronic computers) $x Congresses.
650	_6	Parallélisme (Informatique) $0 (CaQQLa)201-0057785 $x Congrès. $0 (CaQQLa)201-0378208
650	_6	Parallélisme (Informatique) $0 (CaQQLa)201-0057785 $v Congrès. $0 (CaQQLa)201-0378219
650	_7	Parallel processing (Electronic computers) $2 fast $0 (OCoLC)fst01052928
650	17	Parallelle verwerking. $2 gtt
650	17	Parallelprogrammering. $2 gtt
650	_7	Arquitetura de computadores. $2 larpcal
650	_7	Engenharia de programacao (software) $2 larpcal
650	_7	Teoria da computacao. $2 larpcal
650	_7	Parallel processing (Electronic computers) $v Congresses. $2 nli
650	_7	Parallélisme (informatique) $x Congrès. $2 ram
655	_7	Conference papers and proceedings. $2 fast $0 (OCoLC)fst01423772
700	1_	Amestoy, Patrick.
830	_0	Lecture notes in computer science ; $v 1685. $x 0302-9743

856 41 $3 SpringerLink $u https://doi.org/10.1007/3-540-48311-X
856 41 $3 Table of contents $u http://swbplus.bsz-bw.de/bsz079646107inh.htm
856 41 $u http://link.springer-ny.com/link/service/series/0558/tocs/t1685.htm $z Restricted to Springer LINK subscribers
856 41 $u http://link.springer.com/10.1007/3-540-48311-X $z eBook available for UOIT via SpringerLink. Click link to access
856 4_ $3 Cover $u http://swbplus.bsz-bw.de/bsz079646107cov.htm
856 4_ $3 Kapitel 1 $u http://swbplus.bsz-bw.de/bsz079646107kap.htm
856 4_ $x LNCS $u http://www.springerlink.com/openurl.asp?genre=issue&issn=0302-9743&volume=1685 $z Connect to Internet resource
856 4_ $u http://www.link.springer-ny.com/link/service/series/0558/tocs/t1685.htm
938 Brodart $b BROD $n 54710065 $c $125.00
938 Baker and Taylor $b BTCP $n 99040195
938 Otto Harrassowitz $b HARR $n har000724897
938 Ingram $b INGR $n 9783540664437
938 YBP Library Services $b YANK $n 1599946
029 1_ AU@ $b 000020828171
029 1_ AU@ $b 000022701875
029 1_ DEBSZ $b 079646107
029 1_ IG# $b 9783540664437
029 1_ NLGGC $b 185669816
029 1_ NZ1 $b 5327030
029 1_ OHX $b har000724897
029 1_ YDXCP $b 1599946

Action Status Delete Holdings _ Export _ Label _ Produce _ Update Holdings _ Validate _

ATTACHMENT 3B

Springer Link

Search Log in



European Conference on Parallel Processing

Euro-Par 1999: **Euro-Par'99 Parallel Processing** pp 950–960

A Parallel Accelerator Architecture for Multimedia Video Compression

Bertil Schmidt & Manfred Schimmler

Conference paper | First Online: 01 January 1999

136 Accesses | **7** Citations

Part of the Lecture Notes in Computer Science book series (LNCS,volume 1685)

Abstract

This paper describes a parallel architecture for a variety of algorithms for video compression. It has been designed to meet the requirements of encoding and decoding according to the ITU-T standard H.263. The architecture is an implementation of the instruction systolic array (ISA) model which combines the simplicity of systolic arrays with the flexibility of a programmable parallel computer. Although the parallel accelerator unit is implemented on no more than 9 mm2 of silicon it suffices to meet the compression rate necessary to send a

compressed video stream through a standard ISDN terminal interface.

Keywords

Discrete Cosine Transform **Motion Vector**

Systolic Array **Video Compression**

Processor Array

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

Download conference paper PDF

References

1. Video Coding for low bitrate communication, ITU-T Draft Recommendation H.263, Geneva, May (1996)

2. Chang, Y.-T., Wang, C.-L.: New Systolic Array Implementation of the 2-D Discrete Cosine Transform and Its Inverse, IEEE Trans. Circ. Syst. Vid. Tech. 5(2) (1995) 150–157

3. Cheng, S.-C., Hang, H.-M.: A Comparison of Block-Matching Algorithms Mapped to Systolic-

Array Implementation, IEEE Trans. Circ. Syst. Video Tech. 7(5) (1997) 741–757

4. Kunde, M., et al.: The Instruction Systolic Array and its Relation to Other Models of Parallel Computers, *Parallel Computing* 7 (1988) 25–39

5. Lang, H.-W.: The Instruction Systolic Array, a parallel architecture for VLSI, *Integration, the VLSI Journal* 4 (1986) 65–74

6. Lang, H.-W., Maaβ, R., Schimmler, M.: The Instruction Systolic Array-Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers, *Proc. HPCN 94*, LNCS 797, Springer Verlag (1994) 487–488.

7. Pirsch, P., Stolberg, H.-J.: VLSI Implementations of Image and Video Multimedia Processing Systems, *IEEE Trans. Circ. Syst. Video Tech.*, 8(7) (1998) 878–891

8. Schimmler, M., Lang, H.-W.: The Instruction Systolic Array in Image Processing Applications, Proc. *Europto 96*, SPIE 2784 (1996) 136–144

9. Schmidt, B., Schimmler, M., Schröder, H.: Long Operand Arithmetic on Instruction Systolic

Computer Architectures and Its Application to RSA cryptography, *Proc. EuroPar'98*, LNCS 1470, Springer Verlag (1998) 916–922

10. Schmidt, B., Schimmler, M., Schröder, H.: The Instruction Systolic Array in Tomographic Image Reconstruction Applications, Proc. *PART'98*, Springer Verlag (1998) 343–354

Author information

Authors and Affiliations

Institut für Datenverarbeitungsanlagen, TU Braunschweig, Hans-Sommer-Str. 66, 38106, Braunschweig, Germany

Bertil Schmidt & Manfred Schimmler

Editor information

Editors and Affiliations

ENSEEIHT, 2, Rue Camichel, F-31071, Toulouse Cedex 7, France

Patrick Amestoy, Philippe Berger, Michel Daydé & Daniel Ruiz, , &

CERFACS, 42, Av. Gaspard Coriolis, F-31057, Toulouse Cedex 1, France

Iain Duff, Valérie Frayssé & Luc Giraud, &

Rights and permissions

Reprints and Permissions

Copyright information

© 1999 Springer-Verlag Berlin Heidelberg

About this paper

Cite this paper

Schmidt, B., Schimmler, M. (1999). A Parallel Accelerator Architecture for Multimedia Video Compression. In: , *et al.* Euro-Par'99 Parallel Processing. Euro-Par 1999. Lecture Notes in Computer Science, vol 1685. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48311-X_133

.RIS .ENW .BIB

DOI
https://doi.org/10.1007/3-540-48311-X_133

Published	Publisher Name	Print ISBN
06 August 1999	Springer, Berlin, Heidelberg	978-3-540-66443-7

Online ISBN	eBook Packages
978-3-540-48311-3	Springer Book Archive

Share this paper

Anyone you share the following link with will be able to read this content:

Get shareable link

Provided by the Springer Nature SharedIt content-sharing initiative

Not logged in - 76.84.212.16
Big Ten Academic Alliance Formerly (CIC) (3000133814) - University of Nebraska-Lincoln (3000134173)
SPRINGER NATURE

© 2022 Springer Nature Switzerland AG. Part of Springer Nature.

ATTACHMENT 3C

A Parallel Accelerator Architecture for Multimedia Video Compression

Bertil Schmidt and Manfred Schimmler

Institut für Datenverarbeitungsanlagen, TU Braunschweig,
Hans-Sommer-Str. 66, 38106 Braunschweig, Germany
{bschmidt, masch}@ida.ing.tu-bs.de

Abstract. This paper describes a parallel architecture for a variety of algorithms for video compression. It has been designed to meet the requirements of encoding and decoding according to the ITU-T standard H.263. The architecture is an implementation of the instruction systolic array (ISA) model which combines the simplicity of systolic arrays with the flexibility of a programmable parallel computer. Although the parallel accelerator unit is implemented on no more than 9 mm^2 of silicon it suffices to meet the compression rate necessary to send a compressed video stream through a standard ISDN terminal interface.

1 Introduction

Video compression for multimedia systems is a computationally intensive task. The desired compression factor determines the performance requirements of the underlying hardware. Several authors have suggested dedicated VLSI implementations [7] of video compression algorithms. Because of their regularity, high throughput rate, small silicon area, and low power consumption, systolic arrays have been proven as a good candidate structure for these dedicated solutions [2], [3]. Their disadvantage is the lack of flexibility with respect to the implementation of different algorithms and different problem sizes, i.e. each algorithm and each problem size requires an individual hardware solution. ISAs have been developed in order to combine the speed and simplicity of systolic arrays with flexible programmability [5]. Originally, the main application field of ISAs was supposed to be scientific computing. However, in the mid 90s the suitability of the ISA architecture for other applications was recognized, e.g. [6], [8]-[10]. In this paper we illustrate how an ISA architecture can solve all computationally intensive tasks of a multimedia video compression application efficiently. This ISA has been developed in order to compress a source data rate of 15 frames per second (fps) in CIF format according to the ITU-T standard H.263 [1] for a transmission with the given data rate of two ISDN channels.

This paper is organized as follows. The concept of the ISA is explained in Section 2. Section 3 gives an introduction to video compression with H.263. Section 4 presents the new video accelerator architecture. It is documented how the ISA is integrated on an accelerator for videophone applications. The parallel ISA algorithms

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 950-960, 1999.
© Springer-Verlag Berlin Heidelberg 1999

for video compression are explained in Section 5 and their performance is evaluated in Section 6. The outlook to further research topics concludes the paper in Section 7.

2 Principle of the ISA

The ISA is a quadratic array of identical processors, each connected to its four direct neighbours by data wires. The array is synchronized by a global clock. The processors are controlled by instructions, row selectors and column selectors. The instructions are input in the upper left corner of the processor array, and from there they move step by step in horizontal and vertical direction through the array. This guarantees that within each diagonal of the array the same instruction is active during each clock cycle. In clock cycle k+1 processor (i+1,j) and (i,j+1) execute the instruction that has been executed by processor (i,j) in clock cycle k. The selectors also move systolically through the array: the row selectors horizontally from left to right, column selectors vertically from top to bottom. Selectors mask the execution of the instructions within the processors, i.e. an instruction is executed if and only if both selector bits, currently in that processor, are equal to one. Otherwise, a no-operation is executed.

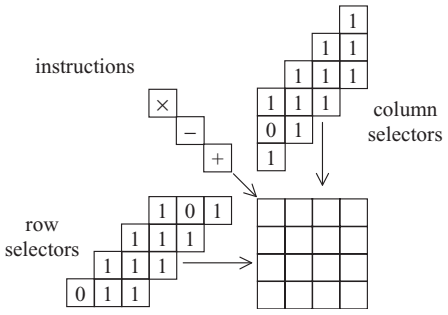


Fig. 1: Control flow in an ISA

Every processor has read and write access to its own memory. Besides that, it has a designated *communication register* (*C-register*) that can also be read by the four neighbour processors. Within each clock phase reading access is always performed before writing access. Thus, two adjacent processors can exchange data within a single clock cycle in which both processors overwrite the contents of their own communication register with the contents of the communication register of its neighbour. This convention avoids read/write conflicts and also creates the possibility to broadcast information across a whole row or column with one single instruction. This property can be exploited for an efficient calculation of row broadcasts, row ringshifts, and row sums which are the key-operations in many algorithms described in Section 5.

3 Multimedia Video Compression

The high amount of visual data associated with typical multimedia services establishes the need for efficient data compression schemes in order to facilitate transmission and storage applications. Several international standards have been introduced for video compression targeting different application fields. Communication applications (e.g. videophone, teleteaching) are covered by the ITU-T standard H.263. The H.263 codec codes video frames using a discrete cosine transform (DCT) on blocks of size 8 × 8 pixels. An initial frame is coded and transmitted as an independent frame. Subsequent frames, which are modelled as changing slowly due to small motions of objects in a scene, are coded efficiently in the inter mode using motion compensation (MC) in which the displacement of groups of pixels from their position in the previous frame are transmitted together with the DCT-coded difference between the predicted and original images. Fig. 2 shows a block diagram of the codec.

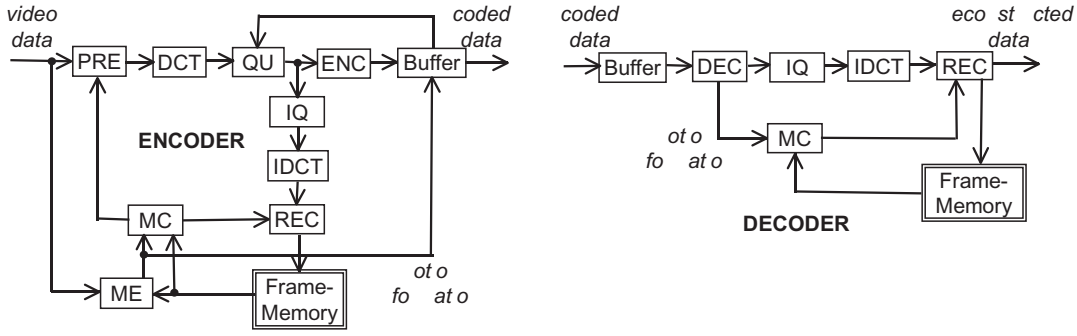


Fig. 2: Tasks and data flow of the H.263 encoder and decoder

The first step in the interframe coder is to calculate a motion vector for the current 16 × 16 pixels macroblock (**MB**) in ME (motion estimation). The motion vector is obtained by minimizing a cost function measuring the mismatch between a candidate MB in the previous frame and the current MB. Although several cost measures have been introduced [7], the most widely used one is the sum-of-absolute-differences (*SAD*) defined by $SAD = \Sigma_{k=0\cdots15}\Sigma_{l=0\cdots15} \left| c_{i,j}(k,l) - r_{i-u,j-v}(k,l) \right|$, where $c_{i,j}(k,l)$ represents the pixel (k,l) of a 16 × 16 MB from the current picture at the spatial location (i,j), and $r_{i-u,j-v}(k,l)$ represents the pixel (k,l) of a candidate MB from a reference picture at the spatial location (i,j) displaced by the vector (u,v). To find the MB producing the minimum mismatch error, we need to calculate SAD at several locations within a search window. The simplest, but the most compute-intensive search method, known as the full search or exhaustive search method, evaluates SAD at every possible pixel location in the search area. To lower the computational complexity, several algorithms that restrict the search area to a few have been proposed [3]. In baseline H.263, one motion vector per MB is allowed for motion compensation. Both, horizontal and vertical components of the motion vectors may be of half pixel accuracy, but their values may lie only in the ±15 range, limiting the search window used in ME.

The predicted MB represented by the calculated motion vector is loaded from the frame memory into MC (motion compensation). If the motion vector is of half pixel accuracy this operation requires an interpolation. The motion compensated prediction error is computed by the difference between the predicted and original MB in PRE (prediction). The resulting difference MB is transformed using a DCT of each 8×8 block, quantization by an adaptive quantizer (QU), entropy encoded using a variable-length coder (ENC), and buffered for a transmission over a fixed rate channel. The quantizer step size is calculated by evaluation of the buffer occupancy. After the quantization process, the original MB is reconstructed by the corresponding inverse operations (IQ, IDCT, REC) and stored in the frame memory. Of course, the reconstructed and original MB are not equal because of the performed lossy quantization. Since the reconstructed MB is available to encoder and decoder, it is used for prediction of the next frame.

The intra/inter mode selection is made at the MB level. If a MB does not change significantly with respect to the reference picture, an encoder can also choose not to encode it, and the decoder will simply repeat the MB located at the subject MB's spatial location. Only the processing instructions and capabilities of the decoder are standardized. The only necessary demand on the encoder is to produce a syntactically correct bitstream. The result is that the quality of H.263 video depends on the encoder implementation. In addition to the discussed baseline encoder algorithms, several negotiable options are offered by H.263 and H.263+. These optional modes allow developers to trade off between compression performance and computational complexity.

4 Accelerator Architecture for Video Compression

The analysis of the different video coding tasks and their processing efforts leads to a fixpoint processor architecture. The processor needs a small local memory for the storage and fast supply of local image data. The wordlength of the data items should be chosen to 8 bits because of the 8 bit input pixels and must also be able to process longer operands, e.g. DCT requires intermediate operands of length up to 24 bit. In order to achieve the real time requirements of the H.263 video codec several of these processors have to work in parallel. Thus, the particular processor has to be optimized with respect to its chip area and power consumption. Figure 3 depicts the processor architecture for the implementation of the H.263 video codec.

Due to the limited chip area the processor has to be very compact. In particular, it implies the choice of a bit-serial data organization. The size of the local memory is 32 internal registers plus 2 communication registers (C-registers). The word length of data items is 8 bits. Furthermore, each processor has two special constant registers 0 and –1. In addition to the registers, there are a zero flag and a negative flag that control the processing units depending on the state of the processor.

Two operand-multiplexers choose two registers of the own local memory or of the C-registers of the neighbours. The operands are propagated to the units required for execution of the current instruction. This units can be the multiplier and the

arithmetic, logical, and conditional unit (ALCU). The corresponding instruction set consists of 24 instructions. The result of the execution is given to the result bus and from there they are written into the local registers or the C-registers or both.

During the execution of one instruction the processor receives the next instruction together with a column selector bit from the upper neighbour and a row selector bit from the left neighbour. The selectors are interpreted. If both are 1 then the execution of the instruction is prepared. If one of the selectors is 0 then the execution of a no-operation is initialized. Instruction and column selector are propagated bit serially to the lower neighbour and the row selector to the right neighbour during the execution of the instruction.

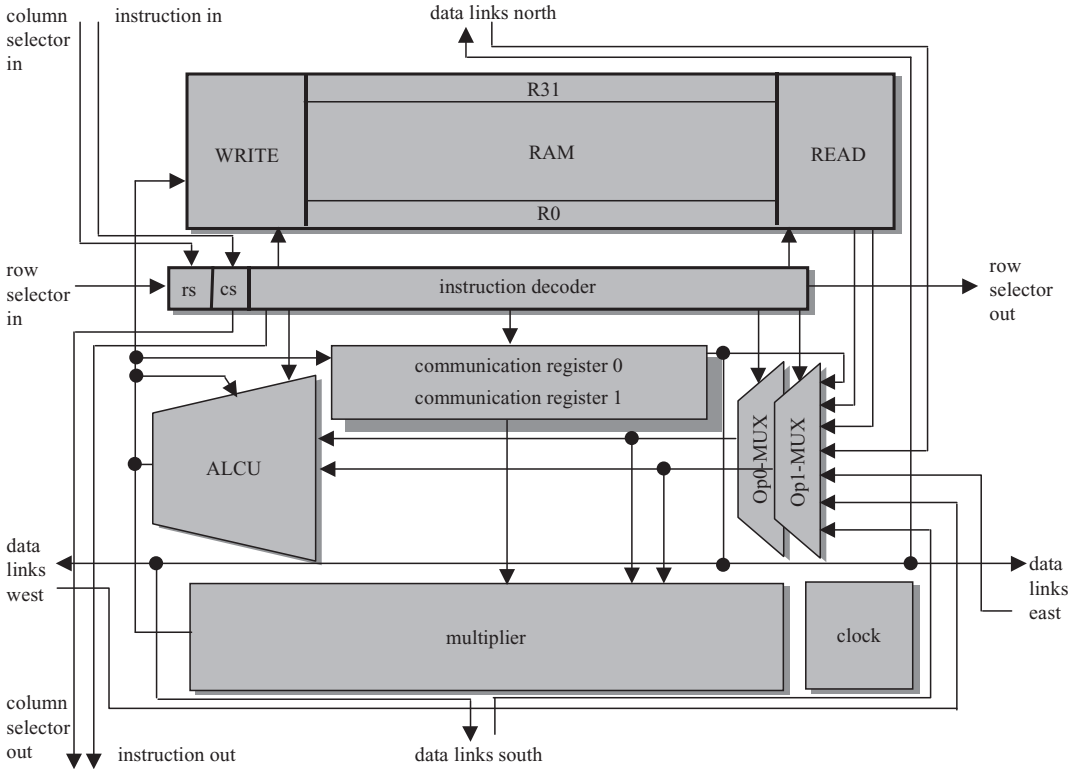


Fig. 3: Architecture of the array processor

The fine grained pipelined execution units are laid out for bitwise 300 MHz true single phase clock. Table 1 shows the areas of the different units of the processor. The full-custom design has been made with a 0.25µ digital CMOS process. An 12×8 ISA of the described processors has been structured in order to provide real-time processing for an H.263 codec at CIF resolution (352×288 pixels) and 15 fps. The resulting silicon area of the processor array is 3.7 mm^2. At a bitwise clock frequency of 300 MHz and a word format of 8 bits the theoretical peak performance of the array is 3.6 GIPS. The corresponding estimated power consumption is 120 mW. In order to exploit the computation capabilities of this unit, it is necessary to provide data and control information at an extremely high speed. Therefore, a cascaded memory

concept is implemented on-chip that forms a fast input and output environment for the parallel processing unit (see Figure 4).

For the fast exchange of data with the processor array each processor has two memory banks. Each memory bank contains 8 interface registers. One of these banks is always assigned to the corresponding processor, the other to the internal RAM by means of a fast data channel. The exchange of data between ISA and the internal RAM is done by bank switching. Both memory banks can be active at the same time, i.e. data transfer between ISA and internal RAM can be done concurrently to the execution of an ISA program. The internal RAM can also communicate bidirectionally with the external SDRAM.

The data transfer is controlled by an on-chip controller. The controller is started by the sequential standard RISC core (e.g. Hitachi SH4) and operates autonomously afterwards. It receives instructions from an instruction queue. The controller supplies the processor array with instructions and selectors that are stored in the ISA program memory. The instruction queue and the ISA program memory are located in the external SRAM, separated from the data. This part of the SDRAM is exclusively available to the controller by means of a fast channel. The internal RAM is organized as single ported static RAM. Its size of 8 KByte is mainly determined by the H.263 implementation (see Section 5). The layout of the complete video accelerator with a 0.25µ digital CMOS process requires an area of 8.1 mm^2 as depicted in Table 2.

Table 1: Area of the individual processor.

RAM	12.144 μm^2
Multiplier	6.398 μm^2
Read-write-logic	2.874 μm^2
ALCU	5.125 μm^2
Instruction decoder	4.011 μm^2
Addressing	1.190 μm^2
Routing	1.674 μm^2
Operand multiplexers	2.133 μm^2
Clock	2.613 μm^2
Sum	**38.135 μ^2**

Table 2: Area of the accelerator.

ISA	3.7 mm^2
Controller	1.1 mm^2
Internal RAM	3.3 mm^2
Sum	**8.1 mm^2**

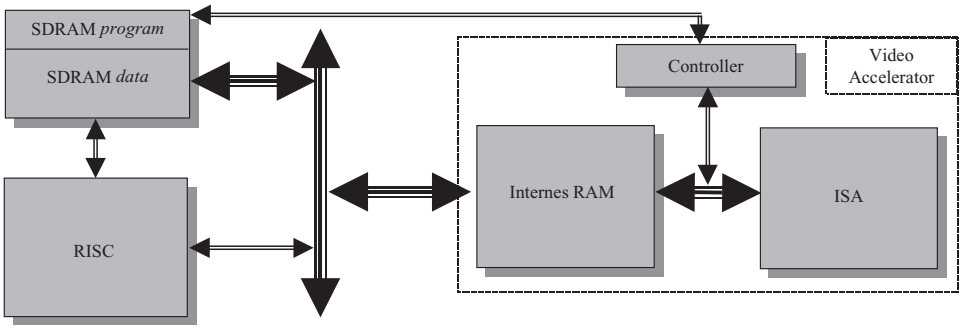


Fig. 4: Video accelerator architecture and data paths to the environment.

5 Parallelization of Video Compression Algorithms

The tasks of the hybrid coding scheme can be subdivided into low level, medium level, and high level tasks. Low level tasks typically operate on pixel data. They are highly regular and offer a large potential of data parallelism. Examples for low level tasks are full search motion estimation, DCT, and prediction. These tasks require a main percentage of the overall processing effort. Medium level tasks like quantization or entropy coding operate on the results of the low level tasks. Computation requirements are significantly lower than for low level tasks. The high level tasks comprise the bitstream handling and several control tasks, primary on encoder side, like quantizer control and decisions for coding strategy. High level tasks show an irregular control flow in conjunction with low computational rates. For control tasks a certain amount of flexibility with respect to modifications of the algorithms is advantageous. The proposed coprocessor architecture for multimedia comprise a flexible general-purpose RISC core with a less flexible accelerator adapted towards a specific type of processing. Thus, the RISC core performs the high level tasks of lower computational requirements, whereas the video accelerator executes the computation intensive but regular low level tasks and a part of the medium level tasks. The parallelization of the H.263 coding tasks on the proposed accelerator architecture is described in the following.

ME: In the 2D array structure of a 4×8 ISA the calculation of one SAD can be efficiently parallelized as follows: Assume, the current MB data $c(k,l)$ and the reference frame candidate MB data $r(k,l)$, $k,l = 0,\ldots,15$, are stored in the processor array such that processor (i,j), $i = 0,\ldots,3$, $j = 0,\ldots,7$, holds $c(4\cdot i,2\cdot j),\ldots,c(4\cdot i+3,2\cdot j+1)$ and $r(4\cdot i,2\cdot j),\ldots,r(4\cdot i+3,2\cdot j+1)$ in 16 internal registers. Now each node (i,j) computes the partial sum $\Sigma_{m=0,\cdots,3,n=0,\cdots,1}|c(4\cdot i+m,2\cdot j+n) - r(4\cdot i+m,2\cdot j+n)|$ within 30 instructions. The result is written into the two C-registers *C1* (most significant byte) and *C0* (least significant byte). The partial sums are added along each processor row within 2 instructions using two efficient row sum operations (see Section 2). The first operation adds up the values of *C0* and the second operation adds up the values of *C1* and the carry-bit of the previous addition. The total sum (SAD) for the candidate MB

is completed by the corresponding two column sum operations in the rightmost processor column and the result is written into *C0* and *C1* of the lower right processor. This SAD is compared in the lower right processor to the stored SAD resulting from the previous comparison. The smaller one as well as the corresponding motion vector is kept for future comparisons.

The preceding computation procedure is repeated until all possible candidate MBs are compared and the final motion vector is obtained. To speed up the computation the calculated SAD value is compared to a preselected threshold value. The search process terminates if the calculated SAD value is smaller than the threshold. Because the processor array of the video accelerator is of size 12×8 three different SAD values for the current MB can be calculated in parallel by a simple replication of the program for the 4×8 array.

Shifting of the candidate MB within the search area depends on the chosen search strategy. In order to avoid delay times in the systolic architecture it is necessary to know these locations in advance, and thus the next candidate MB data can be preloaded from the internal RAM into the ISA during the processing of the current MB. The ±15 pixels 2-step search with subsequent half pixel refinement is suitable and leads to a very good coding quality [3]. In the first step, it evaluates the motion vector candidates with distance two between two nearby search points within the ±15 pixels search area. Thus, 225 SAD values of previously known locations have to be calculated in the worst case. In the second step, the SAD values of the 8 neighbours of the calculated minimum point are evaluated. A subsequent half pixel search around the full pixel accurate displacement vector evaluates the motion vector with half-pixel accuracy. This involves the additional evaluation of 8 candidate blocks per motion vector. The locations of candidate points in the second iteration step and the half pixel search are data dependent. Thus, the data locations to be used at the next step are transferred to the controller when the current step is completed. At the beginning of the new step, the controller loads the corresponding reference data into the ISA. MBs of adjacent candidate vectors overlap quite significantly. In order to reduce memory accesses (bandwidth) only the non-overlapping data have to be loaded into the processor array. Since SAD values for 3 horizontally neighboured motion vectors are computed in the processor array in parallel only 2×20 pixels have to be loaded when the search area moves vertically up in the first step of the 2-step search. Search areas of adjacent MBs also overlap. Thus, only the non-overlapped data of size 48×16 pixels is loaded from the external SDRAM to the internal RAM, which can be performed concurrently to the motion vector computation of the current MB.

MC: The complete reference frame MB corresponding to the calculated motion vector has to be loaded into the ISA. In contrast to the other processing tasks ME is only performed on the luminance data of the MB. For MC and the preceding tasks another mapping of MB data onto the ISA is used. Each processor holds a 2×2 subarray. Thus, each of the six 8×8 blocks of the MB is mapped onto a 4×4 processor subarray of the 12×8 ISA. If the motion vector is of half pixel accuracy linear interpolation is processed in the processor array.

PRE/REC: The motion compensated MB and the original MB are simply subtracted (prediction) resp. added (reconstructed). Only 4 subtraction resp. additions have to be performed within each processor in parallel.

DCT/IDCT: The DCT of an 8×8 block $x_{i,j}$, $i,j = 0,\ldots,7$, can be expressed as

$$y_{k,l} = \sum_{i=0}^{7} c_{i,k} \left[\sum_{j=0}^{7} x_{i,j} \cdot c_{j,l} \right], \text{ where } c_{j,l} = \cos\left((2j+1)l\pi / 16\right) \qquad (1)$$

The core operation in DCT computation is the multiply accumulation (MAC).

$$y_{2l} = \sum_{j=0}^{3} \left(x_j + x_{7-j}\right) \cdot c_{j,2l}; \; y_{2l+1} = \sum_{j=0}^{3} \left(x_j - x_{7-j}\right) \cdot c_{j,2l+1} \qquad \text{for } l = 0,\ldots,3 \qquad (2)$$

Therefore, the multiplier of the introduced ISA processor has been designed to perform MAC efficiently, i.e. simultaneously with multiplication, a third operand can be added to the result without additional delay. As already indicated by the brackets in (1), the 2D transform can be decomposed into cascade of two subsequent 1D transforms applied horizontally and vertically on each row, resp. column. Additional efficiency can be gained, if we split the 1D transform of a sequence x_j, $j = 0,\ldots,7$, into even and odd numbered frequency samples:

This algorithm is applied for the 8×8 DCT within each 4×4 processor subarray. The input values for the 1D DCT are initially permuted within each processor row of length 4, such that each processor j, $0 \le j \le 3$, holds x_j and x_{7-j} and the addition $x_j + x_{7-j}$ and the subtraction $x_j - x_{7-j}$ can be performed in parallel. Now each processor j, $0 \le j \le 3$, computes y_{2j} and y_{2j+1} according to (2) in 4 stages. The computation in each stage requires two MAC operations per processor. Permutation of the values $x_j + x_{7-j}$ and $x_j - x_{7-j}$ between every two stages involves routing of data, which is performed efficiently within each processor row by ringshifting. The coefficient values are precomputed and loaded into the processor array when needed. This load operation takes only a small number of instruction cycles since the values can be broadcasted along the columns of the ISA. The 1D DCT along the processor columns is computed by the *reflected ISA program* for the rows. The IDCT is implemented in a similar way.

QU/IQ: The division operation for quantization is evaluated by multiplication with the reciprocal value in each processor in parallel. The quantizer step size is controlled by loading the corresponding coefficients into the processor array.

ENC/DEC: Due its irregularity the variable length en/decoding and entropy en/decoding is not suitable for an efficient parallelization on the ISA. Thus, the sequential RISC performs these operations. Because of their low computational requirements, their runtime on the RISC is dominated by the processing time of the next/previous MB on the video accelerator.

6 Performance Evaluation

For the runtime determination of the parallel programs described in Section 5 on the introduced accelerator architecture the number of required instructions is multiplied with the corresponding clock cycle time of 26.7 ns. Additionally, the data transfer

between ISA and internal RAM is considered with a throughput of 150 MByte/s. The results are shown Table 3. Note that the times for ME are worst cases, i.e. calculated SAD values are never smaller than the preselected threshold.

Table 3: Worst case runtime on the video accelerator for encoding and decoding per MB. It includes computing time on the ISA and data transfer time between ISA and internal RAM. Data transfer is divided into concurrent transfer, i.e. the transfer time is dominated by the computing time on the ISA, and not concurrent transfer, i.e. during the transfer time no computations are performed on the ISA.

Task		Runtime video accelerator	Data transfer ISA ↔ internal RAM	
			Concurrent to ISA	Not concurrent to ISA
En-coder	ME step 1	103 µs	Load 4080 Bytes	Load 640 Bytes
	step 2	6 µs	Load 36 Bytes	Load 288 Bytes
	half pixel	8.5 µs	Load 18 Bytes	Load 17 Bytes
	MC	5 µs	Load 128 Bytes	Load 451 Bytes
	PRE	0.5 µs		
	DCT and QU	10 µs		
	IQ and IDCT	10 µs	Store 384 Bytes	
	REC	3 µs		Store 384 Bytes
De-coder	IQ and IDCT	10 µs	Load 451 Bytes	Load 384 Bytes
	MC	2 µs		
	REC	3 µs		Store 384 Bytes
Sum per MB		161 µs	5097 Bytes	2584 Bytes
Sum per CIF		64 ms	2.02 Mbytes	1.0 MBytes

The amount of data transfer between external SDRAM and internal RAM per MB is

Encoder: current MB (384 Bytes), overlapping search area (768 Bytes), chrominance blocks of motion compensated MB incl. border pixels (162 Bytes), coded MB (384 Bytes), reconstructed MB (384 Bytes).

Decoder: current MB (384 Bytes), motion compensated MB incl. border pixels (451 Bytes), reconstructed MB (384 Bytes).

The resulting total amount of data transfer of 3301 Bytes per MB, resp. 1.3 MBytes per CIF frame, can be transferred concurrently to the accelerator activities with a reasonable throughput of the SDRAM bus, e.g. 100 MByte/s.

7 Conclusions

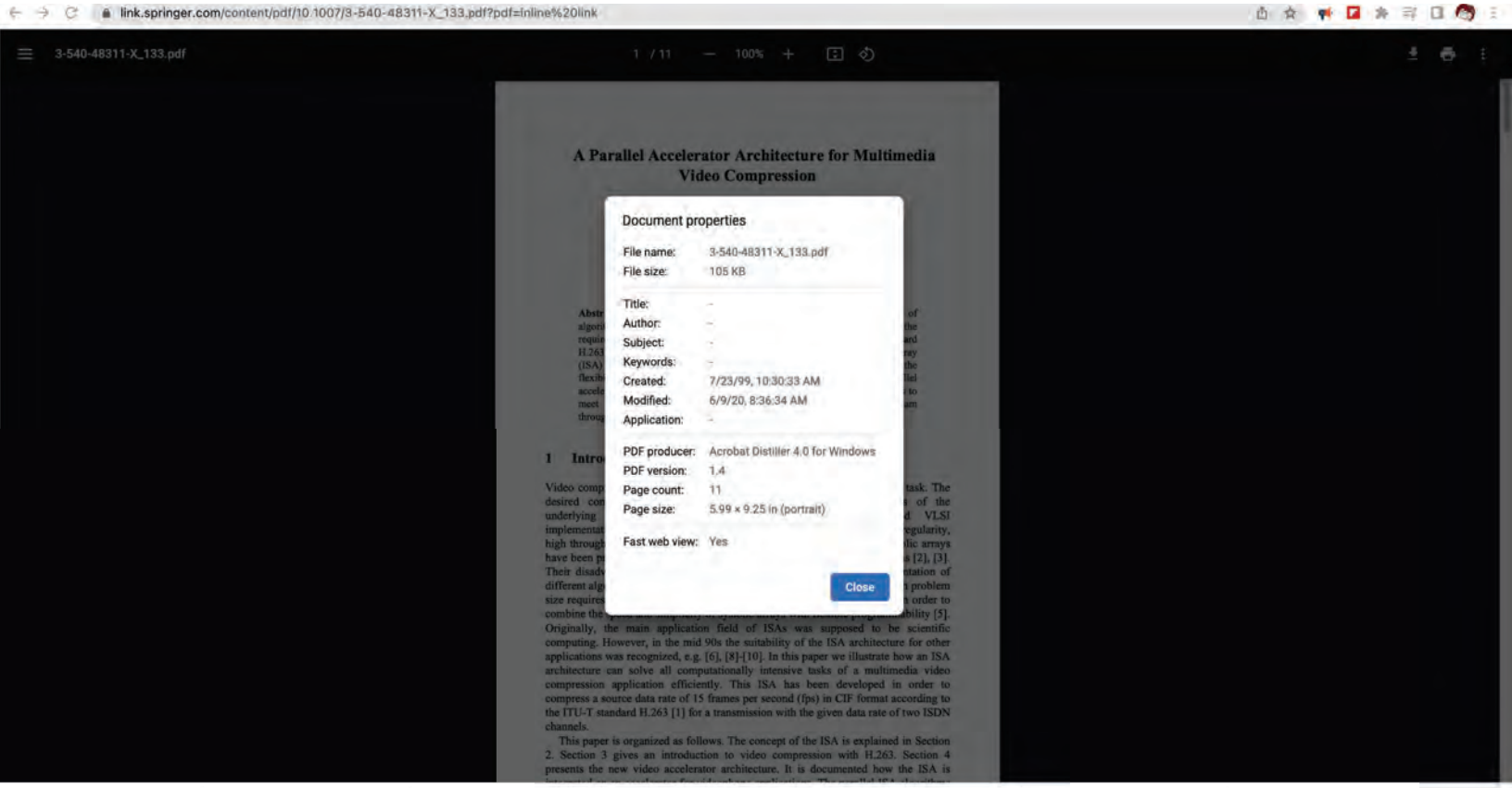
In this paper we have presented an ISA architecture for algorithms required for video compression. The accelerator unit has been implemented on an area of 8.1 mm^2 of silicon using a 0.25µ digital CMOS process. It is capable of on-line encoding and decoding of a video stream of 15 fps in CIF format with the H.263. The global architecture of the accelerator unit has been discussed as well as the detailed implementation of the single processing element of the 12 × 8 array. It has been

shown how the new architecture can be programmed for applications as motion estimation, motion compensation, DCT, and quantization. Apart from its performance figures, the most promising property of the accelerator unit is its flexibility. It would be interesting to study the performance for the new architecture for applications like [8], [9], [10].

References

1. Video Coding for low bitrate communication, ITU-T Draft Recommendation H.263, Geneva, May (1996)
2. Chang, Y.-T., Wang, C.-L.: New Systolic Array Implementation of the 2-D Discrete Cosine Transform and Its Inverse, IEEE Trans. Circ. Syst. Vid. Tech. 5 (2) (1995) 150-157
3. Cheng, S.-C., Hang, H.-M.: A Comparison of Block-Matching Algorithms Mapped to Systolic-Array Implementation, IEEE Trans. Circ. Syst. Video Tech. 7 (5) (1997) 741-757
4. Kunde, M., et al.: The Instruction Systolic Array and its Relation to Other Models of Parallel Computers, *Parallel Computing* 7 (1988) 25-39
5. Lang, H.-W.: The Instruction Systolic Array, a parallel architecture for VLSI, *Integration, the VLSI Journal* 4 (1986) 65-74
6. Lang, H.-W., Maaß, R., Schimmler, M.: The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers, *Proc. HPCN 94*, LNCS 797, Springer Verlag (1994) 487-488.
7. Pirsch, P., Stolberg, H.-J.: VLSI Implementations of Image and Video Multimedia Processing Systems, *IEEE Trans. Circ. Syst. Video Tech.*, 8 (7) (1998) 878-891
8. Schimmler, M., Lang, H.-W.: The Instruction Systolic Array in Image Processing Applications, Proc. *Europto 96*, SPIE 2784 (1996) 136-144
9. Schmidt, B., Schimmler, M., Schröder, H.: Long Operand Arithmetic on Instruction Systolic Computer Architectures and Its Application to RSA cryptography, *Proc. Euro-Par'98,* LNCS 1470, Springer Verlag (1998) 916-922
10. Schmidt, B., Schimmler, M., Schröder, H.: The Instruction Systolic Array in Tomographic Image Reconstruction Applications, Proc. *PART'98*, Springer Verlag (1998) 343-354

ATTACHMENT 3D


link.springer.com/content/pdf/10.1007/3-540-48311-X_133.pdf?pdf=inline%20link
3-540-48311-X_133.pdf
1 / 11
100%
A Parallel Accelerator Architecture for Multimedia Video Compression
Document properties
File name: 3-540-48311-X_133.pdf
File size: 105 KB
Title: -
Author: -
Subject: -
Keywords: -
Created: 7/23/99, 10:30:33 AM
Modified: 6/9/20, 8:36:34 AM
Application: -
PDF producer: Acrobat Distiller 4.0 for Windows
PDF version: 1.4
Page count: 11
Page size: 5.99 × 9.25 in (portrait)
Fast web view: Yes
Close

ATTACHMENT 3E

All locations : No holdings in LDL - 153 other holdings

International EURO-PAR Conference (5th : 1999 : Toulouse, France) Euro-Par'99 parallel processing : 5th International Euro-Par Conference, Toulouse, France, August 31-September 3, 1999 : proceedings / Patrick Amestoy [and others] (eds.). Berlin ; New York : Springer, 1999. xxxii, 1503 pages : illustrations ; 24 cm OCLC:42290131

Holdings Information

STATE	LOCATIONS
AL	ABC
AR	ASU
AZ	AZS AZU
CA	albrs CLU CSL CUI ZAS
CT	YUS
DC	DLC
DE	DLM
FL	EBC FHM FQG FTU
GA	GAT
IL	IAC IAH IAS IAY SOI UIU
IN	IND
KS	KKU
KY	KUK
LA	LRU LUU
MA	AUM HUL mbb TFW
MD	ADB JHE UMC
ME	MEU
MI	a3v EXW
MO	MUU UMS WTU
NC	btcta NDD NRC
NH	ydx
NJ	NJR OAY
NV	UNL
NY	COO NAM NYP RRR SYB VYF YSM ZCU
OH	CWR KSU OSU relva
OR	OHS ORE ORZ
PA	bdx PAU TEU UPM
RI	RBN
SC	SUC
TN	TJC
TX	IUA RCE TXH TXJLF
VA	VGM VWM
VT	VTU
WA	msf NTE XFF
WI	GZM
WV	WVU
AU	atrmh C7F LM1 ln1 LS1 UMV
BB	bwi
BR	uba
AB	UAB UAU
BC	SFB ubc
NL	QE2
ON	cai CAUOI CEF CNECS cnuto OTZ UWW YOU

CN	N5L SLY x8i
DE	dedds dekab dekal depin destu deulu du# ohx T7C uk#
FI	fiefl
FR	fvm
GB	AUD BUB cud eqo EUE mm9 QCL shs snu ukhwv ukqub
HK	cpv cvu HNK HUA
IE	ERD
IL	da$ I8H BRA IL4J6
JP	c6y k4t KEI t4i
MX	i8m
MY	apums
NL	NLCVW NLTUD QGK
NZ	UW1
SE	cuu
TR	b9k
TW	ckt NTU
ZA	OL$



CLOSE WINDOW

ATTACHMENT 4A

WorldCat: High-performance computing and networking

OCLC	1031785684	**No holdings in LDL - 17 other holdings**									
Books	**Rec Stat**	c	**Entered**	19940630	**Replaced**	20220606001528.4					
Type	a	**ELvl**	M	**Srce**	d	**Audn**		**Ctrl**		**Lang**	eng
BLvl	m	**Form**		**Conf**	1	**Biog**		**MRec**		**Ctry**	nyu
		Cont	b	**GPub**		**LitF**	0	**Indx**	1		
Desc	a	**Ills**	a	**Fest**	0	**DtSt**	b	**Dates**	1994,0000		

010		94014716
040		LWA ‡b eng ‡c LWA ‡d OCLCO ‡d OCLCF ‡d CAI ‡d UWO ‡d UMK ‡d OCLCO ‡d OCLCQ ‡d OCLCO
019		1031611395
020		354057980X ‡q (v. 1 ; ‡q soft)
020		9783540579809
020		038757980X ‡q (U.S. ; ‡q v. 1)
020		9780387579801
020		3540579818 ‡q (v. 2 ; ‡q soft)
020		9783540579816
020		0387579818 ‡q (U.S. ; ‡q v. 2)
020		9780387579818
050		QA76.88 ‡b .I57 1994
082	0_	004.1/1 20
090		
049		LDLL
111	2_	International Conference and Exhibition on High-Performance Computing and Networking ‡d (1994 : ‡c Munich, Germany)
245	10	High-performance computing and networking : ‡b International Conference and Exhibition, Munich, Germany, April 18-20, 1994 : proceedings / ‡c Wolfgang Gentzsch, Uwe Harms (eds.).
260		Berlin ; ‡a New York : ‡b Springer-Verlag, ‡c 1994.
300		2 volumes : ‡b illustrations ; ‡c 24 cm
336		text ‡b txt ‡2 rdacontent
337		unmediated ‡b n ‡2 rdamedia
338		volume ‡b nc ‡2 rdacarrier
490	1_	Lecture notes in computer science ; ‡v 796-797
504		Includes bibliographical references and indexes.
505	0_	v. 1. Applications -- v. 2. Networking and tools.
520		"High-performance computing and networking (HPCN) is driven by several initiatives in Europe, the United States, and Japan. In Europe several groups encouraged the Commission of the European Communities to start an HPCN programme. This two-volume work presents the proceedings of HPCN Europe 1994. Volume 2 includes sections on: networking, future European cooperative working possibilities in industry and research, HPCN computer centers aspects, performance evaluation and benchmarking, numerical algorithms for engineering, domain decomposition in engineering, parallel programming environments, load balancing and performance optimization, monitoring, debugging, and fault tolerance, programming languages in HPC, compilers and data parallel structures, architectural aspects, and late papers."--PUBLISHER'S WEBSITE
650	_0	Supercomputers ‡v Congresses.
650	_0	Computer networks ‡v Congresses.
650	_6	Superordinateurs ‡0 (CaQQLa)201-0062859 ‡v Congrès. ‡0 (CaQQLa)201-0378219
650	_6	Réseaux d'ordinateurs ‡0 (CaQQLa)201-0007877 ‡v Congrès. ‡0 (CaQQLa)201-0378219
650	_7	Computer networks. ‡2 fast ‡0 (OCoLC)fst00872297
650	_7	Supercomputers. ‡2 fast ‡0 (OCoLC)fst01138790
655	_7	Conference papers and proceedings. ‡2 fast ‡0 (OCoLC)fst01423772

655 _7 Conference papers and proceedings. ‡2 lcgft
655 _7 Actes de congrès. ‡2 rvmgf ‡0 (CaQQLa)RVMGF-000001049
700 1_ Harms, Uwe, ‡d 1946-
700 1_ Gentzsch, Wolfgang.
830 _0 Lecture notes in computer science ; ‡v 796-797.
856 4_ ‡u http://www.springerlink.com/openurl.asp?genre=issue&issn=0302-9743&volume=797 ‡z Connect to Internet resource for v.2

Action Status Delete Holdings _ Export _ Label _ Produce _ Update Holdings _ Validate _

ATTACHMENT 4B

Springer Link

Search Log in



International Conference on High-Performance Computing and Networking

HPCN-Europe 1994: **High-Performance Computing and Networking** pp 487–488

The instruction systolic array — Implementation of a low-cost parallel architecture as add-on board for personal computers

Architectural Aspects

Manfred Schimmler, Hans-Werner Lang & Rüdiger Maaß

Conference paper | First Online: 01 January 2005

151 Accesses | **2** Citations

Part of the Lecture Notes in Computer Science book series (LNCS,volume 797)

Abstract

In this paper the architecture of a 1024 processor instruction systolic array is described. Besides the presentation of the concept and the architectural details of this novel type of massively parallel system the paper focusses on the problems that had to be solved during the development phase.

This is a preview of subscription content, access via your institution.

Chapter USD 29.95
Price excludes VAT (USA)

- DOI: 10.1007/3-540-57981-8_165
- Chapter length: 2 pages
- Instant PDF download
- Readable on all devices
- Own it forever
- Exclusive offer for individuals only
- Tax calculation will be finalised during checkout

Buy Chapter

Softcover Book USD 129.00

Learn about institutional subscriptions

References

[DiS88] Dittrich, A., Schmeck, H.: Given's Rotation on the Instruction Systolic Array. In: Wolf, G., Legendi, T., Schendel, U. (eds.): PARCELLA '88, Mathematical Research, Bd. 48, Akademie Verlag, Berlin, 340–346 (1988)

[KLSSS88] Kunde, M. et al., The Instruction Systolic Array and its Relation to Other Models of Parallel Computers, Parallel Computing 7, 25–39, (1988)

[La86] Lang, H.-W., The Instruction Systolic Array, a parallel architecture for VLSI, Integration, the VLSI Journal 4, 65–74, (1986)

[La87] Lang, H.-W., ISA and SISA, Two Variants of a General Purpose Systolic Array Architecture, 2nd Int. Conf on Supercomputing, 460–465, (1988)

[Schi87] Schimmler, M.: Fast Sorting on the Instruction Systolic Array, Technical Report No. 8709, Christian-Albrechts-University, Kiel, (1987)

[Schm86] Schmeck, H., A Comparison-Based Instruction Systolic Array, In: Robert, Y. et al. (eds): Parallel Algorithms and Architectures, North Holland (1986)

[Stra90] Strazdins, P.E.: Control Structures for Mesh-Connected Networks, PhD Thesis, The Australian National University, Canberra, Australia, 1990.

Author information

Authors and Affiliations

ISATEC Soft- und Hardware GmbH, Kiel

Manfred Schimmler, Hans-Werner Lang & Rüdiger Maaß

Editor information

Editors and Affiliations

Rights and permissions

Reprints and Permissions

Copyright information

© 1994 Springer-Verlag Berlin Heidelberg

About this paper

Cite this paper

Schimmler, M., Lang, HW., Maaß, R. (1994). The instruction systolic array — Implementation of a low-cost parallel architecture as add-on board for personal computers. In: Gentzsch, W., Harms, U. (eds) High-Performance Computing and Networking. HPCN-Europe 1994. Lecture Notes in Computer Science, vol 797. Springer, Berlin, Heidelberg. https://doi-org.libproxy.unl.edu/10.1007/3-540-57981-8_165

.RIS .ENW .BIB

DOI
https://doi-org.libproxy.unl.edu/10.1007/3-540-57981-8_165

Published	Publisher Name	Print ISBN
26 May 2005	Springer, Berlin, Heidelberg	978-3-540-57981-6

Online ISBN	eBook Packages
978-3-540-48408-0	Springer Book Archive

Not logged in - 129.93.169.156
Big Ten Academic Alliance Formerly (CIC) (3000133814) - University of Nebraska-Lincoln (3000134173)
SPRINGER NATURE

© 2022 Springer Nature Switzerland AG. Part of Springer Nature.

ATTACHMENT 4C

All locations : No holdings in LDL - 16 other holdings

International Conference and Exhibition on High-Performance Computing and Networking (1994 : Munich, Germany) High-performance computing and networking : International Conference and Exhibition, Munich, Germany, April 18-20, 1994 : proceedings / Wolfgang Gentzsch, Uwe Harms (eds.). Berlin ; New York : Springer-Verlag, 1994. 2 volumes : illustrations ; 24 cm OCLC:1031785684

Holdings Information

STATE	LOCATIONS
AZ	AZN
CA	CUT
FL	FQG
GA	GAT
MS	MFM
OH	ggtst
PA	PAU PMC
TX	INT
AB	UAB
ON	cai CEF
GB	ERL eun mm9
ZA	OL$

CLOSE WINDOW

ATTACHMENT 4D

Back

Browse

The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers

Authors: Manfred Schimmler, Hans-Werner Lang, Rüdiger Maaß Authors Info & Claims

HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II: Networking and Tools • April 1994 • Pages 487–488

Published: 18 April 1994 Publication History

2 0

Feedback

ABSTRACT

No abstract available.

Index Terms (auto-classified)

The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers

Computing methodologies

Theory of computation

Comments

DL Comment Policy

Comments should be relevant to the contents of this article, (sign in required).

0 Comments

Sort by Newest

Privacy **Do Not Sell My Data**

View Table Of Contents

Categories **About**

Back

Books
Proceedings
SIGs
Subscription Information
Author Guidelines
Using ACM Digital Library

Browse

Collections
People
ACM Computing Classification System
Digital Library Accessibility

Join

Join ACM
Join SIGs
Subscribe to Publications
Institutions and Libraries

Connect

Contact
Facebook
Twitter
Linkedin
Feedback
Bug Report

The ACM Digital Library is published by the Association for Computing Machinery. Copyright © 2022 ACM, Inc.

Terms of Usage | Privacy Policy | Code of Ethics

ATTACHMENT 5A

WorldCat: Vision systems--sensors, sensor systems, and components

OCLC	35521098	**No holdings in LDL - 68 other holdings**									
Books	**Rec Stat**	c		**Entered**	19960516		**Replaced**	20220602154602.3			
Type	a	**ELvl**		**Srce**	d		**Audn**		**Ctrl**	**Lang**	eng
BLvl	m	**Form**		**Conf**	1		**Biog**		**MRec**	**Ctry**	wau
		Cont	b	**GPub**			**LitF**	0	**Indx**	1	
Desc	a	**Ills**	a	**Fest**	0		**DtSt**	s	**Dates**	1996,	

010 96067692
040 OWpIT $b eng $c DLC $d SCT $d BTCTA $d YDXCP $d CUI $d DEBBG $d IQU $d OCLCF $d OCLCO $d OCLCQ $d NJR $d OCLCQ $d OCLCA $d LVT $d OCLCO
019 1229269450
020 0819421707
020 9780819421708
042 lccopycat
050 00 TA1505 $b .V57 1996
084 ST 330 $2 rvk
090
049 LDLL
245 00 Vision systems--sensors, sensor systems, and components : $b 10-12 June 1996, Besançon, France / $c Otmar Loffeld, chair/editor ; sponsored by CNRS--Centre National de la Recherche Scientifique [and others].
260 Bellingham, Wash. : $b SPIE--the International Society for Optical Engineering, $c ©1996.
300 v, 154 pages : $b illustrations ; $c 28 cm.
336 text $b txt $2 rdacontent
337 unmediated $b n $2 rdamedia
338 volume $b nc $2 rdacarrier
490 1_ Proceedings EurOpt series
490 1_ SPIE proceedings series ; $v 2784
504 Includes bibliographical references and author index.
505 0_ Imaging sensors and applications -- Multisensor data fusion -- Sensors and effects : techniques and applications -- Interferometric techniques -- Special session on components.
650 _0 Imaging systems $v Congresses.
650 _0 Optical detectors $v Congresses.
650 _0 Multisensor data fusion $v Congresses.
650 _6 Imagerie (Technique) $0 (CaQQLa)201-0013575 $v Congrès. $0 (CaQQLa)201-0378219
650 _6 Détecteurs optiques $0 (CaQQLa)201-0022294 $v Congrès. $0 (CaQQLa)201-0378219
650 _6 Fusion multicapteurs $0 (CaQQLa)201-0266002 $v Congrès. $0 (CaQQLa)201-0378219
650 _7 Imaging systems. $2 fast $0 (OCoLC)fst00967605
650 _7 Multisensor data fusion. $2 fast $0 (OCoLC)fst01029095
650 _7 Optical detectors. $2 fast $0 (OCoLC)fst01046682
650 _7 Maschinelles Sehen $2 gnd $0 (DE-588)4129594-8
650 _7 Sensor $2 gnd $0 (DE-588)4038824-4
655 _7 Conference papers and proceedings. $2 fast $0 (OCoLC)fst01423772
655 _7 Kongress. $2 swd
700 1_ Loffeld, Otmar.
710 2_ Centre national de la recherche scientifique (France)
710 2_ Society of Photo-optical Instrumentation Engineers.
776 08 $i Online version: $t Vision systems--sensors, sensor systems, and components. $d Bellingham, Wash. : SPIE--the International Society for Optical Engineering, ©1996 $w (OCoLC)763140154
830 _0 Europto series.
830 _0 Proceedings of SPIE--the International Society for Optical Engineering ; $v v. 2784.

856 41 $u http://link.spie.org/PSISDG/2784/1 $z Rutgers restricted.
856 41 $u https://liverpool.idm.oclc.org/login?url=http://link.spie.org/PSISDG/2784/1 $z <img src="/screens/gifs/go4.gif" alt="Go button" border="0" width="21" height="21" hspace="7" align="middle"> View this e-book online
856 42 $u http://libanswers.liverpool.ac.uk/faq/182315 $z Reading advice (software, printing, accessibility, privacy)
938 Baker and Taylor $b BTCP $n 96067692
938 YBP Library Services $b YANK $n 1311455
029 1_ AU@ $b 000012855149
029 1_ DEBBG $b BV022107989
029 1_ NLGGC $b 154105236
029 1_ NLNZL $b 9918129950302836
029 1_ NZ1 $b 4811903
029 1_ ZWZ $b 026076977

Action Status Delete Holdings _ Export _ Label _ Produce _ Update Holdings _ Validate _

ATTACHMENT 5B

Paper

Select Language ▼

Translator Disclaimer

21 August 1996

Instruction systolic array in image processing applications

Manfred Schimmler, *Hans-Werner Lang*

Author Affiliations + ()

Proceedings Volume 2784, Vision Systems: Sensors, Sensor Systems, and Components; (/conference-proceedings-of-spie/2784.toc) (1996) https://doi.org/10.1117/12.248524 (https://doi.org/10.1117/12.248524)

Event: Lasers, Optics, and Vision for Productivity in Manufacturing I, 1996, Besancon, France

ARTICLE CITED BY ▾

Abstract

The ISATEC parallel computer is the first implementation of an instruction systolic array for the commercial market. The goal i\of integration of 1024 processors on an add-on-board for PCs has been achieved by the development of a low- power/low-area processor architecture whose instruction set is suited particularly for image processing applications. The paper introduces the concept of the instruction systolic array, its implementation and some application examples in the field of image processing.

© (1996) COPYRIGHT Society of Photo-Optical Instrumentation Engineers (SPIE). Downloading of the abstract is permitted for personal use only.

Citation Download Citation ▾

Manfred Schimmler and Hans-Werner Lang "Instruction systolic array in image processing applications", Proc. SPIE 2784, Vision Systems: Sensors, Sensor Systems, and Components, (21 August 1996); https://doi.org/10.1117/12.248524 (https://doi.org/10.1117/12.248524)

ACCESS THE FULL ARTICLE

PERSONAL SIGN IN

Full access may be available with your subscription

Email or Username

your username? (https://spie.org/account/forgotusername?tps%3a%2f%2fwww.spiedigitallibrary.org%2fconference-proceedings-of-f2784%2f0000%2fInstruction-systolic-array-in-image-processing-ions%2f10.1117%2f12.248524.short%3fSSO%3d1&webSyncID=0ba98cda-2a6-d6e4-40e0d866a2e7&sessionGUID=2b539394-9e54-5e1e-e8a3-c67733)

Password

your password? (https://spie.org/account/forgotpassword?tps%3a%2f%2fwww.spiedigitallibrary.org%2fconference-proceedings-of-

PURCHASE THIS CONTENT

SUBSCRIBE TO DIGITAL LIBRARY

50 downloads per 1-year subscription
Members: $195
Non-members: $335
ADD TO CART (/shoppingcart?fuseaction=cartadditem&productId=DI

25 downloads per 1 - year subscription
Members: $145
Non-members: $250
ADD TO CART (/shoppingcart?fuseaction=cartadditem&productId=DI

PURCHASE SINGLE ARTICLE

PROCEEDINGS
9 PAGES

DOWNLOAD PAPER

SAVE TO MY LIBRARY

SHARE | GET CITATION

CITATIONS
Cited by 15 scholarly publications.
Explore citations on Lens.org (https://www.lens.org/lens/scholar/article/002-849-741-169-192/main)

Advertisement

Advertisement

f2784%2f0000%2fInstruction-systolic-array-in-image-processing-
ions%2f10.1117%2f12.248524.short%3fSSO%3d1&webSyncID=0ba98cda-
a6-d6e4-40e0d866a2e7&sessionGUID=2b539394-9e54-5e1e-e8a3-
c67733)

Show

Keep me signed in

SIGN IN

No SPIE account? Create an account (https://spie.org/account/create/accountinfo?webSyncID=0ba98cda-1c9a-3ea6-d6e4-40e0d866a2e7&sessionGUID=2b539394-9e54-5e1e-e8a3-8683e2c67733)

Institutional Access:
Sign in with your institutional credentials (/Account/institutionalsignin?redirect=https%3a%2f%2fwww.spiedigitallibrary.org%2fconference-proceedings-of-spie%2f2784%2f0000%2fInstruction-systolic-array-in-image-processing-applications%2f10.1117%2f12.248524.short%3fSSO%3d1)

Includes PDF, HTML & Video, when available
Members: $17.00
Non-members: $21.00

ADD TO CART (/shoppingcart?urlId=10.1117%2f12.248524)

KEYWORDS

Image processing

Array processing

Clocks

Digital filtering

Computed tomography

Human-machine interfaces

Image filtering

Show All Keywords

RELATED CONTENT

Segmentation criteria in the problem of porosity determination based on... (/conference-proceedings-of-spie/11433/114331E/Segmentation-criteria-in-the-problem-of-porosity-determination-based-on/10.1117/12.2558081.full)
Proceedings of SPIE (January 31 2020)

VLSI Architectures For High Speed Two Dimensional State Space Recursive... (/conference-proceedings-of-spie/1001/0000/VLSI-Architectures-For-High-Speed-Two-Dimensional-State-Space-Recursive/10.1117/12.968973.full)
Proceedings of SPIE (October 25 1988)

Fama Architecture: Implementation Details (/conference-proceedings-of-spie/856/0000/Fama-Architecture-Implementation-Details/10.1117/12.943036.full)
Proceedings of SPIE (October 19 1987)

Expandable computed-tomography architecture for nondestructive inspection (/conference-proceedings-of-spie/1824/0000/Expandable-computed-tomography-architecture-for-nondestructive-inspection/10.1117/12.142903.full)
Proceedings of SPIE (April 01 1993)

Real-time stereovision tools on a SIMD processor (/conference-proceedings-of-spie/2955/0000/Real-time-stereovision-tools-on-a-SIMD-

processor/10.1117/12.262894.full)
Proceedings of SPIE (December 17 1996)

New systolic array architecture for vector median filters (/conference-proceedings-of-spie/3653/0000/New-systolic-array-architecture-for-vector-median-filters/10.1117/12.334742.full)
Proceedings of SPIE (December 28 1998)

✓ Subscribe to Digital Library (/subscribe-page)

! Receive Erratum Email Alert ()

ATTACHMENT 5C

All locations : No holdings in LDL - 68 other holdings

Vision systems--sensors, sensor systems, and components : 10-12 June 1996, Besançon, France / Otmar Loffeld, chair/editor ; sponsored by CNRS--Centre National de la Recherche Scientifique [and others]. Bellingham, Wash. : SPIE--the International Society for Optical Engineering, ©1996. v, 154 pages : illustrations ; 28 cm. OCLC:35521098

Holdings Information

STATE	LOCATIONS
AL	mwf MWR
AZ	AZS
CA	stf ZAS
CO	COA COD
DC	DLC
FL	FTU
GA	GAT
IL	SOI
MA	AUM MYG
MD	ADB nag UMC
MI	a3v EYM EYW
MO	LHL
NC	btcta NDD NRC
NH	ydx
NJ	NJR PUL
NM	IQU snx
NY	BUF COO YRM ZCU
OH	CIN
PA	UPM
TX	RCE ryt tim TXJLF
VA	NAT VPI
WA	NTE WAU XFF
WI	GZM
AU	atutq AUT C7F he1 YQU
BC	ubc
ON	cai CNECS cnuto YOU
DE	dekab deudb tub
FR	fph uy1
GB	BRI cud LVT
HK	HNK HUA
NL	GRU NLTUD
NZ	NZ1 UV0

CLOSE WINDOW

ATTACHMENT 6A

ISCA

ParallAX: an architecture for real-time physics

Authors: Thomas Y. Yeh, Petros Faloutsos, Sanjay J. Patel, Glenn Reinman Authors Info & Claims

ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture • June 2007 • Pages 232–243 • https://doi-org.libproxy.unl.edu/10.1145/1250662.1250691

Published: 09 June 2007 Publication History

31 1,201

eReader PDF

Feedback

ABSTRACT

Future interactive entertainment applications will featurethe physical simulation of thousands of interacting objectsusing explosions, breakable objects, and cloth effects. Whilethese applications require a tremendous amount of performanceto satisfy the minimum frame rate of 30 FPS, there is a dramatic amount of parallelism in future physics workloads.How will future physics architectures leverage parallelismto achieve the real-time constraint?.

We propose and characterize a set of forward-looking benchmarksto represent future physics load and explore the designspace of future physics processors. In response to thedemand of this workload, we demonstrate an architecturewith a set of powerful cores and caches to provide performancefor the serial and coarse-grain parallel components ofphysics simulation, along with a

Furthermore,we perform sensitivity analysis on interconnectalternatives to determine how tightly to couple these cores.

ISCA

References

1. HyperTransport 3.0 Spec. http://www.hypertransport.org/docs/tech/HT3pres.pdf.

2. PCIE Express Spec. http://www.pcisig.com/specifications/pciexpress/.

3. AGEIA. Physx product overview. www.ageia.com.

Show All References

Feedback

Index Terms

ParallAX: an architecture for real-time physics

Computer systems organization

- Architectures
  - Parallel architectures
    - Multiple instruction, multiple data
- Embedded and cyber-physical systems
- Real-time systems

DL Comment Policy

Comments should be relevant to the contents of this



ISCA

0 Comments

Sort by Newest

Nothing in this discussion yet.

View Table Of Contents

Feedback

Categories

Journals
Magazines
Books
Proceedings
SIGs
Conferences
Collections
People

About

About ACM Digital Library
ACM Digital Library Board
Subscription Information
Author Guidelines
Using ACM Digital Library
All Holdings within the ACM Digital Library
ACM Computing Classification System
Digital Library Accessibility

Join

Join ACM
Join SIGs
Subscribe to Publications
Institutions and Libraries

Connect

Contact
Facebook
Twitter
Linkedin
Feedback

The ACM Digital Library is published by the Association for Computing Machinery. Copyright © 2022 ACM, Inc.

Terms of Usage | Privacy Policy | Code of Ethics

ISCA

LIBRARY Computing Machinery

Feedback

ATTACHMENT 6B

ParallAX: An Architecture for Real-Time Physics

Thomas Y. Yeh† Petros Faloutsos† Sanjay J. Patel‡ Glenn Reinman†
†Computer Science Department, UCLA, {tomyeh, pfal, reinman}@cs.ucla.edu
‡AGEIA Technologies, sjp@ageia.com

ABSTRACT

Future interactive entertainment applications will feature the physical simulation of thousands of interacting objects using explosions, breakable objects, and cloth effects. While these applications require a tremendous amount of performance to satisfy the minimum frame rate of 30 FPS, there is a dramatic amount of parallelism in future physics workloads. How will future physics architectures leverage parallelism to achieve the real-time constraint?

We propose and characterize a set of forward-looking benchmarks to represent future physics load and explore the design space of future physics processors. In response to the demand of this workload, we demonstrate an architecture with a set of powerful cores and caches to provide performance for the serial and coarse-grain parallel components of physics simulation, along with a flexible set of simple cores to exploit fine-grain parallelism. Our architecture combines intelligent, application-aware L2 management with dynamic coupling/allocation of simple cores to complex cores. Furthermore, we perform sensitivity analysis on interconnect alternatives to determine how tightly to couple these cores.

Categories and Subject Descriptors:
C.1 [Processor Architectures]: Multiple Data Stream Architectures C.3 [Special-Purpose and Application-based Systems]: Real-time and Embedded Systems

General Terms: Design, Performance.

Keywords: Application Specific Processor, Real-time Physics, Physics Based Animation, Chip Multiprocessor, Stream Processing, Interactive Entertainment.

1. INTRODUCTION AND MOTIVATION

Interactive entertainment (IE) applications demand high performance from all architectural components. In the future, this demand will increase even further. These applications will be composed of a diverse set of computationally demanding tasks. One critical task is modeling how objects and characters move and interact in a virtual environment. Most current IE applications make use of prerecorded motion clips to synthesize the motion of virtual objects (kinematics), but as these objects and their interactions scale up in complexity, pre-recording has become impractical. *Physics-based simulation* has emerged as an attractive alternative to kinematics, providing high levels of physical realism through motion calculation. Future applications targeting a true immersive experience will demand this as a cornerstone of their design.

The benefits of physics-based simulation come with a considerably higher computational cost. To maintain a fluid visual experience, IE applications typically provide at least a 30 frames per second (FPS) display rate (33ms per frame). This is the total time allotted to all game components, including physics-based simulation, graphics display, AI, and game engine code. Conventional cores cannot meet the computational demands of these applications. For example, in Section 6 we show a realistic example where a single-core desktop processor achieves only 2.3 FPS.

The difficulty in meeting the performance demands of physics-based simulation is somewhat mitigated by the high degree of parallelism available in these applications. Our results show that on average 91% of a physics workload can be broken down into parallel subtasks of varying granularities. Although only 9% of the workload is serialized, on a single desktop core this portion can take up to 125% of the available frame time. This argues for an architecture that has sufficient performance to tackle the serial tasks, but also has the flexibility to fully exploit parallel regions. There are existing designs that combine some number of large coarse-granularity (CG) cores with a larger number of fine-granularity (FG) cores, including GPUs connected to a host CPU through a system bus [20], the Cell's SPE's paired with its PPE [10], and a conventional core paired with multiple vector units [14]. However, these designs lack flexibility in how their cores can be utilized, making it difficult for them to *efficiently* meet the demands of real-time physics. This paper proposes a more flexible design that is optimized to meet these demands, based on a design space exploration that includes the number and type of cores required, the amount of cache state required, and the interconnect required between these components.

In this paper, we make the following contributions:

- We devise a forward-looking benchmark suite for interactive, physical simulation. The suite combines rigid body, cloth, debris and other features at a level of scale commensurate with future-generation games.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'07, June 9–13, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-706-3/07/0006 ...$5.00.

- We show that current multi-core architectures will not be able to sustain interactive frame rates even when the benchmark suite is aggressively parallelized. Operating system, cache contention, and control logic area overhead all contribute to this conclusion.
- We propose an architecture with both CG and FG cores that is able to sustain interactive frame rates for physics workloads through efficient area utilization. The key elements of this efficiency are:
  - *Intelligent, application-aware L2 management* – In section 6.1 we examine the L2 requirements for physics simulation and propose a partitioning strategy that reduces the required L2 space by more than half.
  - *Dynamic coupling/allocation of FG cores to CG cores* – In section 7.1 we propose an arbitration policy that balances the maximal utilization of available FG core resources and the exploitation of locality among FG cores working on the same CG task.
  - *Relaxed communication latency of FG and CG cores*– In sections 7.2 and 8.2.2 we explore design alternatives to interconnecting FG and CG cores. The tight coupling of FG and CG cores can restrict where these cores are placed (i.e. on/off chip) and how effectively we can dynamically leverage FG cores. To loosen this coupling, we consider the amount of buffering space and application parallelism required to overlap communication for a variety of interconnection strategies.

The rest of this paper is organized as follows. In section 2 we discuss the related work. Section 3 describes our physics engine and the associated computational load. In section 4 we propose a set of future-thinking benchmarks that represent a wide range of physical actions and entertainment scenarios. Section 5 details the experimental setup. Section 6 explores the performance of this suite on conventional architectures and threading methodologies. In section 7 we outline our proposed physics architecture, and in section 8 we explore its design space. We conclude in section 9.

2. RELATED WORK

RIKEN's MDGRAPE-3 [26] and AGEIA's PhysX [3] are currently the only dedicated physics simulation accelerator designs. While MD-GRAPE targets computational physics with limited programmability, PhysX targets real-time physics for games. Both designs are accelerator boards which connect to the host CPU through a system bus.

Two other closely related bodies of prior work are *vector processing* [9] and *stream computation* [15, 12]. VIRAM[13] has achieved an order of magnitude performance improvement on certain multimedia benchmarks. However, conventional vector architectures are constrained [14] by limitations such as a centralized register file, precise exception handling, and an expensive on-chip memory system. While CODE is scalable, the data shows a plateau at eight clusters with eight lanes, and the cacheless CODE can not satisfy our measured physics workload.

Stream architectures (SAs) aim to enable ASIC-like performance efficiency with high-level language programmability. Stream programs express computation as a signal flow graph with streams of records flowing between computation kernels. While a broad range of designs populate this space, the high-level characteristics of SAs are described in [15]. The Stream Virtual Machine (SVM) architecture model logically consists of three execution engines (control, kernel, and DMA) and three storage structures (local registers, local memory, and global memory). Related designs in this space include IBM's Cell, GPUs, and the Xbox360 system.

Heterogenous CMP designs such as the Cell target *task* specific performance using different cores. However, according to our exploration, both the PPE and SPE designs are not optimal for physics computation. This may be a result of the Cell being designed to execute *all* components of a game, not just physics simulation.

The Graphics Processing Unit (GPU) [20] is another streaming architecture design. GPUs are specialized hardware cores designed to accelerate rendering and display. While Havok's FX allows effect physics simulation on GPUs, GPUs are designed to maximize throughput from the graphics card to the display – data that enters the pipeline and the results of intermediate computations cannot be easily accessed by the CPU. With the host CPU connected to the GPU via a system bus, the long communication latency is problematic for physics simulation working in a continuous feedback loop. This limitation is alleviated in the Xbox360 system [4], which combines a 3-core CMP and GPU shaders. This system allows the GPU to read from the FSB rather than main memory and L2 data compression reduces the required bandwidth.

In terms of physics engine software, both Ageia [3] and Havok [8] provide proprietary SDKs. Open Dynamics Engine (ODE) [7] is the most popular open-source alternative. It has been used in commercial settings, and provides APIs and numerical techniques similar in nature to proprietary engines. ODE is the basis of our physics engine.

3. PHYSICS SIMULATION WORKLOAD

In this section, we discuss physics simulation and identify its main computational phases.

3.1 Our Physics Engine

Our physics engine is a heavily modified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [7]. ODE follows a constraint-based approach for modeling articulated figures, similar to [5, 18], and it is designed for efficiency rather than accuracy. Our implementation supports more complex physical functions, including cloth simulation, pre-fractured objects, and explosions. We have parallelized it using pthreads and a work-queue model with persistent worker threads. Pthreads minimize thread overhead, while persistent threads eliminate thread creation and destruction costs.

The following is the high-level algorithmic flow of ODE, augmented with our changes (shown in italics).

1. Create and setup a dynamics world.
2. While ($time < time_{end}$)
   (a) Apply forces to the objects as necessary (e.g. gravity).
   (b) Calculate all pairs of objects that are in contact.
   (c) For each pair of objects in contact do the following:
      i. Compute the contact points and create the associated contact constraints (joints).

Attachment - Page A147

ii. *If an explosive object makes contact with another object, create a sphere representing blast radius.*

iii. *If a body makes contact with a cloth's bounding volume, insert body on cloth's contact list.*

iv. *If a pre-fractured object is in contact with a blast volume (sphere) break object into debris.*

(d) Form groups (islands) of objects interconnected with joints, i.e. find the connected components.

(e) Forward simulation step: For each island compute the applied loads and the new positions/velocities of each object.

(f) Check all breakable joints: if joint's applied load has exceeded a threshold, break the joint.

(g) *Process all cloth objects by taking a forward simulation step.*

(h) Advance the time: $time = time + \Delta t$

3. End.

At the core (forward step) of the simulation loop lies the constraint solver, which typically uses an iterative relaxation method. There are two key parameters to trade off accuracy for efficiency, the time-step Δt and the number of iterations n. Δt defines the amount of simulated time that separates successive executions of the simulation loop and defines the number of executed loops per simulated second. n controls how many relaxation steps the solver takes in a single step.

For our benchmarks, Δt is 0.01 seconds and 3 steps are executed per frame to ensure stability and prevent fast objects from passing through other objects. We use a n of 20 as recommended by [7].

3.2 Computation Phases of Physics Workload

The steps in the above algorithm form a dataflow of computational phases shown in Figure 1. All phases are serialized with respect to each other, but **unshaded** stages can exploit parallelism within the stage. Simulations such as cloth and fluid are specializations of this. Because cloth simulation presents a conceptually and numerically different load than rigid-body simulation, we considered it a separate phase. Below we describe the 5 computational phases in detail. We will use the terms *phase* and *task* interchangeably.

Broad-phase. This is the first step of *Collision Detection* (CD). Using approximate bounding volumes, it efficiently culls away pairs of objects that cannot possibly collide. While *Broadphase* does not have to be serialized, the most useful algorithms are those that update a spatial representation of the dynamic objects in a scene. And updating these spatial structures (hash tables, kd-trees, sweep-and-prune axes) is not easily mapped to parallel architectures.

Narrow-phase. This is the second step of CD that determines the contact points between each pair of colliding objects. Each pair's computational load depends on the geometric properties of the objects involved. The overall performance is affected by broad-phase's ability to minimize the number of pairs considered in this phase. This phase exhibits massive fine-grain (FG) parallelism since object-pairs are independent of each other. Based on the number of worker threads, we partition the object-pairs into equal sets.

Island Creation. After generating the contact joints linking interacting objects together, the engine serially steps through the list of all objects to create islands (connected components) of interacting objects. This phase is serializing in the sense that it must be completed before the next phase can begin. The full topology of the contacts isn't known until the last pair is examined by the algorithm, and only then can the constraint solvers begin. Practical techniques for parallel island generation are not commonly available.

Island Processing. For each island, given the applied forces and torques, the engine computes the resulting accelerations and integrates them to compute the new position and velocity of each object. This phase exhibits both coarse-grain (CG) and fine-grain (FG) parallelism. Each island is independent, and the constraint solver for each island contains independent iterations of work. We parallelized the engine at both granularities. Only islands with more than 25 degrees-of-freedom removed are inserted into the work-queue – smaller islands execute on the main thread.

Cloth Simulation. We have implemented cloth largely based on Jakobsen's position-based approach[11]. An extension of this approach that handles more general constraints has been proposed by AGEIA[18]. A cloth object is represented using a triangular mesh where each edge represents a length constraint. The constraints are solved using an iterative constraint relaxation solver and the mesh is simulated forward in time using a Verlet integrator. Collision detection is based on a combination of ray casting and axis-aligned bounding volume hierarchies. Collision resolution is based on a vertex projection scheme. This phase also exhibits both CG and FG parallelism. Each cloth object is independent, and the integrator contains independent tasks for each vertex in the cloth object. We parallelized the engine at both the object and vertex levels.

As we will demonstrate in the rest of this paper, physics simulation differs from conventional workloads (i.e. SPEC and multimedia) in the following ways:

- Concrete performance goals - a real-time constraint of at least 30 FPS.
- Discrete phases of the application - with very different levels of parallelism and architectural requirements.
- Tightly coupled application phases - these phases are serial with respect to one another and feature a feedback loop not present in applications such as rendering.

4. FUTURE PHYSICS BENCHMARKS

We propose a comprehensive set of benchmarks to represent the complexity and scale of physics simulation in future game-scenarios. Our benchmark suite can be leveraged by: (1) computer architects/researchers to explore real-time physics hardware/software designs, and (2) application designers to determine gaming platform performance bounds.

Our benchmark design was guided by high-level physical actions and representative scenarios from different game genres. Benchmarks are constructed with parameters for problem-size scaling, and we use the set of features demonstrated in Table 2. Table 3 explains the benchmarks while Table 4 provides some statistical data.

Game physics data scaling has roughly tracked Moore's Law along an exponential path as developers load as much physics onto a game as the minimum-spec system could handle. This trend can be illustrated by three popular games: (a) Unreal Tournament 2003, (b) Unreal Tournament 2004, and (c) Gears of War 2006. From discussion with industry insiders, the estimated number of rigid objects in these games are 75, 100, and 200 respectively. That means that the number of rigid bodies increased by 33% between 2003 and 2004 and by 100% between 2004 and 2006.

Past benchmarks [27] are limited to small scale rigid-body interactions. Our suite has dramatically increased the types

of simulations and number of interacting objects. Based on information from Ageia, future games in development (like Cell Factor Revolution, Auto Assault, and Stoked Rider) targeting the PhysX card use 1000-10,000s of rigid body objects, 10,000s of particles, and 1000s of vertices for deformable meshes (cloth). In contrast, physics targeting a dual-core desktop processor tops out at 500 rigid bodies, 1000s of particles, and no deformable meshes.

5. EXPERIMENTAL SETUP

We used a Simics-based [16] full-system execution-driven simulator. Both the cache and processor timing simulators are from the GEMS toolset [17]. All L2 caches are based on 1MB 4-way banks, and Table 1 shows the configuration parameters used for the coarse-grain cores in our simulations. Fine-grain cores are described in section 8.

Processor Pipeline: 4-wide, 14-stages	Functional Units: 4 int, 2 fp, 2 ld/st
Window/Scheduler: 96, 32 entries	Branch Predictor: 17KB YAGS + 64-entry RAS
Block size: 64 bytes	L1 I/D caches: 32KB, 4-way, 2-cycle
L2 cache/banks: 15-cycle	On-chip network: Point-to-point, 2-cycle per hop
Main Memory: 340 cycles	Clock Frequency: 2GHz

Table 1: Coarse-grain Core Design

All benchmarks are executed for 3 frames of physics simulation due to very long simulation times – the average instruction per frame is shown in table 3. Some benchmarks require more than 4 days to complete a single frame when simulating a 4-core design. The simulation proceeds with 0.01 second time steps. The benchmarks are setup with most activity occuring in the first 10 frames. The frames 5-7 are executed, and the frame with longest execution time is chosen. All benchmarks are warmed up for one physics simulation step prior to execution of the selected frames. The performance target is 30 frames-per-second (FPS).

The cores simulated in Simics are SPARC ISA processors running Solaris 10 in single user mode. All binaries for performance simulation do not include graphical display code and are compiled by gcc 4.1.1 for the SPARC ISA using the following flags: [-mcpu=v9 -mtune=ultrasparc3 -mvis -m32 -pthreads -O2]. For visual verification, we compile a separate set of binaries with visual display code for the x86 ISA. All instrumentation for separating computation phases uses Simics' MAGIC instruction, which is not counted when calculating execution time.

5.1 Interconnect Models

For the on-chip 2D mesh interconnect, we used the data in Table I from [24] for 90 nm technology. The per hop delay is 1 cycle, and the router pipeline is 5 cycles (2GHz clock). This network uses 64-bit flits, and four virtual channels can simultaneously send data. We assume the packet header to be 8-bit long, so each packet's payload is 56-bit.

For the off-chip configurations, we assume the use of either Hypertransport (HTX) [1] or PCI Express (PCIe) [2] to connect the discrete chips. On the fine-grain chip, the same 2D mesh described above connects all cores to the I/O.

6. PERFORMANCE DEMANDS

This section examines physics simulation's performance demand by evaluating single-core performance, the per phase working set, and the limits of coarse-grain (CG) parallelism.

Figure 2(a) shows the total execution time, broken down into per phase contributions, of one frame for single-threaded benchmarks running on a 2GHz single-core desktop-class processor with a 1MB L2 cache. The distribution of execution times shows good complexity scaling ranging from *Periodic* to *Mix*. Only two benchmarks, *Periodic* and *Ragdoll* satisfies the 30 FPS requirement. The most complicated benchmark *Mix* requires over an order of magnitude performance improvement to reach 30 FPS.

While *Broadphase* and *Island Creation* (the difficult to parallelize phases - i.e. serial phases) make up only an average 9% of total execution time, they can still take up to 125% of one frame's worth of time (i.e. 1/30th of a second). This data forces us to optimize both the serial and parallel components of this workload. The instruction mix in figure 7(b) shows that serial phases and *Narrowphase* are integer dominant with large amount of branches while parallel phases *Island Processing* and *Cloth* are FP dominant.

To determine the hardware requirement of future physics applications, we first consider the working set for both the serial and parallel portions. Then we explore the performance impact of exploiting CG thread-level parallelism (TLP).

6.1 Working Set Analysis

After evaluating different core designs and L2 configurations, we found the L2 cache design to be the dominant factor affecting the serial phase performance. Figure 2 (b) shows execution time for the serial phases as the L2 cache is scaled from 1MB to 32MB by adding 4-way 1MB banks. The banks are configured with a point-to-point network and use a 2-level directory based MOESI coherency protocol [17].

Most misses are determined to be capacity misses by using 1024-way 1MB banks. A minimum of 4MB of L2 cache is required to complete the serial phases within a frame's worth of time, and maximum performance requires a fully-associative 16MB or a more realistic 32MB L2. These sizes seem relatively large when we consider the number of objects in these simulations. A majority of L2 misses occur inside the parallel phases, pointing to the possibility that the parallel phase data evicts the serial phase data between simulation steps. To illustrate this, we examine each phase's L2 effective working set size by saving the cache state at the end of a phase and loading this cache state at the beginning of the next step for the same phase. Figures 3 (a) and 4 (a) show that the performance for both serial stages plateaus at 4MB, and the performance is within 7% of a 16MB L2 used by all stages on Figure 2 (b).

Figures 3(b), 4(b), and 5(a) show the isolated L2 scaling performance data for *Narrowphase*, *Island Processing*, and *Cloth* respectively. Both *Island Processing* and *Cloth* are relatively insensitive to L2 cache scaling. For *Narrowphase*, the benchmarks *Explosions* and *Highspeed* show roughly 2X improvement when scaling the L2 from 1MB to 16MB. This behavior can be attributed to the large number of object-pairs shown in Table 4. The increased amount of data required for *Explosions* and *Highspeed* results in their higher sensitivity to L2 cache size.

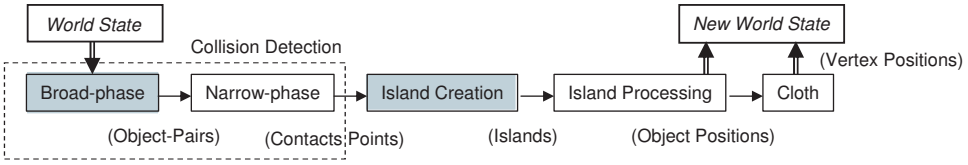


Figure 1: Physics Engine Flow.

Constrained Rigid Bodies	Simulation of articulated objects connected with ideal joints. Virtual humans consist of 16 segments of anthropomorphic dimensions. Cars have a body, rotating wheels, and a suspension system of slider joints.
Terrains	Uneven surfaces described by heightfields or trimeshes.
Breakable Joints	Joints are broken by accumulation of force or a single strong force exceeding a predetermined threshold. Bridges, cars, and robots contain breakable joints.
Prefractured Objects	Each breakable object contains a set amount of debris that can break apart from the object. The object and geom representing each piece of debris is created at startup time and enabled once the object breaks.
Explosions	Each object is marked with an explosive flag. If an explosive object makes contact with any other object, the explosive object is replaced by a sphere representing the blast radius with predetermined duration.
Static Obstacles	Immobile objects that moving objects can come in contact with. They do not participate in forward stepping since they do not move but they do participate in collision detection.
Cloth Simulation	Softbody modeling using constrained vertices that approximate a continuous surface. Large cloth objects use 625 vertices to simulate drapery or netting. Small ones, typically attached to virtual humans, use 25 vertices.

Table 2: Features Found in Our Benchmarks

Benchmark	Avg Inst/Frame	Description
Periodic	34 million	Role-playing scenario with 30 humanoids engaging in hand-to-hand combat.
Ragdoll	36 million	First-person shooter (FPS) scenario with 30 humanoids falling due to projectile impact.
Continuous	47 million	Racing scenario with 30 cars driving on terrain and between obstacles.
Breakable	256 million	FPS scenario with cannons shooting and bombs exploding in 3 areas enclosed by walls.
Deformable	409 million	Sports/Action scenario with 30 uniformed players and 2 large cloth objects.
Explosions	547 million	Real-time strategy scenario with an army fighting in an urban environment with explosives.
Highspeed	518 million	Action scenario with 20 cars crashing and non-explosive high-speed rockets destroying 10 buildings.
Mix	829 million	A combination of all the features/entities used in the previous 7 benchmarks. There are 3 buildings, 6 bridges, 30 uniformed humanoids and 6 vehicles.

Table 3: Our Physics Benchmarking Suite

Benchmark	Obj-Pairs	Islands	Cloth Objs [vertices]	Static Objs	Dynamic Objs	Prefractured Objs	Static Joints
Per	2,633	99	0	0	480	0	480
Rag	2,064	30	0	0	480	0	480
Con	3,182	37	0	1,700	650	0	120
Bre	11,715	97	0	0	1,608	5,652	564
Def	7,871	89	32 [2000]	480	480	0	480
Exp	21,986	58	0	0	3,459	0	200
Hig	21,041	12	0	0	3,309	0	80
Mix	16,367	28	33 [2625]	0	1,608	5,652	564

Table 4: Benchmark Specs

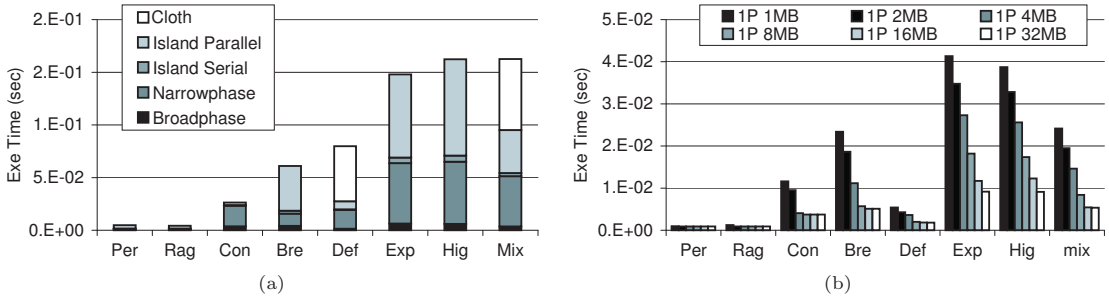


Figure 2: (a) Execution Time Breakdown of 1 Core + 1MB L2 — (b) Single Core Execution of Serial Parts

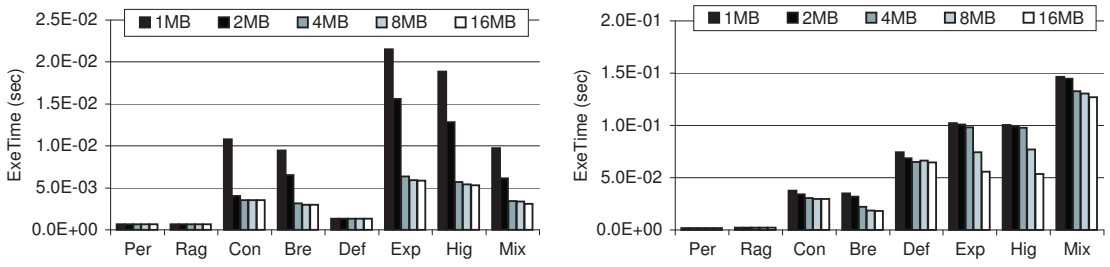


Figure 3: (a) Broadphase Performance [dedicated L2] — (b) Narrowphase Performance [dedicated L2]

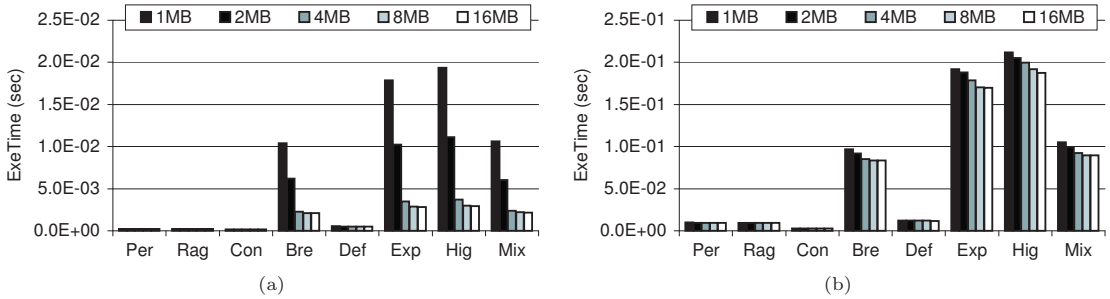


Figure 4: (a) Island Creation Performance [dedicated L2] — (b) Island Processing Performance [dedicated L2]

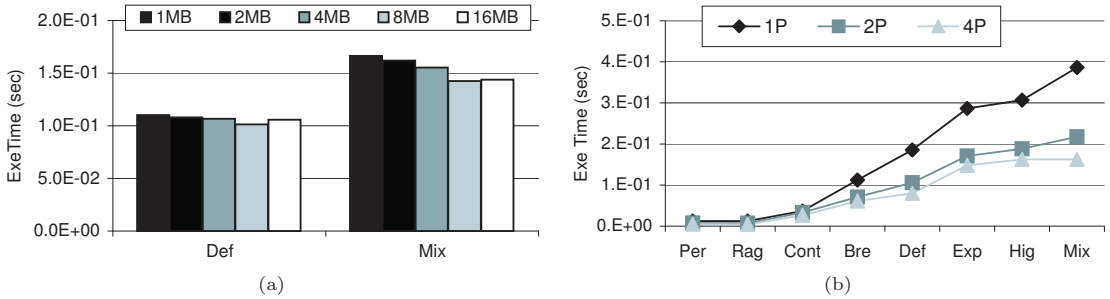


Figure 5: (a) Narrowphase Performance [dedicated L2] — (b) Performance with Processor Scaling

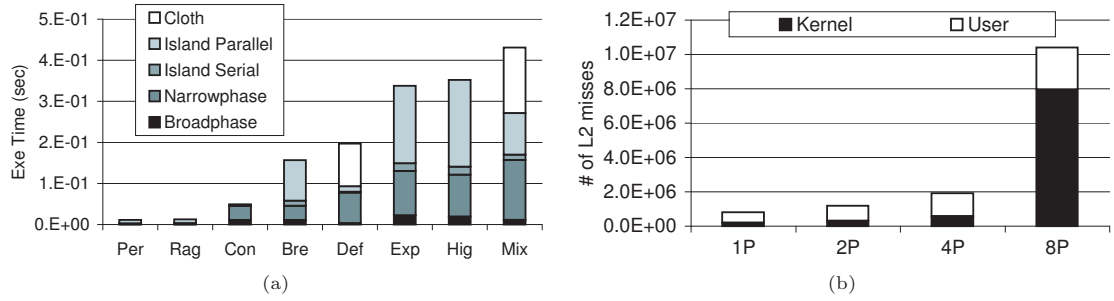


Figure 6: (a) Execution Time Breakdown (4 Core + 12MB L2) — (b) L2 Miss with Thread Scaling

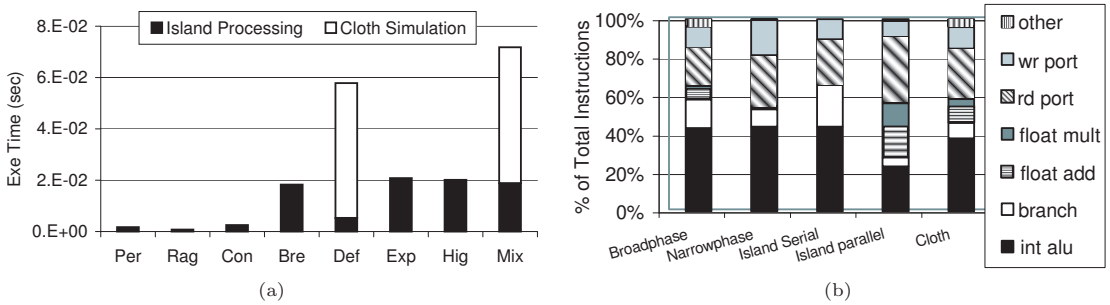


Figure 7: (a) Limit of Coarse-grain Parallelism — (b) Instruction Mix for all 5 Phases

With dedicated cache space for each computation phase, the L2 cache requirement has been significantly reduced. The serial stages now each require 4MB of L2 cache space. Because *Broadphase* uses shape data (geom) for collision detection and *Island Creation* uses object and joint data to create islands, there is little data sharing between these two phases. The memory required per object and geom is 412B and 116B respectively. The memory required per joint varies between 148B to 392B depending on the type. To obtain the performance shown on Figures 3(a) and 4(a), we allocate 8MB of L2 with 4MB dedicated to each serial phase.

When executing the parallel phases in single-thread mode, 1MB of additional L2 space is sufficient to obtain the bulk of the performance. This 1MB can be shared between all three phases since there will be sharing between *Broadphase* and *Narrowphase* and between *Island Creation* and *Island Processing*. The cache space dedicated to the serial phases should be read-only during parallel phases.

However, the cache requirement for parallel phases changes with the number of simultaneously active threads. To better understand this, we vary the number of processor cores that are available to exploit coarse grain parallelism along with the total L2 space. While *Cloth* continues to be insensitive to L2 cache size, both *Narrowphase* and *Island Processing* see an interesting shift in L2 sensitivity. At two threads, these phases improve performance with the same dedicated L2 cache because of the increased TLP. However, at four threads, thrashing between threads grows considerably and drives demand for L2 space higher. For *Island Processing*, 1MB of dedicated L2 cache degrades performance when scaling from 1 to 4 cores. The memory requirement per island depends on the number of joints, bodies, and degrees-of-freedom removed. To satisfy this demand, we allocate an additional 4MB of L2 cache space for the parallel phases in the rest of our experiments.

6.2 Exploiting Coarse-Grain Parallelism

Figure 5 (b) shows the performance as we scale up the number of cores per processor. With every additional core, we add an additional worker thread. The L2 cache space is allocated at a phase granularity according to the earlier observations: (12MB total – organized into three 4MB partitions [25, 21, 28]: one for *Broadphase*, one for *Island Creation*, and one for the parallel sections). Phase identification for physics simulation is trivial as shown by the sequential flow of Figure 1. Depending on the computation phase, different parts of the shared L2 are used for writing data. Future work will examine L2 cache size reduction by prefetching, per-thread cache management, and DMAs.

The L2 is composed of 1MB 4-way cache banks, and the partitioning granularity is done per cache way [6]. Because serial stages are more sensitive to load latency, the 4MB cache partitions for the serial stages are allocated near the CG core used for serial execution. To attain minimal L2 latency for each serial phase, the main thread will execute *Broadphase* on one CG core and *Island Creation* on another.

Figure 5 (b) clearly shows that the performance improvement starts to plateau at 4 cores. On average, scaling from one to two cores improves performance by 53% and scaling from two to four cores improves performance by 29%. Figure 6 (a) shows the execution time breakdown for a four core processor with 12MB total L2 cache space. The performance has improved by roughly 3X, but we still need an additional 5X improvement to satisfy all benchmarks. Performance starts to degrade at eight cores (not shown). This degradation is surprising, but can be attributed to two main causes: an increased working set in the L2 cache and operating system overhead. For all parallel phases, additional threads consume more cache space by simultaneously accessing data that had not needed to co-exist in the cache. The large increase in L2 misses is shown in Figure 6 (b).

Scaling from four to eight threads results in a 5X increase in L2 misses. Kernel memory accesses inside *Island Processing* and *Cloth* make up most of the increase. The *pmap* command in Solaris 10 shows that each worker thread uses approximately 850KB of memory during two and four threaded execution. At eight threads, each worker thread's memory usage jumps to around 5MB. This operating system effect is being investigated further as future work.

Assuming the use of a custom-tuned operating system and programmer defined memory mapping/management, CG scaling may continue past four threads. However, even under ideal conditions (removal of OS overhead and cache contention, unlimited number of cores, and ideal load balancing), CG parallelism will not be sufficient to achieve interactive frame-rates. Figure 7(a) shows that *Mix* and *Deformable* require more than a frame's worth of time just for *Island Processing* and *Cloth*. When considering serial phases, *Explosions* and *Highspeed* barely achieve 30 FPS.

There are two fundamental limitations to CG scaling: (1) conventional threading overhead (synchronization and increased working set) and (2) a poor datapath logic to control logic ratio. Due to conventional threading overhead, island processing is parallelized at the per island level and cloth simulation is parallelized at the cloth level. CG performance scaling is thus bounded by the largest island and cloth. Furthermore, the area ratio of control logic vs datapath logic increases with core complexity. The use of complex cores for physics acceleration will not be scalable as IE applications evolve with more features and complexity. Given the tremendous amount of computation bandwidth required, it will be shown in section 8 that the use of complex cores results in prohibitively large die areas.

7. PARALLAX ARCHITECTURE

Figure 8 demonstrates two potential implementations of our proposed architecture, ParallAX. In both implementations, we have a set of *coarse-grain (CG)* cores handling tasks like serial phase computation, parallel task distribution, memory allocation, and the components of the parallel phases which do not have large amounts of parallelism. A larger pool of *fine-grain (FG)* cores handle the massively parallel components of the computation. The key contributions here are (1) the flexible arbitration policy that provides area-efficient utilization of available core resources (section 7.1), (2) the L2 partitioning strategy that has already been addressed in section 6.1, and (3) an exploration of interconnect sensitivity (sections 7.2 and 8.2.2). The two models of Figure 8 differ in whether they treat the CG cores as auxiliary processing engines or as the main computational core for a given system – this will be explored further in the third component of our key contributions. In the rest of this section we provide more details on our architecture, and then perform a design space exploration in the next section.

Both sets of cores (CG and FG) are connected by 2D meshes. The interconnection between these two sets can be

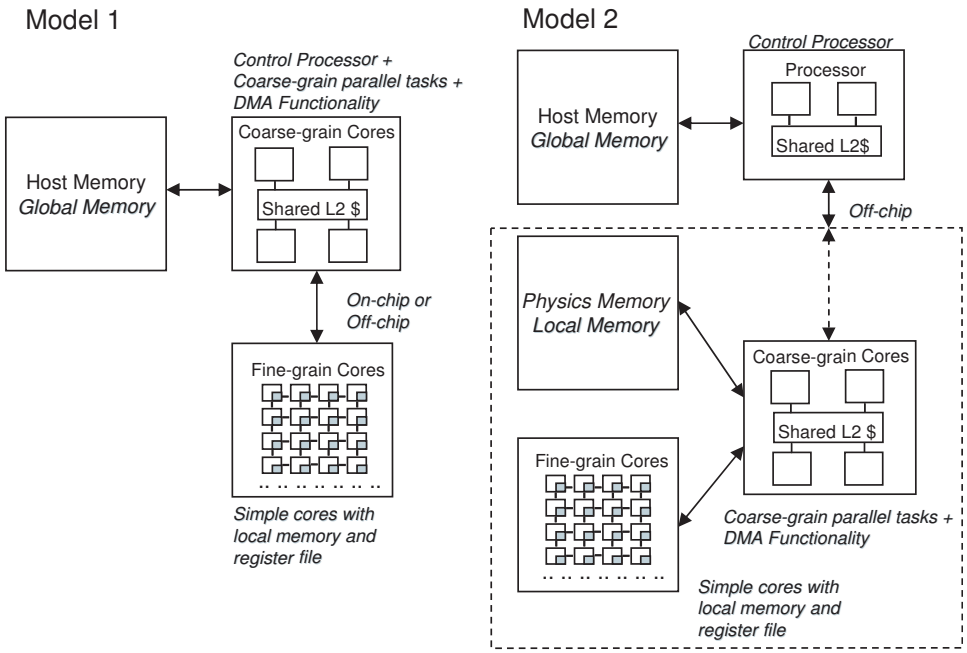


Figure 8: ParallAX - Parallel Physics Accelerator

on-chip or off-chip. The CG cores communicate through the shared L2-cache with banks connected via a 2D mesh. The FG cores are connected via the on-core routers.

In terms of real-time physics tasks, CG cores handle *Broad-phase* execution, providing a set of object-pairs to feed into *Narrowphase* execution. *Narrowphase* has considerable parallelism, and the CG cores start to process individual sets of object-pairs (CG parallelism). Object-pairs are then be farmed out to the FG cores by the CG cores – CG cores move both instructions and data into the FG cores.

FG parallelism refers to breaking computation down into:

- per object-pair for narrowphase collision detection
- per degree-of-freedom removed for LCP solver of island processing
- per vertex in cloth simulation

These granularities were selected to extract loosely coupled data parallelism. Our approach is to design a scalable system by using a large number of area-efficient processing elements. The selected granularities strike a good balance between the lowest level of parallelism (ILP) vs. completely decoupled parallelism (TLP). We refer the reader back to Table 4 and Figure 11 for information on the number of such FG tasks.

FG cores use local instruction and data memories instead of caches, since the instructions and data required for each task are easily determined. Each FG core iterates through all the tasks assigned to it, and no communication is required between these FG cores as there are no data dependencies between iterations. All FG cores execute the same kernel at one time due to the dependency between phases of physics computation. This argues for one pool of shared FG resources for all kernels instead of unique cores dedicated to each kernel. Although this observation is similar to unified shader design [19], physics computation is inherently different from graphics where little pipelining of the phases described in section 3 can occur. There is additional overhead involved in pushing the data to the FG cores and retrieving the data back after computation completes.

7.1 Mapping Fine-Grain to Coarse-Grain Cores

There are a number of design decisions to be made in this architecture. The first is how CG cores should map to FG cores. One alternative would be to statically map some set of FG cores to a single CG core, simplifying task scheduling, but possibly underutilizing our pool of FG cores in cases where one large task dominates the computation – the limiting scenario for CG parallelism. Instead, we allow any CG core to assign tasks to any FG core. However, there are two concerns here: (1) arbitration for FG cores among the CG cores and (2) maximizing FG core locality. This locality exists because FG tasks that comprise the same coarser-level task use some common data. Therefore, when there is balanced demand for FG cores among the CG cores, we would like to evenly distribute FG cores among the CG cores to maximize this locality and reduce data communication.

We propose a hierarchical arbitration policy to schedule tasks to FG cores. We logically divide the FG cores evenly among the CG cores. Each set of FG cores is controlled by an arbiter. The arbiter assigns tasks to FG cores from CG cores in a priority ordering – a different CG core has priority on each arbiter. This ensures even sharing of FG cores when we have an even load across the CG cores. If the top-priority CG core for that arbiter no longer has any FG tasks, or there are idle FG cores for that arbiter, the arbiter checks the next CG core on its priority list (where each arbiter has a unique priority order). This ensures that one CG core with a larger load is able to utilize all FG cores. Detailed load balancing studies will be future work.

7.2 CG to FG Communication Latency

Another design issue we explore is buffering tasks at the FG cores to hide communication latency between CG and FG cores. The more tasks that are sent to each FG core at once, the more potential communication latency we can

hide, and the looser we can make the coupling between CG and FG cores. However, this requires that we have sufficient buffering space and application parallelism to overlap communication with computation. Looser coupling of cores allows the use of less expensive interconnect, enables flexible placement of cores (i.e. off-chip), and facilitates flexible mapping of FG to CG cores.

Our goal is to completely overlap all communication latency with computation except for the startup cost of buffering communication and post process retrieval of results. With the insatiable performance demands of IE, the primary goal of a physics architecture is to maximize performance for a given area. One way to maximize performance is to ensure the cores are maximally utilized – they should never be idle waiting for data to be sent to them. By overlapping communication and computation, we can avoid idle cycles for the cores, and as we will show, the data required to buffer tasks at the cores does not have a large impact on overall area.

Our CG cores and L2 cache banks connect via a 2D mesh, and the FG cores connect to each other through a second 2D mesh. The 2D mesh combines simplicity, area effeciciency, and power efficiency. Its latency and power consumption are slightly worse than a 2D torus, but its simplicity translates to less design effort.

Given the large resource demand of FG cores and prior work's use of off-chip acceleration (i.e. both Ageia's PhysX and GPU-based physics acceleration), one natural question to ask is whether future physics accelerators can be located off-chip, tolerating inter-chip communication delays. To address this, we examine two existing off-chip interconnect protocols to perform CG core to FG core communication: PCI Express (PCIe) and Hypertransport (HTX).

PCIe, a system interconnect with a maximum half-duplex bandwidth of 4 GB/s, is used by both GPUs and PhysX. HTX, a co-processor interconnect with a maximum half-duplex bandwidth of 20.8 GB/s, is used by AMD to connect CPUs and co-processors. Additional local buffering is used to overlap data communication and computation, and data distribution from the I/O ports of the physics chip to the individual FG cores is done via a 2-D mesh topology.

7.3 Programming Model

The orchestration of the FG cores need not require ISA modifications. The memory locations of instructions and data structure need to be remapped into the local memory space of the FG cores, and FG kernel functions need to be inlined. All of this can leverage existing compilers by adding a new back-end customized for the FG cores.

Additional code to do data packing (before sending from FG to CG cores) and data scattering (before sending from CG to FG cores) is also required. The hand-shaking between CG and FG cores for data transfers is similar to network protocols using control and data packets. The control packet includes task id (unique), data-set id (unique per task id), data size, iteration count, and kernel id. Each data packet's header includes task id and data-set id.

The control packet, in conjunction with the previously described arbiters, sets up the flow of data packets to FG cores. Once the a full set of data is received on a FG core, the kernel id chooses the kernel to execute (kernel code already resides in FG cores). The iteration count indicates the number of iterations to execute (the code assumes all FG tasks start from iteration 0 and the data has been packed accordingly). The task id uniquely identifies the CG thread which submitted the FG request, and the data set id uniquely identifies each FG core. This information is tracked on the CG core to identify the results returned back from FG cores. Each CG core has a network interface module to send, receive, and buffer these packets. Each arbiter tracks FG core activity by examining data packet headers.

8. DESIGN EXPLORATION

In this section, we explore the design space for ParallAX, including different types of fine-grain (FG) cores and interconnect strategies for FG and coarse-grain (CG) cores. For each design point, we will determine how many FG cores are required to satisfy our workload and how much local buffering is required at each core to hide communication latency. First, we characterize the FG components of parallel phases, including memory requirements.

8.1 Characterizing Fine-Grain Computation

Figure 9 (a) shows the breakdown of *Mix*'s execution time into serial, CG parallel, and FG parallel components. The first set of three bars shows the data for one core with 9MB of L2, and the 2nd set of three bars shows the data for four cores with 12MB of L2 as described earlier. The serial components' execution time (not shown) does not change significantly with increased number of cores or amount of L2. The CG sections' execution time decreased linearly as we scale from one to four cores, and the FG sections' time decreased by slightly over 50%.

Looking at the four core data, the sum of serial and CG components for *Mix* takes up 68% of one frame's time. This leaves 32% of one frame's worth of time for completing all of the FG computation.

8.1.1 Kernel Characterization

Figure 9 (b) shows the instruction mix for the FG kernels in *Narrowphase*, *Island Processing*, and *Cloth*. NOPs have been filtered out of the instruction mix. For all three, integer operations and memory reads are the top two instruction types. The main difference between these kernels is the percent of instructions dedicated to control-flow (branch and floating point (FP) compare instructions) vs data-flow (FP adds and multiplies).

Narrowphase contains 8% branch instructions and few FP adds and multiplies. This contrasts greatly with both *Island Processing* and *cloth*, where these instructions make up 32% and 28% of the total respectively. *Cloth* differs from *Island Processing* with more branches and its use of integer multiplies, FP divides, and FP squareroot instructions.

8.1.2 Memory Required for Inst and Data Storage

Next, we address how much instruction memory would be required by a FG core. Based on the iterations we sampled, the total number of unique static instructions in each kernel is 277 for *Narrowphase*, 177 for *Island Processing*, and 221 for *Cloth*. With 32-bit instructions, the largest kernel can be stored with 1.1KB of local memory. With 64-bit instructions, the largest kernel can be stored with 2.2KB of local memory. To allow any FG core to be utilized in any parallel phase, we allocate enough memory to store the code for all three kernels. This requires 2.7KB for 32-bit instructions (1.1KB for *Narrowphase*, 0.7KB for *Island Processing*, and 0.9KB for *Cloth*.)

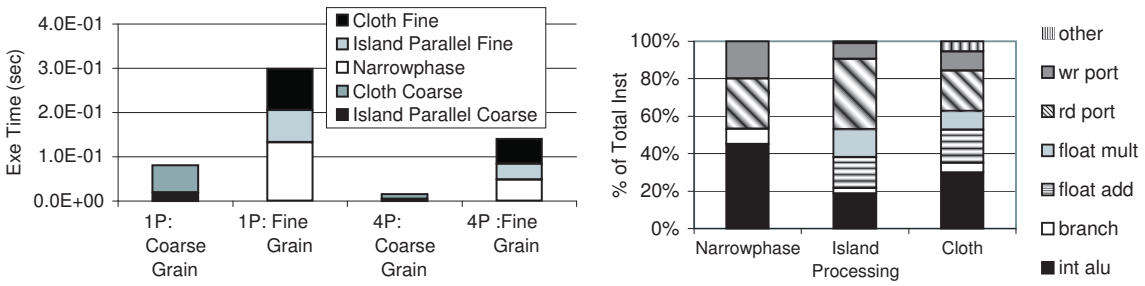


Figure 9: (a) Coarse-grain vs Fine-grain Execution Time — (b) Instruction Mix of Fine-grain Kernels

For data memory, we statistically sampled the total unique data read from memory in each kernel for 100 iterations – we found this to be 1,668B for *NarrowPhase*, 604B for *Island Processing*, and 376B for *Cloth*. The total amount of unique data written to memory in each kernel for 100 iterations is 100B for *Narrowphase*, 128B for *Island Processing*, and 308B for *Cloth*. The required data storage at each core depends on the buffering needed to hide communication latency.

Based on ths memory reuse exploration, the final FG core design contains a total of 8KB local memory.

8.2 Fine-grain Core Design and Requirements

In order to complete the FG computations within the time left, we would like to use as many cores as required to exploit the level of parallelism needed. In addition to evaluating the performance of a desktop core on these kernels, we examine more area-efficient, simpler cores modeled after next generation console and GPU shader designs. An unrealistically large core design is also used as a limit study on the available instruction level parallelism. Table 5 summarizes these designs. As discussed previously, the memory behavior of the FG cores is extremely regular. CG cores will send all data required to the FG cores – i.e. memory requests at the FG cores will always hit in the single-cycle 8KB local memory. All cores run at 2GHz.

Desktop	Intel Core Duo based core with 32-entry Inst Window, 96-entry Reorder Buffer, 17KB YAGS, 4-wide 14-cycle pipeline.
Console	IBM Cell based core with 8-entry Inst Window, 32-entry Reorder Buffer, 17KB YAGS, 2-wide 12-cycle pipeline.
GPU Shader	GPU shader based core with 1-entry Inst Window, 32-entry Reorder Buffer, 1KB YAGS, 1-wide 8-cycle pipeline.
Limit Study	Unrealistic core with 128-entry Inst Window, 512-entry Reorder Buffer, 64KB YAGS, 128-wide 14-cycle pipeline.

Table 5: Our Fine-Grain Core Designs

Figure 10 (a) shows the performance, measured in IPC, on the three kernels. *Island* and *Cloth* have bursty amounts of ILP, as confirmed by the source code and the drastic decrease in IPC from the desktop-class to the console-class cores. The limit study results show an IPC of over 4 for *Island* and 1.5 for *Cloth*. Based on the source code, these two kernels could potentially benefit from SIMD instructions. *Narrowphase* degrades with more resources due to mispredicted branch instructions. Ideal branch prediction (results not shown) resulted in a 30% improvement in performance.

8.2.1 Number of Finegrain Cores Required

Using the average IPC from Figure 10(a) and the dynamic instruction count of FG computation, we show the number of required cores for each design to reach 30 FPS for the most demanding benchmark, *Mix*, in Figure 10(b). This calculation assumes 100% utilization of FG cores during the parallel sections, which we address in the next section. We also assume that there are enough FG tasks to effectively hide communication latency, except the startup and post-process communication costs between CG to FG cores.

The first four sets of data in Figure 10(b) show the requirement if a given % of the total frame time is available for FG computation. Our 4-core CG simulation results (Figure 9(a)) left 32% of the frame time (the final set of bars).

The simulated time constraint using the 2D mesh on-chip interconnect requires 30 desktop-class, 43 console-class, or 150 shader-class cores to achieve 30 FPS. With the HTX off-chip interconnect, the number of shader-class cores increased to 151. With the PCIe interconnect, the number of shader-class cores increases to 153. The desktop-class and console-class core requirements remain the same for both off-chip interconnects. For all interconnections, 2KB of local storage is enough to buffer the minimum amount of data to hide communication latency for all cases. However, these interconnect alternatives differ in the amount of parallelism required to hide communication latency.

Area Estimation:.

By using published die areas and photos of *single* cores from Intel Core Duo 2 [22], IBM Cell [10], and Nvidia G80 [23], we derive area estimates for each core type using 90nm technology. The required interconnect area is derived from Table III of [24]. The area estimates for 30 desktop, 43 console, and 150 shader cores are 1388 mm^2, 926 mm^2, and 591 mm^2 respectively. This argues for the simplest cores as the most area-efficient alternative and points to a severe need of area optimization, which will be a main thrust of our future work.

Current generation architectures certainly lack the computational resources to meet the demands of real-time physics. But simply scaling existing designs that statically map FG cores to CG cores to match the performance demands will require considerably more area. For example, statically mapping GPU shaders only to particular CG cores will require 34% more area (in additional FG cores) than an architecture that can more flexibly and efficiently leverage core resources.

8.2.2 Parallelism to Hide Interconnect Latency

So far, we have optimistically assumed that the parallel phases of our workload could achieve 100% utilization of the fine grain cores. To verify this, we gather the amount of

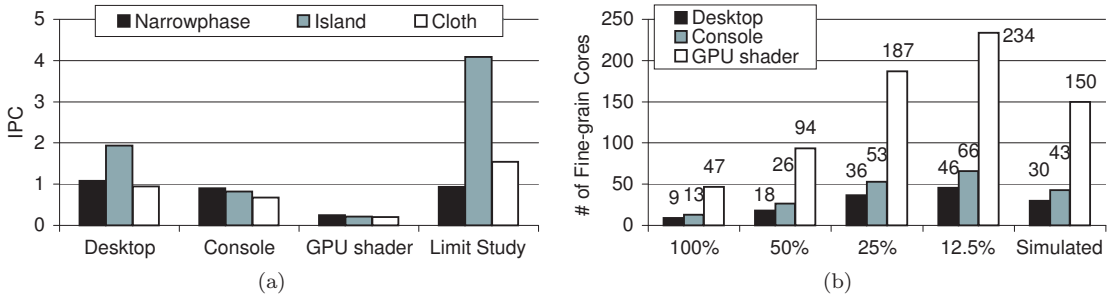


Figure 10: (a) IPC of Fine-grain Core Types — (b) Number of Fine-grain Cores to Achieve 30 FPS

available FG tasks in each phase, assuming four CG threads. As described earlier, the limit of CG parallelism is determined by the size of large islands. Table 6 shows the amount of parallel FG tasks required to hide communication latency for different core types and interconnect technologies.

Narrow, Island, Cloth	On-chip	HTX	PCIe
Desktop	30, 240, 60	30, 540, 120	60, 3000, 1650
Console	43, 215, 86	43, 473, 172	129, 2236, 2408
Shader	150, 600, 300	151, 1510, 755	308, 7700, 9394

Table 6: Number of Fine-Grain Tasks Required to Hide Communication

With four threads, Figure 11 shows all benchmarks contain enough parallel FG tasks to hide on-chip interconnect latency to the number of FG cores indicated on Figure 10 (b) except *Island Processing* for *Continuous* and *Deformable*, and *Cloth* for *Deformable*. Both benchmarks lack large islands with more than 25 parallel FG tasks. Figure 6 (a) had demonstrated that *Continuous* already executes at 30 FPS without the use of any FG cores. For *Deformable*, *Island Processing* does not take up a significant component of one frame's time. Both *Continuous* and *Deformable* can achieve 30 FPS without any FG parallelization of *Island Processing*.

When an off-chip interconnect is used, it becomes difficult to hide communication latency for *Island Processing* and *Cloth*. By filtering islands and cloth with less than 50 FG tasks, HTX's latency can be hidden. This reduces the amount of work executed on FG cores by an average 2% for *Island Processing* and 29% for *Cloth*. For PCIe, it is not possible to hide communication latency for cloth simulation running on the console or shader cores. For *Island Processing*, it becomes necessary to filter out islands with less than 1710 FG tasks in order to hide communication latency. This reduces the amount of work that can be executed on FG cores by an average 59%.

For the configurations where communication latency is fully hidden, only each phase's startup and post-process costs at each simulation step are added to the execution time. When the communication latency is exposed, this requires more FG cores to satisfy our performance bound, which can increase area by 18% for on-chip FG shader cores and 36% for off-chip (HTX) FG shader cores. One alternative to this would be to share local memories among multiple FG cores to leverage data locality and reduce the required communication – an exploration we leave for future work.

Although cloth is shown to have the least amount of FG parallelism for the next generation of games, the number of vertices used to model each cloth will likely scale up when creating realistic cloth movement further into the future.

8.3 Implementation Alternatives

As described in section 2, our design is within the space of streaming computation. Model 1 in Figure 8 shows how the design maps onto the SVM architectural model. CG cores map to the control processor, and FG cores map to the kernel processor. The host memory maps to the global memory, and local storage maps to local memory. The DMA engine, however, is now part of the control processor. Due to real-time physics simulation's extensive use of dynamic memory allocation, the data required for kernel computation is dynamically allocated and then set up for computation. All preparation happens on the CG cores first, then data is sent to the local storage of FG cores. DMA functionality can be implemented on the CG cores.

We evaluated on-chip and off-chip interconnects between the CG and FG cores. The communication latency between CG and FG computation cannot be hidden with the PCIe off-chip interconnect. This conclusion points to the *tightness* of physics simulation's feedback loop between the two granularities of computation. However, by placing the entire physics pipeline, *both* CG and FG resources, onto the *same* discrete chip, off-chip physics accelerators such as MDGRAPE and PhysX with PCIE is feasible. Model 2 in Figure 8 shows such a design. By moving all physics hardware onto a discrete, dedicated accelerator, pin-outs are increased to allow for dedicated physics memory. Dedicated physics memory may enable optimizations to reduce the dynamic memory management and OS overheads described in section 6. The control processor can preload statically allocated data onto physics memory, and only the position and orientation (60B) of each object, position (12B) of each particle, and position (12B) of mesh vertices are communicated at the beginning and end of a frame. This small fixed overhead is easily tolerated when using PCIe (0.00006 seconds for 1,000 objects, 10,000 particles, and 5,000 mesh vertices).

When using a GPU for physics acceleration, the GPU is only effective for FG tasks. The model, as shown on Figure 5 of [15], ties a general purpose CPU to the GPU using an off-chip interconnect. The serial and CG computation are done on a CPU which may not be optimized for this workload, and the off-chip latency from the CPU to the GPU will be exposed many times within a frame (similar to our off-chip evaluation) except when handling massive islands or cloths (with 7700 to 9400 parallel FG tasks).

While first-generation physics accelerators may only need to send object position and orientation back to the main processor core(s), future demands will be more significant.

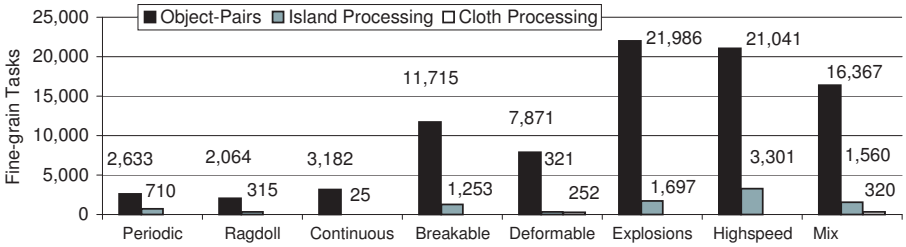


Figure 11: Average Available Fine-grain Tasks

Truly immersive reality may only be possible when the results of physics-guided motion are communicated to other components of IE applications. For example, an object that breaks may generate sound or may alert an AI character.

9. SUMMARY

Physical simulation is becoming a significant component of current and future interactive entertainment applications. We have proposed ParallAX, an architecture that features aggressive coarse-grain (CG) cores and area-efficient fine-grain (FG) cores. The CG cores are designed with sufficient, partitioned, cache space to handle both the serial and coarse grain parallel components of physics simulation. The set of FG cores exploit the massive fine-grain parallelism available for certain components of the computation. FG cores should be flexibly mapped to CG cores, and all cores should either be located on the same silicon die or packaged on separate chips in a multi-component module to successfully overlap communication and computation. With its high performance and programmability, ParallAX can be utilized for other workloads with massive fine-grain parallelism while enjoying the unique economy of scale afforded by interactive entertainment.

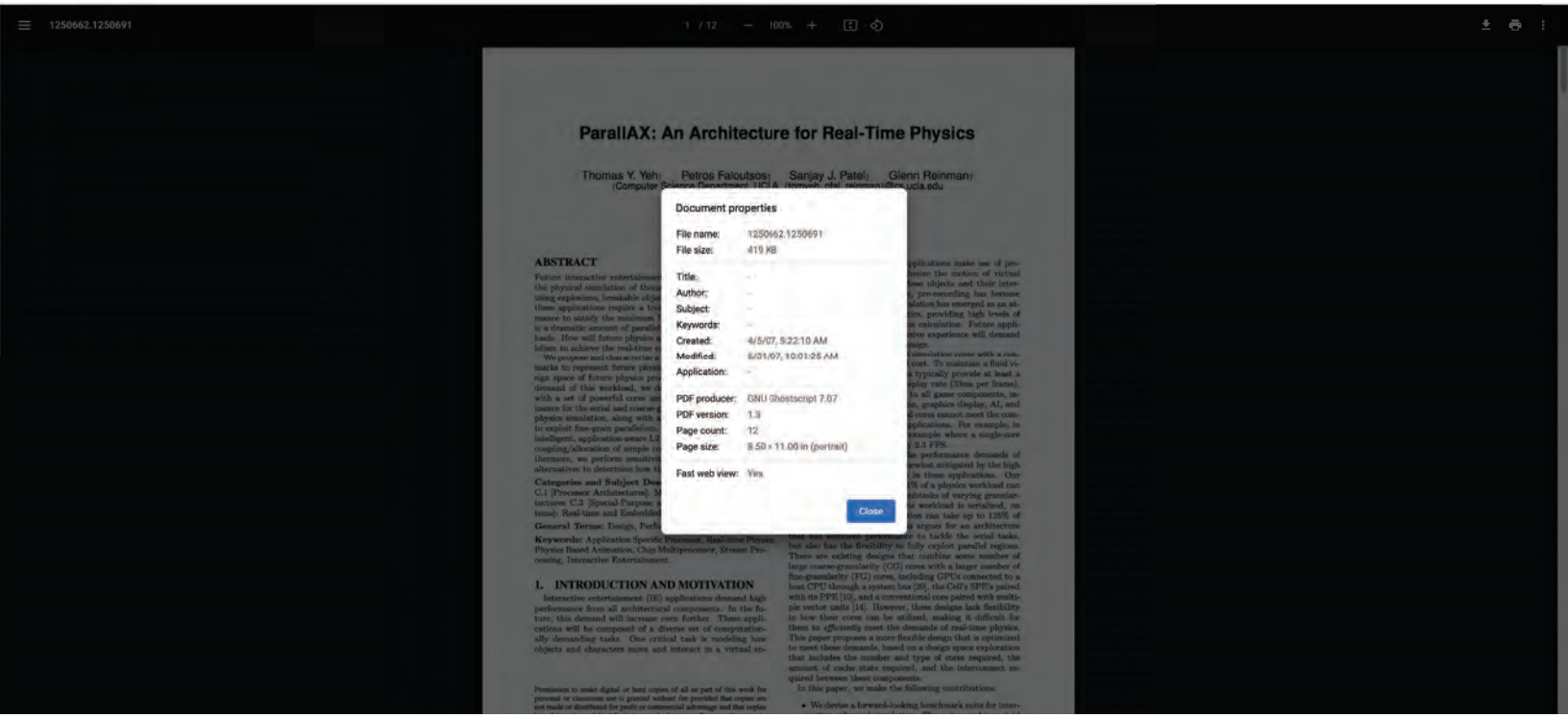
10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments. We are especially grateful to Yuval Tamir for detailed comments and valuable suggestions. We thank Chris Lamb for input on workload scenarios and the rigid body system description. Intel Corp., Microsoft Corp., and ATI Corp. helped us with their generous support through equipment and software grants. This work was partially supported by NSF grants CCF-0429983 and CCF-0133997, and by SRC grant 1317. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or SRC.

11. REFERENCES

[1] HyperTransport 3.0 Spec. http://www.hypertransport.org/docs/tech/HT3pres.pdf.
[2] PCIE Express Spec. http://www.pcisig.com/specifications/pciexpress/.
[3] AGEIA. Physx product overview. www.ageia.com.
[4] J. Andrews and N. Baker. Xbox 360 system architecture. In *IEEE Computer Society*, 2006.
[5] David Baraff. *Physically Based Modeling: Principals and Practice*. SIGGRAPH Online Course Notes, 1997.
[6] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference*, 2000.
[7] Open Dynamics Engine. http://www.ode.org/ode-latest-userguide.html.
[8] Havok. http://www.havok.com/content/view/187/77.
[9] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.
[10] P. Hofstee. Power efficient architecture and the cell processor. In *HPCA11*, 2005.
[11] T Jakobsen. Advanced character physics using the fysix engine. www.gamasutra.com, 2001.
[12] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. In *IEEE Computer*, 2003.
[13] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002.
[14] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *30th Annual International Symposium on Computer Architecture*, 2003.
[15] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. *PACT*, 2004.
[16] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer, Feb 2002*.
[17] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. In *Computer Architecture New, Sep 2005*.
[18] M. Matthias, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. In *Proceedings of the 3rd Workshop in Virtual Reality Interactions and Physical Simulation*, 2006.
[19] V. Moya, C. Gonzalex, J. Roca, A. Fernandez, and R. Espasa. Shader performance analysis on a modern gpu architecture. In *Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, 2005.
[20] M. Pharr and R. Fernando. *GPU Gems 2*. Pearson Education, 2005.
[21] P. Ranganathan, S. V. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, 2000.
[22] P. Schmid and B. Topelt. Game over? core 2 duo knocks out athlon 64. http://www.tomshardware.com/2006/07/14/.
[23] R. Sommefeldt. Nvidia g80: Architecture and gpu analysis. http://www.beyond3d.com/reviews/nvidia/g80-arch/.
[24] V. Soteriou, N. Eisley, H. Wang, B. Li, and L. Peh. Polaris: A system-level roadmap for on-chip interconnection networks. In *Proceedings of the 24th International Conference on Computer Design*, 2006.
[25] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. In *IEEE transactions on Computers*, 1992.
[26] M. Taiji. Mdgrape-3 chip: a 165 gflops application specific lsi for molecular dynamics simulations. In *Hot Chips 16*, 2004.
[27] T. Y. Yeh, P. Faloutsos, and G. Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Video Game Symposium*, 2006.
[28] T. Y. Yeh and G. Reinman. Fast and fair: Data-stream quality of service. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2005.

ATTACHMENT 6C


1250662.1250691
1 / 12
100%
ParallAX: An Architecture for Real-Time Physics
Thomas Y. Yeh
Petros Faloutsos
Sanjay J. Patel
Glenn Reinman
Document properties
File name: 1250662.1250691
File size: 419 KB
Title:
Author:
Subject:
Keywords:
Created: 4/5/07, 5:22:10 AM
Modified: 5/31/07, 10:01:25 AM
Application:
PDF producer: GNU Ghostscript 7.07
PDF version: 1.3
Page count: 12
Page size: 8.50 x 11.00 in (portrait)
Fast web view: Yes
Close
ABSTRACT
1. INTRODUCTION AND MOTIVATION

ATTACHMENT 7A

IEEE.org IEEE Xplore IEEE SA IEEE Spectrum More Sites

Cart Create Account Personal Sign In

Browse My Settings Help

Access provided by: University of Nebraska - Lincoln

Sign Out

Access provided by: University of Nebraska - Lincoln

Sign Out

All

ADVANCED SEARCH

Back to Results

Conferences > 40th Annual IEEE/ACM Internat...

The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration

Publisher: IEEE

Cite This PDF

<< Results

Thomas Yeh ; Petros Faloutsos ; Milos Ercegovac ; Sanjay Patel ; Glenn Reinman All Authors

34 Paper Citations

371 Full Text Views

Alerts

Manage Content Alerts

Add to Citation Alerts

Abstract

Document Sections

1 Introduction and Motivation

2 Background and Related Work

3 Methodology

4 Precision Reduction

5 FPU Sharing

Show Full Outline

Authors

Figures

References

Downl PDF

Abstract: Physics-based animation has enormous potential to improve the realism of interactive entertainment through dynamic, immersive content creation. Despite the massively para... **View more**

Metadata

Abstract:

Physics-based animation has enormous potential to improve the realism of interactive entertainment through dynamic, immersive content creation. Despite the massively parallel nature of physics simulation, fully exploiting this parallelism to reach interactive frame rates will require significant area to place the large number of cores. Fortunately, interactive entertainment requires believability rather than accuracy. Recent work shows that real-time physics has a remarkable tolerance for reduced precision of the significant in floating-point (FP) operations. In this paper, we describe an architecture with a hierarchical floating-point unit (FPU) that leverages dynamic precision reduction to enable efficient FPU sharing among multiple cores. This sharing reduces the area

More Like This

RTL design of parallel FFT with block floating point arithmetic

2008 IEEE Conference on Soft Computing in Industrial Applications

Published: 2008

Design and Development of Efficient Reversible Floating Point Arithmetic Unit

2015 Fifth International Conference on Communication Systems and Network Technologies

Published: 2015

Show More

References

Citations

Keywords

Metrics

More Like This

required by these cores, thereby allowing more cores to be packed into a given area and exploiting more parallelism.

Published in: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)

Date of Conference: 01-05 December 2007

Date Added to IEEE *Xplore*: 26 December 2007

ISBN Information:

ISSN Information:

INSPEC Accession Number: 9879542

DOI: 10.1109/MICRO.2007.9

Publisher: IEEE

Conference Location: Chicago, IL

Contents

SECTION 1
Introduction and Motivation

Physics-based animation (PBA) is becoming one of the most important elements of interactive entertainment applications, such as computer games, largely because of the automation and realism that it offers. The immersive nature of interactive entertainment relies on rapid, dynamic content creation and realistic visual effects. However, the benefits of PBA come at a considerable computational cost, and the demands of interactive entertainment require soft performance bounds of 30–60 frames per second. Fortunately, the physical simulation of complex scenes is massively parallel in nature. Exploiting this parallelism is an active area of research both in terms of software techniques and hardware accelerators. Commercial physics-specific solutions [3], [13] and modem processors [14] already take advantage of the parallel nature of PBA with varying levels of success.

With such massive parallelism, chip multiprocessor [22] (CMP) designs with large numbers of simple cores, such as ParallAX [32], can achieve desirable frame rates. In these designs, the more cores that can be packed into a single piece of silicon, the more parallelism that can be exploited. However, physics simulation uses extensive floating-point (FP) calculation, and could require dedicated floating-point unit (FPU) resources at each simple core. If these resources could be effectively shared across multiple cores, the area savings would result in greater core density and therefore greater exploitation of parallelism. However, contention for shared FP resources can impact performance depending on the degree of sharing.

Fortunately, PBA offers more than just massive parallelism to architects as a means of achieving high perfor-mance. In particular it can be *error tolerant*-interactive entertainment requires perceptually believable results rather than numerically

correct results. The user must be able to suspend disbelief and become immersed in the virtual world, but human perceptual tolerance to small errors is quite high [7], [23]. While the amount of error tolerance in PBA varies with the individual phase of the simulation and the scenario being modeled, prior work [34] has demonstrated that bounding the difference in total energy during a simulation can reliably ensure believability. We propose to leverage this observation to dynamically tune the precision of FP operations in PBA to provide just enough precision to approximately conserve simulation energy, and therefore ensure believability.

Our paper makes the following contributions:

- We propose dynamic FP precision tuning in PBA to leverage perceptual error tolerance.
- We identify and leverage three main benefits of precision reduction:
    1. *Increased Number of Trivial Operations:* Precision reduction enables new conditions and increased chances that a FP computation will not require the use of an FPU-such operations are trivial (e.g., adding shifted out operand, multiplication by one).
    2. *Value Locality:* Precision reduction improves the locality that exists (a) among similar objects in similar scenarios and (b) across iterations during the relaxation of constraints.
    3. *Reduced Precision FPU:* Smaller, faster, and less power hungry FPU s can be used to replace full precision FPUs. However, the varying precision requirement necessitates occasional access to full precision FPUs.
- We propose and investigate the hierarchical sharing of FPU resources among simple cores in PBA. The Ll FPU designs leverage the benefits of precision reduction to execute FP operations locally, and the full-precision L2 FPU s are shared among multiple cores for area reduction. Area savings from FPU sharing can translate directly to performance by adding more simple cores. Per phase performance is increased by up to 55% while per phase energy is simultaneously reduced by up to 50% when compared to a baseline without FPU sharing.

The rest of the paper is organized as follows. Section 2 presents background information on real-time physics simulation and reviews the most related work to this study. Section 3 details our methodology. Section 4 discusses dynamic precision tuning and techniques to exploit perceptual error tolerance. Section 5 details our hierarchical FPU architecture which leverages precision reduction for effective sharing of FPU resources. We conclude with Section 6.

SECTION 2
Background and Related Work

The most relevant work falls into three categories: realtime physics, conjoined cores, and FP optimization.

2.1 Real-Time Physics for Interactive Entertainment

Physics simulation requires the numerical solution of a discrete approximation of the differential equations of motion of all objects in a scene. Articulations between objects, and contact configurations are most often solved with constraint based approaches such as [6], [20], [13].

Physics Acceleration

ParallAX [32] is a heteroge-neous architecture for physics acceleration that combines a smaller set of more powerful cores, for simulation regions with limited or coarse-grain parallelism, with a much larger set of simple cores, for regions with massive parallelism. [15] characterizes physics-based simulation for computer animation and visual special effects used for motion pictures. Simulations with a 64-core chip multiprocessor (CMP) show very high on-die and main memory bandwidth requirements, and most modules have little inter-thread communication.

Error Tolerance for Physics

Barzel et al. [7] is credited with the introduction of the plausible simulation concept, and [23] is a 2004 state of the art survey report on the field of perceptual adaptive techniques proposed in the graphics community. Chapter 4 of [27] compares three physics engines by conducting tests on friction, gyroscopic forces, bounce, constraints, accuracy, scalability, stability, and energy conservation. All tests show significant differences between the three engines, producing different simulation results with the same initial conditions. Even without any error-injection, there is no single *correct* simulation for realtime physics simulation in games as the algorithms are optimized for speed rather than accuracy. Yeh et al. [34] went beyond simple, manually created scenarios used in previous studies to examine complex, physics-engine driven scenarios typical for games. The study provides a methodology to measure believability. After looking at gap and penetration errors, linear and angular velocity, and the angle of deflection for all objects in each scenario, they found that the difference in total energy was a reliable predictor of believability. In other words their results show that approximate energy conservation guarantees believability.

2.2 Conjoined Cores

Kumar et. al [18] proposed having two adjacent cores in a CMP share a single floating-point unit (FPU), crossbar port, instruction cache, and data cache for area savings. They looked at a simple policy where the cores take turns using the shared resources every other cycle, and a more intelligent policy where either core can use a resource in any cycle, but the arbitration priority among the cores switches from cycle to cycle for fairness. Recent work [28] has extended this idea to share other core

structures among more than two cores. Sun's Niagara [16] shares a single FPU among eight cores. The cores access the FPU via a shared bus-the latency seen for an individual FP operation is 40 cycles [29]. In this paper, we will use conjoin to describe FPU sharing.

2.3 Floating-Point Optimization

The IEEE standard single precision FP representation has a single sign bit (S), eight exponent bits (E), and 23 mantissa bits (M). These are packed in a 32-bit register to represent numbers of the form $(-1)^S 2^{E-127}(1.M)$. Precision reduction, as used in this study, refers to the removal of less significant bits from the mantissa using a selected rounding mode.

Not all FP operations need a functional unit to execute-for certain input operands, FP instructions can be made *trivial*. For example, adding any number to zero is a trivial operation: the result is simply the original number. Early studies of trivial instructions [25], [26] focused on FP benchmarks (SPEC 92 and Perfect Club) using a set of 8 trivial computations: multiplication (by 0, 1, and –1), division (*X*/*Y* with $\mathrm{X} = 0, \mathrm{Y}, -\mathrm{Y}$, and square root (of 0 and 1). Most trivial operations were FP multiplies, and the percentage of trivial operations per program ranged from near zero to 7%. They only saw performance improvements in the range of 2–4% for a processor configuration similar to modem cores.

More recent work [35] expands the range of trivializable operations to 26 types and categorize these as bypassable or simplifiable. For SPEC 95/2000 and MediaBench benchmark suites, 13 % and 6% of total dynamic instructions are trivial which enables an 8% average performance improve-ment. [5] studies the energy benefits of bypassing trivial operations. Bypassing trivial operations results in average energy and energy-delay improvement of 5% and 12% re-spectively.

Prior work uses trivialization as a filter for memoization and do not consider the effects of precision reduction. We are the first to leverage trivialization to improve resource sharing, and we propose new conditions which work with dynamic precision tuning to increase the effectiveness of trivialization.

A closely related technique to bypassing trivial operations is memoization or value reuse [25], [10]. Memoization uses an on-chip table that dynamically caches the opcode, input operands, and result of previously executed instructions. For each instruction, the opcode and input operands are checked for a match in the table. If there is a match, the cached result is reused instead of executing the instruction. Fuzzy reuse [4] leverages the error tolerance of multimedia to remove the lower significant bits of FP mantissas for the input operand match. Full precision results are still stored.

Prior work uses reuse to break data dependencies, reduce instruction latency, and conserve energy-but we are the first study to our know ledge to leverage memoization to improve resource sharing.

Another technique related to both trivialization and memoization is value prediction. Last value prediction [19] reduces instruction

latency and breaks data dependencies. Yeh et al. [33] proposed fuzzy value prediction where a certain amount of error can be tolerated without recovery. However, value prediction requires speculative execution to be effective. Therefore, these techniques will likely not be cost effective for the simple, in-order fine-grain cores we are targeting in this paper. We refer to [32] for details on performance-area tradeoff of more complex cores vs the simple ones.

SECTION 3
Methodology

Physics Simulator

Our physics engine is a heavily mod-ified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [1]. ODE follows a constraint-based approach for modeling articulated figures, similar to [6], [20], and it is designed for efficiency rather than accuracy. Our implementation supports more complex physical functions, including cloth simulation, pre-fractured objects, and explosions. We have parallelized it using POSIX threads and a work-queue model with persistent worker threads. POSIX threads minimize thread over-head, while persistent threads eliminate thread creation and destruction costs.

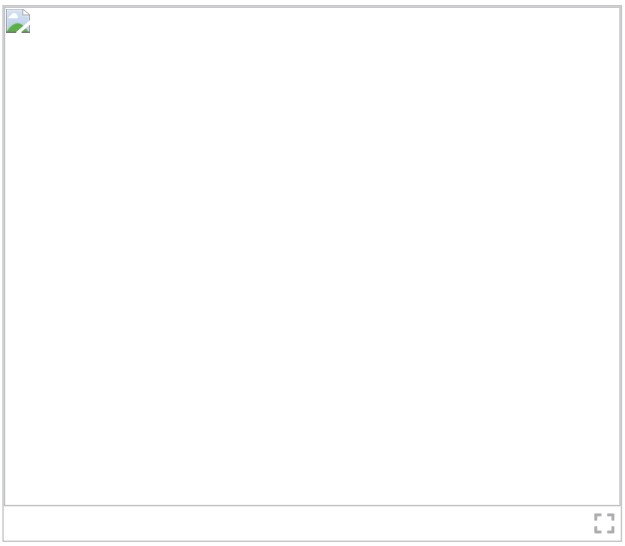


Figure 1. Physics engine flow. All phases are serialized with respect to each other. Unshaded stages can exploit parallelism

Our simulator can be described by the data-flow of computational phases shown in Figure 1. In this study, we focus on the highly parallel phases of physics computation, *Narrow-phase* and *Linear Complementary Problem (LCP) solver*, for which prior work [34] has proposed a perceptual error tolerance evaluation methodology.

Narrow-phase

This is the second step of collision de-tection that determines the contact points between each pair of colliding objects. Each pair's computational load depends on the geometric properties of the objects involved. This phase exhibits massive fine-grain parallelism since object-pairs are independent of each other.

LCP

Rigid body simulation involves the solving of forces within each group of interconnected objects (island). For each island, given the applied forces and torques, the engine computes the resulting accelerations and integrates them to compute the new position and velocity of each ob-ject. Each island is independent, and the LCP solver for each island contains loosely coupled iterations of work.

We leveraged the latest PhysicsBench suite [32], [2] for this study-a set of eight physical scenarios that span different physical actions and situations, covering a wide range of game genres, and exercising a large part of **ODE**. The simulation time-step is 0.01 seconds and 3 steps are executed per frame to ensure stability and prevent fast objects from passing through other objects. We use 20 solver iterations as recommended by [1].

Architectural Simulator

We have modified SESC [24], a cycle accurate architectural simulator, to model the ParallAX [32] architecture. We extend SESC for per phase precision reduction tuning using different rounding modes. All floating-point add, subtract, and multiply operations are executed with the specified precision in both the functional and timing components of SESC. Precision reduction is modeled by rounding both operands, executing the operation, and then rounding the result.

SECTION 4
Precision Reduction

The perceptual error tolerance associated with interactive entertainment applications reduces the required floating point precision in PBA. Exploring and exploiting this observation is a key contribution of our paper and the subject of this section.

4.1 Error Tolerance of FP Operations

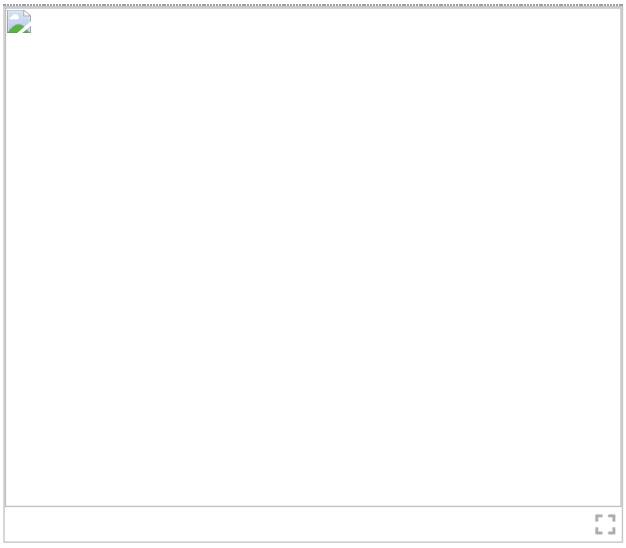
Real-time physics for interactive entertainment must be believable, but may sacrifice some accuracy. As discussed in section 2.1, the total energy in the simulation is one way of measuring the amount of tolerable error. By using the law of energy conservation, the application can compute the energy difference between successive simulation steps to determine whether the simulation is diverging towards instability. It should be noted that this energy conservation takes into account externally injected energy by the player or the game scenario. This energy difference can be compared against empirically-derived thresholds to ensure be-lievability. We added the dynamic monitoring of simulation energy to the physics engine. This additional code is added to the end of the simulation loop after integration. As each object is moved to the new position, its

energy can be calculated and stored. The accumulation of energies can be decoupled from the simulation loop calculation and is performance insensitive. With one 32-bit floating point energy value per object, this does not present a huge bandwidth bot-tleneck. The overhead for computing energy in ODE scales linearly with the number of objects and particles. Our code to compute energy for each object requires 67 instructions, and each particle requires roughly 27 instructions. For the most complicated benchmark in PhysicsBench, *Mix*, this overhead translates to less than 0.3% increase in dynamic instruction count.

4.1.1 Minimum Required Precision

One way to leverage the error tolerance of real-time physics is to reduce the amount of precision used in floating-point (FP) operations. The minimum required number of mantissa bits before the error becomes discernable varies depending on the particular phase of physics simulation and the particular physics scenario being modeled. Table 1 demonstrates this variety across all simulated PhysBench scenarios and across the two different fine-grain phases of ODE. These results are based on 30 frames of simulation following the methodology proposed in [34], and each phase is evaluated independently of the other. We evaluate three different rounding modes for reducing precision: round-to-nearest, jamming, and truncation. Denormal handling remains unchanged.

Table 1. Minimum number of mantissa bits for believable results (RN = Round-to-nearest, J = Jamming, and T = Truncation)



Round-to-nearest and truncation (round-to-zero) are IEEE FP standard rounding modes. There is a significant difference between the minimum required precision of these two rounding modes. While round-to-nearest produces the best result, it requires significant latency and area to round both operands of each FP operation before execution. On the other hand, truncation's negative bias in error injection results in much higher requirements.

To obtain the benefits of both, we look towards jamming, a rounding technique originally proposed by Burks et al. [8] and

used by Fang et al. [11]. Jamming performs a bit-level *OR* of the least significant bit (LSB) and three subsequent guard bits (GBs). The result is placed in the LSB. If LSB is one then the GBs are ignored. If the LSB is zero, then anyone in the GBs will result in rounding the LSB to one. This allows for very simple and fast logic, and the mean of injected error is 0 with no biasing.

Table 1 presents results on the minimum level of precision that is tolerable when either LCP or N arrowphase is considered alone (i.e. only one phase is precision reduced). We use jamming for rounding in the rest of this work.

While independently exploring the precision at each phase gives some intuition for the acceptable levels of precision required, we will be adaptively tuning the precision of each phase simultaneously, and therefore the error injected in one phase will impact the precision tolerance of the other phase. Since 31 % and 13% of dynamics instructions on average are FP for LCP and narrow-phase respectively, we fix LCP's required precision at the minimum found during independent exploration and evaluate the new level of precision required for narrow-phase. Dynamically precision tuning results for narrow-phase are shown in paren-thesis. For some benchmarks, narrow-phase's precision requirement is higher when considering precision reduction for both phases.

4.2 Dynamic Adaptation of Floating Point Precision

The wide range of desirable precision even across phases of physics simulation would be difficult to capture with a static technique. Furthermore, there could be unforseen, pathological situations where the simulation diverges toward instability even when adhering to a statically predetermined threshold.

Given these factors, it makes intuitive sense to dynamically adapt the minimum required precision during execution. We propose a hardware/software co-design solution. At development time, the programmer either chooses a default value or statically profiles the application using the methodology described in [34]. This error tolerance data can be used to select the minimum precisions for different sections of computation based on the developer's design.

At run-time, the application communicates the minimum required precision of the current instruction region to the hardware. The communication between software and hardware is done by setting a control register to indicate the minimum required mantissa width. The value in this register corresponds to the current minimum precision of the executing thread.

Based on dynamic information, the precision is tuned to prevent severe degradation in simulation quality. Leveraging the results from [34], we select the energy difference between successive steps as the dynamic information for tuning. Based on statically determined thresholds of percent energy difference, the application monitors its own simulation quality. We use a threshold of 10% energy difference based on the results from [34] and our observation of the energy data. When the threshold is violated, the application throttles up the significand precision (to full precision) to prevent simulation blow-up. As noted

earlier, this energy difference takes externally injected energy into account. Therefore, the total energy across successive frames may be drastically different without triggering precision adjustment. After the simulation stabilizes, the precision is incrementally reduced after every simulation step until it reaches the minimum value stored in the control register.

There can also be cases where the simulation blows up without warning. In such extreme cases, functional correctness is maintained by re-executing the previous simulation step at full precision. We leave the full study on software algorithms for controlling re-execution to future work.

Our simulations with the selected precisions do not result in simulation energy divergence. This is a fail-safe mechanism to capture unforeseen scenarios.

4.3 Opportunities Enabled by Dynamic Precision Reduction

Our reliable, dynamic adaptation of precision enables significant optimizations which we propose in this section.

4.3.1 Trivial FP Operations

Real-time physics simulation must be flexible enough to handle a wide variety of scenarios, and therefore for different inputs it may have comparisons or computations that are trivial. One example would be a wall composed of stacked bricks-if a projectile hits one part of this wall, bricks distant from the source of impact may not see any effect until time progresses. Such trivial operations provide an interesting opportunity to avoid expensive computation, and precision reduction greatly helps to increase the number of these types of operations. In this section, we will focus on trivial FP add, subtract, and multiply operations. FP addition and multiplication account for 30% of the total dynamic instruction count for the *LCP* phase of real-time physics, illustrating the criticality of these operations.

Table 2. Conventional trivial cases

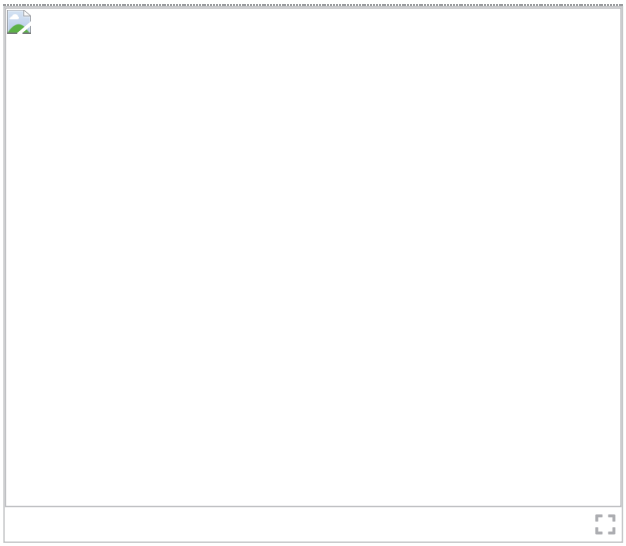
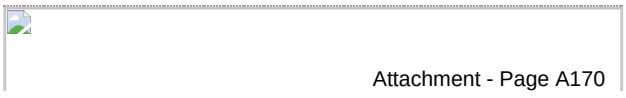
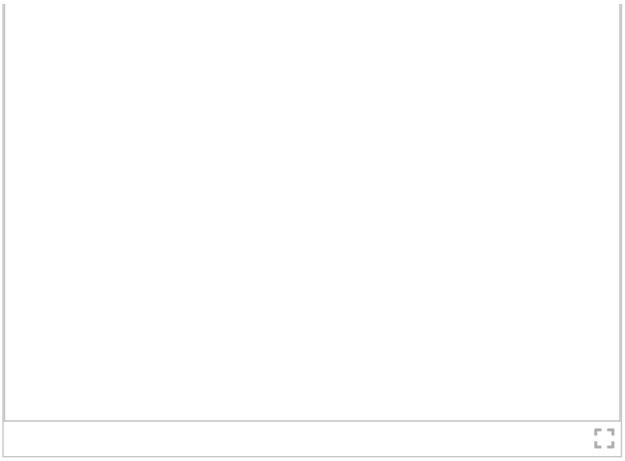


Table 3. Factors increasing trivialization





Conventional trivialization logic for these operations involves detecting operands that are zero, one, or negative one, as illustrated in table 2. We propose three extensions to conventional trivialization of FP operations:

1. *Add/Subtract:* If the magnitude of exponent difference between the operands is greater than the number of valid mantissa bits plus one, then the operation becomes trivial: the result will simply be the larger of the two operands. This is effectively addition/subtraction by zero. The addition of one take into account the implicit one of FP representation. Full precision of the non-trivial operand can be used to minimize injected error.

2. *Multiply:* If the *reduced* mantissa bits of one operand are all zeroes (i.e. the mantissa is 1.0) then the resulting mantissa will simply be the other operand. Full precision of the other operand can be used to minimize injected error. The exponent and sign logic are still ex-ecuted. This is the general case of multiplication by 1×2^E or-1×2^E for any exponent E.

3. *Divide:* If the *full* mantissa bits of the divisor are all zeroes (i.e. the mantissa is 1.0) then the resulting mantissa will simply be the dividend. Full precision of the dividend is used to minimize injected error. The exponent and sign logic are still executed. This is the general case of division by 1×2^E or-1×2^E for any exponent E. Divide could also examine the *reduced* divisor. Because the prior work on perceptual error tolerance [34] only evaluated the precision reduction of FP add, subtract, and multiply, we do not enable trivialization of reduced divisors in this study.

Based on the area for a 64-bit floating-point unit (FPU) in [10], we generate conservative estimates on the cost of trivialization. For the newly proposed trivialization con-ditions, a 8-bit adder is required for the exponent calculation. The area of a parallel prefix adder is $O(m \times logm \times K)$, where m is the number of bits. Assuming that the technology-dependent factor K is independent of precision, a 64-bit adder is about 16 times the area of an 8-bit adder. In this paper, we use the conservative estimate of an area ratio of 16.

4.3.2 Trivialization Details

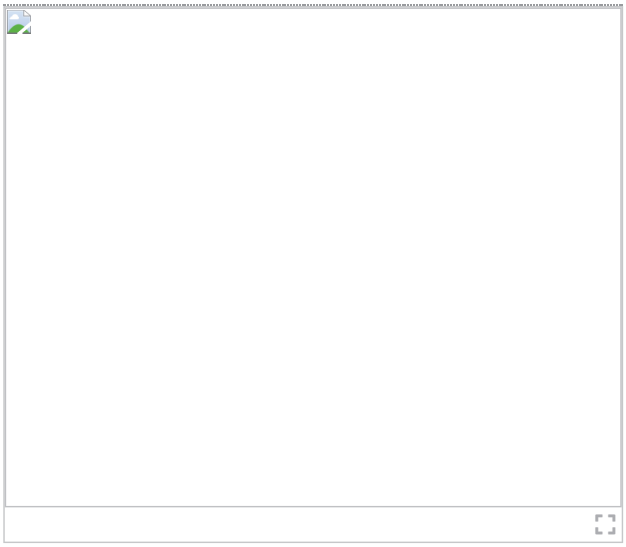
Trivial operations are a result of both the nature of physical simulation and its numerical implementation. For ex-ample, some objects come to rest or move in straight lines, objects rarely rotate continuously, and some objects move slowly, so many delta-quantities (e.g. changes in position) are close to zero within a time-step. Numerically speaking, rotational motion involves trigonometric quantities, such as cosines and sines, whose trivial values are 1, 0, –1. In addition' many computations involve normalized direction vectors which often align with one of the standard unit vectors.

In *LCP*, the core computation involves multiplications of 6-element matrices such as constraint forces, jacobian matrices, and inverse jacobians. For each matrix, three elements refer to the linear components of each axis with val-ues from the normal vector at the contact point. The other three elements refer to angular components of each axis computed as the cross-product of the contact normal with the vector defined by the contact point and the first object's point of reference.

Table 3 presents some factors that contribute to trivial operations, derived from directed tests using two rigid bodies. For example, the collision of objects lacking angular velocity may have more trivial operations than the collision of spinning objects.

Based on simulations of the latest PhysicsBench [32] with object-disabling and round-to-nearest for 200 simulation steps (30 frames), we have compiled the trivialization hit-rate with full precision using conventional conditions versus reduced precision with all conditions in Table 4 for *LCP*. Precision reduction and the new conditions increase the effectiveness of trivialization by 62% for adds and 41 % for multiplies on average. This translates to an additional 15% and 13% of total FP adds and FP multiplies being triv-ializable on average.

Table 4. Percent FP trivialized or memoized with full and reduced precision for adds and multiplies



4.3.3 Locality of FP Operations

While many FP operations in real-time physics are trivializ-able, a small number of non-trivial dynamic instructions are repetitious. Using two 256-entry memoization tables, each with 16-way associativity, we can see that less than 9% of instructions exhibit good locality. One table is used per operation type as suggested by [10]. This is summarized in table 4. We use an XOR of the most significant bits in the mantissas of the operands to index into separate tables for add and multiply. In [10], the SPEC CPU2000 benchmarks were shown to have a much higher hit rate of 20–30%, and even use a smaller table size.

When memoizing precision reduced values, however, there is certainly the immediate gain from the need to store fewer bits per entry. But, there is also an increase in coverage-with reduced precision, there are fewer possible values and combinations to memoize. Values that are very close will become one value with reduced precision, potentially combining multiple table entries into one. By leveraging reduced precision, we see that the memoization hit rate for FP multiply increases to the level seen for SPEC bench-marks. All memoization results are shown in Table 4. Trivializable operations are filtered from accessing the memo-ization tables.

4.3.4 Lookup Table

Although precision reduction significantly improves the average hit-rate of FP multiply memoization, the data points to the fact that the increased hit-rate is due to the reduction in the range of possible operand values. Table 4 shows that precision reduction significantly improve the memoization hit-rates of *Continuous, Deformable, Highspeed*, and *Rag-doll*. All of these require 5 or less mantissa bits. With n-bit mantissa operands, the number of possible unique operations is $2^n \times 2^n = 2^{2n}$. For a 4-bit or 3-bit mantissa, the 256-entry memoization table can store all possible operand pairs resulting in 100% coverage. For a 5-bit mantissa, this table can store 25% of all possible combinations. As the minimum required precision increases to 7 -bit and above, precision reduction fails to improve hit-rates significantly.

This behavior leads us to propose the use of a lookup table for computing narrow-width FP add and mult operations. By using a lookup table instead of memoization ta-bles, area and latency are significantly improved, as shown by Table 5. Instead of storing information about dynamic instruction execution, the lookup table can be populated at boot time. During run-time, no writes to the lookup table are necessary. The index is 11 bits for the 2K entry table, and the most significant bit denotes the type of FP operation (add vs mult). The output of the table is used as the mantissa of the result.

For FP multiply lookup, the concatenation of the reduced operands is used to index the table. If the minimum precision is less than 5 bits, we can still use 5 bits from each operand to index the table for a more accurate result.

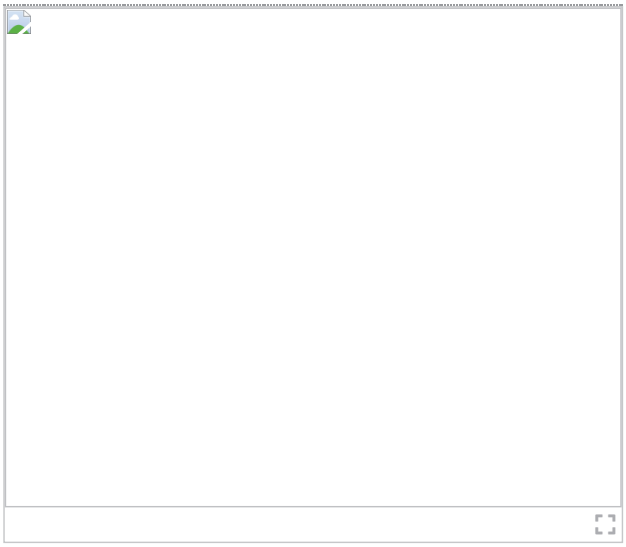
For FP add lookup, the smaller operand is first shifted based on the exponent difference. This operation only requires a small and fast 5-bit shifter. This means that the implicit 1 of the smaller operand will be seen. Then, the concatenation of the larger operand and the shifted operand is used as the index. In FP add lookup entries, an additional bit is stored to indicate the need to increment the exponent by 1 for normalization of a carry out

Since we allocate 8 bits per entry, there is room for such annotation. One comer case for FP add is when the two operands have the same ex-ponent. In this case, indexing with the default scheme will result in aliasing with a smaller operand which has been shifted. One solution is to first detect this by an exponent difference of zero, then appropriately handle the most significant bit after the leading one. This solution does not require any changes to the default scheme.

With the lookup table, there is no tag and only a single rd/wr port is required since the table contents are not modified during execution-all values are preloaded. The mem-oization table requires a read and a write port to support the simultaneous read of an issuing operation and the writing of result from the FPU calculation. However, despite this, we give an advantage to the memoization tables by only charging it for a single rd/wr port in our area/energy estimation, even though we simulate a table with two ports.

All data presented is generated using Cacti 3.0 [30]. For the memoization tables, we use 64 address bits and 32 data bits. For the lookup table, we use 11 address bits and 8 data bits-and disregard the contribution of tag area. For benchmarks requiring precision less than 6 bits, the 2K-entry lookup table (entries are IB each) achieves a better hit-rate than using the two (one for each operation type) 256 entry 16-way memoization tables (entries at 12B each) we simulate. With the lookup table, the area requirement is reduced by 77%. We assume a single cycle latency for lookup. We implement the lookup table in dedicated scratch-pad memory. When the required precision exceeds five bits, the lookup table is no longer useful and the memory can be used for other purposes. Detailed evaluation of this additional opportunity is left for future work.

Table 5. Lookup vs memoization table



SECTION 5
FPU Sharing

Precision reduction, including its effect on trivialization and the opportunity of arithmetic lookup, provides us with the opportunity to avoid or simplify work done by the FPU. In this section, we will exploit this opportunity to facilitate sharing of a single full-precision floating-point unit (FPU) among a number of fine-grain (FG) cores in the ParallAX architecture. The overall goal here is to reduce the area footprint of each FG core, enabling us to bring even more cores to bear for a given silicon budget.

We use simple, in-order shader-class cores with the pa-rameters shown in Table 6 in this study, and assume a core area excluding the floating-point unit (FPU) of $2mm^2$ in 90nm technology. Published data on FPU area ranges widely within the same technology due to parameters such as the design flow, target latency, area allocation, and circuit optimization. To make this exploration more gener-ally applicable, we explore four FPU designs with varying area requirements: $1.5mm^2$, $1.0mm^2$, $0.75mm^2$, and $0.375mm^2$. These area estimates are based respectively on results from a high-level synthesis tool tuned for generating area-efficient floating point data-paths, and published data from [18], [10], and [21].

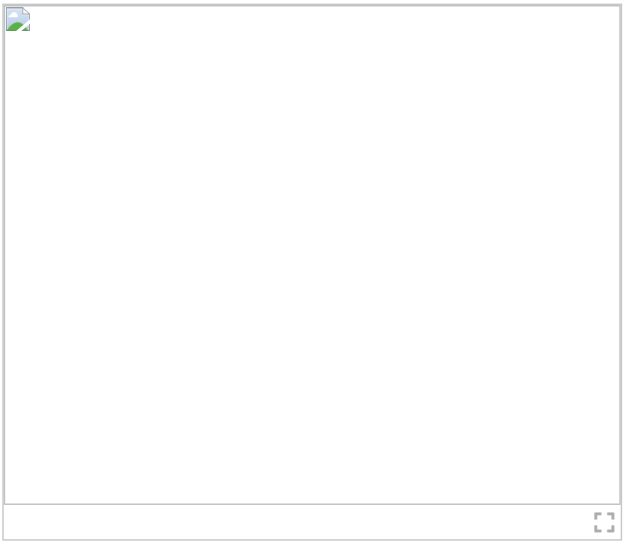


Figure 2. Layout for one, two, four, and eight cores sharing an FPU

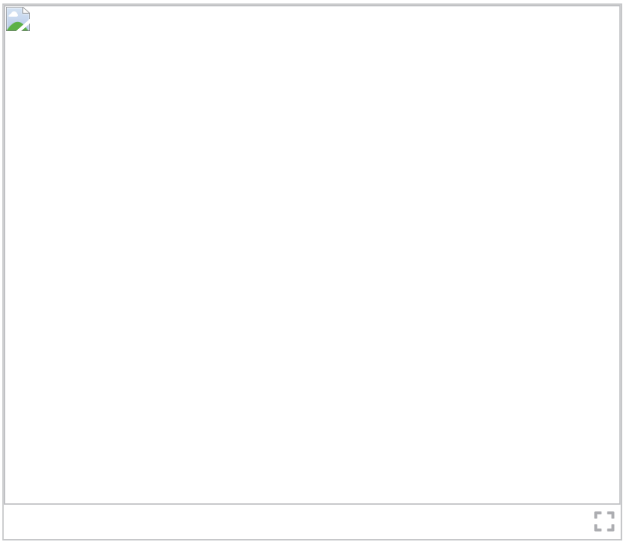
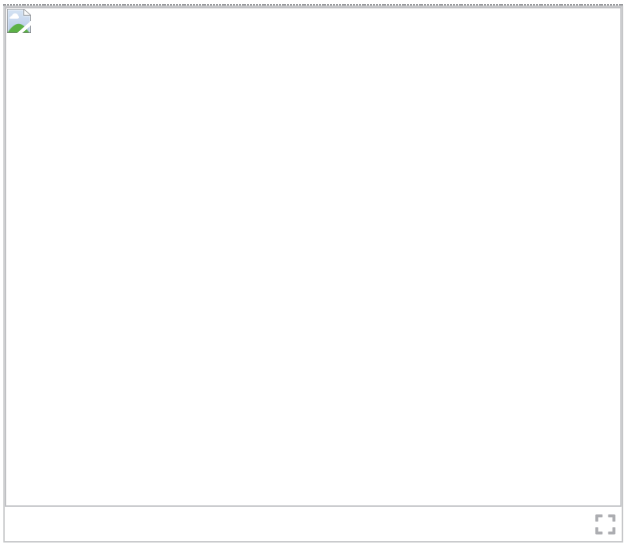


Figure 3. Pipeline for FPU sharing

Table 6. Fine-grain shader core design



The layouts for our assumptions on sharing 2, 4, and 8 cores are shown in figure 2. We adopt a simple policy for arbitration to minimize latency-the cores simply take turns accessing the FPU on alternating cycles for pipelined oper-ations. So when a single FPU is shared among N cores, a given core will get access to the FPU once every N cy-cles. If the core does not require the FPU in that cycle, the opportunity to use the FPU is wasted. For long latency non-pipelined FP operations such as divide, we assume altern ating 3-cycle scheduling windows for each core as described in [18]. These operations consist of a small percentage of total FP operations for physics simulation.

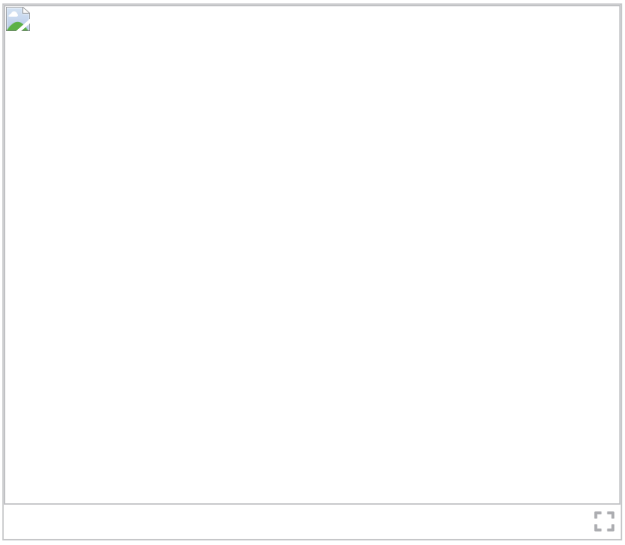


Figure 4. Four cores sharing a single FPU-each core has a trivialization and memoization unit

For sharing among 2 cores, 4 cores and 8 cores, we assume O-cycle, I-cycle, and 2-cycle latency penalties. Based on mirroring of cores as described in [17], [18], sharing the FPU among 2 cores can be accomplished with no added latency. For the 4 core and 8 core sharing, we add conservative latency estimates for the additional wire delay from each core to the FPU. Using a

$2mm \times 2mm$ aspect ratio for the cores, we assume a 2mm wire length for 4-core sharing and a 4mm wire length for 8-core sharing. Based on the 90nm information for the narrowest wire width in [9], the one-way added delays are 0.074 ns and 0.296 ns respectively. In section 5.2, we will evaluate the performance sensitivity to added FPU latency.

5.1 Hierarchical FPUs

To attack the overhead of sharing, we propose a hierarchical FPU (HFPU) architecture. A small, simple Ll FPU is placed local to each core for low latency and high through-put, and the full precision FPU s are shared at the L2 level to maximize area efficiency and to satisfy high-precision ex-ecution. This is similar in concept to a cache hierarchy, but we are sharing execution resources instead of storage resource. If the required level of precision for a particular operation exceeds the capability of the Ll FPU, then the operation must be performed in the L2 FPU. While it may be possible to use software support [12] to execute in the Ll FPU, we do not consider this alternative in this study.

We propose and evaluate the following Ll FPU design alternatives. All Ll designs contain logic to handle simple non-arithmetic FP operations. If a given operation exceeds the capability of the L 1 FPU, it will be executed in the L2 FPU. We list the alternatives by increasing complexity of the Ll FPU design.

1. *Conventional Trivialization:* The Ll FPU only contains conventional trivialization logic-no precision reduction.

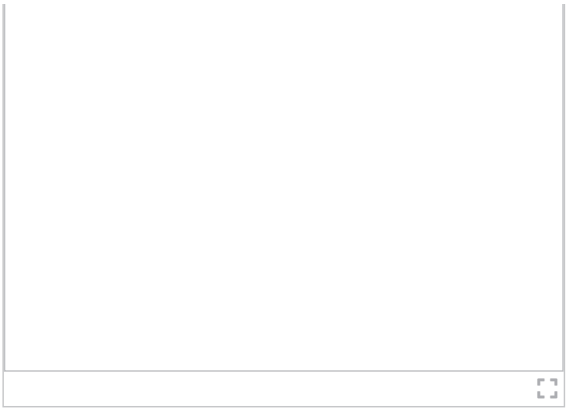
2. *Reduced Precision Trivialization:* The Ll FPU uses our reduced precision trivialization logic. Additional exponent logic is required.

3. *Lookup Table + Reduced Precision Trivialization:* The L 1 FPU has our reduced precision trivialization logic, exponent logic, and a local 2K-entry lookup table. FP add and multiply instructions using mantissa precision of less than six bits make use of the lookup table. Figure 4 demonstrates a version of this design where four cores share a single L2 FPU. Each core has an Ll FPU with trivialization and lookup table hardware. The pipeline change required for this FPU sharing is illustrated in figure 3.

4. *mini-FPU + Reduced Precision Trivialiration:* In addition to our reduced precision trivialization logic, this Ll FPU has a *mini-FPU:* an FPU design that uses a 14-bit mantissa and 8-bit exponents. FP add and multiply operations with mantissa precision of less than 15 bits can execute in this mini-FPU. The 14-bit precision is chosen based on the (benchmark coverage)/(area) ratio derived from Table 1. The intent of this design is to provide potentially higher coverage at the Ll FPU, but at a greater area cost.

Table 7. Variable FP Latency





5.1.1 Variable FP Latency

All components of HFPU's variable FP latency are listed in Table 7. If an operation can be satisfied by trivialization or look-up table, it takes one cycle to complete. If the mini-FPU is available and the operation can be satisfied by it, the operation takes three cycles. If an operation requires the full FPU, the exact latency is determined by accumulating the required latencies of arbitration, interconnect overhead, and full FPU operation.

While the latency for each operation is variable, the latency of a non-trivial operation is known at issue time. This is accomplished by using a local counter to indicate current round-robin arbitration overhead. The interconnect overhead and FPU latencies are fixed.

For the simple in-order, single-issue cores we modeled, there is no dynamic instruction scheduler. Instructions are dispatched in program order, so instruction scheduling is done at compile time. SESe uses the MIPS-ISA, and we compile with the-mips2 option without targeting a specific microarchitecture. We do not leverage the fact that FP op might be single cycle, and the same binary is used for all simulations. If the operation is satisfied by the trivial or look-up table logic, then the operation completes in 1 cycle. If not, the pipeline stalls until the operation is completed. Since the cores are in-order single-issue cores, the performance impact of these stalls is not significant. In the case of the look-up table, if the required precision can be satisfied by the look-up table, 100% of operations sent to the look-up table will be satisfied.

For more complex cores with dynamic instruction sched-ulers, different implementations are possible:

1. Optimistically assumes 1 cycle latency. If the operation is not satisfied by the trivial or look-up table logic, the pipeline stalls until the operation is completed. Once an operation is detected to require the full FPU, the actual number of required cycles is known.

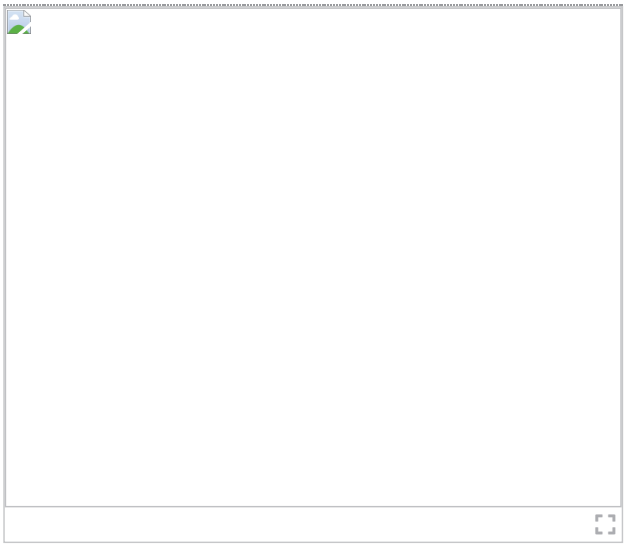
2. Pessimistically assumes the use of the full FPU. The required number of cycles is known at beginning of issue. This scheduling policy eliminates stalls/flushes/replays and retains the energy savings shown in the paper.

5.2 Results

For our performance simulations, we assume a baseline ParallAX architecture with 128 simple (fine-grain) cores, as described in Table 6, connected by a mesh interconnect. Four different FPU designs with varying area requirements (1.5 *mtri"*, 1.0 *mm/'*, 0.75 *mrti"*, and 0.375 mm^2) are eval-uated. Area estimates include per core area (2 mm^2 each), per core mesh interconnect routers (0.19 mm^2 each) [31], the full-precision FPUs, and additional logic and structures for different Ll FPU designs. The details of each configuration are shown in Table 8.

For the 14-bit mantissa mini-FPU, we assume an area requirement of 60% that of the full precision 23-bit mantissa FPU. This ratio is based on the estimates of both prior work on reduced precision FPU [11] and results from a high-level synthesis tool tuned for generating area-efficient floating point data-paths. All area data is for *90nm* technology.

Table 8. Evaluated designs. Conjoin refers to baseline with FPU sharing. Area overhead per core = additional area added to the 2 Trim2 core



First, we consider the lower area cost alternatives, and then we will address the mini-FPU. Figures 5(a) (*LCP*) and 5(b) (*Narrow-phase*) demonstrate the aggregate improvement in throughput over the baseline configuration with no FPU sharing for LCP and narrow-phase respectively. The four clusters along the x-axis represent the different FPU area sizes we evaluate, and each cluster has four data points for each architecture representing sharing a single L2 FPU among one, two, four, and eight cores.

Note that any area saved by FPU sharing is used to add more fine-grain cores, taking into account the interconnect and core area as above. Therefore, two competing trends will influence performance: (1) an increase in the number of cores will mean more exploitation of parallelism and (2) an increase in FPU sharing may mean worse per-core performance due to the overhead of sharing. Since our baseline has 128 cores, the total area available varies with the assumed size of the FPU. For the four FPU sizes we evalu-ate, the total die area is: 472 mm^2 for the 1.5 mm^2 FPU, *408* mm^2 for the 1.0 mm^2 FPU, 376 mm^2 for the

0.75 mm^2 FPU, and 328 mm^2 for the 0.375 mm^2 FPU. Figure 6 shows the total number of cores available to each configuration in the same die area as that of the appropriate baseline configuration with 128 cores.

The data across the two phases from figures 5(a) and 5(b) shows two attractive configurations: *Reduced Precision Trivialization* is better for *Narrow-phase* and *Lookup Table + Reduced Precision Trivialization* is better for *LCP*. As shown by Table 8, the per core IPCs of the two configurations for *Narrow-phase* are the same. The slight degradation with Lookup is due to the lookup table area and the fact that *Narrow-phase* always requires more than five bits of precision-therefore the lookup table is not used. To choose the better design, we also evaluate the energy and power impact of these two HFPU designs.

To estimate the total dynamic energy consumed by FP operations, we leverage the energy data for different subunits of the FPU presented in [10]. For configurations with trivialization, all FP operations are charged the trivialization logic energy. Non-trivial operations are then charged for the FPU energy. The lookup table is activated when the required precision falls below six bits. In these cases, all FP operations are charged the trivialization plus the lookup energies. The diamonds (secondary y-axis) of Figure 6(b) show a clear energy advantage to using the lookup table for *LCP*.

While performance is improved by sharing FPUs, increased per-FPU utilization could worsen the existing *hot spot* power density problem of FPU s [17]. To gauge the per-FPU utilization, we show the % of FP operations that are trivialized by trivialization or table lookup in the bars (pri-mary y-axis) of Figure 6(b). The HFPU design trivializes 53% of FP operations in the FP intensive phase of *LCP*. This suggests that the HFPU can lower per-FPU utilization when sharing one FPU between two cores as compared to the unshared baseline. For sharing among *four* cores with HFPU, per-FPU utilization will be similar to that of the prior work on conjoined CMPs [18] sharing only among *two* cores.

Next we compare the mini-FPU design to our best performing low-overhead Ll FPU. Figures 7(a) and 7(b) demonstrates the improvement in throughput over the 128-core baseline case for these designs-with figure 6 demonstrating the number of cores possible with each architecture. The mini-FPU design provides some improvement over the baseline, but the area overhead of the Ll FPU reduces the performance of this approach in comparison with the *Lookup Table + Reduced Precision Trivialization* Ll FPU.

Physics requires varying amount of numerical accuracy across different phases and scenarios-a clear differentiating factor between the error tolerance of physics and media processing. In the worst case, full precision is required. Therefore the mini-FPU cannot replace a full precision FPU in a physics processor architecture-there must be the ability to do full-precision FP. This makes the mini-FPU design inherently more costly, and argues for possibly sharing Ll FPUs among cores as well. We look at the case where the mini-FPU is shared among either two or four cores. We limit our exploration to configurations where the L2 FPU is shared by at least as many cores as the Ll FPU. For the

Fig-ures, *mini-FPU* refers to no mini-FPU sharing. *mini-FPU* 2 shared one mini-FPU among 2 cores, and *mini-FPU 4* shared one mini-FPU among 4 cores. As before, the x-axis (*Cores per FPU*) refers to the degree of L2 FPU sharing.

When looking at the per core throughput in Table 8, the 14-bit mantissa mini-FPU has the highest IPC. This is a result of a 1-cycle reduction in latency and broad coverage of benchmarks (14-bit mantissa satisfies 7/8 benchmarks for *LCP* and 4/8 benchmarks for *Narrow-phase*). However, the mini-FPU simply cannot pack as many cores into a given silicon area due to its area overhead (as shown in figure 6), resulting in a lower overall throughput. The mini-FPU designs only become more attractive for the most aggressive FPU design ($0.375mm^2$) at the highest level of sharing.

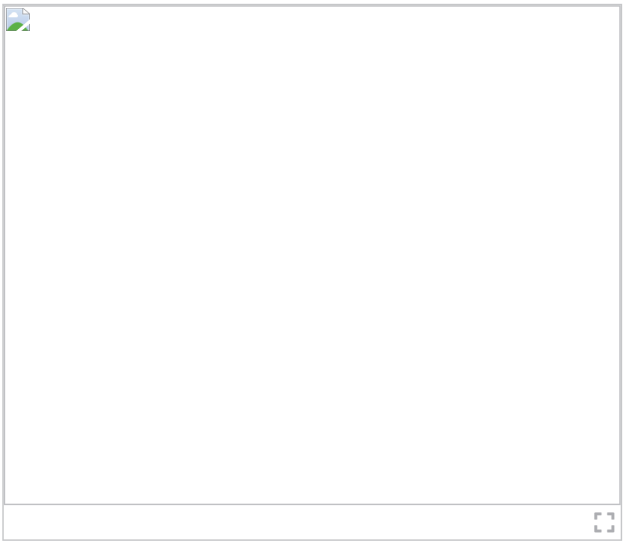


Figure 5. (a) HFPU LCP performance-(b) HFPU narrow-phase performance

As shown by the performance and energy data, the best HFPU design we examined when considering both performance and energy is to share one full-precision FPU among 4 cores using the *Lookup Table + Reduced Precision Trivialization* L1 FPU. On average, HFPU improves *LCP* performance over the baseline by 55%, 40%, 33%, and 20% for the four FPU designs. On average, HFPU improves *Narrow-phase* performance over the baseline by 46%, 32%, 25%, and 13%. Total dynamic energy expended for FP operations is reduced by 50% for *LCP* and 27% for *Narrow-phase*.

There are area and timing overheads when sharing a resource like the FPU among multiple cores regardless of the L1 FPU design-these come from the arbitration logic required to choose which core will access the shared resource and the interconnect cost to connect the cores and resource together. We evaluated the performance sensitivity to increased latency from sharing in Figure 8. The baseline for these figures is the performance of the *Lookup Table + Reduced Precision Trivialization* sharing one FPU among two cores. *LCP* (a) is shown to be more sensitive than *Narrow-phase* (b) in this figure. For *LCP*, the performance of the more aggressively designed FPUs suffers when the latency increases beyond a single cycle.

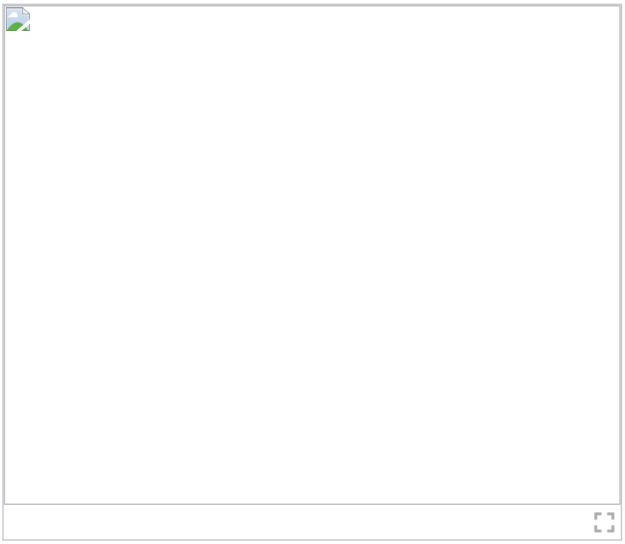


Figure 6. (a) Total number of cores in the same die area as 128-core baseline. The total area used: 472 mm2for 1.5 mm2FPU, 408 mm2for 1.0 mm2FPU, 376 mm2for 0.75 mm2FPU, and 328 mm2for 0.375 mm2FPU.-(b) FP computation % trivialized and energy reduction. C = Cony Triv, R = Reduced Triv, and L = Lookup Table + Reduced Triv

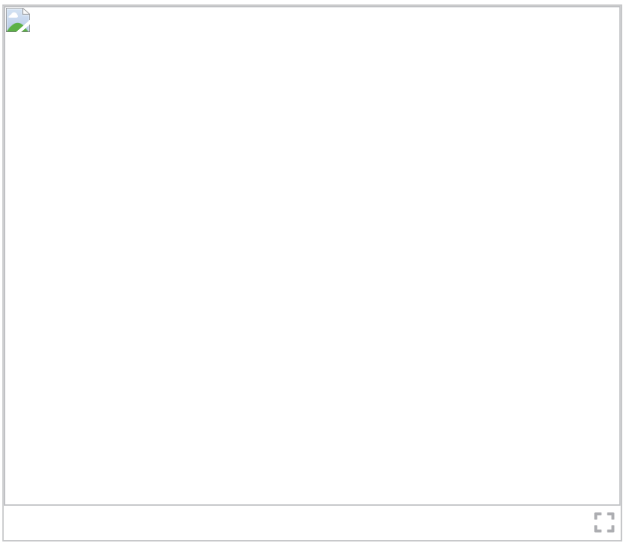


Figure 7. (a) mini-FPU LCP performance-(b) mini-FPU narrow-phase performance





Figure 8. (a) LCP latency sensitivity-(b) Narrow-phase latency sensitivity

SECTION 6
Summary and Future Work

In this study, we exploit the inherent perceptual error tolerance of physics-based animation to dynamically adapt the required precision for floating-point computation. Then, we identify and leverage the benefits made possible by dynamic precision tuning to design a hierarchical floating-point unit (HFPU). This consists of an area-efficient L1 FPU and a full-precision L2 FPU that is shared among a number of cores in the ParallAX architecture. Our best performing L1 FPU design consists of our enhanced trivialization logic and a mantissa lookup table-an area efficient design that allows more cores to be packed into a particular silicon area. For the range of FPU designs we consider, this L1 FPU coupled with an L2 FPU that is shared among four cores provides an improvement in overall throughput up to 55% and reduces energy expended in the FPU by up to 50%.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful comments. Intel Corp., Microsoft Corp., AMD/ATI Corp., and AGEIA Technologies Corp. helped us with their generous support through equipment and software grants. This work was partially supported by NSF grants CCF-0429983 and CCF-0133997, and by SRC grant 1317. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or SRC.

Authors

Figures

References

Citations

Keywords

Metrics

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved.

IEEE Account

» Change Username/Password

» Update Address

Purchase Details

» Payment Options

» Order History

» View Purchased Documents

Profile Information

» Communications Preferences

» Profession and Education

» Technical Interests

Need Help?

» **US & Canada:** +1 800 678 4333

» **Worldwide:** +1 732 981 0060

» Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.

ATTACHMENT 7B

The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration

Thomas Y. Yeh† Petros Faloutsos† Milos Ercegovac† Sanjay J. Patel‡ Glenn Reinman†
†Computer Science Department, UCLA, {tomyeh, pfal, milos, reinman}@cs.ucla.edu
‡AGEIA Technologies, sjp@ageia.com

Abstract

Physics-based animation has enormous potential to improve the realism of interactive entertainment through dynamic, immersive content creation. Despite the massively parallel nature of physics simulation, fully exploiting this parallelism to reach interactive frame rates will require significant area to place the large number of cores. Fortunately, interactive entertainment requires believability rather than accuracy. Recent work shows that real-time physics has a remarkable tolerance for reduced precision of the significand in floating-point (FP) operations. In this paper, we describe an architecture with a hierarchical floating-point unit (FPU) that leverages dynamic precision reduction to enable efficient FPU sharing among multiple cores. This sharing reduces the area required by these cores, thereby allowing more cores to be packed into a given area and exploiting more parallelism.

1 Introduction and Motivation

Physics-based animation (PBA) is becoming one of the most important elements of interactive entertainment applications, such as computer games, largely because of the automation and realism that it offers. The immersive nature of interactive entertainment relies on rapid, dynamic content creation and realistic visual effects. However, the benefits of PBA come at a considerable computational cost, and the demands of interactive entertainment require soft performance bounds of 30-60 frames per second. Fortunately, the physical simulation of complex scenes is massively parallel in nature. Exploiting this parallelism is an active area of research both in terms of software techniques and hardware accelerators. Commercial physics-specific solutions [3, 13] and modern processors [14] already take advantage of the parallel nature of PBA with varying levels of success.

With such massive parallelism, chip multiprocessor [22] (CMP) designs with large numbers of simple cores, such as ParallAX [32], can achieve desirable frame rates. In these designs, the more cores that can be packed into a single piece of silicon, the more parallelism that can be exploited. However, physics simulation uses extensive floating-point (FP) calculation, and could require dedicated floating-point unit (FPU) resources at each simple core. If these resources could be effectively shared across multiple cores, the area savings would result in greater core density and therefore greater exploitation of parallelism. However, contention for shared FP resources can impact performance depending on the degree of sharing.

Fortunately, PBA offers more than just massive parallelism to architects as a means of achieving high performance. In particular it can be *error tolerant* – interactive entertainment requires perceptually believable results rather than numerically correct results. The user must be able to suspend disbelief and become immersed in the virtual world, but human perceptual tolerance to small errors is quite high [7, 23]. While the amount of error tolerance in PBA varies with the individual phase of the simulation and the scenario being modeled, prior work [34] has demonstrated that bounding the difference in total energy during a simulation can reliably ensure believability. We propose to leverage this observation to dynamically tune the precision of FP operations in PBA to provide just enough precision to approximately conserve simulation energy, and therefore ensure believability.

Our paper makes the following contributions:

- We propose dynamic FP precision tuning in PBA to leverage perceptual error tolerance.
- We identify and leverage three main benefits of precision reduction:
    1. *Increased Number of Trivial Operations:* Precision reduction enables new conditions and increased chances that a FP computation will not require the use of an FPU – such operations are trivial (e.g., adding shifted out operand, multiplication by one).

1072-4451/07 $25.00 © 2007 IEEE
DOI 10.1109/MICRO.2007.9

IEEE computer society

Attachment - Page A186

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

2. *Value Locality:* Precision reduction improves the locality that exists (a) among similar objects in similar scenarios and (b) across iterations during the relaxation of constraints.
3. *Reduced Precision FPU:* Smaller, faster, and less power hungry FPUs can be used to replace full precision FPUs. However, the varying precision requirement necessitates occasional access to full precision FPUs.

- We propose and investigate the hierarchical sharing of FPU resources among simple cores in PBA. The L1 FPU designs leverage the benefits of precision reduction to execute FP operations locally, and the full-precision L2 FPUs are shared among multiple cores for area reduction. Area savings from FPU sharing can translate directly to performance by adding more simple cores. Per phase performance is increased by up to 55% while per phase energy is simultaneously reduced by up to 50% when compared to a baseline without FPU sharing.

The rest of the paper is organized as follows. Section 2 presents background information on real-time physics simulation and reviews the most related work to this study. Section 3 details our methodology. Section 4 discusses dynamic precision tuning and techniques to exploit perceptual error tolerance. Section 5 details our hierarchical FPU architecture which leverages precision reduction for effective sharing of FPU resources. We conclude with Section 6.

2 Background and Related Work

The most relevant work falls into three categories: real-time physics, conjoined cores, and FP optimization.

2.1 Real-Time Physics for Interactive Entertainment

Physics simulation requires the numerical solution of a discrete approximation of the differential equations of motion of all objects in a scene. Articulations between objects, and contact configurations are most often solved with constraint based approaches such as [6, 20, 13].

Physics Acceleration. ParallAX [32] is a heterogeneous architecture for physics acceleration that combines a smaller set of more powerful cores, for simulation regions with limited or coarse-grain parallelism, with a much larger set of simple cores, for regions with massive parallelism. [15] characterizes physics-based simulation for computer animation and visual special effects used for motion pictures. Simulations with a 64-core chip multiprocessor (CMP) show very high on-die and main memory bandwidth requirements, and most modules have little inter-thread communication.

Error Tolerance for Physics. Barzel et al. [7] is credited with the introduction of the plausible simulation concept, and [23] is a 2004 state of the art survey report on the field of perceptual adaptive techniques proposed in the graphics community. Chapter 4 of [27] compares three physics engines by conducting tests on friction, gyroscopic forces, bounce, constraints, accuracy, scalability, stability, and energy conservation. All tests show significant differences between the three engines, producing different simulation results with the same initial conditions. Even without any error-injection, there is no single *correct* simulation for real-time physics simulation in games as the algorithms are optimized for speed rather than accuracy. Yeh et al. [34] went beyond simple, manually created scenarios used in previous studies to examine complex, physics-engine driven scenarios typical for games. The study provides a methodology to measure believability. After looking at gap and penetration errors, linear and angular velocity, and the angle of deflection for all objects in each scenario, they found that the difference in total energy was a reliable predictor of believability. In other words their results show that approximate energy conservation guarantees believability.

2.2 Conjoined Cores

Kumar et. al [18] proposed having two adjacent cores in a CMP share a single floating-point unit (FPU), crossbar port, instruction cache, and data cache for area savings. They looked at a simple policy where the cores take turns using the shared resources every other cycle, and a more intelligent policy where either core can use a resource in any cycle, but the arbitration priority among the cores switches from cycle to cycle for fairness. Recent work [28] has extended this idea to share other core structures among more than two cores. Sun's Niagara [16] shares a single FPU among eight cores. The cores access the FPU via a shared bus – the latency seen for an individual FP operation is 40 cycles [29]. In this paper, we will use conjoin to describe FPU sharing.

2.3 Floating-Point Optimization

The IEEE standard single precision FP representation has a single sign bit (S), eight exponent bits (E), and 23 mantissa bits (M). These are packed in a 32-bit register to represent numbers of the form $(-1)^S 2^{E-127}(1.M)$. Precision reduction, as used in this study, refers to the removal of less significant bits from the mantissa using a selected rounding mode.

Not all FP operations need a functional unit to execute – for certain input operands, FP instructions can be made

Attachment - Page A187

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

trivial. For example, adding any number to zero is a trivial operation: the result is simply the original number. Early studies of trivial instructions [25, 26] focused on FP benchmarks (SPEC 92 and Perfect Club) using a set of 8 trivial computations: multiplication (by 0, 1, and -1), division (X/Y with X = 0,Y,-Y, and square root (of 0 and 1). Most trivial operations were FP multiplies, and the percentage of trivial operations per program ranged from near zero to 7%. They only saw performance improvements in the range of 2-4% for a processor configuration similar to modern cores.

More recent work [35] expands the range of trivializable operations to 26 types and categorize these as bypassable or simplifiable. For SPEC 95/2000 and MediaBench benchmark suites, 13% and 6% of total dynamic instructions are trivial which enables an 8% average performance improvement. [5] studies the energy benefits of bypassing trivial operations. Bypassing trivial operations results in average energy and energy-delay improvement of 5% and 12% respectively.

Prior work uses trivialization as a filter for memoization and do not consider the effects of precision reduction. We are the first to leverage trivialization to improve resource sharing, and we propose new conditions which work with dynamic precision tuning to increase the effectiveness of trivialization.

A closely related technique to bypassing trivial operations is memoization or value reuse [25, 10]. Memoization uses an on-chip table that dynamically caches the opcode, input operands, and result of previously executed instructions. For each instruction, the opcode and input operands are checked for a match in the table. If there is a match, the cached result is reused instead of executing the instruction. Fuzzy reuse [4] leverages the error tolerance of multimedia to remove the lower significant bits of FP mantissas for the input operand match. Full precision results are still stored.

Prior work uses reuse to break data dependencies, reduce instruction latency, and conserve energy – but we are the first study to our knowledge to leverage memoization to improve resource sharing.

Another technique related to both trivialization and memoization is value prediction. Last value prediction [19] reduces instruction latency and breaks data dependencies. Yeh et al. [33] proposed fuzzy value prediction where a certain amount of error can be tolerated without recovery. However, value prediction requires speculative execution to be effective. Therefore, these techniques will likely not be cost effective for the simple, in-order fine-grain cores we are targeting in this paper. We refer to [32] for details on performance-area tradeoff of more complex cores vs the simple ones.

3 Methodology

Physics Simulator. Our physics engine is a heavily modified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [1]. ODE follows a constraint-based approach for modeling articulated figures, similar to [6, 20], and it is designed for efficiency rather than accuracy. Our implementation supports more complex physical functions, including cloth simulation, prefractured objects, and explosions. We have parallelized it using POSIX threads and a work-queue model with persistent worker threads. POSIX threads minimize thread overhead, while persistent threads eliminate thread creation and destruction costs.

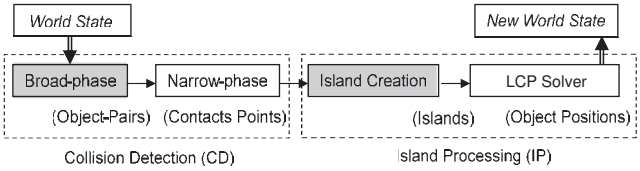


Figure 1. Physics engine flow. All phases are serialized with respect to each other. Unshaded stages can exploit parallelism.

Our simulator can be described by the data-flow of computational phases shown in Figure 1. In this study, we focus on the highly parallel phases of physics computation, *Narrow-phase* and *Linear Complementary Problem (LCP) solver*, for which prior work [34] has proposed a perceptual error tolerance evaluation methodology.

Narrow-phase. This is the second step of collision detection that determines the contact points between each pair of colliding objects. Each pair's computational load depends on the geometric properties of the objects involved. This phase exhibits massive fine-grain parallelism since object-pairs are independent of each other.

LCP. Rigid body simulation involves the solving of forces within each group of interconnected objects (island). For each island, given the applied forces and torques, the engine computes the resulting accelerations and integrates them to compute the new position and velocity of each object. Each island is independent, and the LCP solver for each island contains loosely coupled iterations of work.

We leveraged the latest PhysicsBench suite [32, 2] for this study – a set of eight physical scenarios that span different physical actions and situations, covering a wide range of game genres, and exercising a large part of ODE. The simulation time-step is 0.01 seconds and 3 steps are executed per frame to ensure stability and prevent fast objects from passing through other objects. We use 20 solver iterations as recommended by [1].

Attachment - Page A188

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

Architectural Simulator. We have modified SESC [24], a cycle accurate architectural simulator, to model the ParallAX [32] architecture. We extend SESC for per phase precision reduction tuning using different rounding modes. All floating-point add, subtract, and multiply operations are executed with the specified precision in both the functional and timing components of SESC. Precision reduction is modeled by rounding both operands, executing the operation, and then rounding the result.

4 Precision Reduction

The perceptual error tolerance associated with interactive entertainment applications reduces the required floating point precision in PBA. Exploring and exploiting this observation is a key contribution of our paper and the subject of this section.

4.1 Error Tolerance of FP Operations

Real-time physics for interactive entertainment must be believable, but may sacrifice some accuracy. As discussed in section 2.1, the total energy in the simulation is one way of measuring the amount of tolerable error. By using the law of energy conservation, the application can compute the energy difference between successive simulation steps to determine whether the simulation is diverging towards instability. It should be noted that this energy conservation takes into account externally injected energy by the player or the game scenario. This energy difference can be compared against empirically-derived thresholds to ensure believability. We added the dynamic monitoring of simulation energy to the physics engine. This additional code is added to the end of the simulation loop after integration. As each object is moved to the new position, its energy can be calculated and stored. The accumulation of energies can be decoupled from the simulation loop calculation and is performance insensitive. With one 32-bit floating point energy value per object, this does not present a huge bandwidth bottleneck. The overhead for computing energy in ODE scales linearly with the number of objects and particles. Our code to compute energy for each object requires 67 instructions, and each particle requires roughly 27 instructions. For the most complicated benchmark in PhysicsBench, *Mix*, this overhead translates to less than 0.3% increase in dynamic instruction count.

4.1.1 Minimum Required Precision

One way to leverage the error tolerance of real-time physics is to reduce the amount of precision used in floating-point (FP) operations. The minimum required number of mantissa bits before the error becomes discernable varies depending on the particular phase of physics simulation and the particular physics scenario being modeled. Table 1 demonstrates this variety across all simulated PhysBench scenarios and across the two different fine-grain phases of ODE. These results are based on 30 frames of simulation following the methodology proposed in [34], and each phase is evaluated independently of the other. We evaluate three different rounding modes for reducing precision: round-to-nearest, jamming, and truncation. Denormal handling remains unchanged.

Benchmark	LCP			Narrowphase		
	RN	J	T	RN	J	T
Breakable	8	17	13	17	10 (21)	23
Continuous	4	4	4	9	9 (9)	9
Deformable	3	4	8	9	9 (9)	9
Everything	10	10	23	18	10 (17)	19
Explosions	11	13	9	21	14 (14)	13
Highspeed	3	3	8	9	9 (9)	9
Periodic	13	14	23	22	21 (23)	23
Ragdoll	5	5	9	9	9 (21)	9

Table 1. Minimum number of mantissa bits for believable results (RN = Round-to-nearest, J = Jamming, and T = Truncation).

Round-to-nearest and truncation (round-to-zero) are IEEE FP standard rounding modes. There is a significant difference between the minimum required precision of these two rounding modes. While round-to-nearest produces the best result, it requires significant latency and area to round both operands of each FP operation before execution. On the other hand, truncation's negative bias in error injection results in much higher requirements.

To obtain the benefits of both, we look towards jamming, a rounding technique originally proposed by Burks et al. [8] and used by Fang et al. [11]. Jamming performs a bit-level *OR* of the least significant bit (LSB) and three subsequent guard bits (GBs). The result is placed in the LSB. If LSB is one then the GBs are ignored. If the LSB is zero, then any one in the GBs will result in rounding the LSB to one. This allows for very simple and fast logic, and the mean of injected error is 0 with no biasing.

Table 1 presents results on the minimum level of precision that is tolerable when either LCP or Narrowphase is considered alone (i.e. only one phase is precision reduced). We use jamming for rounding in the rest of this work.

While independently exploring the precision at each phase gives some intuition for the acceptable levels of precision required, we will be adaptively tuning the precision of each phase simultaneously, and therefore the error injected in one phase will impact the precision tolerance of the other phase. Since 31% and 13% of dynamics instructions

Attachment - Page A189

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

on average are FP for LCP and narrow-phase respectively, we fix LCP's required precision at the minimum found during independent exploration and evaluate the new level of precision required for narrow-phase. Dynamically precision tuning results for narrow-phase are shown in parenthesis. For some benchmarks, narrow-phase's precision requirement is higher when considering precision reduction for both phases.

4.2 Dynamic Adaptation of Floating Point Precision

The wide range of desirable precision even across phases of physics simulation would be difficult to capture with a static technique. Furthermore, there could be unforseen, pathological situations where the simulation diverges toward instability even when adhering to a statically predetermined threshold.

Given these factors, it makes intuitive sense to dynamically adapt the minimum required precision during execution. We propose a hardware/software co-design solution. At development time, the programmer either chooses a default value or statically profiles the application using the methodology described in [34]. This error tolerance data can be used to select the minimum precisions for different sections of computation based on the developer's design.

At run-time, the application communicates the minimum required precision of the current instruction region to the hardware. The communication between software and hardware is done by setting a control register to indicate the minimum required mantissa width. The value in this register corresponds to the current minimum precision of the executing thread.

Based on dynamic information, the precision is tuned to prevent severe degradation in simulation quality. Leveraging the results from [34], we select the energy difference between successive steps as the dynamic information for tuning. Based on statically determined thresholds of percent energy difference, the application monitors its own simulation quality. We use a threshold of 10% energy difference based on the results from [34] and our observation of the energy data. When the threshold is violated, the application throttles up the significand precision (to full precision) to prevent simulation blow-up. As noted earlier, this energy difference takes externally injected energy into account. Therefore, the total energy across successive frames may be drastically different without triggering precision adjustment. After the simulation stabilizes, the precision is incrementally reduced after every simulation step until it reaches the minimum value stored in the control register.

There can also be cases where the simulation blows up without warning. In such extreme cases, functional correctness is maintained by re-executing the previous simulation step at full precision. We leave the full study on software algorithms for controlling re-execution to future work.

Our simulations with the selected precisions do not result in simulation energy divergence. This is a fail-safe mechanism to capture unforeseen scenarios.

4.3 Opportunities Enabled by Dynamic Precision Reduction

Our reliable, dynamic adaptation of precision enables significant optimizations which we propose in this section.

4.3.1 Trivial FP Operations

Real-time physics simulation must be flexible enough to handle a wide variety of scenarios, and therefore for different inputs it may have comparisons or computations that are trivial. One example would be a wall composed of stacked bricks – if a projectile hits one part of this wall, bricks distant from the source of impact may not see any effect until time progresses. Such trivial operations provide an interesting opportunity to avoid expensive computation, and precision reduction greatly helps to increase the number of these types of operations. In this section, we will focus on trivial FP add, subtract, and multiply operations. FP addition and multiplication account for 30% of the total dynamic instruction count for the *LCP* phase of real-time physics, illustrating the criticality of these operations.

FP Operation	Representation	Trivial when
Add	X + Y	X=0 or Y=0
Subtract	X - Y	X=0 or Y=0
Multiply	X * Y	X=0 or $\pm$1, or Y=0 or $\pm$1
Divide	X / Y	X=0 or Y=$\pm$1

Table 2. Conventional trivial cases

Conditions for Increased Trivialization
Small mass difference between objects
Zero linear and angular velocities before collision
Small size difference between objects
Simple object shapes
Use of ground and gravity
Higher amount of articulation (human vs box)

Table 3. Factors increasing trivialization

Conventional trivialization logic for these operations involves detecting operands that are zero, one, or negative one, as illustrated in table 2. We propose three extensions to conventional trivialization of FP operations:

1. *Add/Subtract*: If the magnitude of exponent difference between the operands is greater than the number of

Attachment - Page A190

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

valid mantissa bits plus one, then the operation becomes trivial: the result will simply be the larger of the two operands. This is effectively addition/subtraction by zero. The addition of one take into account the implicit one of FP representation. Full precision of the non-trivial operand can be used to minimize injected error.

2. *Multiply*: If the *reduced* mantissa bits of one operand are all zeroes (i.e. the mantissa is 1.0) then the resulting mantissa will simply be the other operand. Full precision of the other operand can be used to minimize injected error. The exponent and sign logic are still executed. This is the general case of multiplication by 1×2^E or -1 $\times 2^E$ for any exponent E.

3. *Divide*: If the *full* mantissa bits of the divisor are all zeroes (i.e. the mantissa is 1.0) then the resulting mantissa will simply be the dividend. Full precision of the dividend is used to minimize injected error. The exponent and sign logic are still executed. This is the general case of division by 1×2^E or -1 $\times 2^E$ for any exponent E. Divide could also examine the *reduced* divisor. Because the prior work on perceptual error tolerance [34] only evaluated the precision reduction of FP add, subtract, and multiply, we do not enable trivialization of reduced divisors in this study.

Based on the area for a 64-bit floating-point unit (FPU) in [10], we generate conservative estimates on the cost of trivialization. For the newly proposed trivialization conditions, a 8-bit adder is required for the exponent calculation. The area of a parallel prefix adder is $O(m \times logm \times K)$, where m is the number of bits. Assuming that the technology-dependent factor K is independent of precision, a 64-bit adder is about 16 times the area of an 8-bit adder. In this paper, we use the conservative estimate of an area ratio of 16.

4.3.2 Trivialization Details

Trivial operations are a result of both the nature of physical simulation and its numerical implementation. For example, some objects come to rest or move in straight lines, objects rarely rotate continuously, and some objects move slowly, so many delta-quantities (e.g. changes in position) are close to zero within a time-step. Numerically speaking, rotational motion involves trigonometric quantities, such as cosines and sines, whose trivial values are $1, 0, -1$. In addition, many computations involve normalized direction vectors which often align with one of the standard unit vectors.

In *LCP*, the core computation involves multiplications of 6-element matrices such as constraint forces, jacobian matrices, and inverse jacobians. For each matrix, three elements refer to the linear components of each axis with values from the normal vector at the contact point. The other three elements refer to angular components of each axis computed as the cross-product of the contact normal with the vector defined by the contact point and the first object's point of reference.

Table 3 presents some factors that contribute to trivial operations, derived from directed tests using two rigid bodies. For example, the collision of objects lacking angular velocity may have more trivial operations than the collision of spinning objects.

Based on simulations of the latest PhysicsBench [32] with object-disabling and round-to-nearest for 200 simulation steps (30 frames), we have compiled the trivialization hit-rate with full precision using conventional conditions versus reduced precision with all conditions in Table 4 for *LCP*. Precision reduction and the new conditions increase the effectiveness of trivialization by 62% for adds and 41% for multiplies on average. This translates to an additional 15% and 13% of total FP adds and FP multiplies being trivializable on average.

Benchmark	Trivial (Add,Mult)		Memo (Add,Mult)	
	23-bit	Reduced	23-bit	Reduced
Bre	36, 34	48, 41	0, 2	1, 8
Con	49, 43	71, 62	0, 1	8, 38
Def	32, 31	61, 64	0, 2	7, 35
Eve	35, 33	43, 38	0, 3	1, 6
Exp	28, 25	38, 29	0, 7	1, 10
Hig	27, 23	54, 49	0, 8	11, 51
Per	32, 32	34, 34	0, 0	0, 0
Rag	34, 33	52, 53	0, 0	2, 28

Table 4. Percent FP trivialized or memoized with full and reduced precision for adds and multiplies.

4.3.3 Locality of FP Operations

While many FP operations in real-time physics are trivializable, a small number of non-trivial dynamic instructions are repetitious. Using two 256-entry memoization tables, each with 16-way associativity, we can see that less than 9% of instructions exhibit good locality. One table is used per operation type as suggested by [10]. This is summarized in table 4. We use an XOR of the most significant bits in the mantissas of the operands to index into separate tables for add and multiply. In [10], the SPEC CPU2000 benchmarks were shown to have a much higher hit rate of 20-30%, and even use a smaller table size.

When memoizing precision reduced values, however, there is certainly the immediate gain from the need to store fewer bits per entry. But, there is also an increase in coverage – with reduced precision, there are fewer possible

Attachment - Page A191

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

values and combinations to memoize. Values that are very close will become one value with reduced precision, potentially combining multiple table entries into one. By leveraging reduced precision, we see that the memoization hit rate for FP multiply increases to the level seen for SPEC benchmarks. All memoization results are shown in Table 4. Trivializable operations are filtered from accessing the memoization tables.

4.3.4 Lookup Table

Although precision reduction significantly improves the average hit-rate of FP multiply memoization, the data points to the fact that the increased hit-rate is due to the reduction in the range of possible operand values. Table 4 shows that precision reduction significantly improve the memoization hit-rates of *Continuous*, *Deformable*, *Highspeed*, and *Ragdoll*. All of these require 5 or less mantissa bits. With n-bit mantissa operands, the number of possible unique operations is $2^n \times 2^n = 2^{2n}$. For a 4-bit or 3-bit mantissa, the 256-entry memoization table can store all possible operand pairs resulting in 100% coverage. For a 5-bit mantissa, this table can store 25% of all possible combinations. As the minimum required precision increases to 7-bit and above, precision reduction fails to improve hit-rates significantly.

This behavior leads us to propose the use of a lookup table for computing narrow-width FP add and mult operations. By using a lookup table instead of memoization tables, area and latency are significantly improved, as shown by Table 5. Instead of storing information about dynamic instruction execution, the lookup table can be populated at boot time. During run-time, no writes to the lookup table are necessary. The index is 11 bits for the 2K entry table, and the most significant bit denotes the type of FP operation (add vs mult). The output of the table is used as the mantissa of the result.

For FP multiply lookup, the concatenation of the reduced operands is used to index the table. If the minimum precision is less than 5 bits, we can still use 5 bits from each operand to index the table for a more accurate result.

For FP add lookup, the smaller operand is first shifted based on the exponent difference. This operation only requires a small and fast 5-bit shifter. This means that the implicit 1 of the smaller operand will be seen. Then, the concatenation of the larger operand and the shifted operand is used as the index. In FP add lookup entries, an additional bit is stored to indicate the need to increment the exponent by 1 for normalization of a carry out. Since we allocate 8 bits per entry, there is room for such annotation. One corner case for FP add is when the two operands have the same exponent. In this case, indexing with the default scheme will result in aliasing with a smaller operand which has been shifted. One solution is to first detect this by an exponent difference of zero, then appropriately handle the most significant bit after the leading one. This solution does not require any changes to the default scheme.

With the lookup table, there is no tag and only a single rd/wr port is required since the table contents are not modified during execution – all values are preloaded. The memoization table requires a read and a write port to support the simultaneous read of an issuing operation and the writing of result from the FPU calculation. However, despite this, we give an advantage to the memoization tables by only charging it for a single rd/wr port in our area/energy estimation, even though we simulate a table with two ports.

All data presented is generated using Cacti 3.0 [30]. For the memoization tables, we use 64 address bits and 32 data bits. For the lookup table, we use 11 address bits and 8 data bits – and disregard the contribution of tag area. For benchmarks requiring precision less than 6 bits, the 2K-entry lookup table (entries are 1B each) achieves a better hit-rate than using the two (one for each operation type) 256 entry 16-way memoization tables (entries at 12B each) we simulate. With the lookup table, the area requirement is reduced by 77%. We assume a single cycle latency for lookup. We implement the lookup table in dedicated scratch-pad memory. When the required precision exceeds five bits, the lookup table is no longer useful and the memory can be used for other purposes. Detailed evaluation of this additional opportunity is left for future work.

Table Type	Latency (ns)	Energy (nJ)	Area (mm^2)
Lookup	0.40	0.03	0.08
Memo	0.88	0.73	0.35

Table 5. Lookup vs memoization table

5 FPU Sharing

Precision reduction, including its effect on trivialization and the opportunity of arithmetic lookup, provides us with the opportunity to avoid or simplify work done by the FPU. In this section, we will exploit this opportunity to facilitate sharing of a single full-precision floating-point unit (FPU) among a number of fine-grain (FG) cores in the ParallAX architecture. The overall goal here is to reduce the area footprint of each FG core, enabling us to bring even more cores to bear for a given silicon budget.

We use simple, in-order shader-class cores with the parameters shown in Table 6 in this study, and assume a core area excluding the floating-point unit (FPU) of $2mm^2$ in 90nm technology. Published data on FPU area ranges widely within the same technology due to parameters such as the design flow, target latency, area allocation, and circuit optimization. To make this exploration more gener-

Attachment - Page A192

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

ally applicable, we explore four FPU designs with varying area requirements: $1.5mm^2$, $1.0mm^2$, $0.75mm^2$, and $0.375mm^2$. These area estimates are based respectively on results from a high-level synthesis tool tuned for generating area-efficient floating point data-paths, and published data from [18], [10], and [21].

Processor Pipeline	1-wide, 5-stages, in-order execution
Functional Units	1 int, 1 fp, 1 ld/st
Window/Scheduler	8,4
Branch Predictor	384B YAGS + 4-entry RAS
Local Inst Memory	4KB, 1-cycle
Local Data Memory	4KB, 1-cycle
Technology	90 nm
Clock Frequency	1GHz
FP Latencies	fpALU (4) fpMult (4) fpDiv (20)
INT Latencies	iALU (1) iMult (6) iDiv (40)

Table 6. Fine-grain shader core design

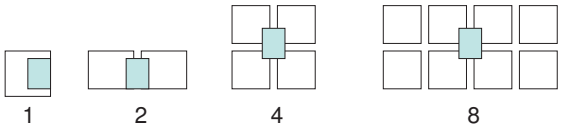


Figure 2. Layout for one, two, four, and eight cores sharing an FPU.

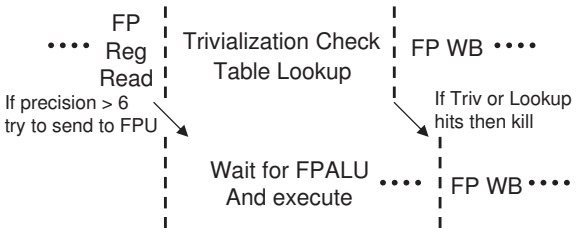


Figure 3. Pipeline for FPU sharing.

The layouts for our assumptions on sharing 2, 4, and 8 cores are shown in figure 2. We adopt a simple policy for arbitration to minimize latency – the cores simply take turns accessing the FPU on alternating cycles for pipelined operations. So when a single FPU is shared among N cores, a given core will get access to the FPU once every N cycles. If the core does not require the FPU in that cycle, the opportunity to use the FPU is wasted. For long latency non-pipelined FP operations such as divide, we assume alternating 3-cycle scheduling windows for each core as described in [18]. These operations consist of a small percentage of total FP operations for physics simulation.

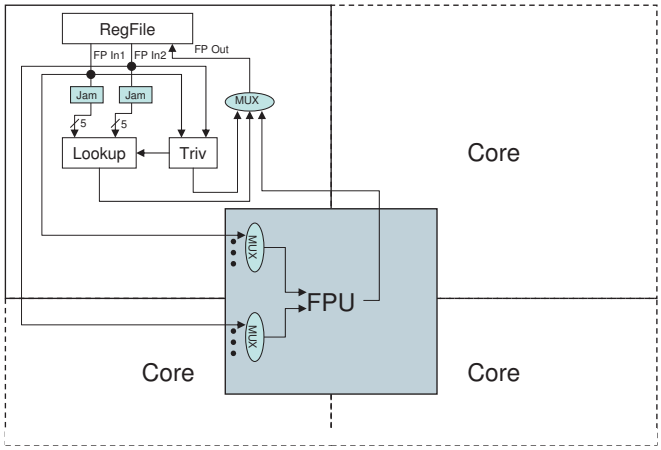


Figure 4. Four cores sharing a single FPU – each core has a trivialization and memoization unit.

For sharing among 2 cores, 4 cores and 8 cores, we assume 0-cycle, 1-cycle, and 2-cycle latency penalties. Based on mirroring of cores as described in [17, 18], sharing the FPU among 2 cores can be accomplished with no added latency. For the 4 core and 8 core sharing, we add conservative latency estimates for the additional wire delay from each core to the FPU. Using a $2mm \times 2mm$ aspect ratio for the cores, we assume a 2mm wire length for 4-core sharing and a 4mm wire length for 8-core sharing. Based on the 90nm information for the narrowest wire width in [9], the one-way added delays are 0.074 ns and 0.296 ns respectively. In section 5.2, we will evaluate the performance sensitivity to added FPU latency.

5.1 Hierarchical FPUs

To attack the overhead of sharing, we propose a hierarchical FPU (HFPU) architecture. A small, simple L1 FPU is placed local to each core for low latency and high throughput, and the full precision FPUs are shared at the L2 level to maximize area efficiency and to satisfy high-precision execution. This is similar in concept to a cache hierarchy, but we are sharing execution resources instead of storage resource. If the required level of precision for a particular operation exceeds the capability of the L1 FPU, then the operation must be performed in the L2 FPU. While it may be possible to use software support [12] to execute in the L1 FPU, we do not consider this alternative in this study.

We propose and evaluate the following L1 FPU design alternatives. All L1 designs contain logic to handle simple non-arithmetic FP operations. If a given operation exceeds the capability of the L1 FPU, it will be executed in the L2

Attachment - Page A193

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

FPU. We list the alternatives by increasing complexity of the L1 FPU design.

1. *Conventional Trivialization:* The L1 FPU only contains conventional trivialization logic – no precision reduction.

2. *Reduced Precision Trivialization:* The L1 FPU uses our reduced precision trivialization logic. Additional exponent logic is required.

3. *Lookup Table + Reduced Precision Trivialization:* The L1 FPU has our reduced precision trivialization logic, exponent logic, and a local 2K-entry lookup table. FP add and multiply instructions using mantissa precision of less than six bits make use of the lookup table. Figure 4 demonstrates a version of this design where four cores share a single L2 FPU. Each core has an L1 FPU with trivialization and lookup table hardware. The pipeline change required for this FPU sharing is illustrated in figure 3.

4. *mini-FPU + Reduced Precision Trivialization:* In addition to our reduced precision trivialization logic, this L1 FPU has a *mini-FPU*: an FPU design that uses a 14-bit mantissa and 8-bit exponents. FP add and multiply operations with mantissa precision of less than 15 bits can execute in this mini-FPU. The 14-bit precision is chosen based on the (benchmark coverage)/(area) ratio derived from Table 1. The intent of this design is to provide potentially higher coverage at the L1 FPU, but at a greater area cost.

Required Mechanism	Latency (cycles)
Trivialization or Look-up Table	1
mini-FPU	3
Full FPU Arbitration	0-1 for 2-core sharing 0-3 for 4-core sharing 0-7 for 8-core sharing
Interconnect Overhead	0 for 2-core sharing 1 for 4-core sharing 2 for 8-core sharing
Full FPU Latency	4 for fpALU and fpMult 20 for fpDiv

Table 7. Variable FP Latency

5.1.1 Variable FP Latency

All components of HFPU's variable FP latency are listed in Table 7. If an operation can be satisfied by trivialization or look-up table, it takes one cycle to complete. If the mini-FPU is available and the operation can be satisfied by it, the operation takes three cycles. If an operation requires the full FPU, the exact latency is determined by accumulating the required latencies of arbitration, interconnect overhead, and full FPU operation.

While the latency for each operation is variable, the latency of a non-trivial operation is known at issue time. This is accomplished by using a local counter to indicate current round-robin arbitration overhead. The interconnect overhead and FPU latencies are fixed.

For the simple in-order, single-issue cores we modeled, there is no dynamic instruction scheduler. Instructions are dispatched in program order, so instruction scheduling is done at compile time. SESC uses the MIPS-ISA, and we compile with the -mips2 option without targeting a specific microarchitecture. We do not leverage the fact that FP op might be single cycle, and the same binary is used for all simulations. If the operation is satisfied by the trivial or look-up table logic, then the operation completes in 1 cycle. If not, the pipeline stalls until the operation is completed. Since the cores are in-order single-issue cores, the performance impact of these stalls is not significant. In the case of the look-up table, if the required precision can be satisfied by the look-up table, 100% of operations sent to the look-up table will be satisfied.

For more complex cores with dynamic instruction schedulers, different implementations are possible:

1. Optimistically assumes 1 cycle latency. If the operation is not satisfied by the trivial or look-up table logic, the pipeline stalls until the operation is completed. Once an operation is detected to require the full FPU, the actual number of required cycles is known.

2. Pessimistically assumes the use of the full FPU. The required number of cycles is known at beginning of issue. This scheduling policy eliminates stalls/flushes/replays and retains the energy savings shown in the paper.

5.2 Results

For our performance simulations, we assume a baseline ParallAX architecture with 128 simple (fine-grain) cores, as described in Table 6, connected by a mesh interconnect. Four different FPU designs with varying area requirements (1.5 mm^2, 1.0 mm^2, 0.75 mm^2, and 0.375 mm^2) are evaluated. Area estimates include per core area (2 mm^2 each), per core mesh interconnect routers (0.19 mm^2 each) [31], the full-precision FPUs, and additional logic and structures for different L1 FPU designs. The details of each configuration are shown in Table 8.

For the 14-bit mantissa mini-FPU, we assume an area requirement of 60% that of the full precision 23-bit mantissa FPU. This ratio is based on the estimates of both prior work on reduced precision FPU [11] and results from a high-level

Attachment - Page A194

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

synthesis tool tuned for generating area-efficient floating point data-paths. All area data is for $90nm$ technology.

Architecture	Area Overhead Per Core (mm^2)	Avg Per Core IPC 4 Cores Per L2-FPU Narrowphase, LCP
Baseline (Conjoin)	—	0.347, 0.293
Conv Triv	0.0023	0.376, 0.319
Reduced Triv	0.0079	0.377, 0.334
Reduced Triv Lookup Table	0.0079 + 0.080	0.377, 0.357
Reduced Triv mini-FPU (14bit)	0.0079 + (0.6 × FP Area)	0.382, 0.364

Table 8. Evaluated designs. Conjoin refers to baseline with FPU sharing. Area overhead per core = additional area added to the 2 mm^2 core.

First, we consider the lower area cost alternatives, and then we will address the mini-FPU. Figures 5(a) (*LCP*) and 5(b) (*Narrow-phase*) demonstrate the aggregate improvement in throughput over the baseline configuration with no FPU sharing for LCP and narrow-phase respectively. The four clusters along the x-axis represent the different FPU area sizes we evaluate, and each cluster has four data points for each architecture representing sharing a single L2 FPU among one, two, four, and eight cores.

Note that any area saved by FPU sharing is used to add more fine-grain cores, taking into account the interconnect and core area as above. Therefore, two competing trends will influence performance: (1) an increase in the number of cores will mean more exploitation of parallelism and (2) an increase in FPU sharing may mean worse per-core performance due to the overhead of sharing. Since our baseline has 128 cores, the total area available varies with the assumed size of the FPU. For the four FPU sizes we evaluate, the total die area is: 472 mm^2 for the 1.5 mm^2 FPU, 408 mm^2 for the 1.0 mm^2 FPU, 376 mm^2 for the 0.75 mm^2 FPU, and 328 mm^2 for the 0.375 mm^2 FPU. Figure 6 shows the total number of cores available to each configuration in the same die area as that of the appropriate baseline configuration with 128 cores.

The data across the two phases from figures 5(a) and 5(b) shows two attractive configurations: *Reduced Precision Trivialization* is better for *Narrow-phase* and *Lookup Table + Reduced Precision Trivialization* is better for *LCP*. As shown by Table 8, the per core IPCs of the two configurations for *Narrow-phase* are the same. The slight degradation with Lookup is due to the lookup table area and the fact that *Narrow-phase* always requires more than five bits of precision – therefore the lookup table is not used. To choose the better design, we also evaluate the energy and power impact of these two HFPU designs.

To estimate the total dynamic energy consumed by FP operations, we leverage the energy data for different subunits of the FPU presented in [10]. For configurations with trivialization, all FP operations are charged the trivialization logic energy. Non-trivial operations are then charged for the FPU energy. The lookup table is activated when the required precision falls below six bits. In these cases, all FP operations are charged the trivialization plus the lookup energies. The diamonds (secondary y-axis) of Figure 6(b) show a clear energy advantage to using the lookup table for *LCP*.

While performance is improved by sharing FPUs, increased per-FPU utilization could worsen the existing *hot spot* power density problem of FPUs [17]. To gauge the per-FPU utilization, we show the % of FP operations that are trivialized by trivialization or table lookup in the bars (primary y-axis) of Figure 6(b). The HFPU design trivializes 53% of FP operations in the FP intensive phase of *LCP*. This suggests that the HFPU can lower per-FPU utilization when sharing one FPU between two cores as compared to the unshared baseline. For sharing among *four* cores with HFPU, per-FPU utilization will be similar to that of the prior work on conjoined CMPs [18] sharing only among *two* cores.

Next we compare the mini-FPU design to our best performing low-overhead L1 FPU. Figures 7(a) and 7(b) demonstrates the improvement in throughput over the 128-core baseline case for these designs – with figure 6 demonstrating the number of cores possible with each architecture. The mini-FPU design provides some improvement over the baseline, but the area overhead of the L1 FPU reduces the performance of this approach in comparison with the *Lookup Table + Reduced Precision Trivialization* L1 FPU.

Physics requires varying amount of numerical accuracy across different phases and scenarios – a clear differentiating factor between the error tolerance of physics and media processing. In the worst case, full precision is required. Therefore the mini-FPU cannot replace a full precision FPU in a physics processor architecture – there must be the ability to do full-precision FP. This makes the mini-FPU design inherently more costly, and argues for possibly sharing L1 FPUs among cores as well. We look at the case where the mini-FPU is shared among either two or four cores. We limit our exploration to configurations where the L2 FPU is shared by at least as many cores as the L1 FPU. For the Figures, *mini-FPU* refers to no mini-FPU sharing. *mini-FPU 2* shared one mini-FPU among 2 cores, and *mini-FPU 4* shared one mini-FPU among 4 cores. As before, the x-axis (*Cores per FPU*) refers to the degree of L2 FPU sharing.

When looking at the per core throughput in Table 8, the 14-bit mantissa mini-FPU has the highest IPC. This is a result of a 1-cycle reduction in latency and broad coverage of benchmarks (14-bit mantissa satisfies 7/8 benchmarks for

Attachment - Page A195

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

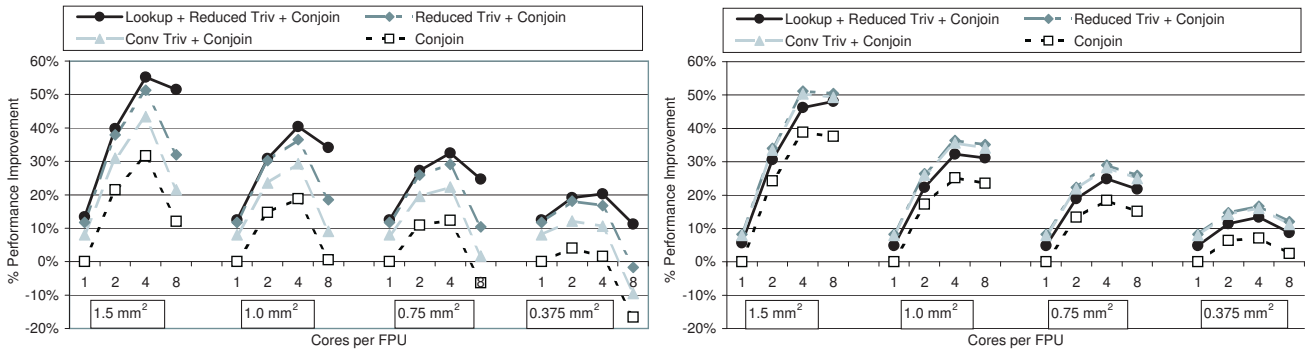


Figure 5. (a) HFPU LCP performance — (b) HFPU narrow-phase performance

LCP and 4/8 benchmarks for *Narrow-phase*). However, the mini-FPU simply cannot pack as many cores into a given silicon area due to its area overhead (as shown in figure 6), resulting in a lower overall throughput. The mini-FPU designs only become more attractive for the most aggressive FPU design ($0.375mm^2$) at the highest level of sharing.

As shown by the performance and energy data, the best HFPU design we examined when considering both performance and energy is to share one full-precision FPU among 4 cores using the *Lookup Table + Reduced Precision Trivialization* L1 FPU. On average, HFPU improves *LCP* performance over the baseline by 55%, 40%, 33%, and 20% for the four FPU designs. On average, HFPU improves *Narrow-phase* performance over the baseline by 46%, 32%, 25%, and 13%. Total dynamic energy expended for FP operations is reduced by 50% for *LCP* and 27% for *Narrow-phase*.

There are area and timing overheads when sharing a resource like the FPU among multiple cores regardless of the L1 FPU design – these come from the arbitration logic required to choose which core will access the shared resource and the interconnect cost to connect the cores and resource together. We evaluated the performance sensitivity to increased latency from sharing in Figure 8. The baseline for these figures is the performance of the *Lookup Table + Reduced Precision Trivialization* sharing one FPU among two cores. *LCP* (a) is shown to be more sensitive than *Narrow-phase* (b) in this figure. For *LCP*, the performance of the more aggressively designed FPUs suffers when the latency increases beyond a single cycle.

6 Summary and Future Work

In this study, we exploit the inherent perceptual error tolerance of physics-based animation to dynamically adapt the required precision for floating-point computation. Then, we identify and leverage the benefits made possible by dynamic precision tuning to design a hierarchical floating-point unit (HFPU). This consists of an area-efficient L1 FPU and a full-precision L2 FPU that is shared among a number of cores in the ParallAX architecture. Our best performing L1 FPU design consists of our enhanced trivialization logic and a mantissa lookup table – an area efficient design that allows more cores to be packed into a particular silicon area. For the range of FPU designs we consider, this L1 FPU coupled with an L2 FPU that is shared among four cores provides an improvement in overall throughput up to 55% and reduces energy expended in the FPU by up to 50%.

7 Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. Intel Corp., Microsoft Corp., AMD/ATI Corp., and AGEIA Technologies Corp. helped us with their generous support through equipment and software grants. This work was partially supported by NSF grants CCF-0429983 and CCF-0133997, and by SRC grant 1317. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or SRC.

References

[1] Open dynamics engine. http://www.ode.org/ode-latest-userguide.html.

[2] Physicsbench. http://sourceforge.net/projects/physicsbench/.

[3] AGEIA. Physx product overview. www.ageia.com.

[4] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. In *IEEE Transactions on Computers*, 2005.

[5] E. Atoofian and A. Baniasadi. Improving energy-efficiency by bypassing trivial computations. In *The 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

Attachment - Page A196

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

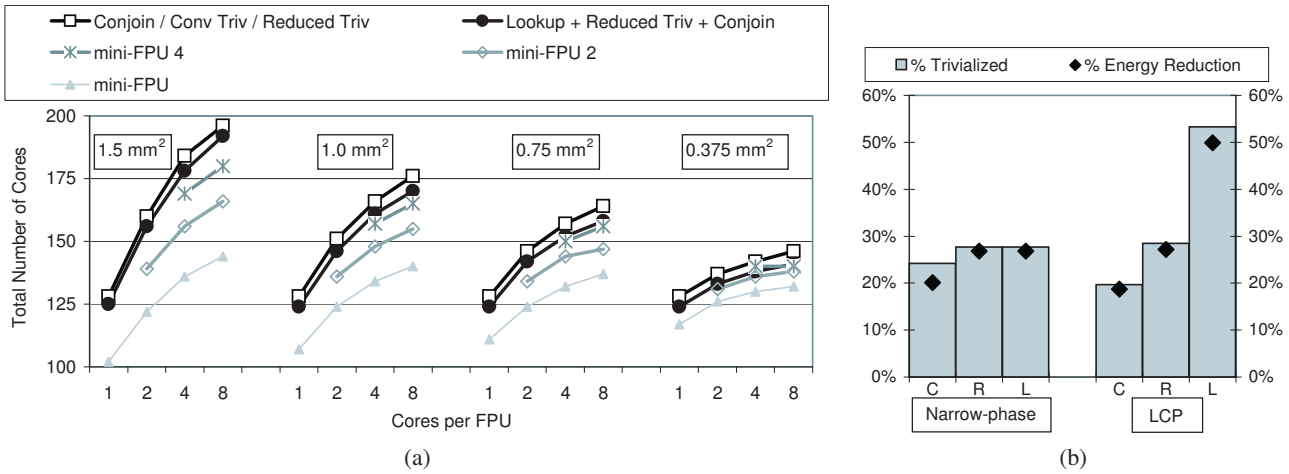


Figure 6. (a) Total number of cores in the same die area as 128-core baseline. The total area used: 472 mm^2 for 1.5 mm^2 FPU, 408 mm^2 for 1.0 mm^2 FPU, 376 mm^2 for 0.75 mm^2 FPU, and 328 mm^2 for 0.375 mm^2 FPU. — (b) FP computation % trivialized and energy reduction. C = Conv Triv, R = Reduced Triv, and L = Lookup Table + Reduced Triv.

[6] D. Baraff. *Physically Based Modeling: Principals and Practice*. SIGGRAPH Online Course Notes, 1997.

[7] R. Barzel, J. Hughes, and D. Wood. Plausible motion simulation for computer graphics animation. In *Computer Animation and Simulation*, 1996.

[8] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronics computing instrument. *Computer Structures: Reading and Examples, McGraw-Hill Inc.*, 1971.

[9] G. Chen, H. Chen, M. Haurylau, N. A. Nelson, D. H. Albonesi, P. M. Fauchet, and E. G. Friedman. On-chip copper-based vs. optical interconnects: Delay uncertainty, latency, power, and bandwidth density comparative predictions. In *IEEE International Interconnect Technology Conference*, 2006.

[10] D. Citron and D. G. Feitelson. Look it up or do the math: An energy, area, and timing analysis of instruction reuse and memoization. In *Workshop on Power-Aware Computer Systems*, 2003.

[11] F. Fang, T. Chen, and R. Rutenbar. Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform. *EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication*, 2002.

[12] D. Goddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in fem simulations. In *International Journal of Parallel, Emergent and Distributed Systems*, 2006.

[13] Havok. http://www.havok.com/content/view/187/77.

[14] P. Hofstee. Power efficient architecture and the cell processor. In *HPCA11*, 2005.

[15] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y. Chen. Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors. In *34th Annual International Symposium on Computer Architecture*, 2007.

[16] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara, a 32-way multithreaded sparc processor. In *IEEE Micro*, 2005.

[17] K. Krewell. Ultrasparc iv mirrors predecessor. In *Microprocessor Report*, 2003.

[18] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *The 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[19] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[20] M. Matthias, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. In *Proceedings of the 3rd Workshop in Virtual Reality Interactions and Physical Simulation*, 2006.

[21] H. Oh, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong. A fully pipelined single-precision floating-point unit in the synergistic processor element of a cell processor. In *IEEE Journal of Solid-State Circuits*, 2006.

[22] K. Olukoton, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII*, 1996.

[23] C. O'Sullivan, S. Howlett, R. McDonnell, Y. Morvan, and K. O'Conor. Perceptually adaptive graphics. In *Eurographics 2004, State of the Art Report*, 2004.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

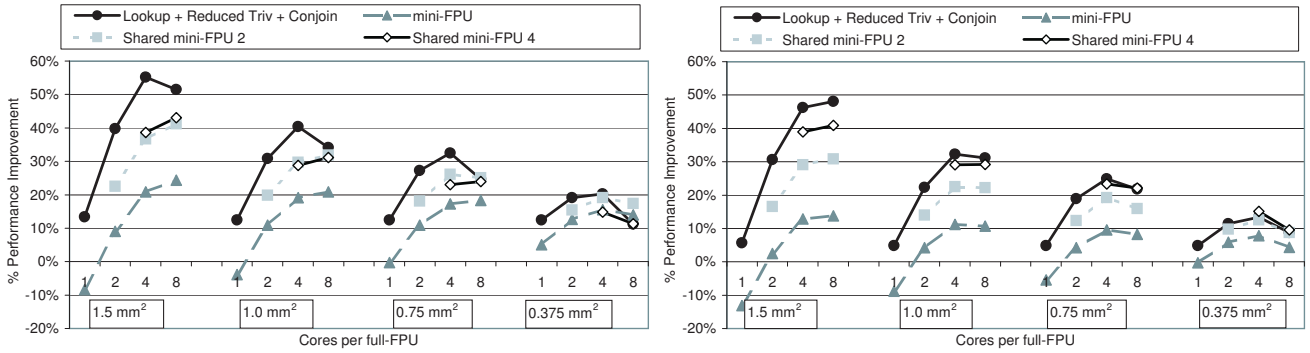


Figure 7. (a) mini-FPU LCP performance — (b) mini-FPU narrow-phase performance

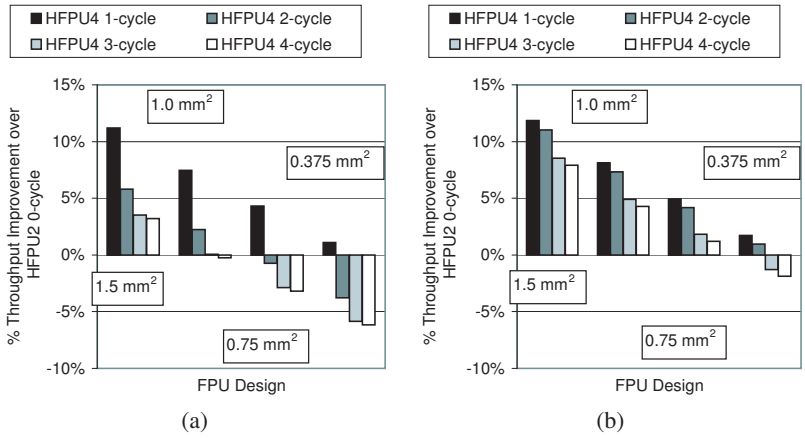


Figure 8. (a) LCP latency sensitivity — (b) Narrow-phase latency sensitivity

[25] S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. In *IEEE Symposium on Computer Arithmetic*, 1993.

[26] S. E. Richardson. Exploiting trivial and redundant computation. 1993.

[27] A. Seugling and M. Rolin. Evaluation of physics engines and implementation of a physics module ina 3d-authoring tool. In *Master's Thesis*, 2006.

[28] A. Shayesteh, G. Reinman, N. Jouppi, S. Sair, and T. Sherwood. Improving the performance and power efficiency of shared helpers in cmps. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.

[29] D. Sheahan. Developing and tuning applications on ultrasparc t1 chip multithreading systems. In *Sun Blue Prints Online*, 2007.

[30] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Compaq WRL 2001/2, 2001.

[31] V. Soteriou, N. Eisley, H. Wang, B. Li, and L. Peh. Polaris: A system-level roadmap for on-chip interconnection networks. In *Proceedings of the 24th International Conference on Computer Design*, 2006.

[32] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman. Parallax: An architecture for real-time physics. In *The 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[33] T. Y. Yeh, P. Faloutsos, and G. Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Video Game Symposium*, 2006.

[34] T. Y. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. In *2007 ACM Transactions on Graphics (TOG) – accepted with major revisions. http://www.cs.ucla.edu/ tomyeh/tog07.pdf*.

[35] J. Yi and D. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *IEEE International Conference on Computer Design: VLSI in computers and Processors (ICCD)*, 2002.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 07,2022 at 16:12:47 UTC from IEEE Xplore. Restrictions apply.

ATTACHMENT 7C

getPDF.jsp
1 / 13
100%

40th IEEE/ACM International Symposium on Microarchitecture

The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration

Thomas Y. Yeh†
†Comput
ren Reinman†
cla.edu

Document properties

File name: getPDF.jsp
File size: 4.7 KB

Title: –
Author: –
Subject: –
Keywords: –
Created: 11/11/07, 5:23:21 PM
Modified: 11/25/22, 12:51:09 PM
Application: –

PDF producer: GNU Ghostscript 7.07; modified using iText® 7.1.1...
PDF version: 1.3
Page count: 11
Page size: 850 × 1100 in (portrait)

Fast web view: No

Close

Physics-based ani
prove the realism of
namic, immersive co
parallel nature of phy
parallelism to reach
significant area so pl
tunately, interactive
rather than accuracy
physics has a remark
of the significand bi
this paper, we descri
cal floating-point unit
sion reduction to enab
tiple cores. This share
cores, thereby allowin
area and exploiting m

1 Introduction

Physics-based anim[illegible] becoming one of the most important elements of interactive entertainment applications, such as computer games, largely because of the automation and realism that it offers. The immersive nature of interactive entertainment relies on rapid, dynamic content creation and realistic visual effects. However, the benefits of PBA come at a considerable computational cost, and the demands of interactive entertainment require soft performance bounds of 30-60 frames per second. Fortunately, the physical simulation of complex scenes is massively parallel in nature. Exploiting this parallelism is an active area of research both in terms of software techniques and hardware accelerators. Commercial physics-specific solutions [3, 13] and modern processors [14] already take advantage of the parallel nature of PBA with varying levels of success.

With such massive parallelism, chip multiprocessor [22]

[illegible]cked into a single
[illegible]t can be exploited.
[illegible]sive floating-point
[illegible]cated floating-point
[illegible]. If these resources
[illegible]ple cores, the area
[illegible]nsity and therefore
[illegible]ever, contention for
[illegible]ance depending on

[illegible]ust massive paral-
[illegible]eving high perfor-
[illegible]olerant – interac-
[illegible] believable results
The user must be
[illegible]mmersed in the vir-
[illegible]ace to small errors
[illegible]f error tolerance in
[illegible]the simulation and
[illegible]. [34] has demon-
[illegible]tial energy during a
[illegible]ily. We propose to
[illegible] fine the precision
of FP operations in PBA to provide just enough precision to approximately conserve simulation energy, and therefore ensure believability.

Our paper makes the following contributions:

- We propose dynamic FP precision tuning in PBA to leverage perceptual error tolerance.
- We identify and leverage three main benefits of precision reduction:
  1. *Increased Number of Trivial Operations:* Precision reduction enables new conditions and increased chances that a FP computation will not require the use of an FPU – such operations are

ATTACHMENT 8A

WorldCat: Multicore processors and systems

OCLC	401151119	**No holdings in LDL - 34 other holdings**									
Books	**Rec Stat**	c		**Entered**	20090618		**Replaced**	20220614200636.1			
Type	a	**ELvl**	7	**Srce**		**Audn**		**Ctrl**		**Lang**	eng
BLvl	m	**Form**		**Conf**	0	**Biog**		**MRec**		**Ctry**	nyu
		Cont	b	**GPub**		**LitF**	0	**Indx**	1		
Desc	a	**Ills**	a	**Fest**	0	**DtSt**	s	**Dates**	2009,		

010		2009930936
040		UKM ‡b eng ‡c DLC ‡d UKM ‡d BTCTA ‡d BWX ‡d PIT ‡d YDXCP ‡d GBVCP ‡d OCLCF ‡d OCLCQ ‡d UV1 ‡d UKMGB ‡d OCLCQ ‡d OCLCO
015		GBA962195 ‡2 bnb
016	7_	994589476 ‡2 DE-101
016	7_	015297121 ‡2 Uk
020		9781441902627 ‡q (hbk. ; ‡q alk. paper)
020		1441902627 ‡q (hbk. ; ‡q alk. paper)
042		lccopycat
050	00	QA76.5 ‡b .M7933 2009
082	04	004.35 ‡2 22
090		
049		LDLL
245	00	Multicore processors and systems / ‡c edited by Stephen W. Keckler, Kunle Olukotun, H. Peter Hofstee.
260		New York : ‡b Springer, ‡c 2009.
300		xviii, 300 pages : ‡b illustrations ; ‡c 24 cm
336		text ‡b txt ‡2 rdacontent
337		unmediated ‡b n ‡2 rdamedia
338		volume ‡b nc ‡2 rdacarrier
490	1_	Integrated circuits and systems
504		Includes bibliographical references and index.
540		British Library not licensed to copy ‡c 0. ‡5 Uk
650	_0	Multiprocessors.
650	_6	Multiprocesseurs. ‡0 (CaQQLa)201-0062858
650	_7	Multiprocessors. ‡2 fast ‡0 (OCoLC)fst01029068
700	1_	Keckler, Stephen W.
700	1_	Olukotun, Oyekunle Ayinde.
700	1_	Hofstee, H. Peter.
830	_0	Series on integrated circuits and systems.
938		Baker and Taylor ‡b BTCP ‡n BK0011835643
938		YBP Library Services ‡b YANK ‡n 3072690
029	1_	AU@ ‡b 000044895956
029	1_	NZ1 ‡b 13629150
029	1_	UKMGB ‡b 015297121

Action Status Delete Holdings _ Export _ Label _ Produce _ Update Holdings _ Validate _

ATTACHMENT 8B

All locations : No holdings in LDL - 34 other holdings

Multicore processors and systems / edited by Stephen W. Keckler, Kunle Olukotun, H. Peter Hofstee. New York : Springer, 2009. xviii, 300 pages : illustrations ; 24 cm OCLC:401151119

Holdings Information

STATE	LOCATIONS
CA	CIT MERUC
DC	DLC
IL	IAI JNA
KS	KSW
MA	HUL
MD	hagcc MUB
MO	ELW UMS
NC	btcta NDD
NH	ydx
NY	COO
OR	bwx
PA	pit
SC	SFU
TX	RCE TXU UHC
VA	VGM
WA	oej
AU	lt1 RM0
GB	BRI BUB ukm
IL	da$
MY	mytmm
NZ	UV1
SE	s3o
TH	culib
TW	thn

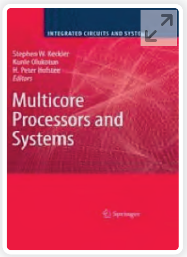
CLOSE WINDOW

ATTACHMENT 8C

Springer Link

Search Log in

Download book PDF



Multicore Processors and Systems pp 73–109

Composable Multicore Chips

Doug Burger, Stephen W. Keckler & Simha Sethumadhavan

Chapter | First Online: 01 January 2009

1699 Accesses

Part of the Integrated Circuits and Systems book series (ICIR)

Abstract

When designing a multicore chip, two primary concerns are how large and powerful to make each core and how many cores to include. A design with a few large cores is attractive for general purpose computing that has coarse-grained threads which depend on instructional-level parallelism for performance. At the other end of the processor granularity spectrum, a large collection of small processors is suited to applications with ample thread-level parallelism. While much of the spectrum

between large cores and tiny cores is being pursued by different vendors, each of these fixed architectures are suited only to the range of workloads that maps well to its granularity of parallelism. An alternative is a class of architectures in which the processor granularity (and the number of processors) can be dynamically configured–at runtime. Called composable lightweight processors (CLPs), these architectures consist of arrays of simple processors that can be aggregated together to form larger more powerful processors, depending on the demand of the running application. Experimental results show that matching the granularity of the processors to the granularity of the tasks improves both performance and efficiency over fixed multicore designs.

Keywords

Direct Memory Access **Data Tile**

Block Address **Register Tile**

Logical Processor

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

> Portions reprinted, with permission, from "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," K. Sankaralingam, R.

> Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, *International Symposium on Microarchitecture* © 2006, IEEE. and "Composable Lightweight Processors," C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, *International Symposium on Microarchitecture* © 2007, IEEE.

Download chapter PDF

References

1. M. Annavaram, E. Grochowski, and J. P. Shen. Mitigating Amdahl's law through EPI throttling. In *International Symposium on Computer Architecture*, pages 298–309, June 2005.

2. S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *International Solid-State Circuits Conference*, pages 88–89, Feb 2008.

3. D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7), 2004.

4. J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *International Solid-State Circuits Conference*, pages 102–103, Feb 2007.

5. J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti. Design of the Power6 microprocessor. In *International Solid-State Circuits Conference*, pages 102–103, Feb 2007.

6. M. Gebhart, B. Maher, P. Gratz, N. Ranganathan, J. Diamond, B. Robatmili, S. Govindan, S. W. Keckler, D. Burger, and K. McKinley. TRIPS System Evaluation. Technical Report TR-08-31, The University of Texas at Austin, Department of Computer Sciences, Sep 2008.

7. S. Ghiasi and D. Grunwald. Aide de Camp: Asymmetric Dual Core Design for Power and Energy Reduction. Technical Report CU-CS-964-

03, The University of Colorado, Department of Computer Science, 2003.

8. E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *International Conference on Computer Design*, pages 236–243, Oct 2004.

9. T. R. Halfhill. Floating point buoys clearspeed. *Microprocessor Report*, 17(11):19–24, Nov 2003.

10. T. Ibaraki and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, 1988.

11. E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez. Core fusion: Accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, June 2007.

12. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), July 2005.

13. R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

14. B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. J. Dally. A Programmable 512 GOPS stream processor for signal, image, and video processing. In *International Solid-State Circuits Conference*, pages 272–273, Feb 2007.

15. C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct 2002.

16. C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture*, pages 381–394, Dec 2007.

17. R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *International Symposium on Microarchitecture*, pages 81–92, Dec 2003.

18. R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *International Symposium on Computer Architecture*, pages 64–75, June 2004.

19. F. Latorre, J. González, and A. González. Back-end assignment schemes for clustered multithreaded processors. In *International Conference on Supercomputing*, pages 316–325, June 2004.

20. E. Lindholm and S. Oberman. The NVIDIA GeForce 8800 GPU. In *Proceedings of the Symposium on High Performance Chips (HotChips-19)*, August 2007.

21. B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *International Symposium on Microarchitecture*, pages 480–491, Dec 2006.

22. M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *International Symposium on Computer Architecture*, pages 46–56, June 2007.

23. S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *Workshop on Microprogramming and Microarchitecture*, pages 60–63, Nov 1988.

24. U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1), Jan 2008.

25. R. M. Rabbah, I. Bratt, K. Asanović, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report MIT-LCS-TM-646, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, June 2004.

26. P. Racunas and Y. N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *International Conference on Supercomputing*, pages 22–31, June 2003.

27. P. Salverda and C. Zilles. Fundamental performance challenges in horizontal fusion of In-order cores. In *International Symposium on*

High-Performance Computer Architecture, pages 252–263, Feb 2008.

28. K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture*, pages 422–433, June 2003.

29. K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *International Symposium on Microarchitecture*, pages 480–491, Dec 2006.

30. S. Sethumadhavan, R. McDonald, R. Desikan, D. Burger, and S. W. Keckler. Design and implementation of the TRIPS primary memory system. In *International Conference on Computer Design*, pages 470–476, Oct 2006.

31. S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: Enabling unordered load – store queues. In

International Symposium on Computer Architecture, pages 347–357, June 2007.

32. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Sep 2001.

33. A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, Mar 2006.

34. A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *International Symposium on Microarchitecture*, pages 89–100, Dec 2006.

35. A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Nov 2000.

36. B. Stackhouse, B. Cherkauer, M. Gowan, P. Gronowski, and C. Lyles. A 65 nm 2-billion-transistor quad-core Itanium processor. In *International Solid-State Circuits Conference*, pages 92–93, Feb 2008.

37. D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-Order Execution using Simple In-Order Cores. Technical Report CS-2007-11, University of Virginia, Department of Computer Science, Aug 2007.

38. M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *International Symposium on Computer Architecture*, pages 2–13, June 2004.

39. M. Tremblay and S. Chaudhry. A third-generation 65 nm 16-core 32-thread plus 32-scout-thread CMT SPARC Processor. In *International Solid-State Circuits Conference*, pages 82–83, Feb 2008.

40. S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100 W TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, Jan 2008.

41. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

Acknowledgments

The authors wish to thank the entire TRIPS hardware team for their contributions to the design and implementation of the TRIPS and TFlex architectures. The team members include: Rajagopalan Desikan, Saurabh Drolia, Mark Gebhart, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Robert McDonald, Ramadass Nagarajan, Nitya Ranganathan, Karthikeyan Sankaralingam, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. The TRIPS project was supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

Author information

Authors and Affiliations

The University of Texas at Austin, 1 University Station C0500, 78712-0233, Austin, TX, USA

Doug Burger & Stephen W. Keckler

Columbia University, New York, NY, USA

Simha Sethumadhavan

Corresponding author

Correspondence to Doug Burger .

Editor information

Editors and Affiliations

College of Natural Sciences, University of Texas, Austin, University Station 1, Austin, 78712-0233, U.S.A.

Stephen W. Keckler

Dept. Electrical Engineering, Stanford University, Stanford, 94305-9510, U.S.A.

Kunle Olukotun

IBM Software Group, Burnet Rd. 11501, Austin, 78758, U.S.A.

H. Peter Hofstee

Rights and permissions

Reprints and Permissions

Copyright information

© 2009 Springer-Verlag US

About this chapter

Cite this chapter

Burger, D., Keckler, S.W., Sethumadhavan, S. (2009). Composable Multicore Chips. In: Keckler, S., Olukotun, K., Hofstee, H. (eds) Multicore Processors and Systems. Integrated Circuits and Systems. Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-0263-4_3

.RIS .ENW .BIB

DOI
https://doi.org/10.1007/978-1-4419-0263-4_3

Published	Publisher Name	Print ISBN
03 August 2009	Springer, Boston, MA	978-1-4419-0262-7

Online ISBN	eBook Packages
978-1-4419-0263-4	Computer Science Computer Science (R0)

Share this chapter

Anyone you share the following link with will be able to read this content:

Get shareable link

Provided by the Springer Nature SharedIt content-sharing initiative

Not logged in - 76.84.212.16
Big Ten Academic Alliance Formerly (CIC) (3000133814) - University of Nebraska-Lincoln (3000134173)
SPRINGER NATURE

© 2022 Springer Nature Switzerland AG. Part of Springer Nature.

ATTACHMENT 8D

INTEGRATED CIRCUITS AND SYSTEMS

Stephen W. Keckler
Kunle Olukotun
H. Peter Hofstee
Editors

Multicore Processors and Systems

Springer

Integrated Circuits and Systems

Series Editor

A. Chandrakasan
Massachusetts Institute of Technology
Cambridge, Massachusetts

For further volumes:
http://www.springer.com/series/7236

Stephen W. Keckler · Kunle Olukotun ·
H. Peter Hofstee
Editors

Multicore Processors and Systems



Editors
Stephen W. Keckler
The University of Texas at Austin
1 University Station, C0500
Austin, TX 78712-0233
USA
skeckler@cs.utexas.edu

H. Peter Hofstee
IBM Systems and Technology Group
11500 Burnet Rd.
Austin, TX 78758
USA
hofstee@us.ibm.com

Kunle Olukotun
Department of Electrical Engineering and
Computer Science
Stanford University
Stanford, CA 94305-9030
USA
kunle@stanford.edu

ISSN 1558-9412
ISBN 978-1-4419-0262-7 e-ISBN 978-1-4419-0263-4
DOI 10.1007/978-1-4419-0263-4
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009930936

© Springer Science+Business Media, LLC 2009
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

While the last 30 years has seen the computer industry driven primarily by faster and faster uniprocessors, those days have come to a close. Emerging in their place are microprocessors containing multiple processor cores that are expected to exploit parallelism. The computer industry has not abandoned uniprocessor performance as the key systems performance driver by choice, but it has been forced in this direction by technology limitations, primarily microprocessor power. Limits on transistor scaling have led to a distinct slowdown in the rate at which transistor switching energies improve. Coupled with clear indications that customers' tolerance level for power per chip has been reached, successive generations of microprocessors can now no longer significantly increase frequency. While architectural enhancements to a single-threaded core cannot pick up the slack, multicore processors promise historical performance growth rates, albeit only on sufficiently parallel applications and at the expense of additional programming effort.

Multicore design has its challenges as well. Replication of cores communicating primarily with off-chip memory and I/O leads to off-chip bandwidth demands inversely proportional to the square of the feature size, while off-chip bandwidths tend to be limited by the (more or less fixed) perimeter from which signals must escape. This trend leads to significant pressure on off-chip signaling rates, which are beginning to exceed the clock frequency of the processor core. High-bandwidth packaging such as that provided by 3D-integration may soon be required to keep up with multicore bandwidth demands.

Compared to a uniprocessor, a multicore processor is somewhat less generic, as uniprocessors are generally more efficient at emulating a parallel processor than vice versa. In the space of emerging applications with inherent concurrency, an ideal multicore processor would become part of a general-purpose system with broad enough markets to sustain the required investments in technology. However, this reality will only come to pass if the programming tools are developed and programmers are trained to deliver the intrinsic performance of these processors to the end applications.

This book is intended to provide an overview of the first generation of multicore processors and systems that are now entering the marketplace. The contributed chapters cover many topics affecting, and associated with, multicore systems including technology trends, architecture innovations, and software and

Attachment - Page A226

programming issues. Chapters also cover case studies of state-of-the-art commercial and research multicore systems. The case studies examine multicore implementations across different application domains, including general purpose, server, signal processing, and media/broadband. Cross-cutting themes of the book are the challenges associated with scaling multicore systems to hundreds and ultimately thousands of cores. We expect that researchers, practitioners, and students will find this book useful in their educational, research, and development endeavors.

We organize the chapters around three major themes: (1) multicore design considerations, (2) programmability innovations, and (3) case studies. While each of these areas are broader than this book can cover, together the chapters introduce the fundamentals of modern and future multicore design. While the third section of this book is dedicated to case studies, the first section also discusses prototype multicore architectures to exemplify the design considerations. In all, this book describes commercial or research implementations of seven different multicore systems or families of systems.

Multicore Design Considerations

Chapter 1 introduces tiled multicore designs, first embodied in the Raw processor chip. Tiling is both a natural and necessary method of constructing a complex system from multiple copies of a simple processor tile. Furthermore, because tiled architectures are inherently partitioned, they are less vulnerable to the deleterious effects of decreasing wire speed relative to transistor speed. Each tile also includes a network router and ports to its neighbors so that scaling a tiled design can be accomplished merely by adding more tiles, logically snapping them together at the interfaces. The perimeter of the tiled array can be used for connections to memory or to connect multiple chips together. The Raw prototype chip consists of a 4×4 array of tiles, with each tile comprising a simple processing pipeline, instruction memory, and data memory. The same design principles apply to Tilera's TILE64 chip, a commercial venture that is descended from Raw.

The key feature of the Raw system is the tight integration of the network into the processor pipeline. Raw exposes inter-tile interaction through register-mapped network ports; reading or writing the designated register name injects an operand into or extracts an operand from the network. Raw also employs a statically programmable router which executes a sequence of routing instructions to route operand packets in concert with the computation instructions executed by the core pipeline. The authors of the chapter show that applications with different characteristics can be executed by the Raw tiled substrate because the underlying computation and communication mechanisms are exposed to software. For example, exposing the operand communication enables serial programs to be partitioned across the tiles with producer/consumer instructions interacting through the operand network. The network also enables fine-grained parallel streaming computation in which small data records are processed in a pipelined fashion across multiple tiles. The Raw chip makes a compelling case for distributed architectures and for exposing

communication to software as a first-class object. These themes recur in different ways throughout the chapters of this book.

Chapter 2 focuses on the design and implementation of on-chip interconnects (networks-on-chip or NOCs), as a critical element of multicore chips. When a chip contains only a few processors and memory units, simple interconnects such as buses and small-scale crossbars suffice. However, as the demand for on-chip communication increases, so does the need for more sophisticated on-chip networks. For example, the IBM Cell chip employs a bidirectional ring network to connect its eight data processors, its Power core, and its off-chip interfaces. The Tilera chip employs a collection of mesh networks to interconnect 64 on-chip processors. This chapter first examines the fundamentals of interconnection network design, including an examination of topologies, routing, flow control, and network interfaces in the context of NOCs. The chapter also discusses the trade-offs for router microarchitecture designed to meet the area, speed, and power constraints for multicore chips.

The chapter also includes two case studies of NOCs deployed in different ways in two recent prototype chips. The TRIPS processor employs a routed network, rather than a conventional broadcast bus, to deliver operands from producing instructions to consuming instructions in a distributed uniprocessor microarchitecture and to connect the processing cores to the level-1 caches. The operand network has 25 terminals and optimizes the routers for latency with a total of one clock cycle per hop. The network is connected directly to the inputs and the outputs of the ALUs and supports fixed length, operand-sized packets. By contrast, the Intel TeraFLOPS chip connects 80 on-chip cores with a high-speed, clock-phase-skew tolerant, inter-router protocol. The TeraFLOPS routers employ a five-stage pipeline and run at 5 GHz in 65 nm. Message injection and reception is explicit with send and receive instructions that use local registers as interfaces to the network. TRIPS and the TeraFLOPS chip make different choices about routing algorithms and flow control based on the demands of their particular deployments. One message from this chapter is that NOCs are not "one-size-fits-all" and should be tailored to the needs of the system to obtain the fastest and the most efficient design. The area of NOC architecture is extremely active at present, with new innovations being published on topologies, routing algorithms, and arbitration, among other topics.

Chapter 3 examines the granularity of cores within a multicore system. A survey of commercial multicore chips shows that core size and number of cores per chip vary by almost two orders of magnitude. At one end of the spectrum are *bulldozer* processors which are large, wide issue, and currently are deployed with only 2–4 cores per chip. At the other end of the spectrum are the *termites* which are small and numerous, with as many as 128 cores per chip in 2008 technology. The trade-off between these two ends of the spectrum are driven by the application domain and the market targeted by the design. Bulldozers are the natural extensions of yesterday's high-performance uniprocessors and target the general-purpose market. Termites typically target a narrower market in which the applications have inherent concurrency that can be more easily exploited by the more numerous yet simpler cores.

This chapter points out that the diversity in multicore granularity is an indicator of greater hardware and software specialization that steps further away from the general-purpose computing platforms developed over the last 30 years. To address this challenge, the chapter describes a general class of composable core architectures in which a chip consists of many simple cores that can be dynamically aggregated to form larger and more powerful processors. The authors describe a particular design consisting of 32 termite-sized cores which can be configured as 32 parallel uniprocessors, one 32-wide, single-threaded processor, or any number and size of processors in between. The chapter describes a set of inherently scalable microarchitectural mechanisms (instruction fetch, memory access, etc.) that are necessary to achieve this degree of flexibility and that have been prototyped and evaluated in a recent test chip. Such a flexible design would enable a single implementation to match the parallelism granularity of a range of applications and even adapt to varying granularity within a single program across its phases. Further, this class of flexible architectures may also be able to feasibly provide greater single-thread performance, a factor that is being sacrificed by many in the multicore space.

Programming for Multicores

While there are a number of challenges to the design of multicore architectures, arguably the most challenging aspect of the transition to multicore architectures is enabling mainstream application developers to make effective use of the multiple processors. To address this challenge we consider in this section the techniques of thread-level speculation (TLS) that can be used to automatically parallelize sequential applications and transactional memory (TM) which can simplify the task of writing parallel programs. As is evident from the chapters that describe these techniques, their efficient implementation requires the close interaction between systems software and hardware support.

Chapter 4 introduces speculative multithreaded or thread-level speculation architectures. The authors first provide motivation for these architectures as a solution to the problems of limited performance and implementation scalability associated with exploiting ILP using dynamic superscalar architectures and the parallel programming problems associated with traditional CMPs.

Speculatively multithreaded architectures eliminate the need for manual parallel programming by using the compiler or hardware to automatically partition a sequential program into parallel tasks. Even though these speculatively parallel tasks may have data-flow or control-flow dependencies, the hardware executes the tasks in parallel but prevents the dependencies from generating incorrect results. The authors describe in detail the approach taken to speculative multithreading in the Multiscalar architecture which was the earliest concrete example of speculative multithreading. A key element of the Multiscalar approach is the use of software and hardware to their best advantage in the implementation of speculative multithreaded execution. Software is used for task partitioning and register data-dependency tracking between tasks, which is possible with static

information. Hardware sequences the tasks at runtime and tracks inter-task memory dependencies, both of which require dynamic information. The hardware support for memory dependency tracking and memory renaming required to support speculative multithreading in the Multiscalar architecture is called the speculative version cache (SVC) and adds considerable complexity to the design of a CMP. As the authors describe, this added complexity can buy significant performance improvements for applications that cannot be parallelized with traditional auto-parallelizing technology. Despite the hardware complexity of speculative-multithreading architectures, the desire to reduce the software disruption caused by CMPs spurs continued interest in these architectures especially in conjunction with dynamic compiler technology.

Chapter 5 presents the design of Transactional Memory (TM) systems. The authors begin by describing how TM can be used to simplify parallel programming by providing a new concurrency construct that eliminates the pitfalls of using explicit locks for synchronization. While this construct is new to the mainstream parallel programming community it has been used and proven for decades in the database community. The authors describe how programming with TM can be done using the *atomic* keyword. With the high-level atomic construct, programmers have the potential to achieve the performance of a highly optimized lock-based implementation with a much simpler programming model.

The authors explain that the key implementation capabilities of a TM system include keeping multiple versions of memory and detecting memory conflicts between different transactions. These capabilities can be implemented in a software TM (STM) or in a hardware TM (HTM) using mechanisms similar to those required for speculative multithreading or a combination of both software and hardware in a hybrid-TM. Because one of the major challenges of STM implementations is performance, the chapter goes into some detail in describing compiler-based optimizations used to improve the performance of an STM by reducing the software overheads of STM operations. The key to efficient optimization is a close coupling between the compiler and the STM algorithm. While STMs support TM on current CMPs, significant performance improvements are possible by adding hardware support for TM. The chapter explains that this support can range from hardware-enhanced STM that reduces software overheads to a pure HTM that completely eliminates all software overheads and achieves the highest performance. The authors conclude that TM deployment will include the use of STMs for existing CMP hardware, but must encompass language, compiler, and runtime support to achieve adequate performance. Ultimately, hardware acceleration will be required to achieve good TM performance.

Case Studies

The last set of chapters comprise four case studies of multicore systems. In each case study, the authors describe not only the hardware design, but also discuss the demands on the systems and application software needed as a part of the multicore

ecosystem. We selected these chapters as they represent different points in the design space of multicore systems and each target different applications or multicore programming models.

Chapter 6 presents the architecture and system design of the AMD Opteron family of general-purpose multicore systems. The authors start by describing the trends and challenges for such general-purpose systems, including power consumption, memory latency, memory bandwidth, and design complexity. While multicore systems can help with some aspects, such as design complexity through replication, others are much more challenging. Power consumption must be shared by multiple processors and with strict power envelopes, not all of the processors can simultaneously operate at their peak performance. Likewise, external bandwidth (memory and interconnect) becomes a greatly constrained resource that must be shared. The challenges are no less severe on the software side, as parallel programming environments for general-purpose applications are in their infancy and system software, such as operating systems, are not yet prepared for increasing levels of concurrency of emerging multicore systems.

The authors then present the Opteron family of multicore systems, which were designed from the outset to be scaled both internally (more processors per core) and externally (more chips per system). These processor cores fit into the category of bulldozers, as they are physically large and can support high single-thread performance. The authors emphasize that because all but small-scale computer systems will be composed of many multicore chips, design for system-level scalability is critical. The chapter describes the details of the system-level architecture that is implemented on the processor chips, including multiple DRAM interfaces, interchip interconnection (Hypertransport), and the system-level cache hierarchy. The lessons from this chapter are that general-purpose multicore systems require a balanced design but that substantial challenges in programmability, system software, and energy efficiency still remain.

Chapter 7 details Sun Microsystems' Niagara and Niagara 2 chips, which are designed primarily for the server space in which job throughput is more important than the execution latency of any one job. The authors make the case that simple processors that lack the speculation and out-of-order execution of high-end uniprocessors are a better match for throughput-oriented workloads. Such processors with shallower pipelines and slower clock rates achieve better power efficiency and can employ multithreading as a latency tolerance technique since independent threads are abundant in these workloads. Because these processors are smaller and more power efficient, more of them can be packed onto a single chip, increasing overall throughput. These processors can be classified as *chainsaws*, as they are smaller than the bulldozers, yet larger and more full-featured than termites.

In their case studies, the authors describe in detail the microarchitecture of Sun's line of multicore chips. The first generation Niagara chip employs eight four-way multithreaded cores, for a total of 32 simultaneously executing threads. The processors share a four-banked level-2 cache, with each bank connected to an independent DRAM memory controller. The second-generation Niagara 2 doubles the number of threads per core (for a total of 64 simultaneously executing threads) and doubles the number of L2 banks to increase on-chip memory-level parallelism. While Niagara

had a single floating-point that is shared across all of the cores, each Niagara 2 core has its own floating-point and graphics unit. Both Niagara systems employ cache coherence that spans the cores on the chip and multiple chips in a system. The chapter includes experimental results that show nearly a factor of 2 boost in power efficiency (performance/Watt) over more complex core systems. Like the authors of Chapter 6, the authors of this chapter indicate that the expected increase in cores and threads will demand more system support, such as operating system virtualization.

Chapter 8 describes a family of stream processors, which are multicore systems oriented around a data-streaming execution model. Initially amenable to multimedia and scientific applications, stream processing exploits parallelism by simultaneously operating on different elements of a stream of data in a fashion similar to vector processors. By making data streams first-class objects that are explicitly moved through levels of the memory hierarchy, stream processors eliminate power and area inefficient caches. SIMD-style computation reduces the amount of control logic required for stream processors. The authors claim that these factors provide a 10–30 times improvement in power efficiency over conventional multicore architectures.

The chapter first details a stream program model which partitions programs into computation kernels and data streams. Kernels can be thought of as data filters in which an input stream or streams is transformed into an output stream. A stream program is represented as a stream flow graph which has nodes that correspond to kernel computations and edges that correspond to the streams. The program itself includes explicit operations on streams to move them through the memory hierarchy and instantiations of kernels between stream transfers. High-level programming languages such as Brook and Sequoia have been developed to target stream processing execution models.

The chapter then describes a general stream processor microarchitecture as well as three case studies: (1) Imagine stream processor, (2) Stream Processors Inc. Storm I processor, and (3) Merrimac streaming supercomputer architecture. Each of these designs shares several common design principles including an explicitly managed memory hierarchy, hardware for bulk data transfer operations, and simple lightweight arithmetic cores controlled through a hybrid SIMD/VLIW execution model. These simple processing elements fall into the category of termites, since they have simple control logic and little memory per core. For example, the Storm I processor has a total of 80 ALUs organized into 16 lanes that execute in parallel. Stream processors exemplify performance and power efficiency that can be obtained when the programming model and application domain allows for explicit and organized concurrency.

Chapter 9 describes the Cell Broadband Engine architecture and its first implementations. This family of processors, jointly developed by IBM, Toshiba, and Sony falls at the upper end of the chainsaw category. The Cell Broadband Engine (Cell B.E) is a heterogeneous multiprocessor with two types of programmable cores integrated on a single chip. The Power processor element (PPE) is a Power architecture compliant core that runs the operating system and orchestrates the eight synergistic processor elements (SPEs). Per-core performance of the SPEs on compute-intensive applications is comparable to that of bulldozer cores, but each SPE requires

substantially less power, less area, and fewer transistors to implement, allowing a nine-core Cell B.E. in a chip the size of a typical dual-core bulldozer.

Like the stream processors described in Chapter 8, the SPEs achieve their efficiency by explicitly managing memory. The SPEs manage one additional level of memory than typical processors, a 256 kB local store included in each SPE. SPE DMA operations are the equivalents of the PPE's load and store operations, and access coherent shared memory on the processor in exactly the same way as the PPE's load and stores. Instead of targeting the SPE's register file, however, the DMA operations place code and data in the local store (or copy it back to main memory). The SIMD-RISC execution core of the SPE operates asynchronously on this local store. Each SPE is an autonomous single-context processor with its own program counter capable of fetching its own code and data by sending commands to the DMA unit.

The management of the local store is an added burden on the software. Initial versions of the software development environment for the Cell Broadband Engine required the application programmer to partition code and data and control their movement into and out of the local store memory. A more recent version removes the burden of code partitioning from the programmer. Following the pioneering efforts of a number of research compilers for the Cell B.E. and other multicore processors, standards-based approaches to programming this processor are now available. These programming languages and frameworks abstract the notions of locality and concurrency and can be targeted at a wide variety of multicore CPUs and (GP)GPUs allowing portable approaches to developing high-performance software and providing a path towards bringing heterogeneous computing into the mainstream.

Current and Future Directions

While the case studies in this book describe a range of architectures for different types of applications, we recognize that the design and application space is much broader. For example, we do not discuss the multicore designs of existing and emerging graphics processing units (GPUs), such as those from NVidia, Intel, and AMD, a subject which merits its own book. The graphics application domain has traditionally exposed substantial pixel and object-level parallelism, a good match to the growing number of on-chip processing elements. Interestingly, there is substantial diversity among GPUs in the number and complexity of the processing elements. The NVidia GeForce 8800 employs 128 simple cores that can operate as independent MIMD processors with non-coherent memory. Intel's Larrabee system employs fewer but larger x86-based cores with local cache and a coherent memory. AMD's Radeon HD 2900 employs four parallel stream processing units, each with 80 ALUs controlled using a hybrid of VLIW and SIMD. Successive generations of GPUs have become more programmable and currently require domain-tailored program systems such as NVidia's CUDA and AMD's CAL to achieve

both performance and programmability. The differences in approaches among the architectures targeted at the graphics domain share similarities with the distinctions between some of the architectures described in the chapters of this book.

Looking forward, Moore's law of doubling integrated circuit transistor counts every 18–24 months appears likely to continue for at least the next decade. The implication for multicore systems is the expected doubling of core count per technology generation. Given the starting point of 4–64 processors in today's commercial systems, we expect to see chips that have the capacity to contain hundreds or even thousands of cores within 10 years. However, the challenges that are described throughout this book will remain. Keeping power consumption within thermal limits will not become any easier with increasing transistor density and core counts. Programming these parallel systems is far from a solved problem, despite recent advances in parallel programming languages and tools. Developing solutions to these problems will be critical to the continued growth and performance of computer systems.

As technology matures further, eventually performance improvements will be possible only with gains in efficiency. Besides increased concurrency, the only significant means to gain efficiency appears to be increased specialization. In the context of multicore systems, cores would be specialized through configuration at run-time, at manufacture, or through unique design. Such differentiated (hybrid) multicore chips will likely be even more difficult to build and program than conventional multicore processors. However, if maturing silicon technologies lead to a lengthening of semiconductor product cycles, manufacturers will be able to afford the cost of specialization and other innovations in computer design, perhaps signaling a new golden age of computer architecture.

Acknowledgments

We would like to thank all of the contributing authors for their hard work in producing a set of excellent chapters. They all responded well to editorial feedback intended to provide a level of cohesion to the independently authored chapters. We would also like to thank Anantha Chandrakasan, Chief Editor of Springer's Series on Integrated Circuits and Systems, for recruiting us to assemble this book. Thanks go to the editorial staff at Springer, in particular Katelyn Stanne and Carl Harris for their patience and support of the production of this book.

Texas, USA — Stephen W. Keckler
California, USA — Kunle Olukotun
Texas, USA — H. Peter Hofstee

Contents

1 Tiled Multicore Processors 1
Michael B. Taylor, Walter Lee, Jason E. Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jason S. Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman Amarasinghe, and Anant Agarwal

2 On-Chip Networks for Multicore Systems 35
Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal

3 Composable Multicore Chips 73
Doug Burger, Stephen W. Keckler, and Simha Sethumadhavan

4 Speculatively Multithreaded Architectures 111
Gurindar S. Sohi and T.N. Vijaykumar

5 Optimizing Memory Transactions for Multicore Systems 145
Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha

6 General-Purpose Multi-core Processors 173
Chuck Moore and Pat Conway

7 Throughput-Oriented Multicore Processors 205
James Laudon, Robert Golla, and Greg Grohoski

8 Stream Processors 231
Mattan Erez and William J. Dally

9 Heterogeneous Multi-core Processors: The Cell Broadband Engine 271
H. Peter Hofstee

Index 297

Attachment - Page A235

Contributors

Ali-Reza Adl-Tabatabai Intel Corporation, Hillsboro, OR USA

Anant Agarwal MIT CSAIL, Cambridge, MA USA

Saman Amarasinghe MIT CSAIL, Cambridge, MA USA

Ian Bratt Tilera Corporation, Westborough, MA USA

Doug Burger The University of Texas at Austin, Austin, TX USA

Pat Conway Advanced Micro Devices Inc., Sunnyvale, CA USA

William J. Dally Stanford University, Stanford, CA USA

Mattan Erez The University of Texas at Austin, Austin, TX USA

Matthew I. Frank University of Illinois at Urbana – Champaign, Urbana, IL USA

Robert Golla Sun Microsystems, Austin, TX USA

Ben Greenwald Veracode, Burlington, MA USA

Greg Grohoski Sun Microsystems, Austin, TX USA

Henry Hoffmann MIT CSAIL, Cambridge, MA USA

H. Peter Hofstee IBM Systems and Technology Group, Austin, TX USA

Paul R. Johnson MIT CSAIL, Cambridge, MA USA

Stephen W. Keckler The University of Texas at Austin, Austin, TX USA

Jason S. Kim MIT CSAIL, Cambridge, MA USA

Christos Kozyrakis Stanford University, Stanford, CA USA

James Laudon Google, Madison, WI USA

Walter Lee Tilera Corporation, Westborough, MA USA

Jason E. Miller MIT CSAIL, Cambridge, MA USA

Chuck Moore Advanced Micro Devices Inc., Sunnyvale, CA USA

Attachment - Page A236

Li-Shiuan Peh Princeton University, Princeton, New Jersey, USA

James Psota MIT CSAIL, Cambridge, MA USA

Bratin Saha Intel Corporation, Hillsboro, OR USA

Arvind Saraf Swasth Foundation, Bangalore India

Simha Sethumadhavan Columbia University, New York, NY USA

Nathan Shnidman The MITRE Corporation, Bedford, MA USA

Gurindar S. Sohi University of Wisconsin, Madison, WI USA

Volker Strumpen IBM Austin Research Laboratory, Austin, TX USA

Michael B. Taylor University of California, San Diego, La Jolla, CA USA

Sriram Vangal Intel Corporation, Hillsboro, OR USA

T.N. Vijaykumar Purdue University, West Lafayette, IN USA

David Wentzlaff MIT CSAIL, Cambridge, MA USA

Chapter 3
Composable Multicore Chips

Doug Burger, Stephen W. Keckler, and Simha Sethumadhavan

Abstract When designing a multicore chip, two primary concerns are how large and powerful to make each core and how many cores to include. A design with a few large cores is attractive for general purpose computing that has coarse-grained threads which depend on instructional-level parallelism for performance. At the other end of the processor granularity spectrum, a large collection of small processors is suited to applications with ample thread-level parallelism. While much of the spectrum between large cores and tiny cores is being pursued by different vendors, each of these fixed architectures are suited only to the range of workloads that maps well to its granularity of parallelism. An alternative is a class of architectures in which the processor granularity (and the number of processors) can be dynamically configured–at runtime. Called composable lightweight processors (CLPs), these architectures consist of arrays of simple processors that can be aggregated together to form larger more powerful processors, depending on the demand of the running application. Experimental results show that matching the granularity of the processors to the granularity of the tasks improves both performance and efficiency over fixed multicore designs.

Portions reprinted, with permission, from "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, *International Symposium on Microarchitecture* ©2006, IEEE. and "Composable Lightweight Processors," C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, *International Symposium on Microarchitecture* ©2007, IEEE.

D. Burger (✉)
The University of Texas at Austin, 1 University Station C0500, Austin, TX 78712-0233, USA
e-mail: dburger@cs.utexas.edu

S.W. Keckler et al. (eds.), *Multicore Processors and Systems*, Integrated Circuits and Systems, DOI 10.1007/978-1-4419-0263-4_3, © Springer Science+Business Media, LLC 2009

Attachment - Page A238

3.1 Introduction

Two fundamental questions when designing a multicore chip are (1) how many processor cores to place on a chip and (2) how much area to devote for the structures required to exploit parallelism in each processor core. An apt – if whimsical – metaphorical taxonomy in use for small, medium, and large processor cores is *termites*, *chainsaws*, and *bulldozers*. A design with a few large cores (or bulldozers) is attractive for general purpose computing that has few coarse-grained threads which exploit instructional-level parallelism for performance. At the other end of the spectrum, a large collection of small processors (or termites) is suited to applications with ample thread-level parallelism. Other designs represent a compromise, having a moderate number of moderately complex processors (or chainsaws).

To date, the issue of core granularity – whether to build termites, chainsaws or bulldozers – is driven by the application space and power envelope chosen by a vendor. Many points in the granularity spectrum are represented by commercial multicore chips. Table 3.1 lists a collection of recent multicore chips along with the approximate size of each core. The chips are ordered by core size, largest to smallest, with the bulldozers appearing at the top and termites appearing at the bottom. The last column shows all of the core sizes scaled to a 65 nm technology for those chips implemented with larger feature sizes. The data in that column is computed by linearly scaling the core size in each of the X and Y dimensions. The core sizes and

Table 3.1 Recent multicore chips with estimates of core sizes

Chip	Technology	No. of cores	Die size (mm^2)	Core size (mm^2)	Normalized to 65 nm
Bulldozers					
Intel Itanium [36]	65 nm	4	699	87.3	87.3
IBM Power6 [5]	65 nm	2	341	48.8	48.8
TFlex-16 [16]	—	16	—	—	37.9
AMD Barcelona [4]	65 nm	4	285	35.6	35.6
TRIPS [29]	130 nm	2	336	94.5	23.6
Chainsaws					
Sun Rock [39]	65 nm	16	396	15.1	15.1
Sun T2 [24]	65 nm	8	342	14.3	14.3
TFlex-4 [16]	—	16	—	—	9.5
IBM Cell [12]	90 nm	9	220	14.5	7.6
Termites					
Raw [38]	180 nm	16	331	18.6	3.7
Intel Polaris [40]	65 nm	80	275	2.5	2.5
TFlex-1 [16]	—	16	—	—	2.4
Tilera [2]	90 nm	64	330	3.6	1.9
NVIDIA GeForce 8800 [20]	90 nm	128	470	<3	<1.6
SPI Storm-1 [14]	130 nm	16	164	5.0	1.3
Clearspeed CS301 [9]	130 nm	64	72	1.13	0.28

chip sizes are estimates taken from published papers and examinations of published die photos.

We list cores larger than 20 mm^2 in 65 nm technology as "bulldozer cores," and cores larger than 5 mm^2 as "chainsaw cores," with anything smaller being "termite cores." The chip containing the largest bulldozer core in Table 3.1 is the quad-core Itanium chip, which is designed for high single-threaded performance across a range of desktop and server applications. A typical chainsaw core is found on the Sun Ultrasparc T2 chip, which targets threaded throughput computing for server workloads. Termite cores can be exemplified by those on the NVIDIA GeForce 8800 chip, which includes 128 stream processors and targets data parallel graphics and rendering algorithms.

The problem with fixing the core granularity at design time, of course, is that a fixed number and granularity of cores will run workload mixes poorly, if the mixes differ from the design-time decisions about core granularity. While such concerns may be irrelevant in domain-specific designs, such as graphics processors, general-purpose designs are much more likely to have workload mixes that vary over time, and thus map poorly onto a fixed multicore design at least some of the time.

The only architecture listed in Table 3.1 that has not been implemented in silicon is the TFlex microarchitecture, an example of a composable multicore architecture described in detail later in this chapter. A composable processor can address the fixed-core granularity problem, by aggregating small cores to form medium- or large-grained logical processors, with no change needed in the running software. Thus, while one TFlex core (estimated at 2.4 mm^2 at 65 nm) is in the size range of termite cores, four can be aggregated into a logical 9.5 mm^2 core, and even 16 can be aggregated to form a logical 37.9 mm^2 core that increases average performance over finer granularities for many workloads. Thus, an array of TFlex cores can be dynamically configured as termites, chainsaws, or bulldozers, depending on the needs of the system and running software.

3.1.1 Heterogeneous Processors

Recently, architecture researchers have proposed heterogeneous multicore (also known as asymmetric multicore) processors as a means to improve power efficiency or performance over symmetric multicore architectures with many fixed granularity processors of one type. Effectively, heterogeneous multicore designs address the granularity issue by providing the software with granularity options. However, this approach fixes the processor granularity at design time, which will perform suboptimally with many workload mixes, while also complicating the task of the software scheduler, and increasing communication/coherence overheads.

Kumar et al. initially proposed a single-ISA heterogeneous multicore architecture, which consisted of instances of different generations of Compaq Alpha processors. They suggest that energy efficiency can be maximized by running a program on the least-powerful core required to meet its performance demand. Others, such as Ghiasi and Grunwald [7] and Grochowski et al. [8] also exploit heterogeneity in multicore architectures for power efficiency. Kumar et al. extended their work to the

multiprogramming space, determining that the core asymmetry provided throughput benefits as well [18]. Annavaram et al. examine the viability of heterogeneity to accelerate multithreaded programs [1]. Heterogeneity is also finding its way into the commercial sphere. For example, IBM's Cell processor includes one mid-range PowerPC processor for executing system code and eight SPUs for executing threaded and data-parallel code [12]. We expect this trend to continue as specialized cores for graphics or cryptography are incorporated into processor dies. While efficient for their assigned tasks, such specialized processors do not provide adaptivity to different granularities of parallelism.

3.1.2 Composable Processors

While heterogeneous cores provide hardware specialization that can lead to efficiency improvements, the number and types of cores are fixed at design time. Composable flexible-core processors aim instead to provide adaptivity in both the number and granularity of processors, enabling the system to execute both large and small numbers of tasks efficiently. The basic approach is to aggregate a number of smaller, identical physical processors in a smaller aggregate area to form larger, logical processors. Composing logical processors from multiple simple processors gives a system the ability to adapt to ranges of instruction-level parallelism (ILP) and thread-level parallelism (TLP) dynamically and transparently to application software. Furthermore, a composable processor design that supports scaling to many cores can reduce the burden on the programmer for hard-to-parallelize applications by providing higher single-thread performance and reducing the number of threads a programmer must manage. A multicore chip composed of bulldozer processors will perform well on applications with few high-ILP threads while a swarm of termite processors will perform well on applications with many low-ILP threads. Composable processors can be configured in the field to meet both needs.

Supporting this degree of flexibility requires physical distribution of architectural structures including the register file, instruction window, L1 caches, and operand bypass network. In addition to the partitioning, various distributed control protocols are required to correctly implement instruction fetch, execute, commit, speculation recovery, and other processor actions. If some of these functions are centralized, overall scalability of the composed cores will suffer.

One example of a composable architecture is Core Fusion, which provides mechanisms to enable a limited number (up to four) of out-of-order cores to be fused into a single more powerful core [11]. Core Fusion's principal advantage is that it is ISA compatible with current industrial parts. Its main disadvantage is that its centralized register renaming function limits its scalability. Another recent composable proposal is *Federation*, which "federates" multiple in-order cores to create an out-of-order processor [37], providing more power-efficient building blocks but achieving lower composed performance. Another example of a composable system is virtual hierarchies, which enables a flexible aggregation and isolation of individual cores to form logical multiprocessors (as opposed to logical uniprocessors) [22]. Flexibility

across these dimensions can offer capability and efficiency advantages to emerging multicore systems.

In this chapter, we explore the architecture and capabilities of a particular composable processor design called composable lightweight processors (or CLPs) [16]. A CLP consists of multiple simple, narrow-issue processor cores that can be aggregated dynamically to form more powerful single-threaded processors. Thus, the number and size of the processors can be adjusted on the fly to provide the target that best suits the software needs at any given time. The same software thread can run transparently – without modifications to the binary – on one core, two cores, or up to as many as 32 cores, the maximum we simulate. Low-level run-time software can decide how to best balance thread throughput (TLP), single-thread performance (ILP), and energy efficiency. Run-time software may also grow or shrink processors to match the available ILP in a thread to improve performance and power efficiency.

Figure 3.1 shows a high-level floorplan with three possible configurations of a CLP. The small squares on the left of each floorplan represent a single physical processing core, while the squares on the right half show a banked L2 cache. If a large number of threads are available, the system could run 32 threads, one on each core (Fig. 3.1a). If high single-thread performance is required and the thread has sufficient ILP, the CLP could be configured to use an optimal number of cores that maximizes performance (up to 32, as shown in Fig. 3.1c). To optimize for energy efficiency, for example in a data center or in battery-operated mode, the system could configure the CLP to run each thread at its most energy-efficient point. Figure 3.1b shows an energy (or power) optimized CLP configuration running eight threads across a range of processor granularities.

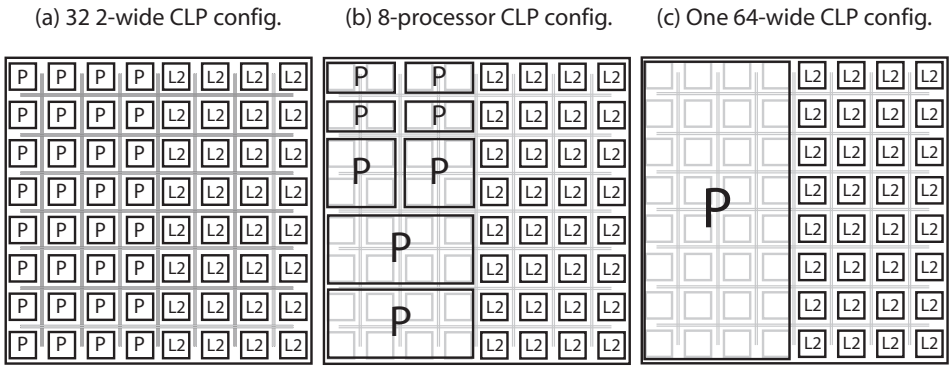


Fig. 3.1 Three dynamically assigned CLP configurations

A fully composable processor shares no structures physically among the multiple processors. Instead, a CLP relies on distributed microarchitectural protocols to provide the necessary instruction fetch, execution, branch prediction, dependence prediction, memory access/disambiguation, and commit capabilities. Full composability is difficult in conventional ISAs, since the atomic units are individual instructions, which require that control decisions be made too frequently to coordinate across a distributed processor. Explicit Data Graph Execution (EDGE) ISAs, conversely, reduce the frequency of control decisions by employing block-based pro-

gram execution [23] and explicit intra-block dataflow semantics, and have been shown to map well to distributed microarchitectures [3]. The particular CLP design that we evaluate, called TFlex, achieves the composable capability by mapping large, structured EDGE blocks across participating cores differently depending on the number of cores that are running a single thread. The TFlex CLP microarchitecture allows the dynamic aggregation of any number of cores – up to 32 for each individual thread – to find the best configuration under different operating targets: performance, area efficiency, or energy efficiency.

3.1.3 Chapter Overview

This chapter describes the instruction-set and microarchitectural support necessary to provide dynamic adaptation of multicore granularity. We first outline the EDGE instruction set architecture which is critical to scaling and composition. The chapter then describes two generations of microarchitectures that support an EDGE ISA. The first microarchitecture is TRIPS, a fixed but scalable microarchitecture that demonstrates the viability of EDGE ISAs. TRIPS has been implemented in a custom 130 nm ASIC and tested on a wide variety of applications. The program execution model and partitioned microarchitecture of TRIPS also provides a foundation for the second microarchitecture, TFlex, which is a highly scalable CLP. The TFlex description includes the features of the microarchitecture necessary for composability, and also includes details of the design of the reprogrammable mapping functions and protocols required for composition and recomposition.

We end this chapter with a set of performance estimates of the TFlex microarchitecture. To summarize the results, on a set of 26 benchmarks, including both high- and low-ILP codes, the best composed configurations range from one to 32 dual-issue cores depending on operating targets and applications. The TFlex design achieves a 1.4× performance improvement, 3.4× performance/area improvement, and 2.0× performance^2/Watt improvement over the TRIPS processor. The TFlex CLP also shows improved parallel flexibility with the throughput of multiprogrammed workloads improving by up to 47% over the best fixed-CMP organization.

3.2 ISAs for Scalable and Multigranular Systems

Designing a composable processor that can scale to many composed cores using a conventional instruction set architecture (RISC or CISC) presents substantial challenges. First, conventional architectures have single points of instruction fetch and commit, which require centralization for those operations. Second, programs compiled for conventional architectures have frequent control-flow changes (branches and jumps occurring as frequently as every 5–6 instructions), which demand fast branch processing and wide distribution of the branch outcome. Third, adding more execution units typically requires a broadcast of ALU outputs to ALU inputs as the

hardware cannot easily track all of the producers and consumers of data. Salverda and Zilles show that effective steering of instructions to multiple cores in a composable, in-order processor is difficult due to both the complexity of instruction-level data flow graphs and fetch criticality [27].

Creating larger logical microarchitectural structures from smaller ones is the principal challenge for the design of a composable architecture. Composing some structures, such as register files and level-one data caches, is straightforward as these structures in each core can be treated as address-interleaved banks of a larger aggregate structure. Changing the mapping to conform to a change in the number of composed processors merely requires adjusting the interleaving factor or function.

However, banking or distributing other structures required by a conventional instruction set architecture is less straightforward. For example, operand bypass (even when distributed) typically requires some form of broadcast, as tracking the ALUs in which producers and consumers are executing is difficult. Similarly, both instruction fetch and commit require a single point of synchronization to preserve sequential execution semantics, including features such as a centralized register rename table and load/store queues. While some of these challenges can be solved by brute force, supporting composition of a large number of processing elements – particularly in an energy- and power-efficient manner – can benefit from instruction set support.

To address the challenges of structure scalability, which inherently supports composability, we developed a new class of instruction sets, called Explicit Data Graph Execution (EDGE) architectures [3]. An EDGE architecture supports execution of serial programs by mapping compiler-generated dataflow graphs onto a distributed substrate of computation, storage, and control units. The two defining features of an EDGE ISA are block-atomic execution and direct communication of instructions within a block, which together enable efficient dataflow-like execution while supporting conventional imperative languages.

3.2.1 TRIPS Blocks

The TRIPS ISA, used both for the TRIPS prototype and for the TFlex composable lightweight processor, is an instance of EDGE ISAs that aggregates up to 128 instructions into a single block that obeys the block-atomic execution model. In this execution model, each block of instructions is logically fetched, executed, and committed as a single entity. This model amortizes the per-instruction book-keeping over a large number of instructions and reduces the number of branch predictions and register file accesses. Furthermore, this model reduces the frequency at which control decisions about what to execute must be made (such as fetch or commit), providing the additional latency tolerance to make distributed execution practical. When an interrupt arrives or an exception is raised, the processor must roll back to the beginning of the excepting block, since a block must be entirely committed or not at all. This model presents similar challenges with respect to I/O as

transactional execution; blocks that produce side-effect-causing accesses to I/O (or loads with store semantics, such as a swap) must be handled carefully and guaranteed to commit once they begin execution. While these atomic blocks may resemble transactions with their notion of atomicity, they are not exposed to the programmer in an API, and are in fact an example of a large "architectural atomic unit" as described by Melvin, Shebanow, and Patt before transactions were first proposed [23]. In Melvin et al.'s taxonomy, an EDGE block corresponds to an architectural atomic unit, and the operations within each block (in the TRIPS prototype) correspond to both compiler and execution atomic units.

The TRIPS compiler constructs blocks and assigns an implicit ID (0–127) to each instruction. This ID determines an instruction's position within the block. The microarchitecture uses these instruction IDs to map each instruction to a reservation station within the execution substrate. Each block is divided into between two and five 128-byte chunks. Every block includes a header chunk which encodes up to 32 `read` and up to 32 `write` instructions that access the 128 architectural registers. The read instructions pull values out of the registers and send them to compute instructions in the block, whereas the write instructions return outputs from the block to the specified architectural registers. In the TRIPS microarchitecture, each of the 32 read and write instructions are distributed across the four register banks, as described in the next section.

The header chunk also holds three types of control state for the block: a 32-bit "store mask" that indicates which of the possible 32 memory instructions are stores, block execution flags that indicate the execution mode of the block, and the number of instruction "body" chunks in the block. The store mask is used to enable distributed detection of block completion.

A block may contain up to four body chunks – each consisting of 32 instructions – for a maximum of 128 instructions, at most 32 of which can be loads and stores. All possible executions of a given block must emit the same number of outputs (stores, register writes, and one branch) regardless of the predicated path taken through the block. This constraint is necessary to detect block completion on the distributed substrate. The compiler generates blocks that conform to these constraints [33, 21].

3.2.2 Direct Instruction Communication

With *direct instruction communication*, instructions in a block send their results directly to intra-block, dependent consumers in a dataflow fashion. This model supports distributed execution by eliminating the need for any intervening shared, centralized structures (e.g. an issue window or register file) between intra-block producers and consumers.

Figure 3.2 shows that the TRIPS ISA supports direct instruction communication by encoding the consumers of an instruction's result as targets within the producing instruction. The microarchitecture can thus determine precisely where the

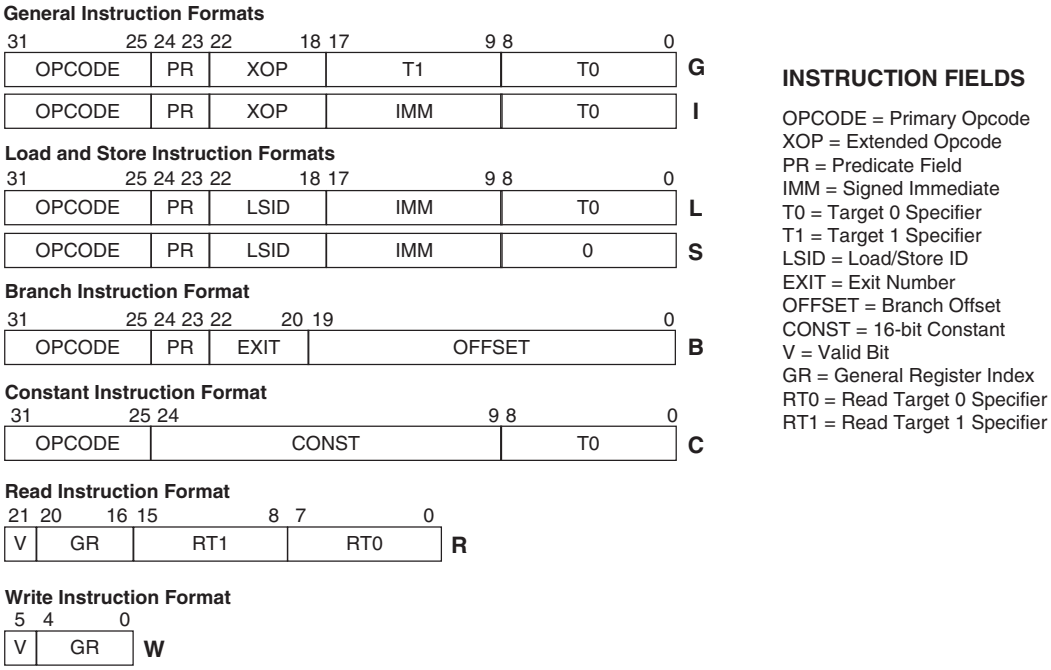


Fig. 3.2 TRIPS instruction formats

consumer resides and forward a producer's result directly to its target instruction(s). The nine-bit target fields (T0 and T1) each specify the target instruction with seven bits and the operand type (left, right, predicate) with the remaining two. A microarchitecture supporting this ISA maps each of a block's 128 instructions to particular coordinates, thereby determining the distributed flow of operands along the block's dataflow graph. The atomic blocks are necessary for direct instruction communication to bound the scope of the dataflow semantics, and therefore the size of each target. Outside of the blocks (i.e. between blocks), values are passed through registers and memory, similar to conventional instruction sets, and do not incorporate architecturally visible dataflow semantics. An instruction's coordinates are implicitly determined by its position in its chunk. Other non-traditional elements of this ISA include the "PR" field, which specifies an instruction's predicate and the "LSID" field, which specifies relative load/store ordering. Smith et al. [34] provide a detailed description of the prediction model in the TRIPS ISA.

3.2.3 Composable Architectures

Because they are well suited for distributed execution, EDGE ISAs are a good match for composable processors. Because of the block-atomic execution model, EDGE ISAs allow the control protocols for instruction fetch, completion, and commit to operate on large blocks – 128 instructions in the TRIPS ISA – instead of individual instructions. This amortization reduces the frequency of control decisions, reduces the overheads of bookkeeping structures, and makes the core-to-core

latencies tractable [28]. For example, since only one branch prediction is made per block, a prediction in TRIPS is needed at most once every eight cycles. This slack makes it possible for one core to make a prediction and then send a message to another core to make a subsequent prediction without significantly reducing performance.

Second, the target encoding of operand destinations eliminates the need for an operand broadcast bus, since a point-to-point network can interpret the identifiers as coordinates of instruction placement within arrays of reservation stations. In a composable architecture, differently sized "logical" processors merely interpret the target instructions' coordinates differently for each composed processor size. Changing the composition of a processor employing an EDGE ISA does not require any recompilation; instead the hardware is reconfigured to interpret register, memory, and target addresses differently for each distinct processor composition. This direct instruction communication also permits out-of-order execution with energy efficiency superior to conventional superscalar architectures: both use a dataflow graph of in-flight instructions for high performance, but the majority of the dataflow arcs in an EDGE processor are specified statically in the ISA (except for a minority of arcs through registers and memory), whereas the entire graph in a superscalar processor must be built and traversed in hardware, at considerable energy expense.

3.3 TRIPS Architecture and Implementation

The first implementation of an EDGE instruction set was the TRIPS microprocessor system, an experimental computer system consisting of custom ASIC chips and custom circuit boards. The TRIPS prototype was intended to demonstrate the viability of the EDGE instruction set and compilers, as well as feasibility of scalable and distributed processor and memory systems. A TRIPS processor has many parallel execution units, no global wires, is built from a small set of reused components on routed networks, and can be extended to a wider-issue implementation without recompiling source code or changing the ISA. Although the TRIPS processors granularity cannot be adjusted after manufacturing, our experience from the design of the prototype and evaluation of applications on TRIPS inspired the development of the flexible TFlex architecture, which is described in detail in Sect. 3.4. This section on TRIPS lays the foundation for distributed and composable architectures and describes the tiles, networks, and protocols needed for a distributed processor. Further detail on the TRIPS implementation can be found in [29].

3.3.1 TRIPS Tiled Microarchitecture

Unlike homogeneous tiles chip multiprocessor architectures such as Raw [41], TRIPS consists of heterogeneous tiles that collaborate to form larger processors and a larger unified level-2 cache. Fig. 3.3 shows the tile-level block diagram of the

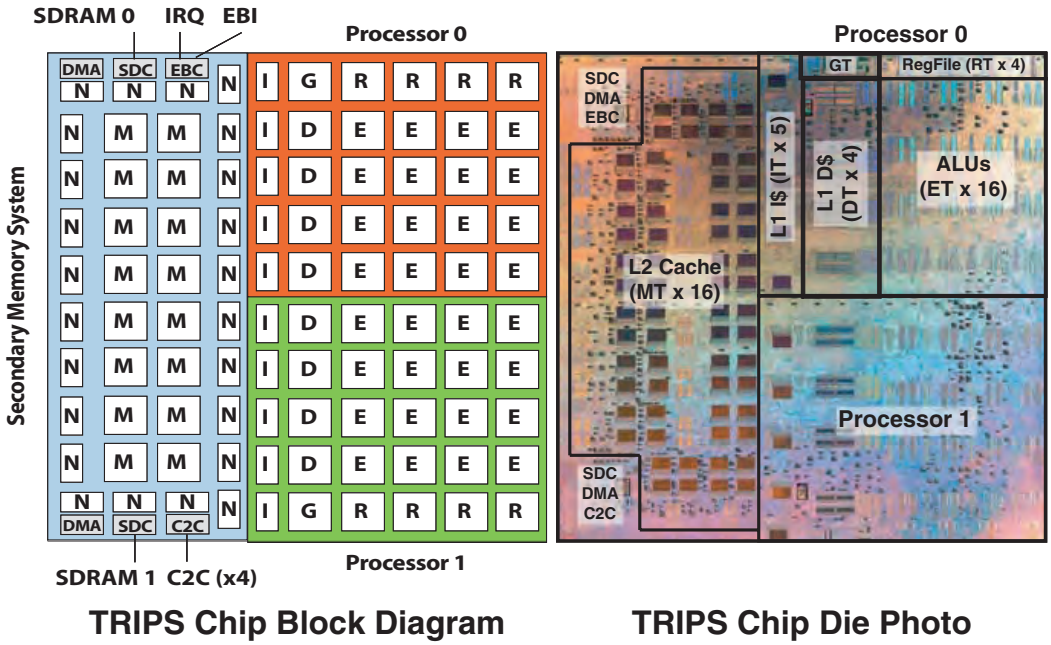


Fig. 3.3 TRIPS prototype block diagram

TRIPS prototype. The three major components on the chip are two processors and the secondary memory system, each connected internally by one or more micronetworks.

Each of the processor cores is implemented using five unique tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The main processor core micronetwork is the operand network (OPN), shown in Fig. 3.4. It connects all of the tiles except for the ITs in a two-dimensional, wormhole-routed, 5×5 mesh topology. The OPN has separate control and data channels, and can deliver one 64-bit data operand per link per cycle. A control header packet is launched one cycle in advance of the data payload packet to accelerate wakeup and select for bypassed operands that traverse the network.

Each processor core contains six other micronetworks, one for instruction dispatch – the global dispatch network (GDN) – and five for control: the global control network (GCN), for committing and flushing blocks; the global status network (GSN), for transmitting information about block completion; the global refill network (GRN), for I-cache miss refills; the data status network (DSN), for communicating store completion information; and the external store network (ESN), for determining store completion in the L2 cache or memory. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. Figure 3.4 shows the links for four of these networks.

This type of tiled microarchitecture is *composable* at design time, permitting different numbers and topologies of tiles in new implementations with only moderate changes to the tile logic, and no changes to the software model. The particular

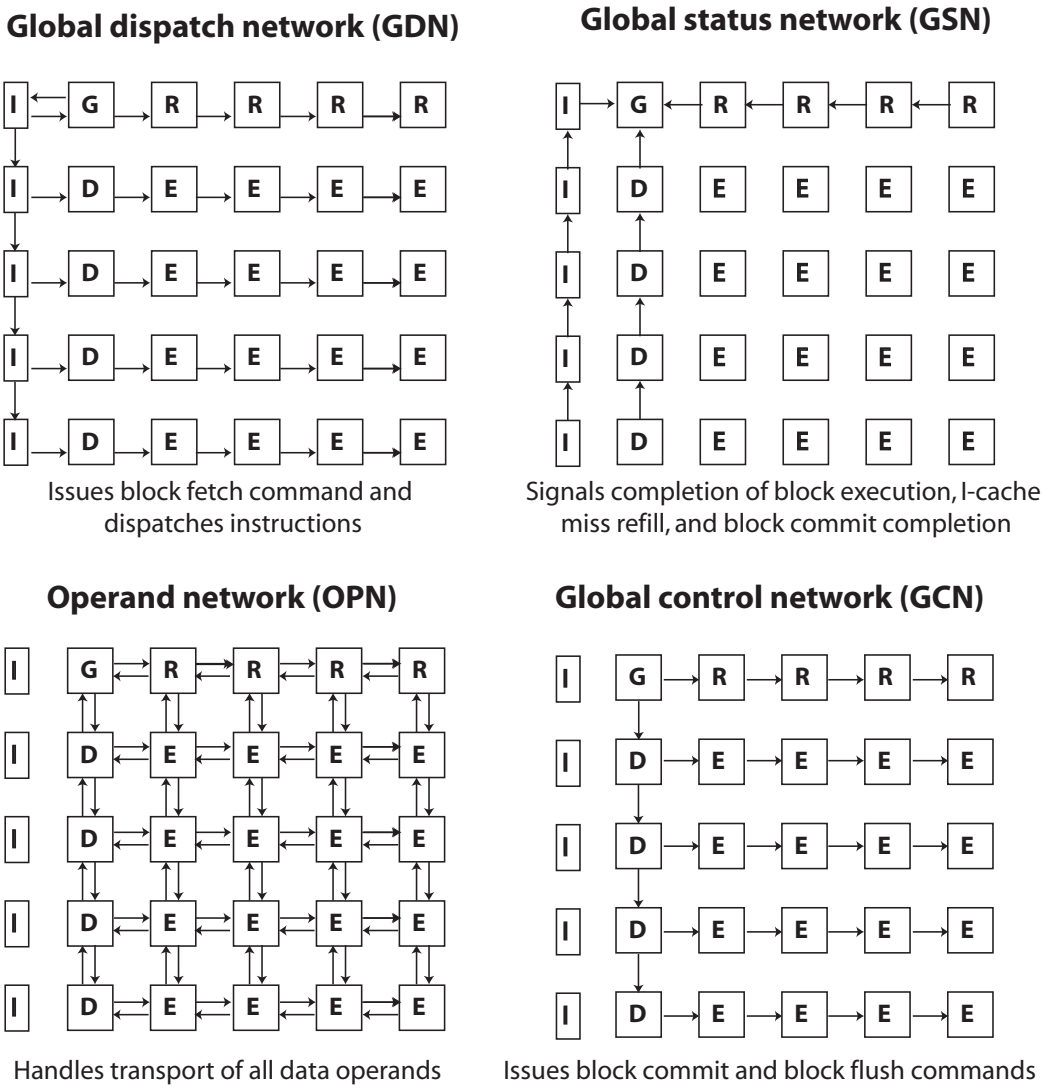


Fig. 3.4 Four TRIPS control and data micronetworks

arrangement of tiles in the prototype produces a core with 16-wide out-of-order issue, 64 KB of L1 instruction cache, and 32 KB of L1 data cache. We extended the register file and control logic to support four SMT threads as well. The microarchitecture supports up to eight TRIPS blocks in flight simultaneously, thus providing an in-flight window of up to 1024 instructions. The block control supports two modes: one non-speculative block and seven speculative blocks if a single thread is running, or two blocks per thread (one non-speculative and one speculative) if two to four threads are running.

The two processors communicate through the secondary memory system, in which the on-chip network (OCN) is embedded. The OCN is a 4×10 wormhole-routed mesh network, with 16-byte data links and four virtual channels. This network is optimized for cache-line sized transfers, although other request sizes are

supported for operations such as loads and stores to uncacheable pages. The OCN is the transport fabric for all inter-processor, L2 cache, DRAM, I/O, and DMA traffic.

The physical design and implementation of the TRIPS chip were driven by the principles of partitioning and replication. The chip floorplan directly corresponds to the logical hierarchy of TRIPS tiles connected only by point-to-point, nearest-neighbor networks. The only exceptions to nearest-neighbor communication are the global reset and interrupt signals, which are latency tolerant and pipelined in multiple stages across the chip.

The TRIPS chip is implemented in the IBM CU-11 ASIC process, which has a drawn feature size of 130 nm and seven layers of metal. The chip itself includes more than 170 million transistors and 1.06 km of wire routing, in a chip area of 18.3 mm by 18.4 mm placed in a 47.5 mm square ball-grid array package. Figure 3.3 shows an annotated die photo of the TRIPS chip, showing the boundaries of the TRIPS tiles. In addition to the core tiles, TRIPS also includes six controllers that are attached to the rest of the system via the on-chip network (OCN). The two 133/266 MHz DDR SDRAM controllers (SDC) each connects to an individual 1 GB SDRAM DIMM. The chip-to-chip controller (C2C) extends the on-chip network to a four-port mesh router that gluelessly connects to other TRIPS chips. These links nominally run at one-half the core processor clock and up to 266 MHz. The two direct memory access (DMA) controllers can be programmed to transfer data to and from any two regions of the physical address space, including addresses mapped to other TRIPS processors. Finally, the external bus controller (EBC) is the interface to a board-level PowerPC control processor. To reduce design complexity, we chose to off-load much of the operating system and runtime control to this PowerPC processor.

3.3.2 TRIPS Tiles

Global Control Tile (GT): Fig. 3.5a shows the contents of the GT, which include the blocks' PCs, the instruction cache tag arrays, the I-TLB, and the next-block predictor. The GT handles TRIPS block management, including prediction, fetch, dispatch, completion detection, flush (on mispredictions and interrupts), and commit. It also holds control registers that configure the processor into different speculation, execution, and threading modes. Thus, the GT interacts with all of the control networks as well as the OPN, which provides access to the block PCs.

The GT also maintains the status of all eight in-flight blocks. When one of the block slots is free, the GT accesses the block predictor, which takes three cycles, and emits the predicted address of the next target block. Each block may emit only one "exit" branch, even though it may contain several predicated branches. The block predictor uses a branch instruction's three-bit exit field to construct exit histories of two bits per block, instead of using per-branch taken/not-taken bits. The predictor has two major parts: an exit predictor and a target predictor. The predictor uses exit histories to predict one of eight possible block exits, employing a tournament

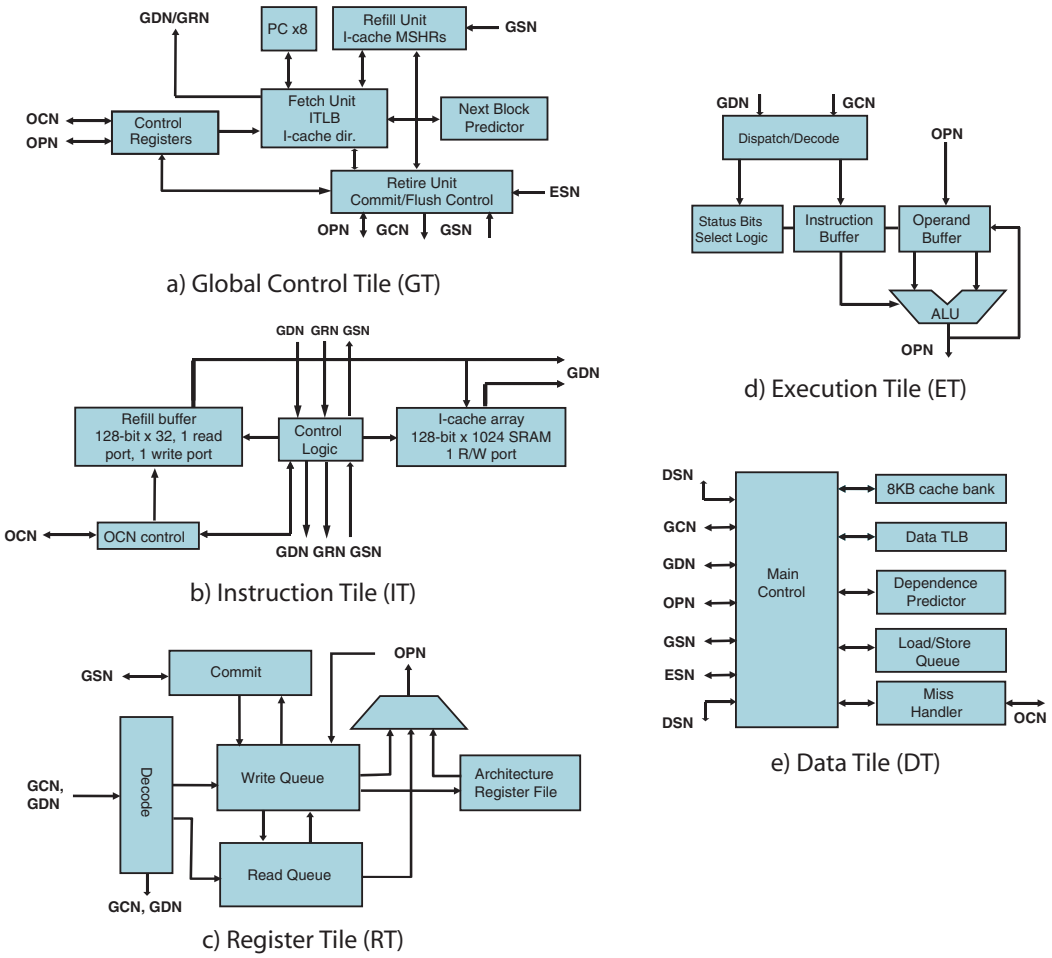


Fig. 3.5 TRIPS block-level tile microarchitectures

local/gshare predictor similar to the Alpha 21264 [13] with 9 K, 16 K, and 12 K bits in the local, global, and tournament exit predictors, respectively. The predicted exit number is combined with the current block address to access the target predictor for the next-block address. The target predictor contains four major structures: a branch target buffer (20 K bits), a call target buffer (6 K bits), a return address stack (7 K bits), and a branch-type predictor (12 K bits). The BTB predicts targets for branches, the CTB for calls, and the RAS for returns. The branch-type predictor selects among the different target predictions (call/return/branch/sequential branch). The distributed fetch protocol necessitates the type predictor; the predictor never sees the actual branch instructions, as they are sent directly from the ITs to the ETs.

Instruction Tile (IT): Figure 3.5b shows an IT, which contains a two-way, 16 KB bank of the total L1 I-cache and acts as a slave to the GT, which holds the single-tag array. Each of the five 16 KB IT banks can hold a 128-byte chunk (for a total of 640 bytes for a maximum-sized block) for each of 128 distinct blocks. Each handles refills as slaves to tag misses in the GT. A control network among the ITs

informs the GT only when all of the ITs notify this network that all four refills have completed.

Register Tile (RT): To reduce power consumption and delay, the TRIPS microarchitecture partitions its many registers into banks, with one bank in each RT. The register tiles are clients on the OPN, allowing the compiler to place critical instructions that read and write from/to a given bank close to that bank. Since many def-use pairs of instructions are converted to intra-block temporaries by the compiler, they never access the register file, thus reducing total register bandwidth requirements by approximately 70%, on average, compared to a RISC or CISC processor. The four distributed banks can thus provide sufficient register bandwidth with a small number of ports; in the TRIPS prototype, each RT bank has two read ports and one write port. Each of the four RTs contains one 32-register bank for each of the four SMT threads that the core supports, for a total of 128 registers per RT and 128 registers per thread across the RTs.

In addition to the four per-thread architecture register file banks, each RT contains a read queue and a write queue, as shown in Fig. 3.5c. These queues hold up to eight read and eight write instructions from the block header for each of the eight blocks in flight, and are used to forward register writes dynamically to subsequent blocks reading from those registers. The read and write queues perform a function equivalent to register renaming for a superscalar physical register file, but are less complex to implement due to the read and write instructions in the TRIPS ISA.

Execution Tile (ET): As shown in Fig. 3.5d, each of the 16 ETs consists of a fairly standard single-issue pipeline, a bank of 64 reservation stations, an integer unit, and a floating-point unit. All units are fully pipelined except for the integer divide unit, which takes 24 cycles. The 64 reservation stations hold eight instructions for each of the eight in-flight TRIPS blocks. The reservation stations are partitioned into banks for instructions, each of two operands, and a bank for the one-bit predicate.

Data Tile (DT): Figure 3.5e shows a block diagram of a single DT. Each DT is a client on the OPN, and holds one two-way, 8 KB L1 data cache bank, for a total of 32 KB across the four DTs. Virtual addresses are interleaved across the DTs at the granularity of a 64-byte cache-line. In addition to the L1 cache bank, each DT contains a copy of the load/store queue (LSQ), a dependence predictor similar to a load-wait predictor, a one-entry backside coalescing write buffer, a data TLB, and a MSHR bank that supports up to 16 requests for up to four outstanding cache lines.[1] Because the DTs are distributed in the network, we implemented a *memory-side* dependence predictor, closely coupled with each data cache bank [30]. Although loads issue from the ETs, a dependence prediction occurs (in parallel) with the cache access only when the load arrives at the DT. An additional challenge for distributed L1-cache designs is the memory disambiguation logic (load/store queue or LSQ) to enable out-of-order execution of unrelated loads and stores. In the TRIPS prototype,

[1] MSHR stands for *miss status handling register* or *miss status holding register*; an MSHR maintains the state in the processor required to resume a cache access that causes a cache miss.

we sized the LSQ in each DT to 256 entries so that space is available in the unlikely event that all in-flight loads and stores target the same DT. Subsequent research has devised more area and power-efficient solutions to this difficult problem [31].

Secondary Memory System: The TRIPS prototype supports a 1 MB static NUCA [15] array, organized into 16 memory tiles (MTs), each one of which holds a four-way, 64 KB bank. Each MT also includes an OCN router and a single-entry MSHR. Each bank may be configured as an L2 cache bank or as a scratch-pad memory, by sending a configuration command across the OCN to a given MT. By aligning the OCN with the DTs, each IT/DT pair has its own private port into the secondary memory system, supporting high bandwidth into the cores for streaming applications. The network tiles (NTs) surrounding the memory system act as translation agents for determining where to route memory system requests. Each of them contains a programmable routing table that determines the destination of each memory system request. By adjusting the mapping functions within the TLBs and the network interface tiles (NTs), a programmer can configure the memory system in a variety of ways, including as a single 1 MB shared level-2 cache, as two independent 512 KB level-2 caches (one per processor), or as a 1 MB on-chip physical memory (no level-2 cache).

3.3.3 Distributed Microarchitectural Protocols

To enable concurrent, out-of-order execution on this distributed substrate, we implemented traditionally centralized microarchitectural functions, including fetch, execution, flush, and commit, with distributed protocols running across the control and data micronets.

3.3.3.1 Block Fetch Protocol

The fetch protocol retrieves a block of 128 TRIPS instructions from the ITs and distributes them into the array of ETs and RTs. In the GT, the block fetch pipeline takes a total of 13 cycles, including three cycles for prediction, one cycle for TLB and instruction cache tag access, and one cycle for hit/miss detection. On a cache hit, the GT sends eight pipelined indices out on the global dispatch network (GDN) to the ITs. Prediction and instruction cache tag lookup for the next block are overlapped with the fetch commands of the current block. Although prediction requires three cycles, as does predictor training, the eight-cycle fetch latency is sufficient to hide the six cycles per block of predictor operations. Thus, running at peak, the machine can issue fetch commands every cycle with no bubbles, beginning a new block fetch every eight cycles.

When an IT receives a block dispatch command from the GT, it accesses its I-cache bank based on the index in the GDN message. In each of the next eight cycles, the IT sends four instructions on its outgoing GDN paths to its associated row of ETs (or RTs). These instructions are written into the read and write queues of the RTs and the reservation stations in the ETs when they arrive at their respective

tiles, and are available to execute as soon as they arrive. Since the fetch commands and fetched instructions are delivered in a pipelined fashion across the ITs, ETs, and RTs, the furthest ET receives its first instruction packet ten cycles and its last packet 17 cycles after the GT issues the first fetch command. While the latency appears high, the pipelining enables a high-fetch bandwidth of 16 instructions per cycle in steady state, with multiple fetched blocks in flight at once, producing one instruction per ET per cycle.

On an I-cache miss, the GT instigates a distributed I-cache refill, using the global refill network (GRN) to transmit the refill block's physical address to all of the ITs. Each IT processes the misses for its own chunk independently, and can simultaneously support one outstanding miss for each executing thread (across four threads).

3.3.3.2 Distributed Execution

Dataflow execution of a block begins with the injection of block inputs by the RTs. An RT may begin to process an arriving read instruction even if the entire block has not yet been fetched. Each RT first searches the write queues of all older in-flight blocks. If no matching, in-flight write to that register is found, the RT simply reads that register from the architectural register file and forwards it to the consumers in the block via the OPN. If a matching write is found, the RT takes one of two actions: if the write instruction has received its value, the RT forwards that value to the read instruction's consumers. If the write instruction is still awaiting its value, the RT buffers the read instruction, which will be woken up by a tag broadcast when the pertinent write's value arrives. While CAMs are typically expensive in terms of energy, they are relatively small, and are accessed less often than the number of instructions, due to the block model.

Arriving OPN operands wake up instructions within an ET, which selects and executes enabled instructions. The ET uses the target fields of the selected instruction to determine where to send the resulting operand. Arithmetic operands traverse the OPN to other ETs, while load and store instructions' addresses and data are sent on the OPN to the DTs. Branch instructions deliver their next block addresses to the GT via the OPN.

An issuing instruction may target its own ET or a remote ET. If it targets its local ET, the dependent instruction can be woken up and executed in the next cycle using a local bypass path to permit back-to-back issue of dependent instructions. If the target is a remote ET, a control packet is formed in the cycle before the operation will complete execution, and is then sent to wake up the dependent instruction early. The OPN is tightly integrated with the wakeup and select logic. When a control packet arrives from the OPN, the targeted instruction is accessed and may be speculatively woken up. The instruction may begin execution in the following cycle as the OPN router injects the arriving operand directly into the ALU. Thus, for each OPN hop between dependent instructions, there will be a one-cycle bubble before the consuming instruction executes.

Figure 3.6 shows an example of how a code sequence is executed on the RTs, ETs, and DTs. The leftmost labels (e.g., N[1]) are included for readability only and

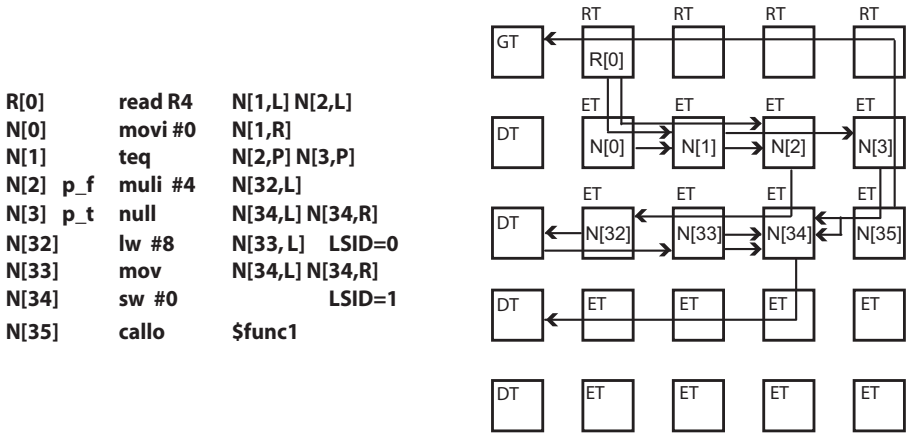


Fig. 3.6 TRIPS program and program execution

are not part of the instructions. Block execution begins when the read instruction R[0] is issued to RT0, triggering delivery of R4 via the OPN to the left operand of two instructions, teq (N[1]) and muli (N[2]). When the test instruction receives the register value and the immediate "0" value from the movi instruction, it fires and produces a predicate which is routed to the predicate field of N[2]. Since N[2] is predicated on false, if the routed operand has a value of 0, the muli will fire, multiply the arriving left operand by four, and send the result to the address field of the lw (load word). If the load fires, it sends a request to the pertinent DT, which responds by routing the loaded data to N[33]. The DT uses the load/store IDs (0 for the load and 1 for the store, in this example) to ensure that they execute in the proper program order if they share the same address. The result of the load is fanned out by the mov instruction to the address and data fields of the store.

If the predicate's value is 1, N[2] will not inject a result into the OPN, thus suppressing execution of the dependent load. Instead, the null instruction fires, targeting the address and data fields of the sw (store word). Note that although two instructions are targeting each operand of the store, only one will fire, due to the predicate. When the store is sent to the pertinent DT and the block-ending call instruction is routed to the GT, the block has produced all of its outputs and is ready to commit. Note that if the store is nullified, it does not affect memory, but simply signals the DT that the store has issued. Nullified register writes and stores are used to ensure that the block always produces the same number of outputs for completion detection.

3.3.3.3 Block/Pipeline Flush Protocol

Because TRIPS executes blocks speculatively, a branch misprediction or a load/store ordering violation causes periodic pipeline flushes. These flushes are implemented using a distributed protocol. The GT is first notified when a mis-speculation occurs, either by detecting a branch misprediction itself or via a GSN message from a DT indicating a memory-ordering violation. The GT then initiates a flush wave on the

GCN which propagates to all of the ETs, DTs, and RTs. The GCN includes an eight-bit block identifier mask indicating which block or blocks must be flushed. The processor must support multi-block flushing because all speculative blocks after the one that caused the misspeculation must also be flushed. This wave propagates at one hop per cycle across the array, holding the flush command high for three cycles at each point to eliminate mispredicted in-flight state. As soon as it issues the flush command on the GCN, the GT may issue a new dispatch command to start a new block. Because both the GCN and GDN guarantee in-order delivery with no contention, the instruction fetch/dispatch command can never catch up with or pass the flush command.

3.3.3.4 Block Commit Protocol

Block commit is the most complex of the microarchitectural protocols in TRIPS, and involves three phases: completion, commit, and commit acknowledgment. In phase one, a block is complete when it has produced all of its outputs, the number of which is determined at compile time and consists of up to 32 register writes, up to 32 stores, and exactly one branch. After the RTs and DTs receive all of the register writes or stores for a given block, they inform the GT using the global status network (GSN). When an RT detects that all block writes have arrived, it informs its west neighbor. The RT completion message is daisy-chained westward across the RTs, until it reaches the GT indicating that all of the register writes for that block have been received. The DTs use a similar protocol for notifying the GT that all stores have completed, but require some additional communication among them because the number of stores that will arrive at each DT depends on the store addresses computed during program execution.

During the second phase (block commit), the GT broadcasts a commit command on the global control network and updates the block predictor. The commit command informs all RTs and DTs that they should commit their register writes and stores to architectural state. To prevent this distributed commit from becoming a bottleneck, the logic supports pipelined commit commands. The GT can legally send a commit command on the GCN for a block in the next cycle after a commit command has been sent for the previous older in-flight block, even if the commit commands for older blocks are still in flight. The pipelined commits are safe because each tile is guaranteed to receive and process them in order. The commit command on the GCN also flushes any speculative in-flight state in the ETs and DTs for that block.

The third phase acknowledges the completion of commit. When an RT or DT has finished committing its architectural state for a given block and has received a commit completion signal from its neighbor on the GSN (similar to block completion detection), it signals commit completion on the GSN. When the GT has received commit completion signals from both the RTs and DTs, it knows that the block is safe to deallocate, because all of the block's outputs have been written to architectural state. When the oldest block has acknowledged commit, the GT initiates a block fetch and dispatch sequence for that block slot.

3.3.4 Discussion

The tiled nature of the TRIPS processor made it easy to implement, even with a design team consisting of a small number of graduate students. Although the TRIPS chip consists of 11 different types of tiles, the reuse (106 instances) and the design style enforced via well-engineered interconnection networks resulted in a straightforward verification strategy and fully functional first silicon.

Our experimental results show that even with a nascent compiler, one TRIPS processor can be competitive with commercial microprocessors on serial programs. Table 3.2 summarizes the performance of the TRIPS prototype relative to two commercial systems; a full performance evaluation of the TRIPS prototype can be found in [6]. Because of the different process technologies and design methodologies, we use cycle count as the performance metric, thus normalizing out the differences in clock rate. The ratio of processor speed to memory speed is approximately the same across all of the systems. We also use the GNU C compiler gcc on the Intel platforms as it has similar scalar optimizations as the TRIPS compiler. In general, TRIPS performs well on computationally intensive applications, approaching the performance of the Core 2 on EEMBC and SpecFP. On less regular applications, TRIPS compiled code performs less well relative to the Core 2 because of more frequent and less predictable branches as well as less inherent instruction-level parallelism in the application.

Table 3.2 Performance relative to Intel Core 2

Benchmark suite	Optimization	Intel Core 2 (gcc)	Intel Pentium 4 (gcc)	TRIPS
Simple	TRIPS hand optimized	1.00	0.51	2.77
EEMBC	Compiled	1.00	0.53	0.73
SpecINT 2000	Compiled	1.00	0.49	0.35
SpecFP 2000	Compiled	1.00	0.47	1.00
Process		65 nm	90 nm	130 nm
Power		65 W (TDP)	84 W (TDP)	24 W (actual)
Scaled power				19 W (scaled)
Clock rate		1.6 GHz	3.6 GHz	366 MHz

However, the TRIPS compiler is still relatively immature and has many opportunities for improvement. To determine the underlying capabilities of the architecture, we extracted key functions from 15 different benchmarks and performed source-code and some assembly code optimizations to enable better block formation. On these programs (labeled *Simple* in Table 3.2), TRIPS often performs three to four times better than the Core 2, and averages 2.8 times better. For these applications, exploiting instruction-level parallelism is effective. We also found that while some of the applications we examined used the abundance of execution and data resources effectively, others had serial program phases that resulted in inefficient use of the TRIPS tiles. Because the fixed TRIPS design was a poor match for these applications, we sought to devise

architectures that could be adapted to the parallelism granularity of the running application: applications with serial threads would use narrow-issue processors while applications with more abundant ILP would employ wider-issue processors. The configurable processor design described in the next section is the result of this investigation.

Because it does not require the large associative structures found in conventional processors, TRIPS has the potential for greater power efficiency. This efficiency is not obvious in the TRIPS prototype, as it lacks any power management such as clock gating or voltage/frequency scaling. A rudimentary scaling model that accounts for power management, technology scaling, clock rate scaling, and additional leakage power shows that TRIPS should consume about 20 W at the same frequency and technology as the Core 2. While 20 W cannot be compared to the thermal design point (TDP) of 65 W for the Core 2, achieving comparable or better power consumption at three times the performance on general purpose codes is within the potential of TRIPS architectures. Further, power efficiency can be achieved by adapting the granularity of the processors to the concurrency of the application.

3.4 Composable Lightweight Processor Architectures

TFlex is a second-generation EDGE microarchitecture that provides the capability of dynamically composing cores to form variable-granularity processors. Figure 3.7 shows an array of 32 TFlex tiles and the internals of one of the cores. The L2 cache banks and the peripheral circuitry are not shown in this diagram. In the figure, each TFlex core includes two execution units with a dual-issue pipeline, an

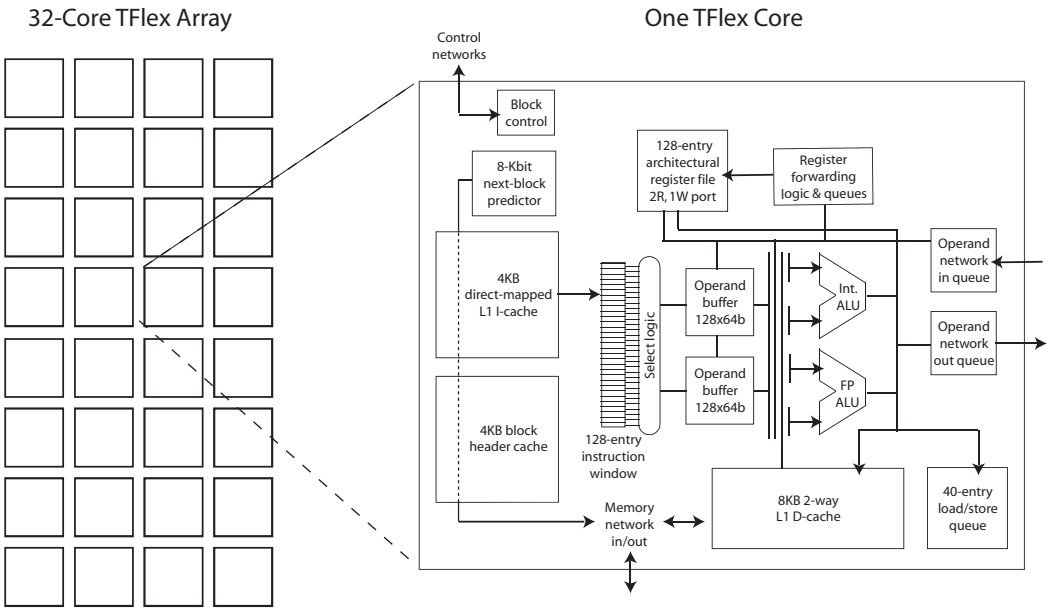
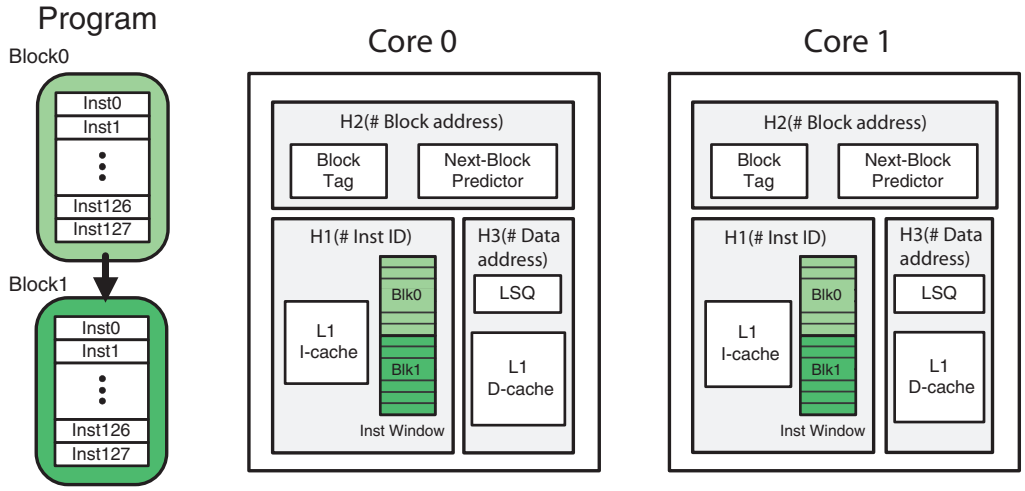


Fig. 3.7 TFlex core microarchitecture

8 Kbit next-block predictor, a 4 KB instruction cache, an 8 KB data cache, a 128-instruction window, a 40-entry load/store queue, and ports to networks that connect to other cores. A TFlex area model estimates that a single TFlex core would consume 2.4 mm^2 in a 65 nm technology, which would make it smaller than two of the termite cores in Table 3.1.

A truly composable processor has microarchitectural structures that linearly increase (or decrease) in capacity as participating cores are added (or removed). For example, doubling the number of cores should double the number of useful load/store queue entries, the usable state in the branch predictors, and the cache capacities. The key mechanisms for dynamically composing cores into differently sized logical processors are (1) composing multiple microarchitecture structures into a single logical but distributed microarchitecture structure, (2) interleaving the addresses to the structure across its different banks, and (3) employing a programmable hashing function that changes the interleaving depending on the number of cores that compose the logical processor. For example, when two CLP cores form a single logical processor, the level-1 data cache addresses are interleaved across the banks using the low-order bits of the index to select the core containing the right cache bank. If two more CLP cores are added, the hashing function is changed to interleave the cache-line addresses across four banks. Concretely, the TFlex microarchitecture (Fig. 3.8) uses three distinct hash functions for interleaving across three classes of structures.



Hash1 (# Inst ID) = (# Inst ID) % (# of participating cores)

Hash2 (# Block address) = ((# Block address) >> 12) % (# of participating cores)

Hash3 (# Data address) =
((High-order bits from # data address) XOR (Low-order bits from # data address)) %
(# of participating cores)

Fig. 3.8 Interleaving of different microarchitectural structures for a two-core processor

Block starting address: The next-block predictor resources (e.g., BTBs and local history tables) and the blog I-cache tag structures are partitioned based on the starting virtual address of a particular block, which corresponds to the program counter

in a conventional architecture. Predicting control flow and fetching instructions in TFlex occur at the granularity of a block, rather than individual instructions.

Instruction ID within a block: A block contains up to 128 instructions, which are numbered in order. Instructions are interleaved across the partitioned instruction windows and instruction caches based on the instruction ID, theoretically permitting up to 128 cores each holding one instruction from each block.

Data address: The load–store queue (LSQ) and data caches are partitioned by the low-order index bits of load/store addresses, and registers are interleaved based on the low-order bits of the register number.

The register names are interleaved across the register files. However, because a single core must have 128 registers to support single-block execution, register file capacity goes unused when multiple cores are aggregated. Because interleaving is controlled by bit-level hash functions, the number of cores that can be aggregated to form a logical processor must be a power of two.

In addition, composability requires distributed protocols to accomplish traditionally centralized microarchitecture operations. We adapted the distributed protocols of the TRIPS microarchitecture to a system consisting of homogeneous cores which are described in the subsequent sections.

3.4.1 Overview of TFlex Operation

Since TFlex employs an EDGE instruction set, its block execution model is similar to TRIPS. However, when cores are composed, TFlex uses a distributed fetch protocol in which each in-flight block is assigned a single *owner core*, based on a hash of the block address. The owner core is responsible for initiating fetch of that block and predicting the next block. Once the next-block address is predicted, the owner core sends that address to the core that owns the next predicted block. For each block that it owns, an owner core is also responsible for launching pipeline flushes, detecting block completion, and committing the block.

Figure 3.9 provides an overview of TFlex execution for the lifetime of one block. It shows two threads running on non-overlapping sets of eight cores each. In the block fetch stage, the block owner accesses the I-cache tag for the current block and broadcasts the fetch command to the I-cache banks in the participating cores (Fig. 3.9a). Upon receiving a fetch command from a block owner, each core fetches its portion of the block from its local I-cache, and dispatches fetched instructions into the issue window. In parallel, the owner core predicts the next block address and sends a control message – plus the next block address and global history register – to the next block owner to initiate fetch of the subsequent block (Fig. 3.9b). Up to eight blocks (one per core) may be in flight for eight participating cores. Instructions are executed in dataflow order when they are ready (Fig. 3.9c). When a block completes, the owner detects completion, and when it is notified that it holds the oldest block, it launches the block commit protocol, shown in Fig. 3.9d. While Fig. 3.9e–h shows the same four stages of execution for the next block controlled by

Attachment - Page A260

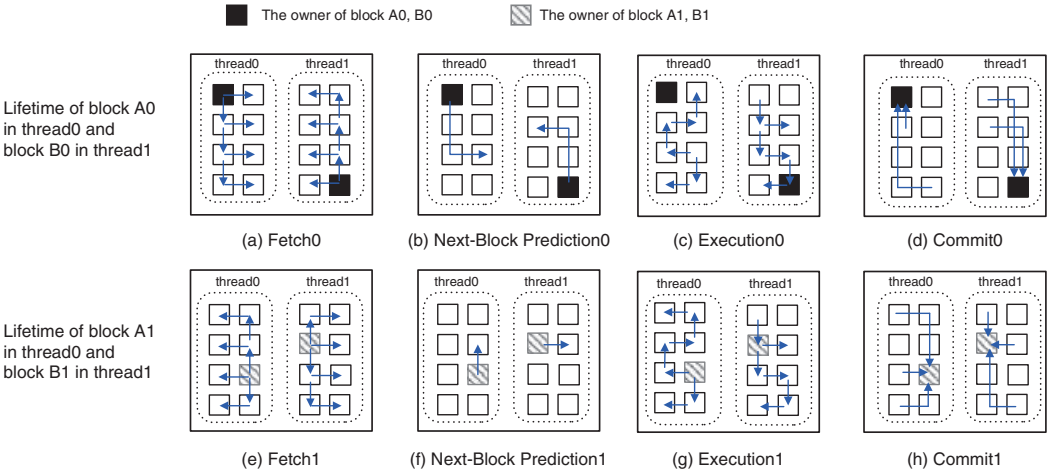


Fig. 3.9 TFlex execution stages: execution of two successive blocks (A0, A1) and (B0,B1) from two different threads executing simultaneously on a 16-core TFlex CLP with each thread running on eight cores

a different owner, fetch, execution, and commit of the blocks are actually pipelined and overlapped.

3.4.2 Composable Instruction Fetch

In the TFlex microarchitecture, instructions are distributed across the private I-caches of all participating cores, but fetches are initiated by each of the instruction block's owner cores. The owner core manages the tags on a per-block basis, and functions as a central registry for all operations of the blocks it owns. The cache tags for a block are held exclusively in the block owner core. In a 32-core configuration, each core caches four instructions from a 128-instruction block.

In TFlex, the fetch bandwidth and the I-cache capacity of the participating cores scale linearly as more cores are used. While this partitioned fetch model amplifies fetch bandwidth, conventional microarchitectures cannot use it because current designs require a centralized point to analyze the fetch stream (for register renaming and instruction number assignment) to preserve correct sequential semantics. This constraint poses the largest challenge to composable processors built with conventional ISAs. In contrast, EDGE ISAs overcome these deficiencies by explicitly and statically encoding the dependences among the instructions within a block; the dynamic total order among all executing instructions is obtained by concatenating the block order and the statically encoded order within a block. Given large block sizes, distributed fetching is feasible because intra-block instruction dependence relations are known a priori.

3.4.3 Composable Control-Flow Prediction

Control-flow predictors are some of the most challenging structures to partition for composability since the predictor state has traditionally been physically centralized to facilitate few cycles between successive predictions. The TFlex composable predictor treats the distributed predictors in each composed core as a single logical predictor, exploiting the block-atomic nature of the TRIPS ISA to make this distributed approach tenable. Similar to the TRIPS prototype microarchitecture, the TFlex control-flow predictor issues one next-block prediction for each 128-instruction hyperblock – a predicated single entry, multiple exit region – instead of one prediction per basic block.

Figure 3.10 illustrates the distributed next-block predictor which consists of eight main structures. Each core has a fully functional block predictor which is identical to all others across the cores. The block predictor is divided into two main components: the exit predictor, which predicts which exit will be taken out of the current block; and the target predictor, which predicts the address of the next block. Each branch in a block contains three *exit* bits in its instruction, which are used to form histories without traditional taken/not taken bits.

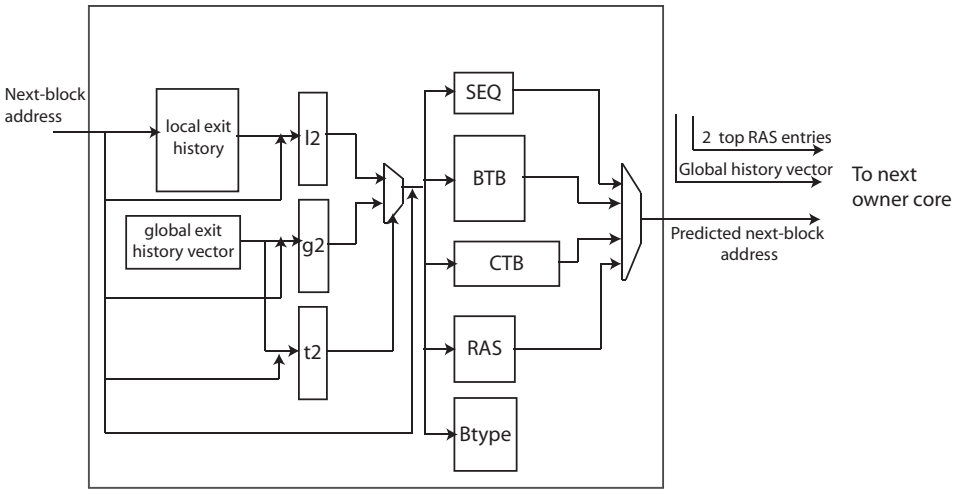


Fig. 3.10 TFlex next-block predictor

The exit predictor is an Alpha 21264-like tournament-style hybrid predictor [13] and is composed of traditional two-level local, global, and choice predictors that use local and global block exit histories. Speculative history updates are maintained and used to make predictions, updating local and global histories when blocks commit. The target address is predicted by first predicting the type of the exit branch – a call, a return, a next sequential block, or a regular branch – using the branch type (Btype) predictor. The Btype predictor result selects one of those four possible next-block targets, which are provided by a next-block adder (SEQ), a branch target buffer (BTB) to predict branch targets, a call target buffer (CTB) to predict call targets, and a return address stack (RAS) to predict return targets.

The predictor organization described above is naturally composable. The local histories are composable since, for a fixed composition, the same block address will always map to the same core. Local predictions do not lose any information even though they are physically distributed. Similarly, the Btype, BTB, and CTB tables hold only the target addresses of the blocks owned by that core. With this organization, the capacity of the predictor increases as more cores are added, assuming that block addresses tend to be distributed among cores equally.

To enable prediction with current global histories, the predicted next-block address is sent from the previous blocks' owner to the owning core of the predicted block along with the global exit history. This forwarding enables each prediction to be made with the current global history, without additional latency beyond the already incurred point-to-point latency to transmit the predicted next-block address from core to core.

The return address stack (RAS) is maintained as a single logical stack across all the cores, since it represents the program call stack which is necessarily logically centralized. The TFlex microarchitecture uses the RAS entries across different tiles to enable a deeper composed RAS. Instead of using address interleaving, the RAS is sequentially partitioned across all the cores (e.g., a 32-entry stack for two cores would have entries 0–15 in core 0 and 16–31 in core 1). The stacks from the participating cores form a logically centralized but physically distributed global stack. If the exit branch type is predicted as a call, the corresponding return address is pushed on to the RAS by sending a message to the core holding the current RAS top. If the branch is predicted as a return, the address on the stack is popped off by sending a pop-request to the core holding the RAS top. Since prediction is a time-critical operation, the latency of predicting return addresses is optimized by passing the two top RAS values from owner core to owner core. Recovery upon a misprediction is the responsibility of the mispredicting owner, which rolls back the mis speculated state and sends the updated histories and RAS pointers to the next owner core, as well as the corrected top-of-stack RAS information to the core that will hold the new RAS top.

3.4.4 Composable Instruction Execution

Each instruction in an EDGE block is statically assigned an identifier between 0 and 127 by the TRIPS compiler. A block's instructions are interleaved across the cores of a composable processor by the hash function described above. Changing the mapping function when cores are added (or removed) achieves composability of instruction execution. Figure 3.11 shows the mechanism that the TFlex design uses to support dynamic issue across a variable number of composed cores. Each instruction in an EDGE ISA block contains at least one nine-bit *target* field, which specifies the location of the dependent instruction that will consume the produced operand. Two of the nine bits specify which operand of the destination instruction is targeted (left, right, or predicate), and the other seven bits specify which of the

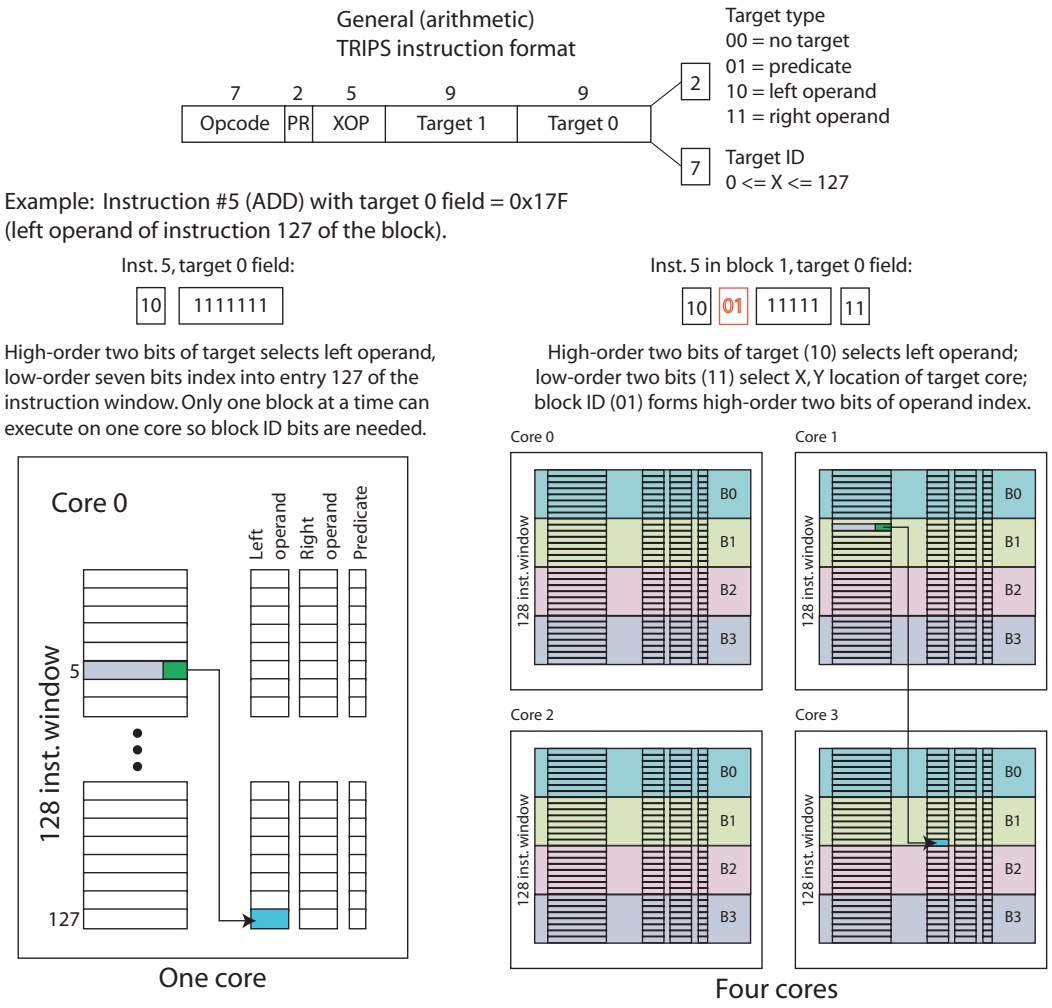


Fig. 3.11 Block mapping for one-core and four-core processors

128 instructions is targeted. Figure 3.11 shows how the target bits are interpreted in single-core mode, with instruction five targeting the left operand of instruction 127. In single-core mode, all seven target identifier bits are used to index into a single 128-instruction buffer.

Figure 3.11 also shows how the microarchitecture interprets the target bits when running in a four-core configuration. The four cores can hold a total of four instruction blocks, but each block is striped across the four participating cores. Each core holds 32 instructions from each of the four in-flight blocks. In this configuration, the microarchitecture uses the two low-order bits from the target to determine which core is holding the target instruction, and the remaining five bits to select one of the 32 instructions on that core. The explicit dataflow target semantics of EDGE ISAs make this operation simple compared to what would be required in a RISC or CISC ISA, where the dependences are unknown until an instruction is fetched and analyzed.

3.4.5 Composable Memory System

As with clustered microarchitectures [19, 26], L1 data caches in a composable processor can be treated as a single logical cache by treating the individual caches as address-partitioned banks. When running in one-core mode, each thread accesses only its bank. When multiple cores are composed, the L1 cache becomes a cache-line interleaved aggregate of all the participating L1 caches. With each additional core, a thread obtains more L1 D-cache capacity and an additional memory port. The cache bank accessed by a memory instruction is determined by the low-order bits of the set index; the number of bits is determined by the number of participating cores. All addresses within a cache line will always map to the same bank in a given configuration. When a core computes the effective address of a load, the address and the target(s) of the load are routed to the appropriate cache bank, the lookup is performed, and the result is directly forwarded to the core containing the target instruction of the load.

One of the microarchitectural challenges to support a composable memory system is to handle memory disambiguation efficiently on a distributed substrate with a variable number of cores. The TFlex microarchitecture relies on load–store queues (LSQs) to disambiguate memory accesses dynamically. As more cores are aggregated to construct a larger window, more entries in the LSQ are required to track all in-flight memory instructions. Partitioning LSQ banks by address and interleaving them with the same hashing function as the data caches is a natural way to build a large, distributed LSQ. However, unless each LSQ bank is maximally sized for the worst case (the instruction window size), a system must handle the situation when a particular LSQ bank is full, and cannot slot an incoming memory request (called "LSQ overflow"). TFlex uses a low-overhead NACK mechanism to handle LSQ overflows efficiently [31].

3.4.6 Composable Instruction Commit

Block commits in TFlex are staged across four phases. First, the block owner detects that a block is complete when participating cores inform the owner core that the block has emitted all of its outputs – stores, register writes, and one branch. Register files send a message to the owner core when they have received all of their register writes, and each completing store sends a control message to the owner core, which can then determine when all outputs have been received. Second, when the block becomes the oldest one, the block owner sends out a *commit* command on the control network to the participating cores. Third, all distributed cores write their outputs to architectural state, and when finished, respond with *commit acknowledgment* signals. Finally, the owner core broadcasts a resource deallocation signal, indicating that the youngest block owner can initiate its own fetch and overwrite the committed block with a new block.

3.4.7 Coherence Management

For a level-two (L2) cache organization, we consider a 4 MB shared design (shown in Fig. 3.1) that contains 32 cache banks connected by a switched mesh network. To maintain coherence among private L1 caches, the shared L2 cache uses a standard on-chip directory-based cache coherence protocol with L1 sharing vectors stored in the L2 tag arrays. The composition of the cores on the chip does not affect how the coherence protocol is implemented. A sharing status vector in the L2 tag keeps track of L1 coherence by treating each L1 cache as an independent coherence unit. When a composition changes – adding cores to some composed processors and removing them from others – the L1 caches need not be flushed; the new interleaved mapping for the composed L1 D-cache banks will result in misses, at which point the underlying coherence mechanism forwards the request to the lines stored in the old L1 cache banks, invalidating them or forwarding a modified line.

3.5 Experimental Evaluation

To explore the benefits of flexibility and composability, we compare the performance of the composable TFlex architecture to a configuration similar to the TRIPS architecture on single-threaded applications. Then we employ a set of multiprogrammed workloads to measure the benefits of composability on throughput-oriented workloads. These analyses use a validated cycle-accurate simulator with a set of both compiler-generated and hand-optimized benchmarks shown in Table 3.3. For the TFlex configurations, the programs are scheduled assuming a 32-core composable processor.

Table 3.3 Single-core TFlex microarchitecture parameters

Parameter	Configuration
Instruction supply	Partitioned 8 KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8 K+256 bits, 3 cycle latency) with speculative updates; Num. entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP).
Data supply	Partitioned 8 KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 44-entry LSQ bank; 4 MB decoupled S-NUCA L2 cache [15] (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.
Simulation	Execution-driven simulator validated within 7% of TRIPS hardware
Hand-optimized	3 kernels (conv, ct, genalg), 7 EEMBC benchmarks (a2time, autocor, basefp, bezier, dither, rspeed, tblook), 2 Versabench (802.11b, 8b10b) [25]
Compiled	14 SPEC CPU benchmarks currently supported (8 Integer, 6 FP), simulated with single simpoints of 100 million cycles [32].

The sizes of the structures in the core ensure that one core can execute and atomically commit one EDGE block. Each core has a 128-entry instruction window to accommodate all instructions in a block, 128 registers, and an LSQ large enough to hold at least 32 loads/stores. The sizes of the remaining structures, including the I-cache, D-cache, and the branch predictor, were sized to balance area overhead and performance. The simulated baseline TRIPS microarchitecture matches that described by Sankaralingam et al. [29], with the exception that the L2 capacity of the simulated TRIPS processor is 4 MB to support a more direct comparison with TFlex on the multiprogramming workloads.

The TFlex architecture also includes two microarchitectural optimizations that could be applied to improve the baseline performance of the TRIPS microarchitecture. First, the bandwidth of the operand network is doubled to reduce inter-ALU contention. Second, TFlex cores support limited dual issue – two integer instructions but only one floating-point instruction issued per cycle – as opposed to the single-issue execution tiles in TRIPS.

3.5.1 Serial Programs

Figures 3.12 and 3.13 show the performance of the TRIPS prototype architecture and that of TFlex compositions on low and high ILP benchmarks, for 2–32 cores. The performance of each benchmark is normalized to the performance of a single TFlex core. On low-ILP benchmarks, TFlex compositions with fewer cores typically meet the performance of more cores as the exposed ILP does not justify wide issue. On some benchmarks (such as sixtrack), four-core performance exceeds that of eight or more since the protocol overheads executing across a larger number of

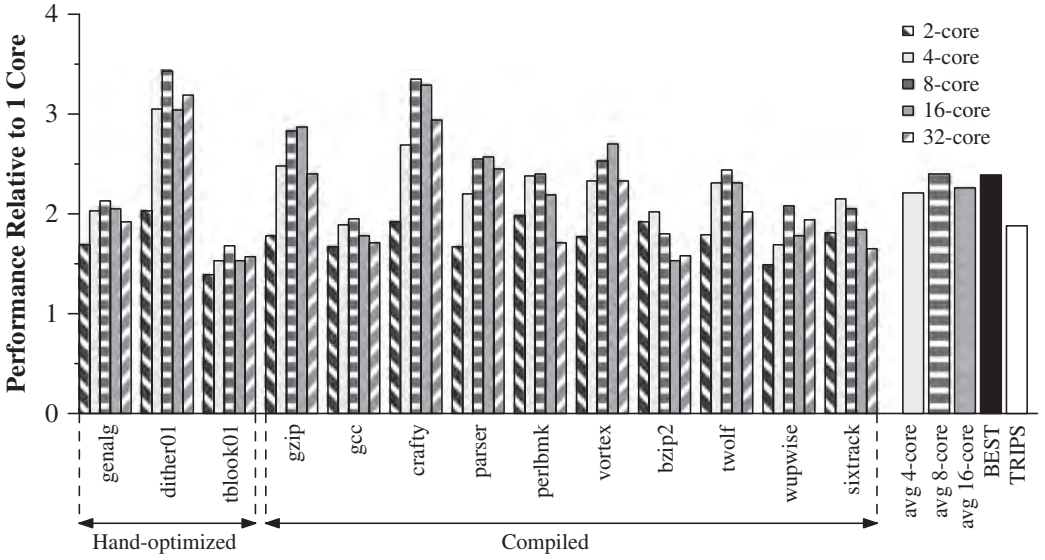


Fig. 3.12 Low ILP benchmarks: normalized performance of applications running on 2–32 CLP cores

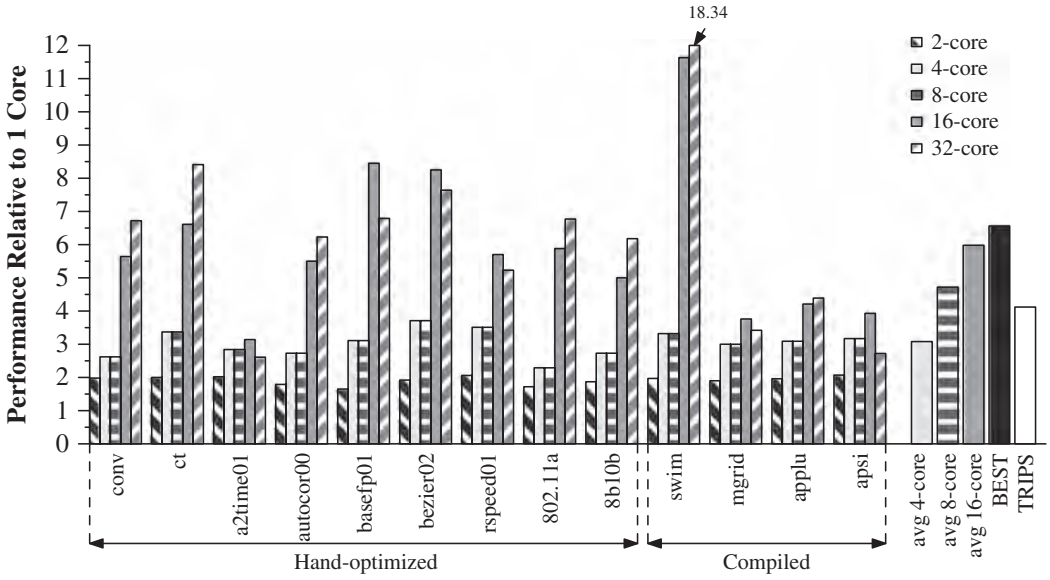


Fig. 3.13 High ILP benchmarks: normalized performance of applications running on 2–32 CLP cores

distributed processors can hinder performance. In general, eight cores shows the highest average performance on the low-ILP benchmarks.

On high-ILP benchmarks, the best single composition ranges from 16 to 32 cores, depending on the application. Averaged across all of the low and high ILP benchmarks, the 16-core TFlex configuration performs best and shows a factor of 3.5 speedup over a single TFlex core. Additional performance can be obtained by adjusting the number of TFlex cores per processor on a per-application basis. This configuration, represented by the bar "BEST," provides an additional 13% performance increase and an overall speedup over a single TFlex core of a factor of four. These results indicate that a combination of scalable processor mechanisms and the capability to dynamically adjust the processor granularity provides a system that can run sequential applications effectively across multiple cores to achieve substantial speedups.

On average, an eight-core TFlex processor, which has area and issue width comparable to the TRIPS processor, outperforms TRIPS by 19%, reflecting the benefits of additional operand network bandwidth as well as twice the L1 cache bandwidth stemming from fine-grained distribution of the cache. The best per-application TFlex configuration outperforms TRIPS by 42%, demonstrating that adapting the processor granularity to the application granularity provides significant improvements in performance.

3.5.2 Multitasking Workloads

To quantify throughput gains provided by the increased flexibility of CLPs, we measured the performance of multi-programming workloads on both the TFlex design

and fixed-granularity CMPs. To isolate the contributions of composability to flexibility, we modeled fixed-granularity CMPs by configuring a 32-core TFlex system to use fixed numbers of logical processors composed of equal numbers of cores. For example, an experiment labeled "CMP-4" is a TFlex configuration that has eight composed processors with four cores each.

The TFlex simulator models inter-processor contention for the shared L2 cache and main memory. The 12 benchmarks used in this experiment were selected from the hand-optimized benchmark suite, which exhibit considerable diversity in ILP. We varied the workload size (i.e., degree of multiprogramming) from two to 16 applications running simultaneously.

We measured throughput performance for TFlex on multiple workload sizes (number of threads) and compared them to different fixed-core configurations using a *weighted speedup* (WS) metric [35]. Weighted speedup results are normalized to a single-processor TFlex configuration that produces the best performance. We used an optimal dynamic programming algorithm to find the core assignments that maximize WS on TFlex, rather than employing an on-line algorithm [10]. The dynamic programming algorithm uses the results from Fig. 3.12 and 3.13 to compute individual benchmark speedup as a function of the number of cores.

For each workload size, Fig. 3.14 reports the average weighted speedup on TFlex and on the set of fixed-granularity CMPs. The best CMP granularity changes depending on the workload size (i.e., the number of threads). CMP-16 is the best for workloads with two threads, CMP-8 is the best for four-thread workloads, CMP-4 is the best for 6–8 threads, and CMP-2 is the best for 12–16 threads. Because of the capability of selecting the best granularity for each workload, the TFlex array consistently outperforms the best fixed-granularity CMP. The set of bars labeled AVG show the average WS across all workloads. While CMP-4 is the best granularity for fixed CMPs, the TFlex design produces an average of 26% higher WS and a maximum of 47% higher WS.

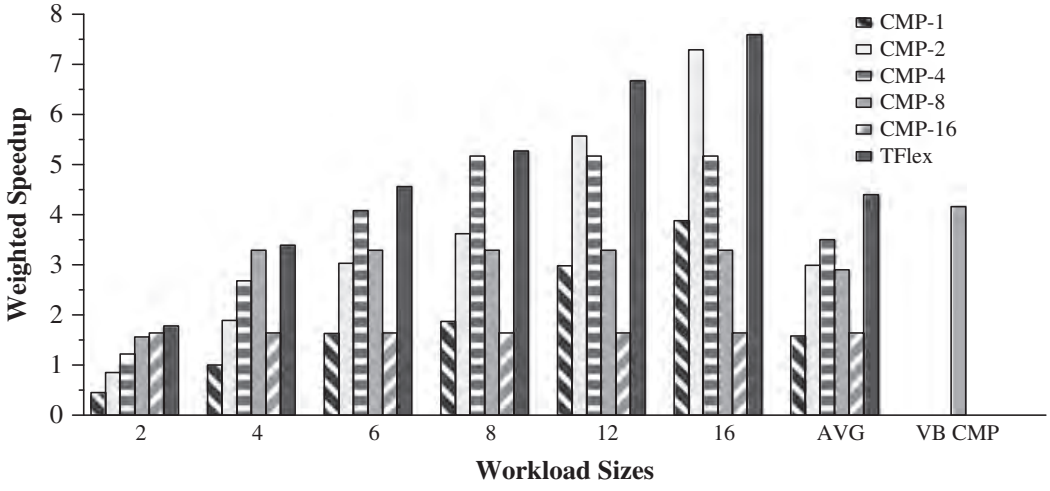


Fig. 3.14 Weighted speedup comparison between TFlex and fixed-granularity CMPs

Figure 3.14 also demonstrates the value of asymmetric processor composition in TFlex by comparing to a hypothetical flexible CMP that can change its granularity dynamically but is symmetric, requiring each processor to be composed from equal number of cores. The results for this hypothetical system, labeled VB CMP (Variable Best CMP), show that the capability of composing processors of different granularities for different simultaneously executing programs results in a 6% improvement in throughput. For a given granularity, even when the number of threads exactly matches the number of processors, some threads may under-utilize their own processor's resources. In such a scenario, TFlex is able to allocate a portion of these under-utilized resources to applications which can make better use of them. Table 3.4 lists the fraction of applications at each workload size that are assigned to a given processor granularity and demonstrates that the optimal processor granularity varies even within a given workload size. For workloads of size 8, TFlex allocates four cores to the majority of the applications, but also allocates two cores to 28% of them and eight cores to 14% of them. A CMP-4 configuration, conversely, would allocate only four cores regardless of the application characteristics. The difference between the VB CMP bar and the TFlex average bar represents the performance benefit of allocating different numbers of cores to applications running simultaneously.

Table 3.4 Percentage of applications mapped to each composed processor in TFlex for each workload size

	Cores (%)					
Workload size	1	2	4	8	16	32
2	0	0	13	35	53	0
4	0	0	31	54	15	0
6	0	18	43	37	2	0
8	0	28	59	14	0	0
12	9	82	9	0	0	0
16	32	52	16	0	0	0

3.5.3 Composability Overheads and Benefits

Since the TRIPS prototype processor and the proposed TFlex implementation share the same ISA and and several microarchitectural structures, a comparison of TFlex and TRIPS implementations can reveal the area overhead for providing composability. Table 3.5 presents the area of different microarchitectural components in an eight-core TFlex processor and the 30 tiles in a single TRIPS processor. The area of each microarchitectural component in a single TFlex core was estimated from the post-synthesis netlist (before final place-and-route) of the 130 nm ASIC implementation of the TRIPS prototype.

The TRIPS and TFlex designs differ in a number of features that are unrelated to composability. For example, the techniques in TFlex for reducing the LSQ capacity per tile could be applied to TRIPS. The difference in cache and branch predictor capacities are also independent of composability. The only area costs for compos-

Table 3.5 Area (mm^2) estimate for eight-core TFlex and area of a single TRIPS core

	eight TFlex cores (16-issue)		Single TRIPS processor (16-issue)	
Structures	Size (8 tiles)	Area	Size (30 tiles)	Area
Fetch (block predictor, I-cache)	66 K-bit predictor, 64 KB I-cache	10.9	80 K-bit predictor, 80 KB I-cache	7.7
Register files	1024 entries	6.5	512 entries	3.0
Execution resources (issue window, ALUs)	1 K-entry issue window 16-INT ALU, 8-FP ALU	23.6	1 K-entry issue window 16-INT ALU, 16-FP ALU	39.4
L1 D-cache subsystem	64 KB D-cache		32 KB D-cache	
(D-cache, LSQ, MSHR)	352-entry LSQ	27.8	1024-entry LSQ	33.4
Routers		7.0		11.0
Total		75.8		94.5

ability are (1) larger register file, (2) some additional control logic to adjust the hashing functions, and (3) some additional bits in control networks to manage the distributed protocols. We conservatively estimate that this area overhead comes to less than 10% in the processors, and less than that when the on-chip memory system is included.

Despite the register file overhead, TFlex has better performance/area because the features used for supporting composability simultaneously improve performance (by aggregating resources) and reduce area (by distribution). In addition, we estimate that the total power consumption of eight TFlex cores is 15% less than a single TRIPS processor, indicating that composability imposes little if any power penalty. Results also show that TFlex achieves twice the power efficiency ($performance^2$/Watt) of TRIPS and 25% better efficiency than the best fixed-core design, because of its ability to adapt the processor granularity to the application's concurrency [16].

3.6 Summary

Since clock rate scaling can no longer sustain computer system performance scaling, future systems will need to mine concurrency at multiple levels of granularity. Depending on the workload mix and number of available threads, the balance between instruction-level and thread-level parallelism will likely change frequently for general-purpose systems, rendering the design-time freezing of processor granularity (bulldozers, chainsaws, or termites) in traditional CMPs an undesirable option. Composable lightweight processors (CLPs) provide the flexibility to allocate resources dynamically to different types of concurrency, ranging from running a single thread on a logical processor composed of many distributed cores

to running many threads on separate physical cores. The small size of an individual TFlex CLP core places it near the regime of termites so that it can exploit substantial threaded parallelism. When the application or environment demands, CLP cores can be aggregated to form bulldozer-sized processors. In addition to pure performance, the system can also use energy and/or area efficiency to choose the best-suited processor configuration. We envision multiple methods of controlling the allocation of cores to threads. At one end of the spectrum, the operating system could make such decisions based on the number of threads to run and their criticality. The OS could even monitor how each thread uses its allocated resources and reallocate them among the threads as necessary. At the other end of the spectrum, the hardware could potentially adjust the number of cores per thread in an automated fashion. This capability would further blur the distinction between conventional uniprocessors and multiprocessors, which we view as a promising direction.

Acknowledgments The authors wish to thank the entire TRIPS hardware team for their contributions to the design and implementation of the TRIPS and TFlex architectures. The team members include: Rajagopalan Desikan, Saurabh Drolia, Mark Gebhart, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Robert McDonald, Ramadass Nagarajan, Nitya Ranganathan, Karthikeyan Sankaralingam, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. The TRIPS project was supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

References

1. M. Annavaram, E. Grochowski, and J. P. Shen. Mitigating Amdahl's law through EPI throttling. In *International Symposium on Computer Architecture*, pages 298–309, June 2005.
2. S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *International Solid-State Circuits Conference*, pages 88–89, Feb 2008.
3. D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7), 2004.
4. J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. In *International Solid-State Circuits Conference*, pages 102–103, Feb 2007.
5. J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti. Design of the Power6 microprocessor. In *International Solid-State Circuits Conference*, pages 102–103, Feb 2007.
6. M. Gebhart, B. Maher, P. Gratz, N. Ranganathan, J. Diamond, B. Robatmili, S. Govindan, S. W. Keckler, D. Burger, and K. McKinley. TRIPS System Evaluation. Technical Report TR-08-31, The University of Texas at Austin, Department of Computer Sciences, Sep 2008.
7. S. Ghiasi and D. Grunwald. Aide de Camp: Asymmetric Dual Core Design for Power and Energy Reduction. Technical Report CU-CS-964-03, The University of Colorado, Department of Computer Science, 2003.
8. E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *International Conference on Computer Design*, pages 236–243, Oct 2004.

9. T. R. Halfhill. Floating point buoys clearspeed. *Microprocessor Report*, 17(11):19–24, Nov 2003.
10. T. Ibaraki and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, 1988.
11. E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez. Core fusion: Accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, June 2007.
12. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), July 2005.
13. R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.
14. B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. J. Dally. A Programmable 512 GOPS stream processor for signal, image, and video processing. In *International Solid-State Circuits Conference*, pages 272–273, Feb 2007.
15. C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct 2002.
16. C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture*, pages 381–394, Dec 2007.
17. R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *International Symposium on Microarchitecture*, pages 81–92, Dec 2003.
18. R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *International Symposium on Computer Architecture*, pages 64–75, June 2004.
19. F. Latorre, J. González, and A. González. Back-end assignment schemes for clustered multithreaded processors. In *International Conference on Supercomputing*, pages 316–325, June 2004.
20. E. Lindholm and S. Oberman. The NVIDIA GeForce 8800 GPU. In *Proceedings of the Symposium on High Performance Chips (HotChips-19)*, August 2007.
21. B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *International Symposium on Microarchitecture*, pages 480–491, Dec 2006.
22. M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *International Symposium on Computer Architecture*, pages 46–56, June 2007.
23. S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *Workshop on Microprogramming and Microarchitecture*, pages 60–63, Nov 1988.
24. U. G. Nawathe, M. Hassan, K. C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-core, 64-thread, power-efficient SPARC server on a chip. *IEEE Journal of Solid-State Circuits*, 43(1), Jan 2008.
25. R. M. Rabbah, I. Bratt, K. Asanović, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report MIT-LCS-TM-646, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, June 2004.
26. P. Racunas and Y. N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *International Conference on Supercomputing*, pages 22–31, June 2003.
27. P. Salverda and C. Zilles. Fundamental performance challenges in horizontal fusion of In-order cores. In *International Symposium on High-Performance Computer Architecture*, pages 252–263, Feb 2008.
28. K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture*, pages 422–433, June 2003.

29. K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *International Symposium on Microarchitecture*, pages 480–491, Dec 2006.
30. S. Sethumadhavan, R. McDonald, R. Desikan, D. Burger, and S. W. Keckler. Design and implementation of the TRIPS primary memory system. In *International Conference on Computer Design*, pages 470–476, Oct 2006.
31. S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: Enabling unordered load – store queues. In *International Symposium on Computer Architecture*, pages 347–357, June 2007.
32. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Sep 2001.
33. A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, Mar 2006.
34. A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *International Symposium on Microarchitecture*, pages 89–100, Dec 2006.
35. A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Nov 2000.
36. B. Stackhouse, B. Cherkauer, M. Gowan, P. Gronowski, and C. Lyles. A 65 nm 2-billion-transistor quad-core Itanium processor. In *International Solid-State Circuits Conference*, pages 92–93, Feb 2008.
37. D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-Order Execution using Simple In-Order Cores. Technical Report CS-2007-11, University of Virginia, Department of Computer Science, Aug 2007.
38. M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *International Symposium on Computer Architecture*, pages 2–13, June 2004.
39. M. Tremblay and S. Chaudhry. A third-generation 65 nm 16-core 32-thread plus 32-scout-thread CMT SPARC Processor. In *International Solid-State Circuits Conference*, pages 82–83, Feb 2008.
40. S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100 W TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, Jan 2008.
41. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

ATTACHMENT 9A

IEEE.org IEEE *Xplore* IEEE SA IEEE Spectrum More Sites

Cart Create Account Personal Sign In

Browse My Settings Help

Access provided by: **University of Nebraska - Lincoln**

Sign Out

Access provided by: **University of Nebraska - Lincoln**

Sign Out

All

ADVANCED SEARCH

Back to Results

Conferences > 1999 IEEE Pacific Rim Confere...

Tradeoff analysis of integer multiplier circuits implemented in FPGAs

Publisher: IEEE

Cite This PDF

<< Results

M.A. Thornton ; J.D. Gaiche ; J.V. Lemieux **All Authors**

1 Paper Citation

1 Patent Citation

36 Full Text Views

Alerts

Manage Content Alerts

Add to Citation Alerts

More Like This

Field Programmable Gate Array Design and Implementation for Fast Fourior Transform Processor

2010 International Conference on E-Business and E-Government

Published: 2010

Superconducting Magnetic Field Programmable Gate Array

IEEE Transactions on Applied Superconductivity

Published: 2018

Show More

Abstract

Authors

References

Citations

Keywords

Metrics

More Like This

Downl PDF

Abstract: Integer multiplication is a necessary operation for performing many tasks relevant to multimedia and telecommunications processes. Here, we discuss the results of an inve... **View more**

Metadata

Abstract:

Integer multiplication is a necessary operation for performing many tasks relevant to multimedia and telecommunications processes. Here, we discuss the results of an investigation into the effectiveness of automated synthesis tools as related to a sample of modern Programmable Logic Devices (PLDs). Although it is generally accepted that superior results in terms of required area and circuit delay can generally be obtained through manual implementation of such circuits, the exclusive use of automated synthesis tools based upon an original specification in terms of a Hardware Description Language (HDL) is presented here. The results of several different approaches to multiplier architectures are presented.

Published in: 1999 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM 1999). Conference Proceedings (Cat. No.99CH36368)

Date of Conference: 22-24 August 1999

Date Added to IEEE *Xplore*: 06 August 2002

Print ISBN:0-7803-5582-2

INSPEC Accession Number: 6505166

DOI: 10.1109/PACRIM.1999.799536

Publisher: IEEE

Conference Location: Victoria, BC, Canada

Authors

References

Citations

Keywords

Metrics

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved.

IEEE Account

» Change Username/Password

» Update Address

Purchase Details

» Payment Options

» Order History

» View Purchased Documents

Profile Information

» Communications Preferences

» Profession and Education

» Technical Interests

Need Help?

» **US & Canada:** +1 800 678 4333

» **Worldwide:** +1 732 981 0060

» Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.

ATTACHMENT 9B

Tradeoff Analysis of Integer Multiplier Circuits Implemented in FPGAs

M. A. Thornton
Mississippi State University
J. D. Gaiche
Acxiom Corporation
J. V. Lemieux
University of Arkansas

Abstract

Integer multiplication is a necessary operation for performing many tasks relevant to multimedia and telecommunications processes. Here, we discuss the results of an investigation into the effectiveness of automated synthesis tools as related to a sample of modern Programmable Logic Devices (PLDs). Although it is generally accepted that superior results in terms of required area and circuit delay can generally be obtained through manual implementation of such circuits, the exclusive use of automated synthesis tools based upon an original specification in terms of a Hardware Description Language (HDL) is presented here. The results of several different approaches to multiplier architectures are presented.

1 Introduction

Field Programmable Gate Array (FPGA) and Complex Programmable Logic Devices (CPLDs) devices have matured to the point that they present a viable alternative for the implementation of general arithmetic and digital signal processing (DSP) circuitry [4] [2]. DSP processing tasks have traditionally been realized using microprocessors, Application Specific ICs (ASIC) or special purpose DSP processors. However, the logic density and speeds of modern FPGA/CPLDs have now reached levels where they can be competitive with special purpose DSP devices. In terms of custom solutions, the FPGA/CPLD approach offers immediate availability compared to the relatively lengthy development and turn around time for the design and fabrication of an ASIC. As a result of these benefits, many developers are investigating the feasibility of porting DSP applications into FPGA/CPLD based circuitry.

Digital signal processing algorithms are generally multiplication intensive tasks and many different multiplication circuit architectures exist [1] [5] [9]. It is therefore crucial that the particular multiplier circuit employed must make efficient use of the internal FPGA/CPLD architecture for a DSP task to yield acceptable performance. Furthermore, the multiplication architectures which are utilized in ASIC's are not necessarily optimal for a FPGA/CPLD implementation because of their fixed internal structure. More information is needed to help determine which algorithms are best suited for FPGA/CPLD implementations and to further clarify the division of hardware and software functions in the circuitry being implemented. A survey of various multiplier circuits is presented in [1]. The motivation for this study is that dynamically reconfigurable computing solutions for DSP tasks will require automatically generated multipliers of various sizes and resource requirements. In some cases, throughput is enhanced with the use of many small but slow multipliers versus a large fast one [7].

The suitability of several multiplication algorithms for use in FPGAs is explored by describing the circuit structure using the Hardware Descriptive Language (HDL), *Verilog*, followed by simulation and synthesis runs. Each multiplier in this investigation is capable of multiplying two 16-bit integer numbers. Multiplier circuits that are investigated include a Scaling Accumulator Multiplier, a Addition Array Multiplier, a Computed Partial Product Multiplier, a Wallace Tree Multiplier, and a Partial Product Look-Up Table Multiplier. These particular architectures were chosen since they represent differing resource requirements and all can be scaled to various wordlengths.

The paper is organized as follows. First, a description of example multiplication circuits is presented. Next, the experimental results are presented and analyzed. Finally, conclusions and future efforts are addressed.

2 Multiplier Circuit Architecture

Descriptions of the integer multiplication architectures are provided in this section. As a point of reference, a purely behavioral description of a 16-bit multiplier is synthesized and results are given for the purpose of comparison to the other approaches considered here. An attempt was made to choose candidate architectures that represent diverse approaches. We have included structures that have traditionally yielded the best results in terms of ASIC implementation (addition tree and array based approaches), techniques that are known for speed but require large amounts of circuitry (combinational logic implementations such as the look-up table based approaches), and, word-serial implementations that require

0-7803-5582-2/99/$10.00 ©1999 IEEE

Attachment - Page A279

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 18:53:59 UTC from IEEE Xplore. Restrictions apply.

smaller amounts of circuitry but multiple clock cycles.

2.1 Behavioral

The behavioral multiplier is implemented using a *Verilog* statement of the form: `prod = mplier * mcand;`. When the synthesis tools encounter this statement, they default to built in multiplier models for the target device. Thus, these results are useful for comparing against the RTL level multiplier descriptions of the other architectures. The *Verilog* code for this model is shown in Figure 1.

Figure 1: *Verilog* Description of Behavioral Multiplier

```
module behmult(mcand,mplier,prod,clk,
         reset,din_valid,dout_valid);
// I/O Declarations
input   [15:0]  mcand, mplier;
input           clk, rest, din_valid;
output  [31:0]  prod;
output          dout_valid;
// Data Type Declarations;
reg     [31:0]  prod;
reg             dout_valid;

always @(posedge clk)
   if (reset == 1'b1)
      begin
        prod = 0;
        dout_valid = 1'b0;
      end
   else
      if (din_valid == 1)
         begin
           prod = mplier * mcand;
           dout_valid = 1'b1;
         end
endmodule
```

2.2 Scaled Accumulator

This is a "shift and add" approach to generating the product. The advantages are a small amount of circuitry, but a number of clock cycles equal to the wordsize of the multiplier operand.

2.3 Computed Partial Product

This circuit stores the multiplicand, twice the multiplicand and three times the multiplicand in three internal registers. These are generated very efficiently through the use of a single-bit left-shift and an adder. The multiplier word is then used two bits at a time to select the appropriate multiple of the multiplicand. As each successive bit pair from the multiplier word is brought in, the final product is accumulated in a shift register. This approach is a generalization of the scaled accumulator where a digit size of two bits versus a single bit is used. Figure 2 contains a block diagram of the computed partial product multiplier.

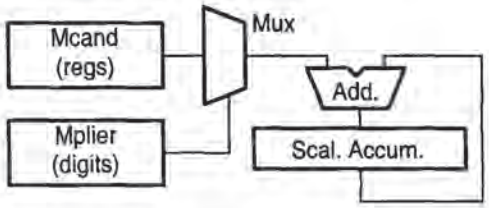


Figure 2: Block Diagram of Parallel Multiplier (array/tree type)

2.4 Addition Array

This is a array based multiplier circuit [6] using full and half single-bit adders. This approach is known for producing fast multipliers with a regular layout pattern and is popular in ASIC designs. It operates by generating all the partial products in parallel and adding them together using a rectangular array of single bit addition circuits. This is a fully parallel approach in that a product is returned at each clock cycle. Figure 3 contains a diagram of this type of multiplier.

2.5 Addition Tree

This approach accumulates the partial products through the use of a tree of 3:2 and 2:2 compressors (or single bit adders) [8] [3]. While these approaches often yield the fastest multipliers in ASIC designs, they are known for large area requirements and usually do not map well to the regular layouts inherent in FPGAs and CPLDs. In terms of operational functionality, this approach is very similar to the array multiplier described above with the difference being that a tree of adders is used to accumulate the partial products. Figure 3 also represents the addition tree multiplier.

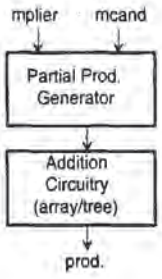


Figure 3: Block Diagram of Parallel Multiplier (array/tree type)

2.6 Memory Based Lookup Table

Here, external memory is used to store the product with the address serving as the concatenation of the multiplier and

Attachment - Page A280

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 18:53:59 UTC from IEEE Xplore. Restrictions apply.

multiplicand. To reduce the size of the required memory, four partial products are stored versus the entire 32-bit product and programmable device logic is used to scale (shift) and accumulate the final products. Due to the small amount of logic required, this approach is generally limited in speed due to the memory access times. This approach also places additional demands on FPGA/CPLD I/O requirements since pins are needed to drive the memory device address and to read in the resulting data for scaling and accumulation.

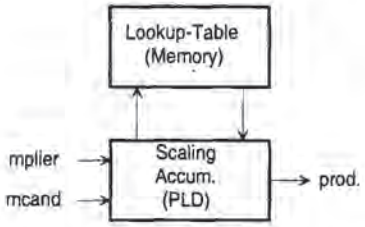


Figure 4: Block Diagram of Memory Based LUT Multiplier

3 Experimental Results

Table 1 contains results that were generated using the place and route tools from Altera MaxPlusII (ver. 9.1.2) and Xilinx Foundation Express (ver. 1.5). The Verilog descriptions were synthesized using the Synopsys FPGA Express tools bundled with MaxPlusII and Foundation, and the Synplicity Synplify tool (ver. 5.1.4). The Verilog models were tested using the PC-based Model Tech and the Unix-based Cadence Verilog-XL simulators. Estimated operating frequencies and logic resource utilization metrics are compiled based on the place and route tool log files. The place route tools were set to place approximately equal emphasis on area and delay. Since the various FPGA/CPLD manufacturers market devices with different internal architectures, each multiplier circuit is synthesized onto a variety of target devices.

In the data presented below, each multiplier type is designated using a three-letter acronym. `beh` is the behavioral multiplier, `aam` is the addition array multiplier, `atm` is the addition tree multiplier, `cpp` is the computed partial product multiplier, `sca` is the scaling accumulator approach, and, `lpp` is the look-up table based circuit. The table entries `io` and `lb` refer to inadequate device resources in terms of required I/O pins and logic blocks respectively. The columns labeled IOB, CLB, IO and LB contain the percentage of the device resources needed to implement the circuit. For those circuits that were implemented as a block of combinational logic with a register for the product, clock speed was computed as the inverse of the critical path delay plus the output register setup time. The frequencies for the `lpp` type circuits are given assuming a 10 ns SRAM, a 40 ns DRAM and a 120 ns EPROM are used for storing the partial products.

4 Conclusion

Various scalable multipliers have been synthesized and analyzed in terms of speed and area for implementation in programmable devices. It should be noted that the purpose here is not to compare the effectiveness of logic synthesis tools or programmable device architectures, but rather to get a feel for the trend in resource requirements given the various approaches. A fair comparison of the synthesis tools would require careful adherence to coding styles required by each tool and the to compare the architectures, details about internal features (such as dedicated carry-ripple logic) would need to be exploited in the HDL descriptions.

These results are useful for researchers in the area of reconfigurable computing where tradeoffs between speed and logic usage must be evaluated dynamically. The results can also be of use to designers of dedicated programmable logic circuits in the preliminary phases of resource estimation and the evaluation of candidate designs with respect to functional requirements.

References

[1] R. Andraka. *Multiplication in FPGAs. Web Page*, `http://users.ids.net/ randraka/multipli.htm`, 1999.

[2] Altera Corporation. Improving fixed-point dsp processor system performance with plds as a dsp coprocessor. *Altera Application Brief*, `http://www.altera.com/document/papers/cpdsp.pdf`, 1999.

[3] L. Dadda. Some Schemes for Parallel Multipliers. *Alta Freq.*, 34:349–356, 1965.

[4] S. K. Knapp. Using programmable logic to accelerate dsp functions. *Xilinx Application Brief*, `http://www.xilinx.com/appnotes/dspintro.pdf`, 1995.

[5] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall Publishers, Englewood Cliffs, New Jersey, 1993.

[6] S. D. Pezaris. A 40-ns 17-bit by 17-bit Array Multiplier. *IEEE Transactions on Computers*, C-20:442–447, 1971.

[7] J. Pihl. Tradeoffs Between Parallel and Serial Architectures in High Performance Digital Signal Processing. *Proceedings of the International Symposium on Integrated Circuits*, 1997.

[8] C. S. Wallace. A Suggestion for a Fast Multiplier. *IRE Transactions on Electronic Computers*, EC-13:14–17, 1964.

[9] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehart and Winston, New York, 1982.

Attachment - Page A281

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 18:53:59 UTC from IEEE Xplore. Restrictions apply.

Table 1: Results Using Various FPGA/CPLDs

Xilinx		Synopsys Express			Synplicity Synplify		
Arch.	%IOB	% CLB	CLK (MHz)	Rate (MHz)	% CLB	CLK (MHz)	Rate (MHz)
V300PQ240-4 (Virtex)							
beh	40	5	21.4	21.4	4	42.4	42.4
aam	40	12	10.0	10.0	9	9.1	9.1
atm	40	11	18.8	18.8	11	15.8	15.8
cpp	31	5	22.7	2.8	4	42.0	5.3
sca	31	1	57.7	3.6	1	69.7	4.4
lpp	50	2	36.6	14.9/3.7/2.0	1	65.1	18.1/3.9/2.0
S30VQ100-3 (Spartan)							
beh	88	43	10.6	10.6	39	25.6	25.6
aam	88	68	4.7	4.7	86	4.3	4.3
atm	88	60	7.6	7.6	65	12.2	12.2
cpp	70	25	17.2	2.2	22	25.3	3.2
sca	68	12	27.4	1.7	6	35.4	2.2
lpp	109	9	io	io	5	io	io
S05XLPC84-4 (Spartan XL)							
beh	111	250	io,lb	io,lb	214	io/lb	io/lb
aam	111	381	io,lb	io,lb	295	io/lb	io/lb
atm	111	332	io,lb	io,lb	348	io/lb	io/lb
cpp	88	144	lb	lb	116	lb	lb
sca	86	72	44.9	2.8	39	52.4	3.3
lpp	137	54	io	io	32	io	io
4028EXHQ208 ex-2 (4000EX)							
beh	42	24	10.0	10.0	22	23.1	23.1
aam	42	38	4.0	4.0	28	8.2	8.2
atm	42	33	7.9	7.9	35	12.3	12.3
cpp	33	14	22.4	2.8	12	23.2	2.9
sca	33	7	32.0	2.0	3	35.8	2.2
lpp	51	5	22.4	11.8/3.5/1.9	3	38.3	15.1/3.8/2.0
40150XVHQ240 xv-09 (4000XV)							
beh	38	4	8.9	8.9	4	33.0	33.0
aam	38	7	3.4	3.4	5	10.2	10.2
atm	38	6	6.0	6.0	7	15.5	15.5
cpp	30	2	19.2	2.4	2	38.6	4.8
sca	29	1	17.0	1.1	1	61.5	3.8
lpp	46	1	11.4	7.8/3.1/1.8	1	59.2	17.6/3.9/2.0
3020APC68-6 (3000A)							
beh	115	414	io,lb	io,lb	607	io,lb	io,lb
aam	115	525	io,lb	io,lb	576	io,lb	io,lb
atm	115	648	io,lb	io,lb	637	io,lb	io,lb
cpp	91	337	lb	lb	343	lb	lb
sca	89	185	lb	lb	92	135.1	8.4
lpp	143	98	io	io	84	io	io
Altera		Synopsys Express			Synplicity Synplify		
Arch.	%IO	% LC	CLK (MHz)	Rate (MHz)	% LC	CLK (MHz)	Rate (MHz)
EPF10K20TC144-3 (Flex 10k)							
beh	64	71	9.2	9.2	48	26.0	26.0
aam	64	67	6.1	6.1	54	6.1	6.1
atm	64	67	10.4	10.4	lb	lb	lb
cpp	50	52	12.2	1.5	28	33.9	4.2
sca	48	13	20.0	1.3	15	49.5	3.1
lpp	81	10	17.2	10.2/3.4/1.9	8	55.2	17.2/3.9/2.0
EPF81188AQC208-2 (Flex 8k)							
beh	45	81	8.7	8.7	55	23.8	23.8
aam	45	77	5.3	5.3	61	4.7	4.7
atm	45	76	9.6	9.6	80	8.3	8.3
cpp	36	59	10.7	1.3	34	21.6	2.7
sca	35	15	17.5	1.1	14	34.7	2.2
lpp	56	12	16.5	9.9/3.3/1.8	9	41.5	15.6/3.8/2.0
EPF6010ATC100-1 (Flex 6k)							
beh	95	93	12.3	12.3	62	23.0	23.0
aam	95	89	7.1	7.1	lb	lb	lb
atm	95	87	13.4	13.4	lb	lb	lb
cpp	74	68	14.1	1.8	38	33.6	4.2
sca	73	17	25.4	1.6	16	41.3	2.6
lpp	io	io	io	io	io	io	io

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 18:53:59 UTC from IEEE Xplore. Restrictions apply.

ATTACHMENT 9C

Tradeoff Ana FPGAs

Document properties

File name:	getPDF.jsp
File size:	363 KB
Title:	Tradeoff analysis of integer multiplier circuits impl...
Author:	M.A. Thornton;J.D. Gaiche;J.V. Lemieux
Subject:	1999 IEEE Pacific Rim Conference on Communicat...
Keywords:	-
Created:	3/22/04, 3:06:21 PM
Modified:	11/18/22, 10:40:02 AM
Application:	Acrobat Capture 3.0
PDF producer:	Adobe PDF Library 4.0; modified using iText® 7.1....
PDF version:	1.3
Page count:	4
Page size:	Varies
Fast web view:	No

Close

Integer multiplication
forming many tasks relev
nications processes. Here
tigation into the effectiv
as related to a sample o
vices (PLDs). Although i
results in terms of requir
erally be obtained throug
circuits, the exclusive use
upon an original specific
scription Language (HD
of several different approa
presented.

1 Introduction

Field Programmable G
Programmable Logic Devices (CPLDs) devices have matured to the point that they present a viable alternative for the implementation of general arithmetic and digital signal processing (DSP) circuitry [4] [2]. DSP processing tasks have traditionally been realized using microprocessors, Application Specific ICs (ASIC) or special purpose DSP processors. However, the logic density and speeds of modern FPGA/CPLDs have now reached levels where they can be competitive with special purpose DSP devices. In terms of custom solutions, the FPGA/CPLD approach offers immediate availability compared to the relatively lengthy development and turn around time for the design and fabrication

More information
ms are best suited
further clarify the
s in the circuitry
multiplier circuits
study is that dy-
ons for DSP tasks
tipliers of various
cases, throughput
t slow multipliers

algorithms for use
cuit structure us-
IDL), *Verilog*, fol-
Each multiplier in
g two 16-bit inte-
vestigated include
tion Array Multi-
er, a Wallace Tree
Table Multiplier.
These particular architectures were chosen since they represent differing resource requirements and all can be scaled to various wordlengths.

The paper is organized as follows. First, a description of example multiplication circuits is presented. Next, the experimental results are presented and analyzed. Finally, conclusions and future efforts are addressed.

2 Multiplier Circuit Architecture

Descriptions of the integer multiplication architectures are

ATTACHMENT 10A

IEEE.org IEEE *Xplore* IEEE SA IEEE Spectrum More Sites

Cart Create Account Personal Sign In

Browse My Settings Help

Access provided by: **University of Nebraska - Lincoln**

Sign Out

Access provided by: **University of Nebraska - Lincoln**

Sign Out

All

ADVANCED SEARCH

Back to Results | Next >

A fully-pipelined single-precision floating point unit in the synergistic processor element of a CELL processor

Publisher: IEEE

Cite This PDF

<< Results | Next >

Hwa-Joon Oh ; S.M. Mueller ; C. Jacobi ; K.D. Tran ; S.R. Cottier ; B.W. Michael ; H. … **All Authors**

1 Paper Citation

1 Patent Citation

236 Full Text Views

Alerts

Manage Content Alerts

Add to Citation Alerts

Abstract

Document Sections

- Introduction
- Challenges in 11FO4 design
- The Multiplier and the Aligner
- The Fraction Adder and the Leading Zero Anticipator
- The Normalizer and the Result Multiplexer

Show Full Outline

Authors

Downl PDF

Abstract: The floating point unit in the synergistic processor element of a CELL processor is a fully-pipelined 4-way SIMD unit designed to accelerate media and data streaming. It ... **View more**

Metadata

Abstract:

The floating point unit in the synergistic processor element of a CELL processor is a fully-pipelined 4-way SIMD unit designed to accelerate media and data streaming. It supports 32-bit single-precision floating point and 16-bit integer operands with two different latencies, optimizing the performance of critical single-precision multiply-add operations. It employs fine-grained clock gating for power saving. Architecture, logic, circuits and integration are co-designed to meet the performance, power, and area goals.

Published in: Digest of Technical Papers. 2005 Symposium on VLSI Circuits, 2005.

More Like This

Highly Linear Silicon-on-Insulator CMOS Digitally Programmable Capacitor Array for Tunable Antenna Matching Circuits

IEEE Microwave and Wireless Components Letters

Published: 2013

26.5-30-GHz resistive mixer in 90-nm VLSI SOI CMOS technology with high linearity for WLAN

IEEE Transactions on Microwave Theory and Techniques

Published: 2005

Show More

Authors
Figures
References
Citations
Keywords
Metrics
More Like This

Date of Conference: 16-18 June 2005 **INSPEC Accession Number:** 8573458

Date Added to IEEE *Xplore*: 25 July 2005 **DOI:** 10.1109/VLSIC.2005.1469325

Publisher: IEEE

Print ISBN:4-900784-01-X

Conference Location: Kyoto, Japan

ISSN Information:

Contents

Introduction

The Synergistic Processing Element (SPE) [1] of a CELL processor is the first implementation of a new processor architecture designed to optimize the multimedia applications, such as 3D graphics, media streaming, and signal processing [2]. Real time multimedia applications demand single precision performance significantly exceeding that of conventional processors. A single-precision floating-point unit (FPU) of the SPE is a 4-way single-instruction multiple-data (SIMD) design. Most instructions in the FPU process 128-bit operands, divided into four 32-bit words. Each of the 4 slices supports 32-bit single-precision and 16-bit integer multiply-add instructions and converts between floating-point and integer. The FPU has two different latencies, 6-cycle for single-precision and 7-cycle for integer and converts including result forwarding (Fig. 1).

Challenges in 11FO4 design

The amount of logic per pipeline stage is 11 fanout-of-4 (FO4) inverter delays. Since result forwarding takes 6FO4, the delay budget including latches is 60FO4 for the whole FPU logic. A state-of-art FPU has a latency of around 100FO4 [3]. Saving 40FO4 and being power efficient for the high volume market required optimizations at all design levels: architecture, logic, circuits, layout, and placement.



Fig. 1. Overview of the FPU: Design challengers are presented at each block to make a 6-cycle single-precision operation and 7-cycle integer operation with 11FO4 cycle time.

The FPU is optimized for single-precision multiply-add type instructions [4]. With truncation rounding and extended data set, denormal numbers are flushed to zero and NaNs are handled as normal numbers. An extra format stage is added to pre-process the integer and convert instructions while a late zero correction in aligner and multiplier allows single-precision operations to bypass the format stage, saving one cycle.

Latch insertion delay of 2FO4 to 3FO4 occupies a significant portion of the 11FO4 cycle. In order to minimize this delay, NAND logic is merged with latch function. Also, there is a special type of latch which has multiplexers (MUX) at the input terminals [5]. Both types of latches are heavily used at the FPU to improve timing. CMOS static gates are used to implement most of the logic.

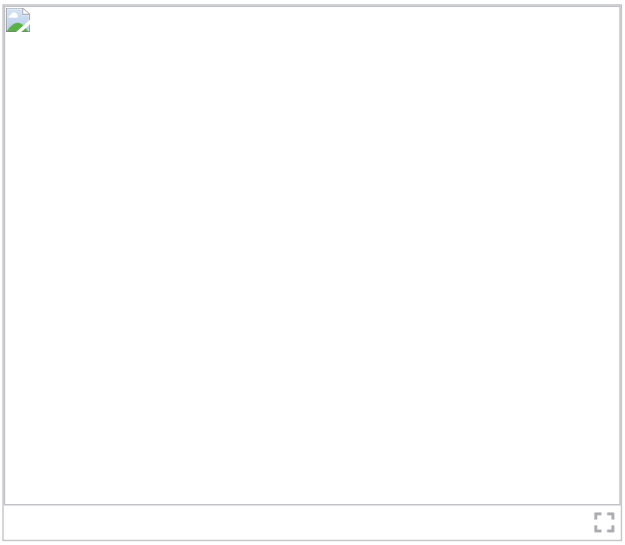


Fig. 2. Radix-4 Booth recording and partial product reduction trees at the multiplier. The 3:2 counter (Full Adder) circuits have static AOI and transmission-gate circuits.

The Multiplier and the Aligner

The 2-cycle multiplier (Fig. 2) uses a radix-4 Booth encoding; it supports unsigned and 2's complement numbers. Partial product reduction tree uses static full adders (FA) consisting of transmission gate XOR and AOI. The static implementation has advantages over dynamic circuits of reduced wiring need and improved timing optimization. Dynamic adders need true and complement signals; static circuits cut the wire density in half

The timing is optimized exploiting the different path delays in a static FA [6]; in a dynamic FA every signal has the same block delay. Since the FA has the slowest path from x to sum and the fastest path from z to carry, the carry output of one stage is connected to the x input of the next stage. By doing so, we are able to put 4 FA reduction tree stages in 8.5FO4.

The aligner block shifts the addend based on the exponent difference (ED) in order to align it with the product. The shifter is broken into several 4:1 MUX stages. The exponent difference is usually generated by an adder then decoded to select signals for the MUX stages, such that both, adder and decoder, add critical path delay to the aligner (Fig. 3). Our 2-cycle aligner uses a special sum-addressed scheme; adding and decoding of the exponent difference are done in parallel. The last MUX stage is integrated into the special MUX latch, reducing the effective latch insertion delay.

Main 4:1 MUX stages at the aligner consist of 4-input transmission-gate multiplexers (TG-MUX) (Fig. 3). Its output inverter driver has better driving capability over the long horizontal wires than that of the AOI-type MUX. Its short cycle time (e.g., 11FO4) forces to insert non-scan staging latches in the middle of the select signal generations. To solve the reliability problem, special circuits have been implemented at local clock buffer; start latching function as soon as possible when the VDD voltage level becomes stable.

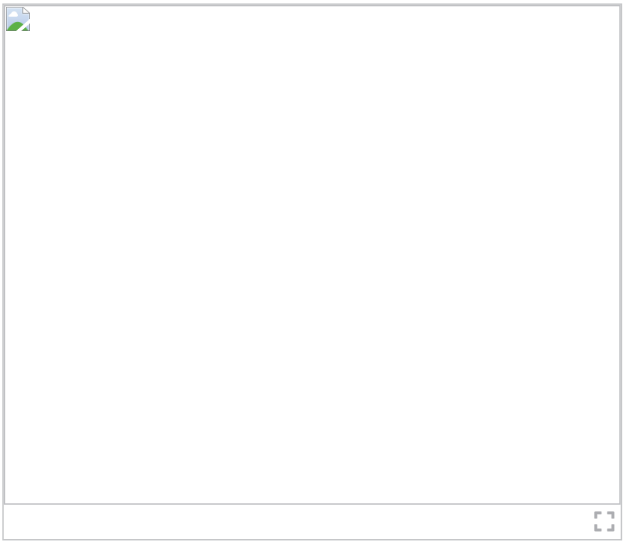


Fig. 3. Sum-address decoder speeds up the shifting stages at the aligner. Main shifting stage consists of 4-input TG-MUXes.





Fig. 4. Critical path of the adder: generating select signals of the multiplexer.

The Fraction Adder and the Leading Zero Anticipator

The fraction adder (Fig. 4) computes the sum or absolute difference of its inputs using the end-around-carry concept. It is partitioned as incrementer, adder and sticky part. The main adder generates sum (sum0) and sum+1 (sum1). Instruction type, shift amount in the aligner, the sticky bit, and the carryout (cout) of the adder determine whether sum0 or sum1 is chosen, and whether complement of the result is needed. An incrementer generates sumi1 incremented by one from the 25-bit MSB of addend (sumi0). The subtraction-type instructions re-complement both sumi0 and sumi1. The selection of sum0, sum1, complement of sum0, sumi0, and sumi1 is implemented with the special6-port MUX latch. The path delay of this multiplexing is hidden by the latch insertion delay. The computation of the select signals based on the cout is the critical path of the adder. An extra carry-only adder is used to speed up signal cout.

The leading zero anticipator (LZA) estimates the number of leading zeros in the adder result. LZA correction logic checks whether the estimate is accurate or too large by one bit.

The Normalizer and the Result Multiplexer

The normalizer block normalizes the fraction, shifting out the leading zeros. Moving one shift stage to the adder and integrating the last shift stage in a special MUX latch allows for a 1-cycle normalizer.

The FPU only supports round toward zero and non-trap exception conditions. That simplifies the rounding step considerably; the fraction is truncated and no exponent wrapping is needed. The fraction rounder is implemented as a result-MUX. Exponent rounder (ERND) adjusts the exponent, detects exceptions and controls the result-MUX. To match the fast rounder, the ERND delay was improved by 15FO4 using highly optimized checkers and dynamic circuits. ERND uses multiple pre-computed exponent copies (e, e+1, e-1, and a special MUX latch selects the appropriate copy of the ERND results.



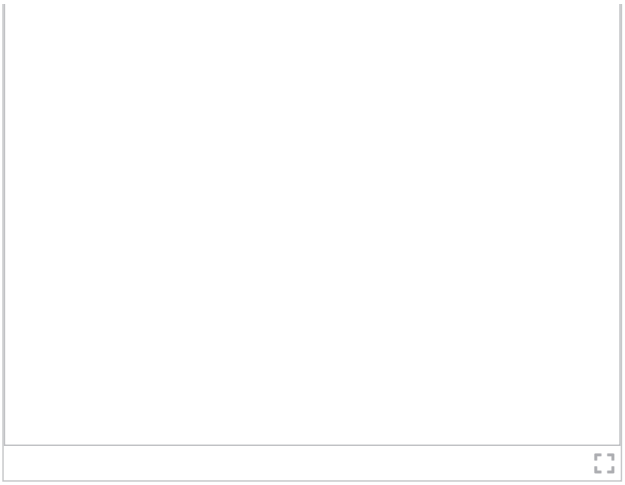


Fig. 5. Static-to-dynamic multiplexer generates the dynamic signals, and then the dynamic result MUX converts the signals to static forwarding signals.

The result-MUX requires four select signals: a) results from integer or double precision, b) a zero result, c) single-precision results which need LZA correction, and d) single-precision results which do not need LZA correction. Since the last cycle has only 5FO4 budget due to the result forwarding, a dynamic 4:1 MUX (Fig. 5) was designed. The fraction data and the select signals from ERND use a local clock signal (50% duty-cycle). The output of the result-MUX is converted to static signals through cross-coupled NAND2 circuits.

Physical Designs

Physical design considerations such as wiring, pin locations, and floor-plan are carefully considered (Fig. 7). The physical width of each FPU slice is 46 bits and aligned to the operand sources such as the register file. This is very narrow for a FPU since intermediate results require 11-bit exponent and 96-bit fraction. The data flow is split into an exponent and a fraction part. Most fraction macros are folded and their layouts use two different bit images, 16-track/bit and 12-track/bit. 16-track/bit regions assign 9 bits for exponent and 37 bits for fraction, whereas 12-track/bit regions assign 12 and 49 bits, respectively.

From early in the project, wire and re-powering delays were limited to 30% of a cycle. Since wires determine the size of the unit and have timing impacts, strict wiring assignments are applied. Macro level wiring was restricted to Metal-3 and below. Some macros have mandatory empty wiring tracks at Metal-2 for global wiring.

Clock Gating to Save Power Consumption

Clock-gating is extensively used to reduce the active power consumption. Only pipeline stages with valid instructions are

activated by controlling the local clock buffer. Also, circuit blocks are clocked down based on instruction type and on operand values, e.g., the multiplier is bypassed for add-type instructions, and the aligner is idle for multiply-type instructions.

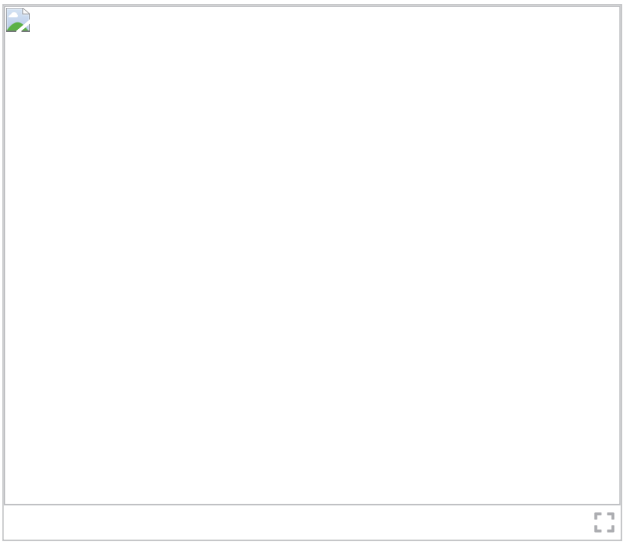


Fig. 6. Voltage vs. Frequency Schmoo: A Synergistic Processing Element is running single-precision floating point computations for lighting and transform workloads.

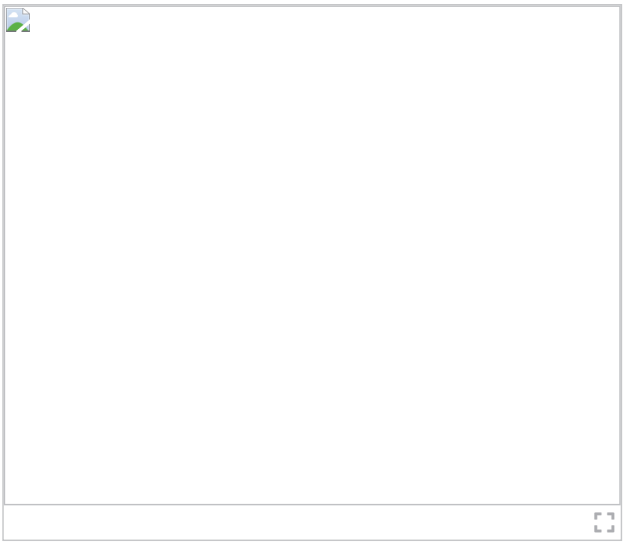


Fig. 7. Macro placement and Die photo: Macro placements are shown at left figure. Two different bit images are used, 16track/bit and 12track/ bit. Die photo of FPU in the Synergistic Processing Element are shown at right photo.

Conclusions

The design has roughly 450K transistors in 1.3mm2, fabricated in 90nm SOI technology with 8 levels of copper inter-connects. Correct operation has been observed up to 5.6GHz, 1.4V of supply voltage, and 56°C, delivering a peak performance of

44.8GFlops. An interpolated hardware power of 1.4W at 4GHz (e.g., 32GFlops), gives 43.75mW/GFlop.

The VDD voltage vs. operating frequency schmoo plot is shown in Fig. 6., when a SPE is running single-precision floating point computations for lightning and transform workloads.

There are 3 key enablers for this low latency, power efficient, and high frequency FPU: architecture and implementation are optimized for target applications, trading uncritical function for overall performance; logic, circuits and integration are co-designed; the pipeline stages are carefully bal-anced, achieving the maximum path delay difference of 3% between stages.

ACKNOWLEDGMENT

We thank Osamu Takahashi, Gordon C. Fossum and Barry L. Minor for design efforts, Gilles Gervais and SPU verification team for supports, and Linda V. Grinsven and Robert Putney for the management team.

Authors
Figures
References
Citations
Keywords
Metrics

IEEE Personal Account
CHANGE USERNAME/PASSWORD

Purchase Details
PAYMENT OPTIONS
VIEW PURCHASED DOCUMENTS

Profile Information
COMMUNICATIONS PREFERENCES
PROFESSION AND EDUCATION
TECHNICAL INTERESTS

Need Help?
US & CANADA: +1 800 678 4333
WORLDWIDE: +1 732 981 0060
CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved.

IEEE Account
» Change Username/Password

Purchase Details
» Payment Options

Profile Information
» Communications Preferences

Need Help?
» **US & Canada:** +1 800 678 4333

» Update Address
» Order History
» Profession and Education
» **Worldwide:** +1 732 981 0060
» View Purchased Documents
» Technical Interests
» Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.
© Copyright 2022 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.

ATTACHMENT 10B

A Fully-Pipelined Single-Precision Floating Point Unit in the Synergistic Processor Element of a CELL Processor

Hwa-Joon Oh[1], Silvia M. Mueller[2], Christian Jacobi[2], Kevin D. Tran[1], Scott R. Cottier[1], Brad W. Michael[1], Hiroo Nishikawa[3], Yonetaro Totsuka[4], Tatsuya Namatame[5], Naoka Yano[5], Takashi Machida[5], Sang H. Dhong[1]

[1]IBM System and Technology Group, Austin, TX
[2]IBM Entwicklung GmbH, Boeblingen, Germany
[3]IBM Engineering and Technology Services, Yasu, Japan
[4]Sony Computer Entertainment of America, Austin, TX
[5]Toshiba America Electronic Components, Austin, TX

Abstract

The floating point unit in the synergistic processor element of a CELL processor is a fully-pipelined 4-way SIMD unit designed to accelerate media and data streaming. It supports 32-bit single-precision floating point and 16-bit integer operands with two different latencies, optimizing the performance of critical single-precision multiply-add operations. It employs fine-grained clock gating for power saving. Architecture, logic, circuits and integration are co-designed to meet the performance, power, and area goals.

(Keywords: VLSI, CMOS, SOI, Microprocessor, Static circuits, and Floating point unit)

Introduction

The Synergistic Processing Element (SPE) [1] of a CELL processor is the first implementation of a new processor architecture designed to optimize the multimedia applications, such as 3D graphics, media streaming, and signal processing [2]. Real time multimedia applications demand single precision performance significantly exceeding that of conventional processors. A single-precision floating-point unit (FPU) of the SPE is a 4-way single-instruction multiple-data (SIMD) design. Most instructions in the FPU process 128-bit operands, divided into four 32-bit words. Each of the 4 slices supports 32-bit single-precision and 16-bit integer multiply-add instructions and converts between floating-point and integer. The FPU has two different latencies, 6-cycle for single-precision and 7-cycle for integer and converts including result forwarding (Fig. 1).

Challenges in 11FO4 design

The amount of logic per pipeline stage is 11 fanout-of-4 (FO4) inverter delays. Since result forwarding takes 6FO4, the delay budget including latches is 60FO4 for the whole FPU logic. A state-of-art FPU has a latency of around 100FO4 [3]. Saving 40FO4 and being power efficient for the high volume market required optimizations at all design levels: architecture, logic, circuits, layout, and placement.

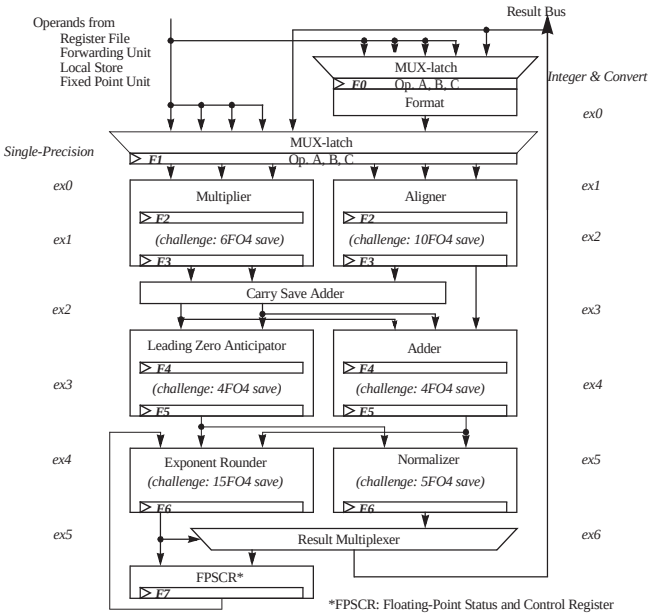


Fig. 1. Overview of the FPU: Design challengers are presented at each block to make a 6-cycle single-precision operation and 7-cycle integer operation with 11FO4 cycle time.

The FPU is optimized for single-precision multiply-add type instructions [4]. With truncation rounding and extended data set, denormal numbers are flushed to zero and NaNs are handled as normal numbers. An extra format stage is added to pre-process the integer and convert instructions while a late zero correction in aligner and multiplier allows single-precision operations to bypass the format stage, saving one cycle.

Latch insertion delay of 2FO4 to 3FO4 occupies a significant portion of the 11FO4 cycle. In order to minimize this delay, NAND logic is merged with latch function. Also, there is a special type of latch which has multiplexers (MUX) at the input terminals [5]. Both types of latches are heavily used at the FPU to improve timing. CMOS static gates are used to implement most of the logic.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 16:51:34 UTC from IEEE Xplore. Restrictions apply.

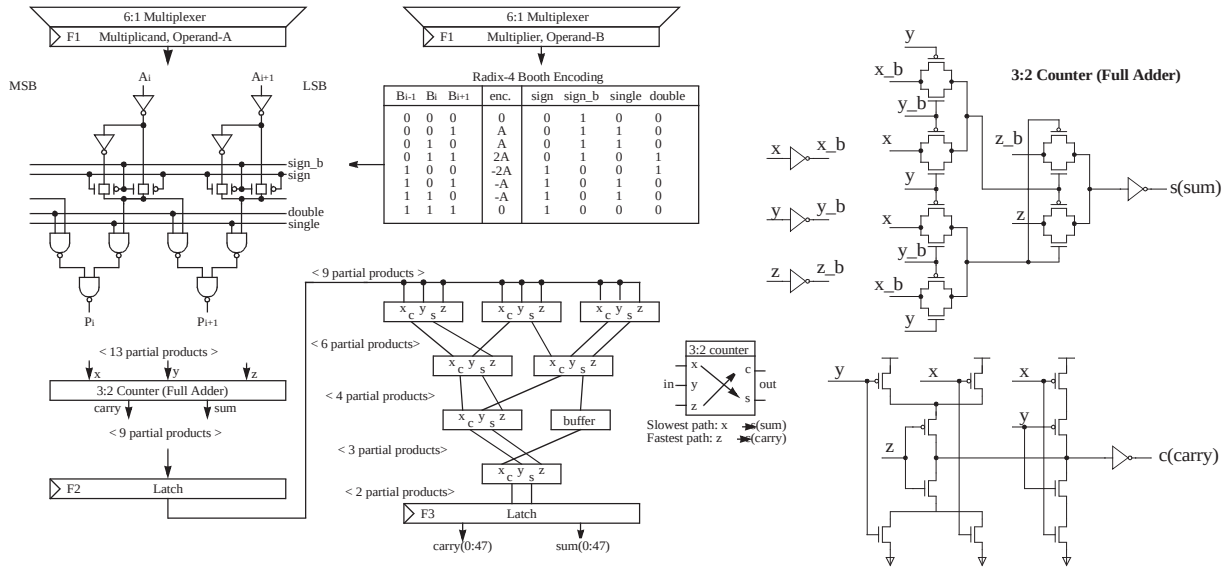


Fig. 2. Radix-4 Booth recording and partial product reduction trees at the multiplier. The 3:2 counter (Full Adder) circuits have static AOI and transmission-gate circuits.

The multiplier and the aligner

The 2-cycle multiplier (Fig. 2) uses a radix-4 Booth encoding; it supports unsigned and 2's complement numbers. Partial product reduction tree uses static full adders (FA) consisting of transmission gate XOR and AOI. The static implementation has advantages over dynamic circuits of reduced wiring need and improved timing optimization. Dynamic adders need true and complement signals; static circuits cut the wire density in half. The timing is optimized exploiting the different path delays in a static FA [6]; in a dynamic FA every signal has the same block delay. Since the FA has the slowest path from x to sum and the fastest path from z to carry, the carry output of one stage is connected to the x input of the next stage. By doing so, we are able to put 4 FA reduction tree stages in 8.5FO4.

The aligner block shifts the addend based on the exponent difference (ED) in order to align it with the product. The shifter is broken into several 4:1 MUX stages. The exponent difference is usually generated by an adder then decoded to select signals for the MUX stages, such that both, adder and decoder, add critical path delay to the aligner (Fig. 3). Our 2-cycle aligner uses a special sum-addressed scheme; adding and decoding of the exponent difference are done in parallel. The last MUX stage is integrated into the special MUX latch, reducing the effective latch insertion delay.

Main 4:1 MUX stages at the aligner consist of 4-input transmission-gate multiplexers (TG-MUX) (Fig. 3). Its output inverter driver has better driving capability over the long horizontal wires than that of the AOI-type MUX. Its short cycle time (e.g., 11FO4) forces to insert non-scan staging latches in the middle of the select signal generations. To solve the reliability problem, special circuits have been implemented at local clock buffer; start latching function as soon as possible when the VDD voltage level becomes stable.

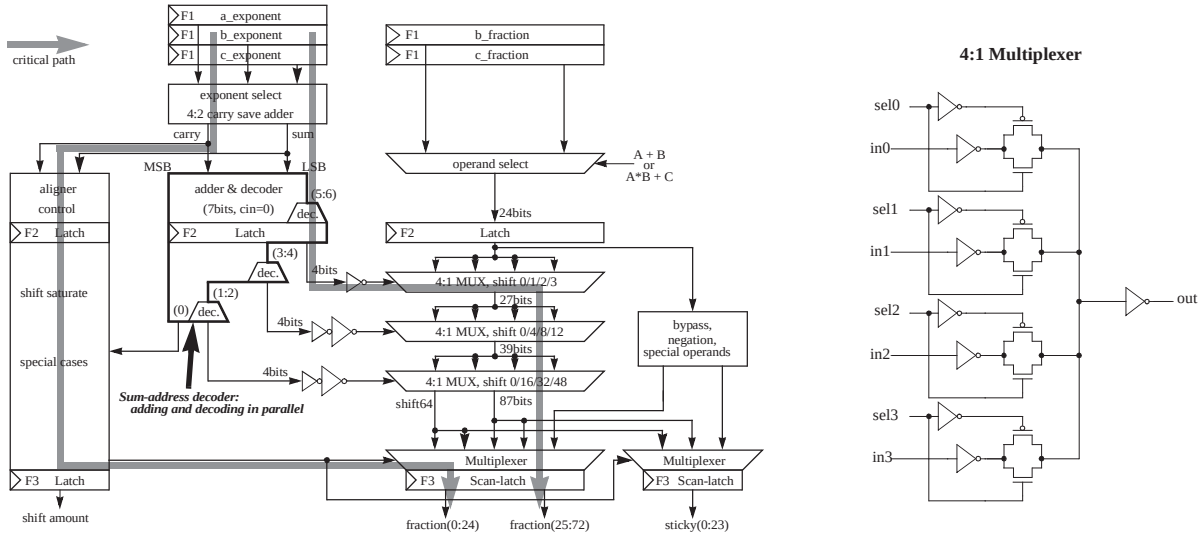


Fig. 3. Sum-address decoder speeds up the shifting stages at the aligner. Main shifting stage consists of 4-input TG-MUXes.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 16:51:34 UTC from IEEE Xplore. Restrictions apply.

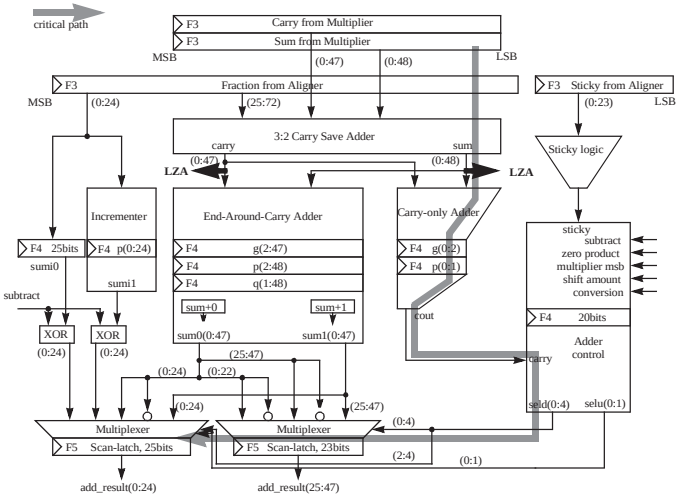


Fig. 4. Critical path of the adder: generating select signals of the multiplexer.

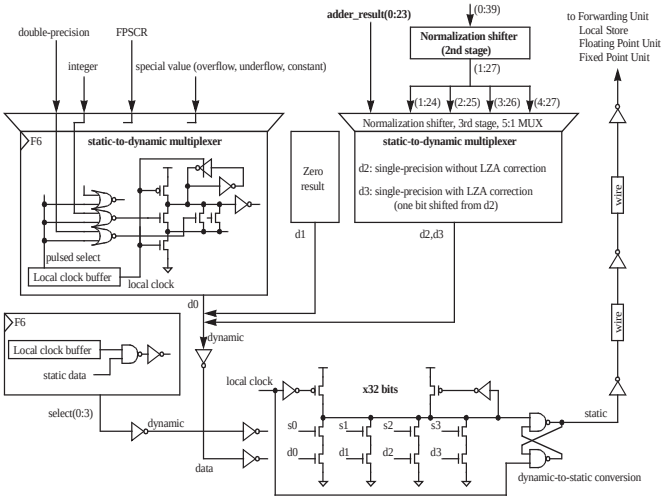


Fig. 5. Static-to-dynamic multiplexer generates the dynamic signals, and then the dynamic result MUX converts the signals to static forwarding signals.

The fraction adder and the leading zero anticipator

The fraction adder (Fig. 4) computes the sum or absolute difference of its inputs using the end-around-carry concept. It is partitioned as incrementer, adder and sticky part. The main adder generates sum (sum0) and sum+1 (sum1). Instruction type, shift amount in the aligner, the sticky bit, and the carry-out (cout) of the adder determine whether *sum0* or *sum1* is chosen, and whether complement of the result is needed. An incrementer generates *sumi1* incremented by one from the 25-bit MSB of addend (sumi0). The subtraction-type instructions re-complement both *sumi0* and *sumi1*. The selection of *sum0*, *sum1*, *complement of sum0*, *sumi0*, and *sumi1* is implemented with the special 6-port MUX latch. The path delay of this multiplexing is hidden by the latch insertion delay. The computation of the select signals based on the *cout* is the critical path of the adder. An extra carry-only adder is used to speed up signal *cout.*

The leading zero anticipator (LZA) estimates the number of leading zeros in the adder result. LZA correction logic checks whether the estimate is accurate or too large by one bit.

The normalizer and the result multiplexer

The normalizer block normalizes the fraction, shifting out the leading zeros. Moving one shift stage to the adder and integrating the last shift stage in a special MUX latch allows for a 1-cycle normalizer.

The FPU only supports round toward zero and non-trap exception conditions. That simplifies the rounding step considerably; the fraction is truncated and no exponent wrapping is needed. The fraction rounder is implemented as a result-MUX. Exponent rounder (ERND) adjusts the exponent, detects exceptions and controls the result-MUX. To match the fast rounder, the ERND delay was improved by 15FO4 using highly optimized checkers and dynamic circuits. ERND uses multiple pre-computed exponent copies (e, e+1, e-1), and a special MUX latch selects the appropriate copy of the ERND results.

The result-MUX requires four select signals: a) results from integer or double precision, b) a zero result, c) single-precision results which need LZA correction, and d) single-precision results which do not need LZA correction. Since the last cycle has only 5FO4 budget due to the result forwarding, a dynamic 4:1 MUX (Fig. 5) was designed. The fraction data and the select signals from ERND use a local clock signal (50% duty-cycle). The output of the result-MUX is converted to static signals through cross-coupled NAND2 circuits.

Physical designs

Physical design considerations such as wiring, pin locations, and floor-plan are carefully considered (Fig. 7). The physical width of each FPU slice is 46 bits and aligned to the operand sources such as the register file. This is very narrow for a FPU since intermediate results require 11-bit exponent and 96-bit fraction. The data flow is split into an exponent and a fraction part. Most fraction macros are folded and their layouts use two different bit images, 16-track/bit and 12-track/bit. 16-track/bit regions assign 9 bits for exponent and 37 bits for fraction, whereas 12-track/bit regions assign 12 and 49 bits, respectively.

From early in the project, wire and re-powering delays were limited to 30% of a cycle. Since wires determine the size of the unit and have timing impacts, strict wiring assignments are applied. Macro level wiring was restricted to Metal-3 and below. Some macros have mandatory empty wiring tracks at Metal-2 for global wiring.

Clock gating to save power consumption

Clock-gating is extensively used to reduce the active power consumption. Only pipeline stages with valid instructions are activated by controlling the local clock buffer. Also,

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 16:51:34 UTC from IEEE Xplore. Restrictions apply.

circuit blocks are clocked down based on instruction type and on operand values, e.g., the multiplier is bypassed for add-type instructions, and the aligner is idle for multiply-type instructions.

Conclusions

The design has roughly 450K transistors in 1.3mm2, fabricated in 90nm SOI technology with 8 levels of copper interconnects. Correct operation has been observed up to 5.6GHz, 1.4V of supply voltage, and 56°C, delivering a peak performance of 44.8GFlops. An interpolated hardware power of 1.4W at 4GHz (e.g., 32GFlops), gives 43.75mW/GFlop.

The VDD voltage vs. operating frequency schmoo plot is shown in Fig. 6., when a SPE is running single-precision floating point computations for lightning and transform workloads.

There are 3 key enablers for this low latency, power efficient, and high frequency FPU: architecture and implementation are optimized for target applications, trading uncritical function for overall performance; logic, circuits and integration are co-designed; the pipeline stages are carefully balanced, achieving the maximum path delay difference of 3% between stages.

Acknowledgments

We thank Osamu Takahashi, Gordon C. Fossum and Barry L. Minor for design efforts, Gilles Gervais and SPU verification team for supports, and Linda V. Grinsven and Robert Putney for the management team.

References

[1] B. Flachs et. al., "The microarchitecture of the streaming processor for a CELL processor," accepted for 2005 IEEE International Solid-State Circuits Conference (ISSCC-2005).

[2] O. Takahashi et. al., "The circuits and physical design of the synergistic processor element of a CELL processor", submitted to 2005 VLSI Circuits Symposium (VLSI-2005).

[3] N. Rohrer et. al., "PowerPC 970 in 130nm and 90nm technologies," ISSCC Dig. Tech. Papers, pp.68-69, Feb. 2004.

[4] S. M. Mueller et. al., "The vector floating-point unit in a synergistic processor element of a CELL processor," submitted to 2005 IEEE 17th Symposium on Computer Arithmetic (ARITH-17).

[5] D. Pham et. al., "The design and implementation of a first-generation CELL processor," accepted for 2005 IEEE International Solid-State Circuits Conference (ISSCC-2005).

[6] V. G. Oklobdzija et. al., "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," IEEE Transactions on Computers, Vol. 45, No.3, pp. 294-305, March 1996

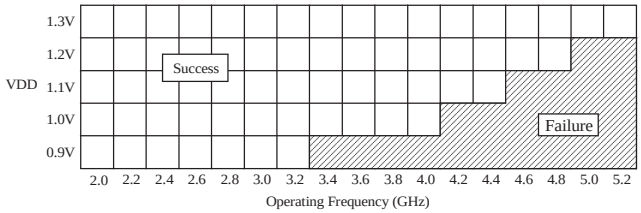


Fig. 6. Voltage vs. Frequency Schmoo: A Synergistic Processing Element is running single-precision floating point computations for lighting and transform workloads.

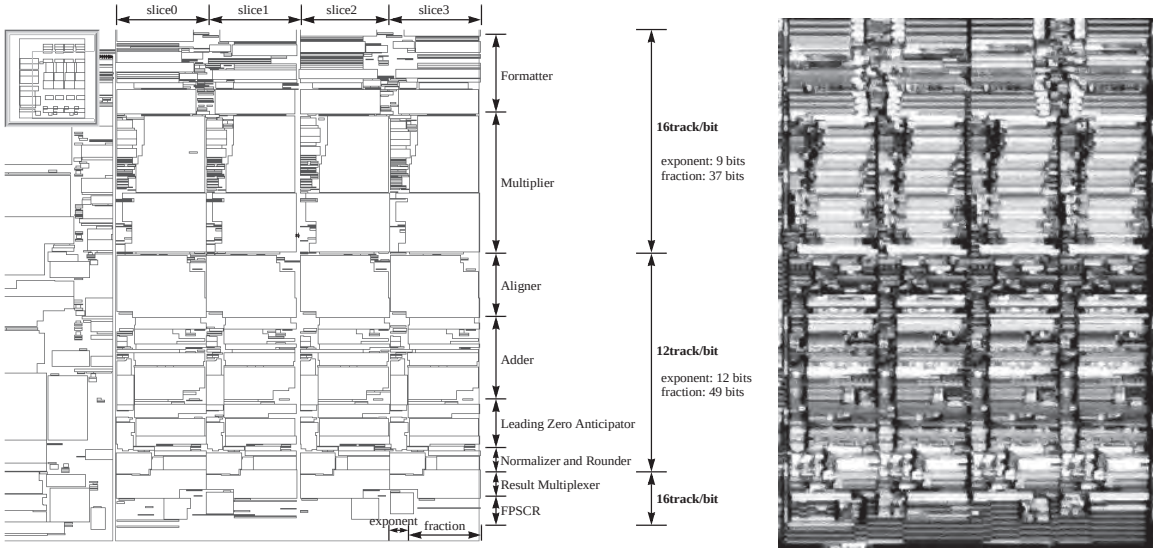


Fig. 7. Macro placement and Die photo: Macro placements are shown at left figure. Two different bit images are used, 16track/bit and 12track/bit. Die photo of FPU in the Synergistic Processing Element are shown at right photo.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 16:51:34 UTC from IEEE Xplore. Restrictions apply.

ATTACHMENT 10C

2-4

A Fully-Pipelined Single-Precision Floating Point Unit in the Synergistic Processor Element of a CELL Processor

Hwa-Joon [illegible] Cottier[1], Brad V[illegible]nc[5],

Document properties

File name:	getPDF.jsp
File size:	472 KB
Title:	A fully-pipelined single-precision floating point unit...
Author:	Hwa-Joon Oh; S.M. Mueller; C. Jacobi; K.D. Tran; S....
Subject:	Digest of Technical Papers. 2005 Symposium on V...
Keywords:	-
Created:	5/19/05, 9:34:49 PM
Modified:	11/18/22, 10:51:34 AM
Application:	Creator
PDF producer:	Acrobat Distiller 7.0 (Windows); modified using iTe...
PDF version:	1.6
Page count:	4
Page size:	8.26 × 11.14 in (portrait)
Fast web view:	No

Close

The floating point u[illegible] of a CELL processor [illegible] designed to accelerate [illegible] 32-bit single-precision [illegible] ands with two different [illegible] of critical single-precisi[illegible] fine-grained clock gat[illegible] logic, circuits and integr[illegible] formance, power, and ar[illegible]

(Keywords: VLSI, CMO[illegible] and Floating point unit)

The Synergistic Processing Element (SPE) [1] of a CELL processor is the first implementation of a new processor architecture designed to optimize the multimedia applications, such as 3D graphics, media streaming, and signal processing [2]. Real time multimedia applications demand single precision performance significantly exceeding that of conventional processors. A single-precision floating-point unit (FPU) of the SPE is a 4-way single-instruction multiple-data (SIMD) design. Most instructions in the FPU process 128-bit operands, divided into four 32-bit words. Each of the 4 slices supports 32-bit single-precision and 16-bit integer multiply-add instruc-

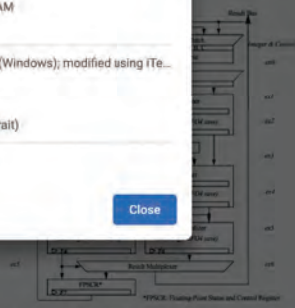


Fig. 1. Overview of the FPU: Design challengers are presented at each block to make a 6-cycle single-precision operation and 7-cycle integer operation with 11FO4 cycle time.

The FPU is optimized for single-precision multiply-add

ATTACHMENT 11A

DAC

Thousand core chips: a technology perspective

Author: Shekhar Borkar Authors Info & Claims

DAC '07: Proceedings of the 44th annual Design Automation Conference • June 2007 • Pages 746–749 • https://doi.org/10.1145/1278480.1278667

Published: 04 June 2007 Publication History

420 3,737

Get Access

Feedback

ABSTRACT

This paper presents the many-core architecture, with hundreds to thousands of small cores, t unprecedented compute performance in an affordable power envelope. We discuss fine grain power management, memory bandwidth, on die networks, and system resiliency for the many-core system.

PDF

Help

References

About Cookies On This Site

We use cookies to ensure that we give you the best experience on our website.

Learn more

Got it!

Show All References

DAC

Index Terms

Thousand core chips: a technology perspective

Computer systems organization

Comments

DL Comment Policy

Comments should be relevant to the contents of this article, (sign in required).

0 Comments

Sort by Newest

PDF

Help

Nothing in this discussion yet.

Privacy **Do Not Sell My Data**

About Cookies On This Site

We use cookies to ensure that we give you the best experience on our website.

Learn more

Got it!

Books
Proceedings
SIGs
Subscription Information
Author Guidelines
Using ACM Digital Library
DAC
Collections
People
ACM Computing Classification System
Digital Library Accessibility

Join

Join ACM
Join SIGs
Subscribe to Publications
Institutions and Libraries

Connect

Contact
Facebook
Twitter
Linkedin
Feedback
Bug Report

The ACM Digital Library is published by the Association for Computing Machinery. Copyright © 2022 ACM, Inc.

Terms of Usage | Privacy Policy | Code of Ethics



PDF
Help

About Cookies On This Site

We use cookies to ensure that we give you the best experience on our website.

Learn more

Got it!

ATTACHMENT 11B

Thousand Core Chips—A Technology Perspective

Shekhar Borkar
Intel Corp, Microprocessor Technology Lab, JF2-04, 2111 NE 25th Ave, Hillsboro, OR 97124.
Shekhar.Y.Borkar@intel.com

ABSTRACT
This paper presents the many-core architecture, with hundreds to thousands of small cores, to deliver unprecedented compute performance in an affordable power envelope. We discuss fine grain power management, memory bandwidth, on die networks, and system resiliency for the many-core system.

Categories and Subject Descriptors
C.0 *Computer Systems Organization, General*

General Terms
Design, Performance, Reliability.

Keywords
CMOS, Power, Memory, Variability, Reliability.

1. Introduction
Moore's Law continues with technology scaling, improving transistor performance to increase frequency, increasing transistor integration capacity to realize complex architectures, and reducing energy consumed per logic operation to keep power dissipation within limit. Advances in software technology, such as rich multimedia applications and run time systems, exploited this performance explosion, delivering end users with higher productivity, seamless internet connectivity, and even multimedia & entertainment. The technology treadmill will continue, providing integration capacity of billions of transistors; however, with several fundamental barriers [1]. In this paper, we will examine some of the barriers, ways to get around them, how it changes the landscape, and how the future advances in technology, architecture, and software, all together could help continue this treadmill.

2. Is Multi-Core enough?
Integration capacity of billions of transistors exists today, and will double every two years. This trend is shown in Figure 1, starting from 2001 with 130nm technology generation, with a 300mm^2 die capable of integrating one billion transistors.

Assuming about half the die area being allocated for logic, and the other half for large memory arrays such as caches, the trend shows that by 2015 you will have 100B transistors on a 300mm^2 die, with almost 1.5B transistors available for logic. The logic transistors tend to be larger than transistors in the memory, take larger space, and consume more power.

How will you employ these logic transistors to deliver performance? The evolutionary approach is to continue today's trend with a few large processor cores, each employing 20 to 100 million logic transistors, and a large shared cache.

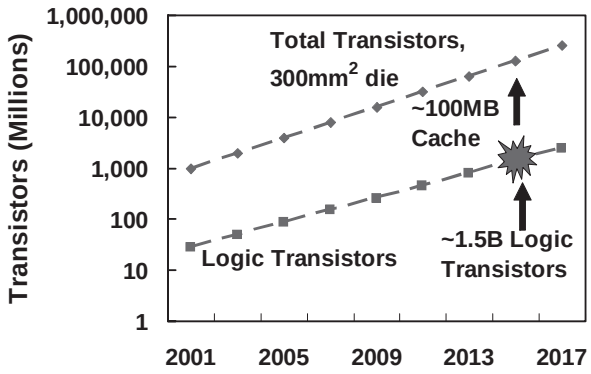


Figure 1: Transistor integration capacity

Performance increase by microarchitecture alone is governed by Pollack's Rule, which states that performance increase is roughly proportional to square root of increase in complexity. In other words, if you double the logic in a processor core, then it delivers only 40% more performance—as evidenced by all the leading processors in the past as shown in Figure 2. It plots integer performance increase of new microarchitectures against area (power) increase from the previous generation microarchitecture, in the same process technology.

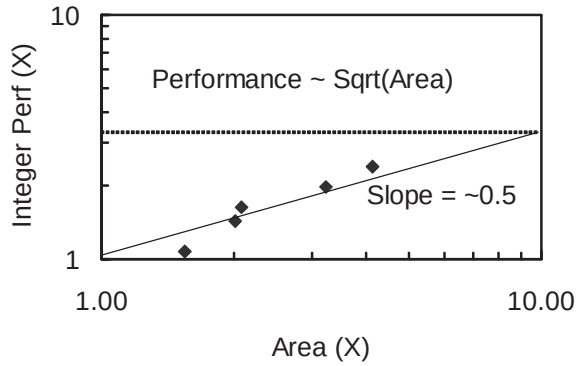


Figure 2: Pollack's Rule

A multi-core microarchitecture, on the other hand, has potential to provide near linear performance improvement with complexity and power. Two smaller processor cores, instead of a large monolithic processor core, can potentially provide 70-80% more performance, as compared to only 40% from a large monolithic core. Multiprocessors have several other benefits as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2007, June 4–8, 2007, San Diego, California, USA.
Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...$5.00.

well: (1) each processor core can be individually turned on or off, thereby saving power; (2) each processor core can be run at its own optimized supply voltage and frequency; (3) easier to load balance among processor cores to distribute heat across the die; and (4) can potentially produce lower die temperatures improving reliability and leakage.

As technology scales further, transistor performance will not increase at the historic rates, due to excessive sub-threshold leakage current, and supply voltage scaling slowing down [1]. Taking these effects into consideration, Figure 3 estimates power consumption of a 300mm^2 processor die.

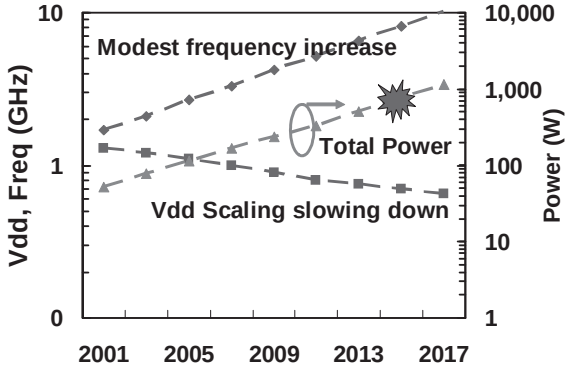


Figure 3: Frequency and Power Consumption

Notice that such a multi-core die will consume almost 1,000 watts of power, which is unreasonable. Therefore, we need to go beyond multi-core, and apply Pollack's rule to the extreme to deliver compute performance in a reasonable power envelope.

3. From Multi—to Many-Cores

Therefore, business as usual is not an option. You cannot simply follow the path of multi-core evolution, integrating multiple complex cores on a die. Instead, we propose that you integrate lots of smaller cores. Each small core delivers lower performance than a large complex core; however, the total compute throughput of the system is much higher as follows.

If you have 1B logic transistor budget, instead of integrating 10 large 100M transistor cores, we propose to integrate 100 medium 10M transistor cores, or even 1,000 small 1M transistor cores. Applying Pollack's rule inversely, performance of a smaller core reduces as square-root of the size, but power reduction is linear, resulting in smaller performance degradation with much larger power reduction. Overall, the compute throughput of the system, on the other hand, increases linearly with the larger number of small cores.

A many-core system on a die does not necessarily have to be symmetric or homogenous. An asymmetric system may have a few large cores to deliver higher single-thread performance, but will predominantly have large number of small cores. A heterogeneous system may even integrate diverse special purpose cores for hardware acceleration, e.g. graphics engines.

Figure 4 illustrates such a heterogeneous many-core system with general purpose cores (GP), and special purpose cores (SP), each core having local cache memory, and all cores connected together with an on-die interconnection network.

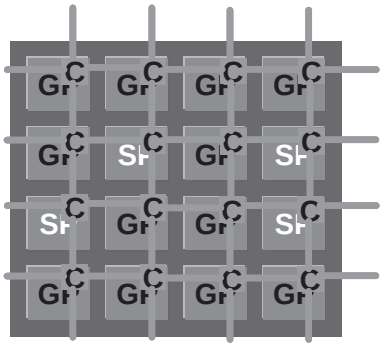


Figure 4: Illustration of a Many Core System

4. Performance of a Many-Core System

Although it is true that a many-core system will deliver higher compute throughput than a multi-core system for the same die size and in the same power envelope, it may be difficult to harvest the performance. The limitation is Amdahl's Law, which states that the parallel speedup is limited by the serial code in a program:

Parallel Speedup = 1/(Serial% +(1-Serial%)/N)

If the serial percentage in a program is large, then parallel speedup saturates with small number of cores. Figure 5 illustrates impact of serial percentage of code on parallel speedup.

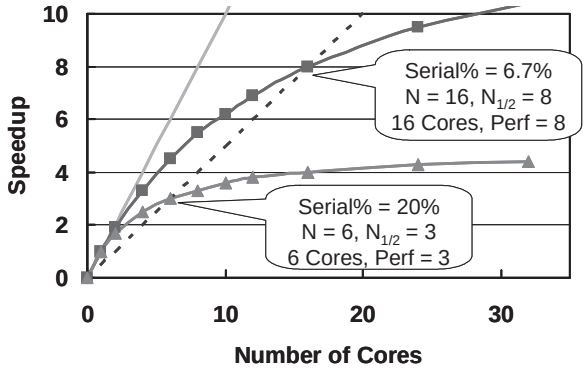


Figure 5: Amdahl's Law limits parallel speedup

Notice that even 7% serial code impacts parallel speedup adversely with diminishing return beyond 16 cores, delivering the performance of only 8 cores, thereby limiting application of inverse Pollack's rule up to only 16 cores. This limitation is true if only one application is running on the system at any given time, and you try to parallelize a single application across all the cores. In practice there are multiple applications running, each with multiple tasks and multiple threads, and thus there exists opportunity to harvest the performance of a many core system.

To explore this concept further, consider three systems on a chip comprising 1: 12 large (60MT) cores, 2: 48 medium (15MT) cores, and 3: 144 small (5MT) cores, all of them with the same amount of total cache, and all of them in the same power envelope. Figure 6 compares their relative performance.

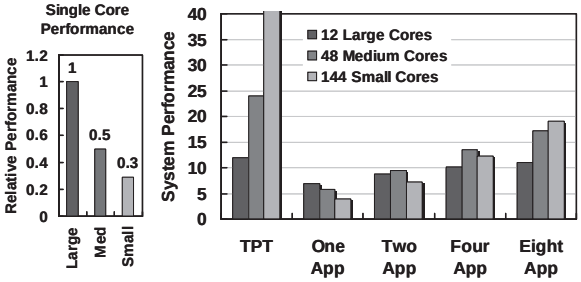


Figure 6: Performance of Large, Medium, and Small Cores

Small core is 12 times smaller than the large core, but its performance is only about a third of the large core. Total throughput of the small core system (TPT), however, is much higher than the large core system. If you parallelize one application for a 12 large core system and a 144 small core system then the small core system performs poorly, as expected from Amdahl's Law. Notice that as the number of applications increase, total system performance of medium and small core systems increase. Therefore, to harvest the performance of a many-core system, we cannot just depend on parallelizing a single application, but must utilize task level and application level parallelism.

5. Power and Energy

So far we have discussed how to maximize compute performance using many-cores, using hundreds or even thousands of small cores. Now we discuss how to fit the system in an affordable power envelope.

The best lever to reduce power with minimal impact on performance is to use voltage scaling. Applying voltage scaling indiscriminately to the entire system would lower power, but may not be optimal. Instead we propose to exploit the fact that there are hundreds to thousands of cores, and each core can be voltage scaled individually, thus employing fine grain power management. Individual cores can be voltage and frequency scaled to any arbitrary voltage and frequency in the possible range, but this could be difficult and cumbersome. Different cores running at different voltages and frequency would create asynchronous interfaces, adding latency, meta-stability, and would require complex power delivery system.

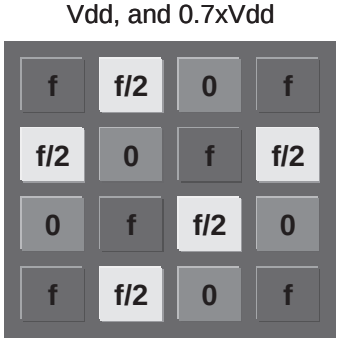


Figure 7: Fine grain power management

We propose a much simpler method of fine grain power management illustrated in Figure 7. A core operates at one of the two frequencies: f and f/2, where f is the maximum frequency of operation. When a core operates at f/2, it uses lower voltage, say 0.7xVdd, thus consuming only 25% of the maximum power. Idle cores could be turned off to save leakage power. Thus you need only two supply voltages, two sleep transistors [2] in each core to select a supply voltage, and a frequency divider in each core to select the frequency of operation. Since the number of cores is large, this coarse grain power management of a core enables fine grain power management of the entire many core system, with only two supply voltages, simplifying design and power delivery.

If all the cores were operating at half the frequency then the power consumption would be approximately 200W. Therefore, we estimate power consumption of about 300W for a 300mm^2 die, or 100W for a 100mm^2 die, fitting in the affordable power envelope.

6. Design Considerations

Since a core should be capable of operating at a much lower voltage, it warrants careful design practices. In the past, designs had to operate at the highest possible frequency to deliver high performance, but not anymore; it's the system performance that matters. Hence, design styles, such as domino logic, which are power hungry and do not scale well with voltage should not be employed. Instead, simple and robust static CMOS logic, which consumes much lower power, must be used. Large memory arrays such as caches and register files tend to be sensitive to lower voltages too, often due to design tradeoffs to increase their density. These arrays will have to be designed for robustness for low voltage operation, sacrificing some density.

7. On die network

The backbone of the many-core system is the network on the chip, connecting all the cores together, and carrying memory and IO traffic. In a shared memory system, this network carries cache coherence traffic to keep caches coherent, which is bandwidth intensive and latency sensitive. The network can be a ring [3] or a higher dimensional network such as a mesh [4] to reduce latency. It is typically implemented as a packet switched network carrying packets buffered at each node.

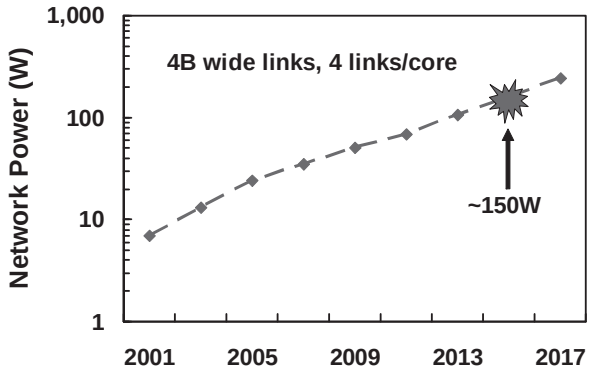


Figure 8: Network power estimate

These packet switched networks tend to be power hungry, consuming almost 500mW of power at each node. Assuming

even narrow 4 byte wide links, 4 per core, with 1MT small cores, Figure 8 shows estimated network power for a many core die, confirming that the network power is substantial.

Power management in the network is difficult, if not impractical, because any power management technique, such as clock gating or sleep-states, incurs wake up latency, impacting system performance. On the one hand, smaller cores give you higher throughput performance in the same power envelope as discussed before, but on the other hand, it also increases the number of network nodes, increasing the network power. A careful study of the system power and performance is necessary to balance the size of the core, and the number of cores.

8. Memory Bandwidth

A many core system with thousands of cores will demand 100's of GB of memory bandwidth, and a traditional memory subsystem solution is not sufficient. A memory bus IO circuit consumes ~25mW/Gbps, and consequently a memory bus alone would consume about 25W to deliver 100GB/s memory bandwidth, which is excessive. The IO circuits employ sophisticated signal processing techniques to attain high data rate, consuming much of the power. If the bus lengths are somehow made substantially smaller, say a few millimeters, then the buses behave like lumped capacitor, rather than as transmission lines, and the IO circuits would consume substantially lower power, of the order of 1-2mW/Gbps.

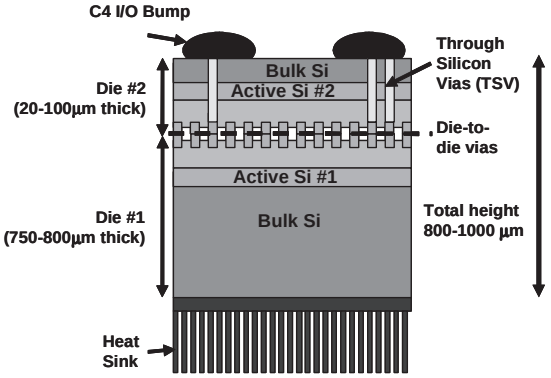


Figure 9: Three dimensional interconnect with stacking

A three dimensional integration (3D) of a memory with processor is a potential solution, where a thinned memory die is placed between the processor and the package [4]. Signals and power to the processor are routed through the memory die using through silicon vias, as shown in Figure 9 & Figure 10.

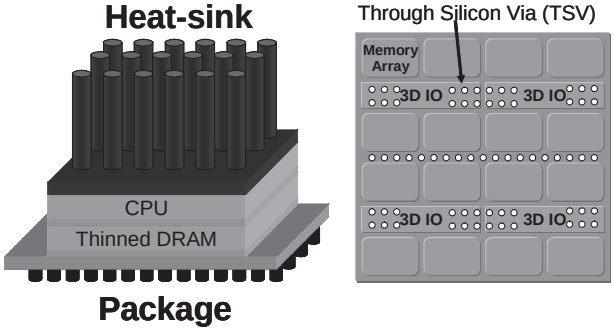


Figure 10: Assembly of 3D memory

This 3D integration of memory will consume only a few watts of IO power, and provide substantial memory bandwidth to the many-core system. Of course, architecturally, this adds another level of memory hierarchy and needs further attention.

9. Resiliency

As technology scales further, variations become prominent [5]. Chips will encounter dynamic variations of supply voltage and temperature; frequent and intermittent soft-errors; and transistors that slowly age and degrade over time, degrading circuit performance [6]. Despite these difficulties, users expect the system to remain reliable and to continue to deliver the rated performance. This challenge will undoubtedly require a major paradigm shift in all aspects of VLSI design—fabrication, design, microarchitecture, testing, software, and applications. Many-cores, with hundreds to thousands of cores provide resiliency to combat this problem.

Many-cores in a system will provide redundancy with spare cores, and functional redundancy checking employed at a coarse-grained level. For example, one core could check results produced by several cores; of course, software and applications will have to support this concept whenever possible. We could distribute test functionality as a part of the hardware to dynamically detect errors, or to isolate and correct aging and faulty cores, by replacing from the spare cores.

This microarchitecture strategy, with many-cores to assist in redundancy, is called resilient microarchitecture. It continually detects errors, isolates faults, confines faults, reconfigures the hardware, and thus adapts. If we can make such a strategy work, there is no need for one-time factory testing or burn-in, since the system is capable of testing and reconfiguring itself to make itself work reliably throughout its lifetime.

10. Conclusion

The many-core architecture with hundreds to thousands of small cores delivers unprecedented compute performance in an affordable power envelope. Fine grain system power management, an optimized on-die-network, and 3D memory technology are vital, and a many-core system also provides resiliency to combat variability and reliability.

11. References

[1] Shekhar Borkar, "Design Challenges of Technology Scaling, IEEE Micro", July-August 1999.

[2] Tschanz J., et al, " Dynamic sleep transistor and body bias for active leakage power control of microprocessors", IEEE Journal of Solid State Circuits, November 2003.

[3] Pham D. et al, "The design and implementation of a first-generation CELL processor", ISSCC 2005.

[4] Vangal et al, "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS", ISSCC 2007.

[5] Shekhar Borkar et al, "Parameter Variations and Impact on Circuits and Microarchitecture", Proceedings, DAC 2003.

[6] Shekhar Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation", IEEE Micro, November-December 2005.

ATTACHMENT 11C

Thousand Core Chips—A Technology Perspective

Intel Corp, Micro ro, OR 97124.

ABSTRACT

This paper presents the t thousands of small core performance in an afford grain power management and system resiliency for

Categories and Su
C.0 *Computer Systems Or*
General Terms
Design, Performance, Rel
Keywords
CMOS, Power, Memory

1. Introduction

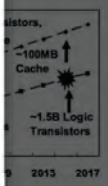
Moore's Law continues transistor performance transistor integration cap and reducing energy co power dissipation wi technology, such as rich systems, exploited this users with higher produ and even multimedia & will continue, providing transistors, however, wi this paper, we will exam around them, how it cha advances in technology, architecture, and software, all together could help continue this treadmill.

2. Is Multi-Core enough?

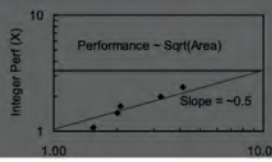
Integration capacity of billions of transistors exists today, and will double every two years. This trend is shown in Figure 1, starting from 2001 with 130nm technology generation, with a 300mm² die capable of integrating one billion transistors.

Assuming about half the die area being allocated for logic, and the other half for large memory arrays such as caches, the trend shows that by 2015 you will have 100B transistors on a 300mm² die, with almost 1.5B transistors available for logic. The logic transistors tend to be larger than transistors in the memory, take

is to continue today's nch employing 20 to ared cache.



es capacity
alone is governed by formance increase is ease in complexity. In processor core, then it evidenced by all the in Figure 2. It plots oarchitectures against previous generation microarchitecture, in the same process technology.



Document properties

File name:	1278480.1278667
File size:	218 KB
Title:	Thousand Core Chips - A Technology Perspective
Author:	-
Subject:	42.1
Keywords:	-
Created:	5/2/07, 5:59:17 AM
Modified:	12/4/22, 12:27:31 PM
Application:	AFPL Ghostscript 8.54 PDF Writer
PDF producer:	AFPL Ghostscript 8.54 ps2pdf.com; modified usin...
PDF version:	1.3
Page count:	4
Page size:	8.50 × 11.00 in (portrait)
Fast web view:	No

Close

ATTACHMENT 12A

IEEE.org IEEE *Xplore* IEEE SA IEEE Spectrum More Sites

Cart Create Account Personal Sign In

Browse My Settings Help

Access provided by: **University of Nebraska - Lincoln**

Sign Out

Access provided by: **University of Nebraska - Lincoln**

Sign Out

All

ADVANCED SEARCH

< Previous | Back to Results

Conferences > 2008 IEEE International Solid...

A 65nm 2-Billion-Transistor Quad-Core Itanium® Processor

Publisher: IEEE

Cite This PDF

<< Results | < Previous

Blaine Stackhouse ; Brian Cherkauer ; Mike Gowan ; Paul Gronowski ; Chris Lyles **All Authors**

23 Paper Citations

1 Patent Citation

400 Full Text Views

Alerts

Manage Content Alerts

Add to Citation Alerts

Abstract

Authors

Figures

References

Citations

Keywords

Metrics

Media

More Like This

Downl

PDF

Abstract: The next generation in the Itanium processor family is introduced. The processor has 4 dual-threaded cores integrated on die with a system interface and 30MB of cache in ... **View more**

Metadata

Abstract:

The next generation in the Itanium processor family is introduced. The processor has 4 dual-threaded cores integrated on die with a system interface and 30MB of cache in an 8M 65nm process. The 21.5times32.5mm ^{2} die contains 2.05 billion transistors. The silicon is designed to operate the cores up to 2.0GHz and the system interface at 2.4GHz, with a thermal design power (TDP) of 170W at 110degC.

Published in: 2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers

Date of Conference: 03-07 February **INSPEC Accession Number:**

More Like This

Process induced damages from various integrated circuit interconnection designs - limitations of antenna rule under practical integrated circuit layout practice

2003 8th International Symposium Plasma- and Process-Induced Damage.

Published: 2003

Quasi static analysis of hybrid and monolithic integrated circuits interconnections

Proceedings of IEEE 43rd Electronic Components and Technology Conference (ECTC '93)

Published: 1993

Show More

2008

10047804

Date Added to IEEE *Xplore*: 09 January 2009

DOI: 10.1109/ISSCC.2008.4523072

Publisher: IEEE

ISBN Information:

Conference Location: San Francisco, CA, USA

ISSN Information:

Contents

The next generation in the Itanium processor family is introduced. The processor has 4 dual-threaded cores integrated on die with a system interface and 30MB of cache in an 8M 65nm process. The $21.5 \times 32.5\mathrm{mm}^2$ die contains 2.05 billion transistors (Fig. 4.6.1). The silicon is designed to operate the cores up to 2.0GHz and the system interface at 2.4GHz, with a thermal design power (TDP) of 170W at 110°C.

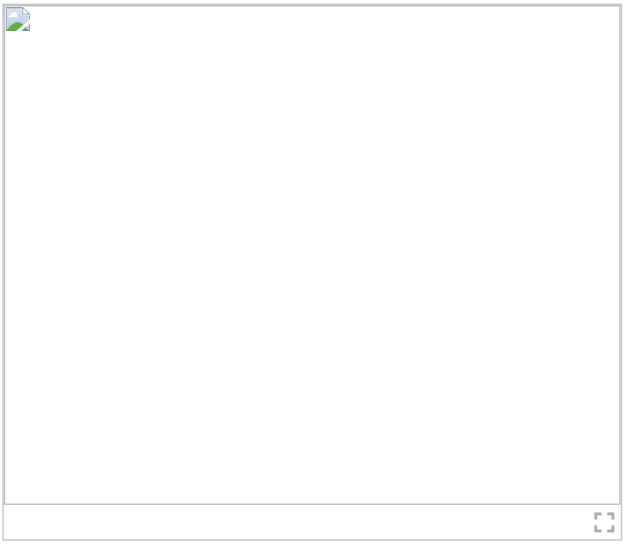


Figure 4.6.1:
Chip floorplan and statistics.

Improvements over previous designs include the integration of two additional cores on the die [1], [2] with circuit methodology enhancements for low-voltage operation. The caches are optimized for improved $\mathrm{V}_{\mathrm{CCMIN}}$ and low standby power operation. New to the chip is the inclusion of high-speed interconnects, the integration of a system interface using SER-tolerant storage elements, the implementation of a fully deterministic power-frequency management system, and the design of a tunable di/dt controller.

This is the first Itanium architecture chip to include the QuickPath interconnect, which allows each processor design to implement a different number of interconnects operating at different frequencies. This design implements 4 full-width and 2 half-width interconnects that enable processor-to-processor and

processor-to-I/O communication with a peak bandwidth of 96GB/s. The chip also implements 4 duplex FBDIMM channels capable of accessing memory at a maximum bandwidth of 34GB/s. Together, these high-speed interconnects deliver $6\times$ the memory bandwidth and 9x the compute bandwidth of previous Itanium processors.

Each interconnect is designed with a mixture of complex analog circuits and static SER-tolerant storage for the queues and controllogic (Fig. 4.6.2). These circuits operate across multiple voltage and frequency domains and the I/O signals are double-pumped at a maximum frequency 4.8GT/s using low-jitter clock elements. Each PLL and DLL in an interconnect implements a duty-cycle correction system on its output using a differential amplifier comparator (Fig. 4.6.3). The physical layer consumes 123mm^2, contains 26 million transistors, and contains 237K SER-tolerant scan latches.

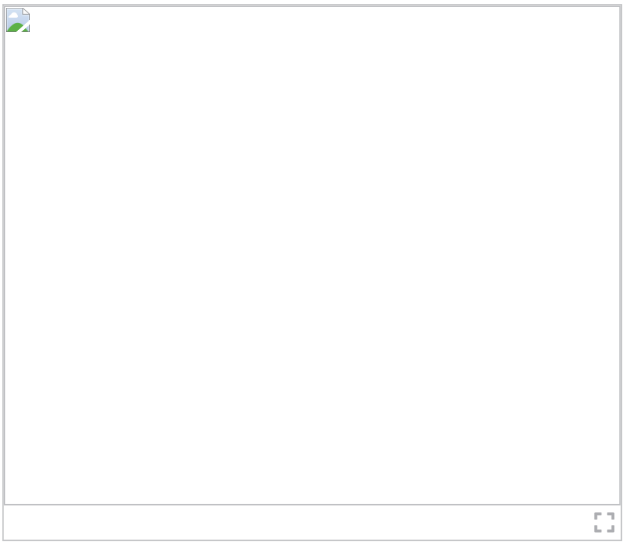


Figure 4.6.2:
I/O transmitter and receiver floorplan.

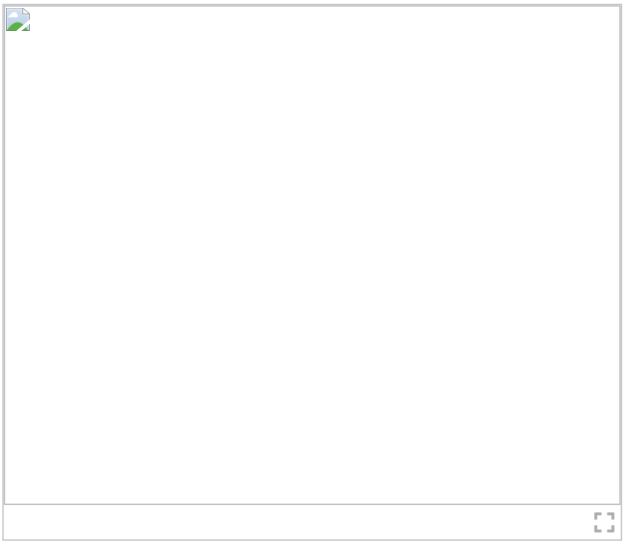


Figure 4.6.3:
Clock duty cycle correction amplifier.

The caches are redesigned to improve performance relative to previous Itanium processors (Fig 4.6.4). All cache memory cells are placed on a separate high-voltage supply to improve V_{CCMIN} and eliminate sensitivity to erratic bits. Low-latency voltage-conversion circuits allow synchronous accesses of the cache without frequency penalties. Redundancy is added through more layers of the hierarchy to improve manufacturing yield and reliability. Finally, the 24MB L3 cache implements PMOS-based RAM-cell sleep circuitry to lower leakage power consumption when the arrays are in standby mode.

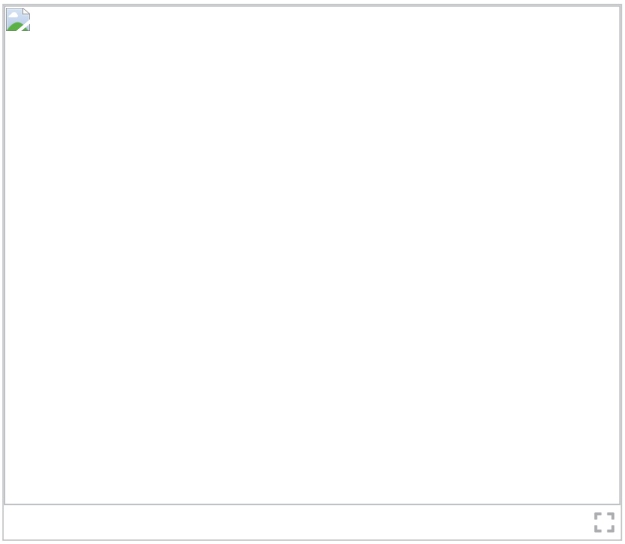


Figure 4.6.4:
Cache feature summary.

The system interface features a 12-port switch/router that is responsible for concurrent full-duplex intra- and inter-processor communications. Dual integrated memory controllers serve as home agents for the directory-based cache coherence protocol and interface the microprocessor to memory. A system-utilities controller performs functions such as processor configuration access and implements SMBus master and off-chip flash ROM interfaces.

The system interface is implemented with fully static circuits. To reduce design time, automation is heavily used. Control blocks with embedded register file arrays are fully synthesized. Datapaths are implemented with an automated compiler. A memory compiler is used to generate all register files and CAM arrays.

Because enterprise servers require highly reliable systems, all state in the system interface is protected against soft errors. Large register files are protected with ECC, medium-sized register files are protected with SER-hardened RAM cells and parity, and small register files are protected with SER-hardened RAM cells. Furthermore, 99% of latches and flip-flops in the system interface are SER-hardened [3].

To connect the system interface to the cores and I/O physical layer, the chip implements a synchronizer and routing architecture that is distributed across the die (Fig. 4.6.5).

Synchronizer FIFOs handle data crossings and control when data is sent between the different clock domains of the cores, QuickPath interconnects, and the system interface. This flow control is then used by the router to optimize its arbitration logic. Each 16-entry synchronizer FIFO is configured at reset to transfer data based on a gearing scheme that accounts for frequency mismatch and route flight time, while the distributed synchronizers align to the clock system's heartbeat signal to start up in lockstep.

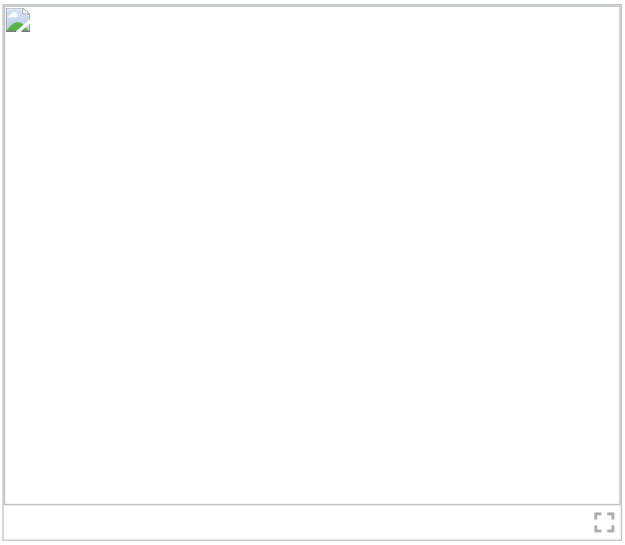


Figure 4.6.5:
Synchronizer locations at clock region interfaces.

Synchronizers are implemented in pairs, with a FIFO for each direction of an interface physically located near its receiving block. This allows use of local clocks to read out, while writing in with source-synchronous clocks routed with the input data. To enable source-synchronous routing for 15mm across the die, retiming stations are implemented at points along the routes to realign data and clocks.

The design methodology prefers lower-leakage devices to maintain the power-limited nature of the chip. Excluding SRAM, 79% of the core transistors and 84% of the system interface transistors are lower-leakage devices.

The chip includes a new design for power-frequency management based on digital activity sensing and discrete voltage-frequency pairs. In each of the 4 cores, approximately 120 architectural events are monitored, each weighted to represent the relative amount of capacitance switched by that event. A central control unit accumulates these events and makes a decision approximately every $6\mu s$ of which core voltage-frequency pair is appropriate for that activity. The capacitance thresholds are stored, along with voltage-frequency pairs, in a look-up table fused in production. By keeping weights and switching thresholds identical among parts, the frequency decisions of every part are identical, and not subject to part-to-part and environmental variability.

Frequency changes are accomplished by two parallel PLLs that are mux-selected. One drives the core clock while the other locks to the next requested ratio. When the ratio change is executed, the core clock is halted momentarily, the PLL mux select is switched, and the core clock is restarted at the new ratio.

The design includes a tunable di/dt controller. The controller prevents sudden changes in current by limiting how quickly the issue rate of high-power instructions increases and by injecting artificial activity to smooth out sudden drops in the issue rate of these instructions. The controller is designed to reduce first droop in the cores by 30 to 50%. The simulated step from minimum to maximum current draw in the cores, with and without the controller enabled, is shown in Fig. 4.6.6. The controller breaks the one large current step into three smaller steps staggered in time to limit voltage droop.

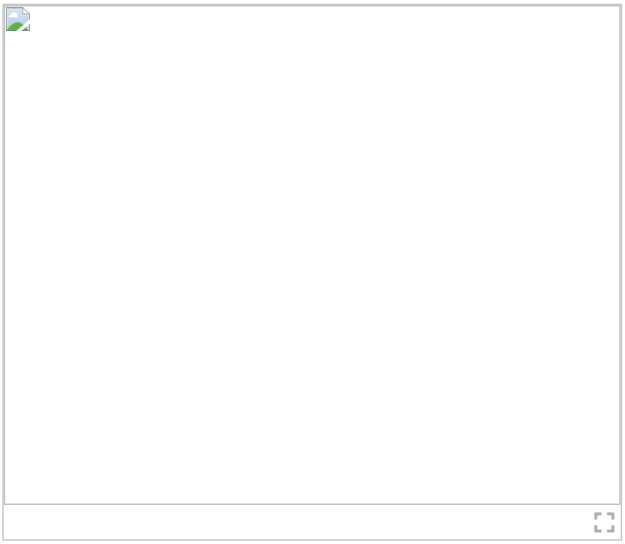


Figure 4.6.6:
Simulation results of di/dt controller.

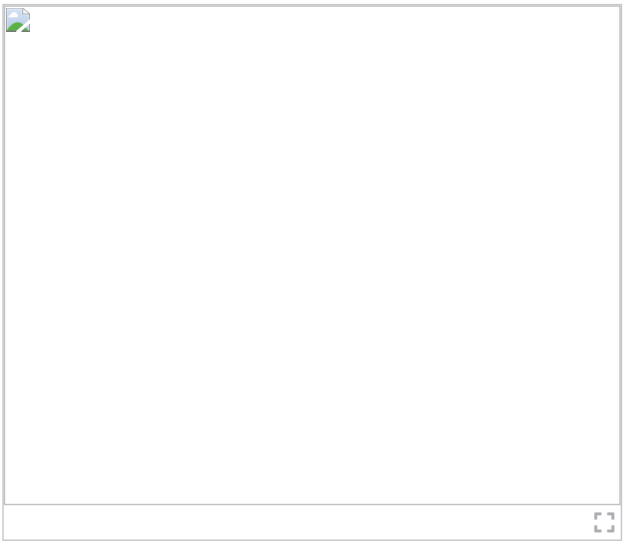


Figure 4.6.7:
Die micrograph.

ACKNOWLEDGEMENTS

The authors thank the entire design team from Fort Collins, CO and Hudson, MA.

Authors

Figures

References

Citations

Keywords

Metrics

Media

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved.

IEEE Account

- Change Username/Password
- Update Address

Purchase Details

- Payment Options
- Order History
- View Purchased Documents

Profile Information

- Communications Preferences
- Profession and Education
- Technical Interests

Need Help?

- **US & Canada:** +1 800 678 4333
- **Worldwide:** +1 732 981 0060
- Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.

ATTACHMENT 12B

Please click on paper title to view Visual Supplement.

4.6 A 65nm 2-Billion-Transistor Quad-Core Itanium® Processor

Blaine Stackhouse[1], Brian Cherkauer[1], Mike Gowan, Paul Gronowski[2], Chris Lyles[1]

[1]Intel, Fort Collins, CO, [2]Intel, Hudson, MA

The next generation in the Itanium processor family is introduced. The processor has 4 dual-threaded cores integrated on die with a system interface and 30MB of cache in an 8M 65nm process. The 21.5×32.5mm² die contains 2.05 billion transistors (Fig. 4.6.1). The silicon is designed to operate the cores up to 2.0GHz and the system interface at 2.4GHz, with a thermal design power (TDP) of 170W at 110°C.

Improvements over previous designs include the integration of two additional cores on the die [1,2] with circuit methodology enhancements for low-voltage operation. The caches are optimized for improved V_{CCMIN} and low standby power operation. New to the chip is the inclusion of high-speed interconnects, the integration of a system interface using SER-tolerant storage elements, the implementation of a fully deterministic power-frequency management system, and the design of a tunable di/dt controller.

This is the first Itanium architecture chip to include the QuickPath interconnect, which allows each processor design to implement a different number of interconnects operating at different frequencies. This design implements 4 full-width and 2 half-width interconnects that enable processor-to-processor and processor-to-I/O communication with a peak bandwidth of 96GB/s. The chip also implements 4 duplex FBDIMM channels capable of accessing memory at a maximum bandwidth of 34GB/s. Together, these high-speed interconnects deliver 6× the memory bandwidth and 9× the compute bandwidth of previous Itanium processors.

Each interconnect is designed with a mixture of complex analog circuits and static SER-tolerant storage for the queues and control logic (Fig. 4.6.2). These circuits operate across multiple voltage and frequency domains and the I/O signals are double-pumped at a maximum frequency 4.8GT/s using low-jitter clock elements. Each PLL and DLL in an interconnect implements a duty-cycle correction system on its output using a differential amplifier comparator (Fig. 4.6.3). The physical layer consumes 123mm², contains 26 million transistors, and contains 237K SER-tolerant scan latches.

The caches are redesigned to improve performance relative to previous Itanium processors (Fig 4.6.4). All cache memory cells are placed on a separate high-voltage supply to improve V_{CCMIN} and eliminate sensitivity to erratic bits. Low-latency voltage-conversion circuits allow synchronous accesses of the cache without frequency penalties. Redundancy is added through more layers of the hierarchy to improve manufacturing yield and reliability. Finally, the 24MB L3 cache implements PMOS-based RAM-cell sleep circuitry to lower leakage power consumption when the arrays are in standby mode.

The system interface features a 12-port switch/router that is responsible for concurrent full-duplex intra- and inter-processor communications. Dual integrated memory controllers serve as home agents for the directory-based cache coherence protocol and interface the microprocessor to memory. A system-utilities controller performs functions such as processor configuration access and implements SMBus master and off-chip flash ROM interfaces.

The system interface is implemented with fully static circuits. To reduce design time, automation is heavily used. Control blocks with embedded register file arrays are fully synthesized. Datapaths are implemented with an automated compiler. A memory compiler is used to generate all register files and CAM arrays. Because enterprise servers require highly reliable systems, all state in the system interface is protected against soft errors. Large register files are protected with ECC, medium-sized register files are protected with SER-hardened RAM cells and parity, and small register files are protected with SER-hardened RAM cells. Furthermore, 99% of latches and flip-flops in the system interface are SER-hardened [3].

To connect the system interface to the cores and I/O physical layer, the chip implements a synchronizer and routing architecture that is distributed across the die (Fig. 4.6.5). Synchronizer FIFOs handle data crossings and control when data is sent between the different clock domains of the cores, QuickPath interconnects, and the system interface. This flow control is then used by the router to optimize its arbitration logic. Each 16-entry synchronizer FIFO is configured at reset to transfer data based on a gearing scheme that accounts for frequency mismatch and route flight time, while the distributed synchronizers align to the clock system's heartbeat signal to start up in lockstep.

Synchronizers are implemented in pairs, with a FIFO for each direction of an interface physically located near its receiving block. This allows use of local clocks to read out, while writing in with source-synchronous clocks routed with the input data. To enable source-synchronous routing for 15mm across the die, retiming stations are implemented at points along the routes to realign data and clocks.

The design methodology prefers lower-leakage devices to maintain the power-limited nature of the chip. Excluding SRAM, 79% of the core transistors and 84% of the system interface transistors are lower-leakage devices.

The chip includes a new design for power-frequency management based on digital activity sensing and discrete voltage-frequency pairs. In each of the 4 cores, approximately 120 architectural events are monitored, each weighted to represent the relative amount of capacitance switched by that event. A central control unit accumulates these events and makes a decision approximately every 6µs of which core voltage-frequency pair is appropriate for that activity. The capacitance thresholds are stored, along with voltage-frequency pairs, in a look-up table fused in production. By keeping weights and switching thresholds identical among parts, the frequency decisions of every part are identical, and not subject to part-to-part and environmental variability.

Frequency changes are accomplished by two parallel PLLs that are mux-selected. One drives the core clock while the other locks to the next requested ratio. When the ratio change is executed, the core clock is halted momentarily, the PLL mux select is switched, and the core clock is restarted at the new ratio.

The design includes a tunable di/dt controller. The controller prevents sudden changes in current by limiting how quickly the issue rate of high-power instructions increases and by injecting artificial activity to smooth out sudden drops in the issue rate of these instructions. The controller is designed to reduce first droop in the cores by 30 to 50%. The simulated step from minimum to maximum current draw in the cores, with and without the controller enabled, is shown in Fig. 4.6.6. The controller breaks the one large current step into three smaller steps staggered in time to limit voltage droop.

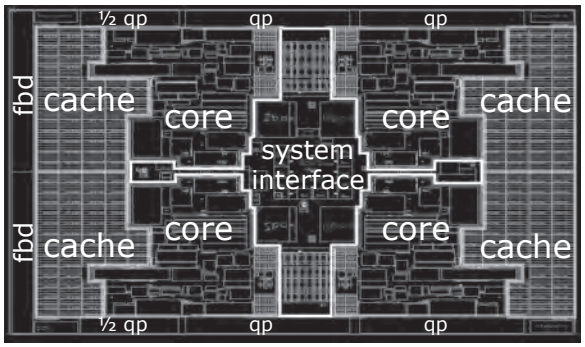
Acknowledgements:
The authors thank the entire design team from Fort Collins, CO and Hudson, MA.

References:

[1] S. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, et al., "The Implementation of the Itanium 2 Microprocessor", *IEEE J. of Solid State Circuits*, vol. 37, no. 11, pp. 1448-1460, Nov. 2002.
[2] S. Naffziger, B. Stackhouse, T. Grutkowski, D. Josephson, et al., "The Implementation of a 2-Core, Multi-Threaded Itanium® Family Microprocessor", *IEEE J. of Solid State Circuits*, vol. 41, no. 1, pp. 197-209, Jan. 2006.
[3] D. Krueger, E. Francom, J. Langsdorf, "Circuit Design for Voltage Scaling and SER Immunity on a Quad-Core Itanium® Family Processor", *ISSCC Dig. Tech. Papers,* pp. 94-95, Feb. 2008.

 978-1-4244-2011-7/08/$25.00 ©2008 IEEE

Attachment - Page A322
Please click on paper title to view a Visual Supplement.
Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 18:55:17 UTC from IEEE Xplore. Restrictions apply.



	FET count	Voltage	Power
Core logic	430M	0.9-1.15V	100W
Sys Int	157M	0.9-1.15V	30W
L3 cache	1,420M	1.10V	20W
IO logic	39M	1.10V	20W
Chip Total	**2.046B**		**170W**

Figure 4.6.1: Chip floorplan and statistics.

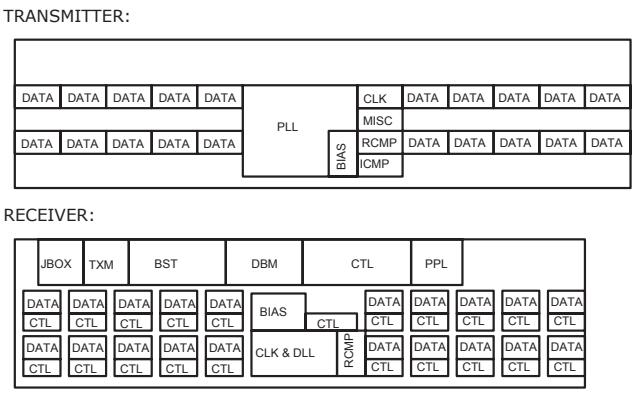


Figure 4.6.2: I/O transmitter and receiver floorplan.

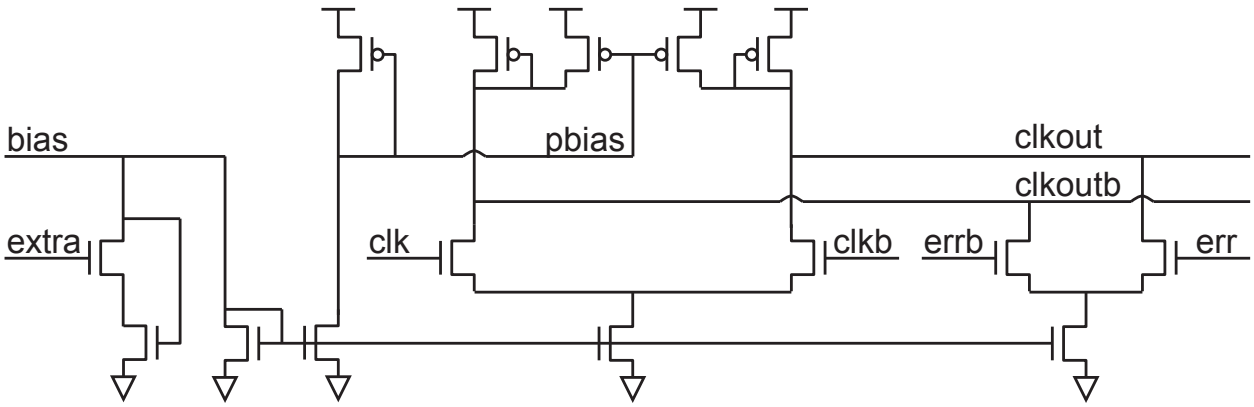


Figure 4.6.3: Clock duty cycle correction amplifier.

Cache Array	Size	ECC/ Parity	Phase/ Cycle Access	Voltage Domain	Ramcell Sleep	Redun-dancy
L3	6MB/ Core	ECC	Cycle	Vcache	Yes	Yes
L3 Tag	250KB/ Core	ECC	Phase	Vcache	No	Yes
L2 Instr	512KB/ Core	Parity	Cycle	Vcache	No	Yes
L2 Instr Tag	20KB/ Core	Parity	Phase	Vcache	No	No
L2 Data	256KB/ Core	ECC	Phase	Vcache	No	Yes
L2 Data Tag & L1	Various	Various	Various	Vcpu	No	No
Directory Cache	1.9MB/ Die	Parity	Cycle	Vcache	No	Yes

Changes from Montecito in Grey

Figure 4.6.4: Cache feature summary.

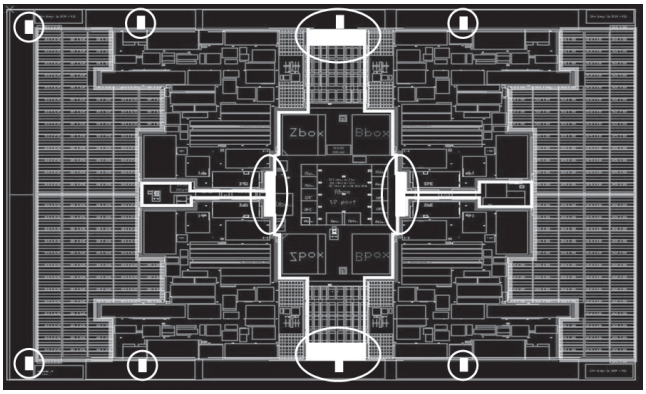


Figure 4.6.5: Synchronizer locations at clock region interfaces.

Continued on Page 598

Attachment - Page A323

Please click on paper title to view a Visual Supplement.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 18:55:17 UTC from IEEE Xplore. Restrictions apply.

Please click on paper title to view Visual Supplement.

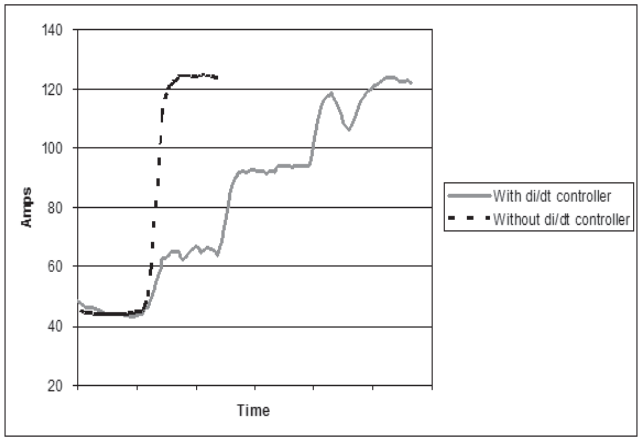


Figure 4.6.6: Simulation results of di/dt controller.

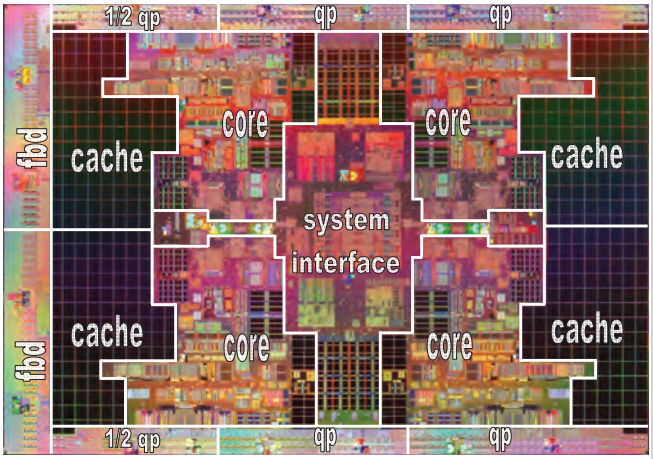


Figure 4.6.7: Die micrograph.

 978-1-4244-2011-7/08/$25.00 ©2008 IEEE

Please click on paper title to view a Visual Supplement.
Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on November 18,2022 at 18:55:17 UTC from IEEE Xplore. Restrictions apply.

ATTACHMENT 12C

Please click on paper title to view Visual Supplement.

4.6 A 65nm 2-Billion-Transistor Quad-Core Itanium® Processor

Blaine Stackhouse¹, Brian
Paul Gronowski², Chris Ly
¹Intel, Fort Collins, CO, ²Intel

The next generation in
duced. The processor ha
with a system interfac
process. The 21.5x32.5m
(Fig. 4.6.1). The silicon
2.0GHz and the system
design power (TDP) of 1

Improvements over pre
two additional cores on
enhancements for low-
mized for improved V_{CC}
to the chip is the inclus
gration of a system int
ments, the implementa
quency management sy
controller.

This is the first Itani
QuickPath interconnect
implement a different n
ferent frequencies. This
half-width interconnects
processor-to-I/O comm
96GB/s. The chip also i
capable of accessing m
34GB/s. Together, these
memory bandwidth and
Itanium processors.

Each interconnect is de
circuits and static SER-
trol logic (Fig. 4.6.2). These circuits operate across multiple volt-
age and frequency domains and the I/O signals are double-
pumped at a maximum frequency 4.8GT/s using low-jitter clock
elements. Each PLL and DLL in an interconnect implements a
duty-cycle correction system on its output using a differential
amplifier comparator (Fig. 4.6.3). The physical layer consumes
123mm², contains 26 million transistors, and contains 237K SER-
tolerant scan latches.

The caches are redesigned to improve performance relative to
previous Itanium processors (Fig 4.6.4). All cache memory cells
are placed on a separate high-voltage supply to improve V_{CCMIN}
and eliminate sensitivity to erratic bits. Low-latency voltage-con-

Because enterprise servers require highly reliable systems, all
against soft errors.
medium-sized regis-
M cells and parity,
ER-hardened RAM
ops in the system

s and I/O physical
d routing architec-
6.5). Synchronizer
when data is sent
cores, QuickPath
flow control is then
agic. Each 16-entry
ransfer data based
ncy mismatch and
unters align to the
lockstep.

h is FIFO for each
near its receiving
ut, while writing in
the input data. To
m across the die,
along the routes to

e devices to main-
luding SRAM, 79%
terface transistors

uency management
voltage-frequency
120 architectural
resent the relative
t. A central control
decision approxi-
ncy pair is appro-
priate for that activity. The capacitance thresholds are stored,
along with voltage-frequency pairs, in a look-up table fused in
production. By keeping weights and switching thresholds identi-
cal among parts, the frequency decisions of every part are identi-
cal, and not subject to part-to-part and environmental variability.

Frequency changes are accomplished by two parallel PLLs that
are mux-selected. One drives the core clock while the other locks
to the next requested ratio. When the ratio change is executed,
the core clock is halted momentarily, the PLL mux select is
switched, and the core clock is restarted at the new ratio.

The design includes a tunable di/dt controller. The controller pre-
vents sudden changes in current by limiting how quickly the

Document properties

File name:	getPDF.jsp
File size:	1,054 KB
Title:	A 65nm 2-Billion-Transistor Quad-Core Itanium® P...
Author:	Blaine Stackhouse; Brian Cherkauer; Mike Gowan; ...
Subject:	2008 IEEE International Solid-State Circuits Confer...
Keywords:	-
Created:	2/24/08, 10:06:11 AM
Modified:	11/18/22, 12:55:17 PM
Application:	Gnostice PDFOne
PDF producer:	Acrobat Distiller 8.1.0 (Macintosh); modified using...
PDF version:	1.5
Page count:	3
Page size:	8.25 × 11.75 in (portrait)
Fast web view:	No

Close

ATTACHMENT 13A

WorldCat: A practical programming model for the multi-core era

OCLC	272306953	LDL Holdings - 361 other holdings							
Books	Rec Stat c	Entered 20081117		Replaced 20220718132306.9					
Type	a	ELvl		Srce	d	Audn		Ctrl		Lang	eng
BLvl	m	Form	o	Conf	1	Biog		MRec		Ctry	gw
		Cont	b	GPub		LitF	0	Indx	1		
Desc	a	Ills	a	Fest	0	DtSt	s	Dates	2008,		

007 c ‡b r ‡d m ‡e n
040 GW5XE ‡b eng ‡e pn ‡c GW5XE ‡d OSU ‡d CEF ‡d OCLCQ ‡d C$Q ‡d GZM ‡d PUL ‡d LEAUB ‡d OCLCO ‡d OCLCQ ‡d OCLCA ‡d OCLCF ‡d BEDGE ‡d OCLCO ‡d YDXCP ‡d IDEBK ‡d NUI ‡d OCLCQ ‡d OCLCO ‡d OCL ‡d OCLCO ‡d EBLCP ‡d OCLCQ ‡d UAB ‡d ESU ‡d OCLCQ ‡d BUF ‡d OCLCQ ‡d U3W ‡d ICG ‡d OCLCQ ‡d ERF ‡d WURST ‡d OCLCQ ‡d EUN ‡d OCLCQ ‡d OCLCO ‡d COM ‡d OCLCO ‡d OCLCQ
019 246740113 ‡a 488389706 ‡a 1112544702
020 9783540693031
020 3540693033
020 ‡z 9783540693024
020 ‡z 3540693025
024 7_ 10.1007/978-3-540-69303-1. ‡2 doi
050 _4 QA76.642 ‡b .I5893 2007eb
072 _7 QA ‡2 lcco
072 _7 UKN. ‡2 bicssc
072 _7 COM075000. ‡2 bisacsh
082 04 004 ‡2 22
090
049 LDLL
111 2_ IWOMP (Conference) ‡n (3rd : ‡d 2007 : ‡c Beijing, China)
245 12 A practical programming model for the multi-core era : ‡b 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007 : proceedings / ‡c Barbara Chapman [and others] (eds.).
246 30 IWOMP 2007
260 Berlin ; ‡a New York : ‡b Springer, ‡c ©2008.
300 1 online resource (xii, 206 pages) : ‡b illustrations (some color)
336 text ‡b txt ‡2 rdacontent
337 computer ‡b c ‡2 rdamedia
338 online resource ‡b cr ‡2 rdacarrier
347 text file
347 ‡b PDF
490 1_ Lecture notes in computer science, ‡x 0302-9743 ; ‡v 4935
490 1_ LNCS sublibrary: SL 1 - Theoretical computer science and general issues
504 Includes bibliographical references and index.
588 0_ Print version record.
505 0_ A Proposal for Task Parallelism in OpenMP -- Support for Fine Grained Dependent Tasks in OpenMP -- Performance Evaluation of a Multi-zone Application in Different OpenMP Approaches -- Transactional Memory and OpenMP -- OpenMP on Multicore Architectures -- Supporting OpenMP on Cell -- CMP Cache Architecture and the OpenMP Performance -- Exploiting Loop-Level Parallelism for SIMD Arrays Using OpenMP -- OpenMP Extensions for Irregular Parallel Applications on Clusters -- Optimization Strategies Using Hybrid MPI+OpenMP Parallelization for Large-Scale Data Visualization on Earth Simulator -- An Investigation on Testing of Parallelized Code with OpenMP -- Loading OpenMP to Cell: An Effective Compiler Framework for Heterogeneous Multi-core Chip -- OpenMP Implementation of Parallel Linear Solver for Reservoir Simulation -- Parallel Data Flow Analysis for OpenMP Programs -- Design and Implementation of OpenMPD: An OpenMP-Like Programming

Language for Distributed Memory Systems -- A New Memory Allocation Model for Parallel Search Space Data Structures with OpenMP -- Implementation of OpenMP Work-Sharing on the Cell Broadband Engine Architecture -- Toward an Automatic Code Layout Methodology -- An Efficient OpenMP Runtime System for Hierarchical Architectures -- Problems, Workarounds and Possible Solutions Implementing the Singleton Pattern with C++ and OpenMP -- Web Service Call Parallelization Using OpenMP -- Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution.

520 This book constitutes the thoroughly refereed post-workshop proceedings of the Third International Workshop on OpenMP, IWOMP 2007, held in Beijing, China, in June 2007. The 14 revised full papers and 8 revised short papers presented were carefully reviewed and selected from 28 submissions. The papers address all topics related to OpenMP, such as OpenMP performance analysis and modeling, OpenMP performance and correctness tools and proposed OpenMP extensions, as well as applications in various domains, e.g., scientific computation, video games, computer graphics, multimedia, information retrieval, optimization, text processing, data mining, finance, signal and image processing, and numerical solvers.

650 _0 Parallel programming (Computer science) ‡v Congresses.
650 _0 Application program interfaces (Computer software) ‡v Congresses.
650 _6 Programmation parallèle (Informatique) ‡0 (CaQQLa)201-0062860 ‡v Congrès. ‡0 (CaQQLa)201-0378219
650 _6 Interfaces de programmation d'applications ‡0 (CaQQLa)201-0316133 ‡v Congrès. ‡0 (CaQQLa)201-0378219
650 _7 Informatique. ‡2 eclas
650 _7 Application program interfaces (Computer software) ‡2 fast ‡0 (OCoLC)fst00811704
650 _7 Parallel programming (Computer science) ‡2 fast ‡0 (OCoLC)fst01052939
653 00 algoritmen
653 00 algorithms
653 00 computeranalyse
653 00 computer analysis
653 00 programmeren
653 00 programming
653 00 simulatiemodellen
653 00 simulation models
653 00 computerwetenschappen
653 00 computer sciences
653 00 computerwiskunde
653 00 computational mathematics
653 00 numerieke wiskunde
653 00 numerical mathematics
653 00 software engineering
653 10 Information and Communication Technology (General)
653 10 Informatie- en communicatietechnologie (algemeen)
655 _2 Congress ‡0 (DNLM)D016423
655 _4 Electronic books.
655 _7 Conference papers and proceedings. ‡2 fast ‡0 (OCoLC)fst01423772
655 _7 Conference papers and proceedings. ‡2 lcgft
655 _7 Actes de congrès. ‡2 rvmgf ‡0 (CaQQLa)RVMGF-000001049
700 1_ Chapman, Barbara, ‡d 1954-
776 08 ‡i Print version: ‡a IWOMP 2007 (2007 : Beijing, China). ‡t Practical programming model for the multi-core era. ‡d Berlin : Springer, 2008 ‡z 9783540693024 ‡w (OCoLC)232569716
830 _0 Lecture notes in computer science ; ‡v 4935.
830 _0 LNCS sublibrary. ‡n SL 1, ‡p Theoretical computer science and general issues.

856 40 ‡3 MyiLibrary ‡u http://www.myilibrary.com?id=184945
856 40 ‡3 ProQuest Ebook Central ‡u https://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=3068454
856 40 ‡3 SpringerLink ‡u https://doi.org/10.1007/978-3-540-69303-1
856 40 ‡3 SpringerLink ‡u http://springerlink.com/openurl.asp?genre=issue&issn=0302-9743&volume=4935
856 40 ‡3 SpringerLink ‡u https://link.springer.com/book/10.1007%2F978-3-540-69303-1
856 4_ ‡u https://discover.gcu.ac.uk/discovery/openurl?institution=44GLCU_INST&vid=44GLCU_INST:44GLCU_VU2&?u.ignore_date_coverage=true&rft.mms_id=991002485245903836 ‡p 5371347790003836 ‡x StGlGCU
938 ProQuest Ebook Central ‡b EBLB ‡n EBL3068454
938 ProQuest MyiLibrary Digital eBook Collection ‡b IDEB ‡n cis23927234
938 YBP Library Services ‡b YANK ‡n 3024426
029 1_ AU@ ‡b 000043648551
029 1_ AU@ ‡b 000048718831
029 1_ NZ1 ‡b 12550556
029 1_ NZ1 ‡b 13658240

006 fields for Computer Files **Computer File**
Audn **Form** o **File** d **GPub**

Action Status Delete Holdings _ Export _ Label _ Produce _ Update Holdings _ Validate _

ATTACHMENT 13B

Springer Link

Search Log in

Download book PDF



Conference proceedings | © 2008

A Practical Programming Model for the Multi-Core Era

International Workshop on OpenMP, IWOMP 2007 Beijing, China, June 3-7, 2007, Proceedings

Editors: Barbara Chapman, Weiming Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, Dongsheng Wang

Part of the book series: Lecture Notes in Computer Science (LNCS, volume 4935)

Part of the book sub series: Theoretical Computer Science and General Issues (LNTCS)

Conference series link(s): IWOMP: International Workshop on OpenMP

12k Accesses | **83** Citations | **3** Altmetric

Conference proceedings info: IWOMP 2007.

Sections

Table of contents
Other volumes
About this book
Keywords
Bibliographic Information

Table of contents (22 papers)

Search within book

← Previous Page 1 of 2 Next →

Front Matter PDF

A Proposal for Task Parallelism in OpenMP PDF
Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli et al.
Pages 1-12

Support for Fine Grained Dependent Tasks in OpenMP PDF
Oliver Sinnen, Jsun Pe, Alexei Vladimirovich Kozlov
Pages 13-24

Performance Evaluation of a Multi-zone Application in Different OpenMP Approaches PDF

Haoqiang Jin, Barbara Chapman, Lei Huang
Pages 25-36

Transactional Memory and OpenMP PDF
Miloš Milovanović, Roger Ferrer, Osman S. Unsal, Adrian Cristal, Xavier Martorell, Eduard Ayguadé et al.
Pages 37-53

OpenMP on Multicore Architectures PDF
Christian Terboven, Dieter an Mey, Samuel Sarholz
Pages 54-64

Supporting OpenMP on Cell PDF
Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, Tao Zhang
Pages 65-76

CMP Cache Architecture and the OpenMP Performance PDF
Jie Tao, Kim D. Hoàng, Wolfgang Karl
Pages 77-88

Exploiting Loop-Level Parallelism for SIMD Arrays Using OpenMP PDF
Con Bradley, Benedict R. Gaster
Pages 89-100

OpenMP Extensions for Irregular Parallel Applications on Clusters PDF
Jue Wang, Changjun Hu, Jilin Zhang, Jianjiang Li
Pages 101-111

Optimization Strategies Using Hybrid MPI+OpenMP Parallelization PDF

for Large-Scale Data Visualization on Earth Simulator

Li Chen, Issei Fujishiro
Pages 112–124

An Investigation on Testing of Parallelized Code with OpenMP PDF

Robert Barnhart, Christian Trefftz, Paul Jorgensen, Yonglei Tao
Pages 125–128

Loading OpenMP to Cell: An Effective Compiler Framework for Heterogeneous Multi-core Chip PDF

Haitao Wei, Junqing Yu
Pages 129–133

OpenMP Implementation of Parallel Linear Solver for Reservoir Simulation PDF

Changjun Hu, Jilin Zhang, Jue Wang, Jianjiang Li
Pages 134–137

Parallel Data Flow Analysis for OpenMP Programs PDF

Lei Huang, Girija Sethuraman, Barbara Chapman
Pages 138–142

Design and Implementation of OpenMPD: An OpenMP-Like Programming Language for Distributed Memory Systems PDF

Jinpil Lee, Mitsuhisa Sato, Taisuke Boku
Pages 143–147

A New Memory Allocation Model for Parallel Search Space Data Structures with OpenMP PDF

Christophe Jaillet, Michaël Krajecki
Pages 148-152

Implementation of OpenMP Work-Sharing on the Cell Broadband Engine Architecture PDF

Jun Sung Park, Jung-Gyu Park, Hyo-Jung Song
Pages 153-156

Toward an Automatic Code Layout Methodology PDF

Joseph B. Manzano, Ziang Hu, Yi Jiang, Ge Gan, Hyo-Jung Song, Jung-Gyu Park
Pages 157-160

An Efficient OpenMP Runtime System for Hierarchical Architectures PDF

Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, Pierre-André Wacrenier
Pages 161-172

← Previous Page 1 of 2 Next →

Back to top ↑

Other Volumes

1. A Practical Programming Model for the Multi-Core Era

Back to top ↑

About this book

The Third International Workshop on OpenMP, IWOMP 2007, was held at Beijing,China.This year'sworkshopcontinuedits traditionofbeingthe premier opportunity to learn more about OpenMP, to obtain practical experience and to interact with OpenMP users and developers. The workshop also served as a forum for presenting insights gained by practical experience, as well as research ideas and results related to OpenMP. A total of 28 submissions were received in response to a call for papers. Each submissionwasevaluatedbythreereviewersandadditionalreviewswerereceived for some papers. Based on the feedback received, 22 papers were accepted for inclusion in the proceedings. Of the 22 papers, 14 were accepted as full papers. We also accepted eight short papers, for each of which there was an opportunity to givea shortpresentationat the workshop,followed byposter demonstrations. Each paper was judged according to its originality, innovation, readability, and relevance to the expected audience. Due to the limited scope and time of the workshop and the high number of submissions received, only 50% of the total submissions were able to be included in the ?nal program. In addition to the contributed papers, the IWOMP 2007 program featured several keynote and banquet speakers: Trevor Mudge, Randy Brown, and Shah, Sanjiv. These speakers were selected due to their signi?cant contributions and reputation in the ? eld. A tutorial session and labs were also associated with IWOMP 2007.

Back to top ↑

Keywords

OpenMP **Performance** **Processing**

computer **computer graphics**

image processing **model** **modeling**

multi-core **multimedia** **optimization**

programming

algorithm analysis and problem complexity

Back to top ↑

Bibliographic Information

Book Title	Book Subtitle	Editors
A Practical Programming Model for the Multi-Core Era	International Workshop on OpenMP, IWOMP 2007 Beijing, China, June 3-7, 2007, Proceedings	Barbara Chapman, Weiming Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, Dongsheng Wang
Series Title	**DOI**	**Publisher**
Lecture Notes in Computer Science	https://doi.org/10.1007/978-3-540-69303-1	Springer Berlin, Heidelberg
eBook Packages	**Copyright Information**	**Softcover ISBN**

Computer Science, Computer Science (R0)

Springer-Verlag Berlin Heidelberg 2008

978-3-540-69302-4

eBook ISBN
978-3-540-69303-1

Series ISSN
0302-9743

Series E-ISSN
1611-3349

Edition Number
1

Number of Pages
XII, 207

Topics
Computer Communication Networks, Programming Techniques, Software Engineering, Algorithms, Mathematics of Computing, Computer Modelling

Back to top ↑

Not logged in - 76.84.212.16
Big Ten Academic Alliance Formerly (CIC) (3000133814) - University of Nebraska-Lincoln (3000134173)
SPRINGER NATURE

© 2022 Springer Nature Switzerland AG. Part of Springer Nature.

ATTACHMENT 13C

Exploiting Loop-Level Parallelism for SIMD Arrays Using *OpenMP*

Con Bradley and Benedict R. Gaster

ClearSpeed Technology Plc
3110 Great Western Court
Hunts Ground Road
Bristol BS34 8HP UK
{ceb,brg}@clearspeed.com

Abstract. Programming SIMD arrays in languages such as C or FORTRAN is difficult and although work on automatic parallelizing programs has achieved much, it is far from satisfactory. In particular, almost all 'fully' automatic parallelizing compilers place fundamental restrictions on the input language.

Alternatively *OpenMP* provides an approach to parallel programming that supports incremental improvements to applications that places restrictions in the context of preserving semantics of a parallelized section. *OpenMP* is limited to a thread based model that does not define a mapping for other parallel programming models.

In this paper, we describe an alternative approach to programming SIMD machines using an extended subset of *OpenMP*(*OpenMP SIMD*), allowing us to model naturally the programming of arbitrary sized SIMD arrays while retaining the semantic completeness of both *OpenMP* and the parent languages.

Using the CSX architecture we show how *OpenMP SIMD* can be implemented and discuss future extensions to both the language and SIMD architectures to better support explicit parallel programming.

1 Introduction

With the rise of both SIMD processors, such as ClearSpeed's SIMD Array Processor [1], and the current unstoppable trend to put more and more cores on to a single die, parallel programming is back with a bang! For anyone who has tried to write a reasonably large parallel program then this will be seen with at least a slight frown. Why? In a nutshell parallel programming is hard and a choice must be made as to which of the many parallel programming languages to use.

While there is no magic wand to wave that can easily resolve the problem of parallel programming, there has been a number of recent proposals for programming languages, including Brook [2], Cilk [3], and C^n [4], and a number of simple language extensions for parallel programming, including *OpenMP* [5] and X10 [6]. Other than Brook, which introduces a programming model based on Streams, C^n, *OpenMP*, and X10 are all based upon existing programming

B. Chapman et al. (Eds.): IWOMP 2007, LNCS 4935, pp. 89–100, 2008.
© Springer-Verlag Berlin Heidelberg 2008

languages C[1] [7] and Java [8], respectively, and as such provide parallel programming in the context of well understood and widely used programming paradigms. It is only *OpenMP* that has seen wide adoption in the marketplace and is the only major parallel programming extension to C that is supported by the major compiler vendors, including Intel (Intel C Compiler) and Microsoft (Visual Studio).

OpenMP as defined is intended as a light-weight approach to programming threads. In this paper, we describe an alternative interpretation of *OpenMP*, called *OpenMP SIMD*, that rather than mapping to the standard runtime parallelism of threads looks instead at how it can be deployed for programming SIMD machines. In particular, we show that the loop level parallelism described by *OpenMP* maps well to data-parallelism as an alternative to the runtime approach described traditionally by *OpenMP*.

While we believe this alternative semantics for *OpenMP* is valid for any SIMD architecture, including Intel's Streaming SIMD Extensions [9], we focus here on programming the CSX architecture [1]. The CSX architecture is a SIMD array with a control unit and a number of processing elements (PE), with each PE having a relatively small private storage and supports an asynchronous I/O mechanism to access larger shared memory.

To date the CSX architecture would have been programmed using the explicit parallel programming language C^n but such an approach has some draw backs:

- Any code must be completely ported to use explicit parallelism before it can be run; and
- Users have large amounts of C and *OpenMP* code that they do not want to re-write.

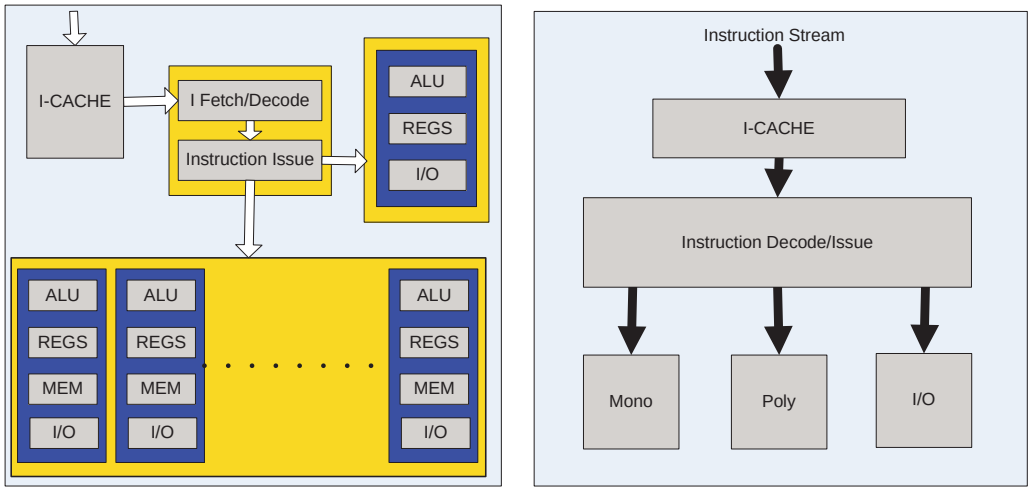
A mapping from *OpenMP* to either C^n or directly to the CSX architecture provides an alternative approach as *OpenMP* provides a piecemeal approach to parallelizing a given program and by definition supports any legacy C and *OpenMP* code.

The remaining sections of this paper are as follows; Section 2 introduces the CSX architecture, including how it is intended to be programmed; Section 3 is broken into a number of sub-sections describing the translation of *OpenMP SIMD* to the CSX architecture; Section 4 gives two examples of *OpenMP SIMD* in practice, highlighting both the translation and the small but necessary extensions to *OpenMP*; and finally, Section 5 concludes.

2 The CSX Architecture

The CSX family of co-processors [1] are based around a multi-threaded array processor (MTAP) core optimized for high performance, floating point intensive applications. The most recent product includes an MTAP processor, DDR2 DRAM interface, on-chip SRAM and high speed inter-processor I/O ports.

[1] *OpenMP* provides models for C++ and FORTRAN but we choose to focus on the C++ variant.



(a) CSX MTAP Processor (b) Instruction Execution

Fig. 1. CSX Architecture

Figure 1(a), visualizes the MTAP architecture showing a standard, RISC-like, control unit with instruction fetch, caches, and I/O mechanisms. A single instruction stream comprising of fairly standard, three operand instructions, is made up of mono (non-parallel) instructions executed by a single RISC-style processor and SIMD instructions (poly) executed, in lock-step, by a number of poly execution (PE) cores[2].

As noted above the processor executes a single instruction steam; each instruction is sent to one of the functional units, which may be in the mono, or poly execution units or one of the I/O controllers. Figure 1(b) shows this visually.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to 'variables' which have an instance on every PE core and are operated on in parallel: we call this poly data. Intuitively, this can be thought of as a data vector which is distributed across the PE core array.

Variables which only need to have a single instance (e.g. loop control variables) are known as mono variables; they behave exactly like normal variables on a sequential processor and can be 'broadcast' to all PEs if required in the poly domain.

Code for the mono and poly execution units is handled similarly, except the handling of conditional code execution. The mono unit uses conditional jumps to branch around code like a standard RISC architecture. This affects both mono and poly operations. The poly unit uses an enable register to control execution of each PE. If one or more of the bits of that PE enable register is zero, then the PE core is disabled and most instructions it receives will perform as though a *NOP* has been issued. The enable register is a stack, and a new bit, specifying the result of a test, can be pushed onto the top of the stack allowing nested predicated execution. The bit can later be popped from the top of the stack to

[2] The current CSX600 processor has 96 PEs.

remove the effect of the condition. This makes handling nested conditions and loops very efficient.

In order to provide fast access to the data being processed, each PE core has its own local memory and register file. Each PE can directly access only its own storage and additionally can communicate with its left and right neighbors via what is known as the swazzle path, forming a network of PEs. Data is transferred between PE (poly) memory and registers via load/store instructions while a family of instructions is provided for swazzle communication.

In the following mapping of *OpenMP* to the CSX architecture it is important to note that mono memory (i.e. card memory) and local PE (poly) memory are allocated separate address spaces which require 32 and 16 bit pointers, respectively.

3 *OpenMP SIMD*

As discussed in the introduction we are looking at using *OpenMP* to exploit loop level parallelism and as such the translation described below supports only one parallel construct, namely, the work-sharing loop. For this construct, we retain the fork-join 'threading' model. The master thread runs on the mono processor and is responsible for creating the 'thread' gang to execute the iterations of the work-sharing loop. Where the translation differs from the multi-core threading model traditionally associated with *OpenMP* is that each 'thread' is modeled as a PE element. At the end of the loop there is an implicit barrier. *OpenMP* expects the master thread to participate in the loop calculations but in our model this is not the case due to the asymmetry between the PE and mono processor architecture. We assume that all housekeeping operations are carried out on the mono processor.

3.1 Work-Sharing Constructs

The *OpenMP SIMD* translation actually provides two alternative mappings descrined in the following to sections.

Poly Loop Translation. *OpenMP* states that the index variable of the loop to be parallelized becomes private for the duration and thus a straightforward translation of the *OpenMP* loop pragma to *OpenMP SIMD* is to make the control variable poly and to increment by the number of PEs each time around the loop. Such a translation has the effect of unrolling the loop by the number of PEs. Mapping the test expression of the loop to use the poly control variable and increment with the defined increment plus the number of PEs in the final clause of the loop definition, provides all that is needed for the loop control definition. All that is remaining is to translate the body of the loop. The mapping is defined as follows:

```
#pragma omp parallel for clause clause ...
for (i = e1; i op e2; i = i op_s incr) { stmts; }
```

is translated to[3]:

```
{ poly int i_p = e_1;
  for (i_p = i_p op_s get_penum() * incr; i_p op e_2; s_1p )
    { stmts_p; }
}
```

Here *OpenMP* requires that *op* is one of $<$, $>$, $\leq$, or $\geq$ and op_s is $+$ or $-$. With this s_{1_p} becomes[4] $i_p = i_p\ ops\ (incr\ *\ \_\_NUM\_PES\_\_)$.

Finally, we must provide a translation for $stmts_p$. Firstly, all automatics defined locally to the scope of the loop become poly, i.e. privates in *OpenMP*, and any reference to the index variable i is replaced with i_p. Secondly, the translation must replace all accesses to arrays indexed by newly introduced poly expressions with poly temporaries, and insert calls to runtime routines that copy data between memory spaces, accounting for the strided access implied by parallelizing by the number of PEs. For example, consider the following loop statement which uses i to index into an array of mono doubles:

```
double tmp = V[i];
```

In the first part of the translation this becomes:

```
poly double tmp_p = V[i_p];
```

Assuming that V is defined globally and thus in *OpenMP* terminology shared, then this expression becomes the list of statements[5]:

```
poly double tmp_p;
double * poly V_p = ((double * poly) V) + i_p;
memcpym2p(&tmp_p, V_p, sizeof(double));
```

where *memcpym2p* is one of a family of memory transfer routines forming part of the *OpenMP SIMD* runtime, with this routine in particular performing data transfers from mono memory into local PE memory.

Mono Loop Translation. The poly loop translation given above is both obvious and straightforward but in practice may not be the most effective approach to implementing the loop on a SIMD style architecture.

To illustrate this consider the simple poly loop[6]:

```
for (i_p = 0; i_p < bound; i_p++) { ... }
```

As all control flow is handled in the mono processor a poly loop must be implemented by evaluating the conditional $i_p < bound$ such that all, some, or none of the PEs are disabled. Then a mono instruction is issued to determine if all of the PEs are disabled and if so then exit the loop, otherwise continue with

[3] The *OpenMP SIMD* runtime function poly int *get_penum*(*void*) returns the unique number assigned to each PE, given in the sequence 0..95.

[4] *__NUM_PES__* is defined statically as the number of PEs for a given implementation.

[5] The type expression *double *poly* should be read as a poly pointer to a mono double.

[6] Assuming *bound* to be mono.

the next iteration. The overhead of checking wheather the PEs are disabled or not is expensive and is often not necessary, as a mono loop can be used instead.

To explain the mono loop translation we consider again the loop from the previous section:

```
#pragma omp parallel for clause clause ...
for (i = e1; i op e2; i = i op incr) { stmts; }
```

which can be translated into the mono loop:

```
for (i = e1; i op (e2 − (e2%__NUM_PES__));
     i = i op (incr ∗ __NUM_PES__))
    { stmts_p }
```

Finally a translation for $stmts_p$ is required, which is the same as in the previous section except in the case when the control variable i is referenced. In this case the translation needs to account for the fact that $OpenMP$, and thus the translation, requires that for the duration of the loop i is marshaled as a private and to account for the unrolling factor the PE number must be added into any expression referencing i. Again considering the loop defined statement:

```
double tmp = V[i];
```

the first part of the translation gives:

```
poly double tmp_p = V[i];
```

As i must be treated as a private, accounting for the stride due to loop unrolling, the final translated code becomes:

```
poly double tmp_p;
double * poly V_p = ((double * poly) V) +
                                    (i + get_penum() ∗ incr);
memcpym2p(&tmp_p, V_p, sizeof(double));
```

If one is not careful it is easy to think at this point that the translation for mono loops provided above is enough, but we must consider the case when the number of PEs is not a multiple of the number of PEs. To see that there may in fact be a problem, consider again the translated test condition for the loop $i\ op\ (e_4 - (e_4\%\_\_NUM\_PES\_\_))$.

The expression $(e4 - (e_4\%\_\_NUM\_PES\_\_))$ always results in an expression that is a multiple of the number of PEs. It is straightforward to see that any remaining number of iterations will not be performed by the loop, and as is the case with optimizations like software-pipelining [10] the mono loop translation requires some additional post loop code to account for these missing iterations. This is easily achieved by augmenting the translation with the post loop code:

```
if (get_penum() < (e4%__NUM_PES__)) { stmts_p; }
```

Where $stmts_p$ is the same set of statements from the translated loop body and the conditional simply disables any PEs not required for the remaining calculation. An important consideration is to note that as the conditional is

poly then $stmts_p$ will always be executed. If the condition $get\_penum() < e_4\%\_\_NUM\_PES\_\_$ evaluates to false this could be expensive depending on the size of the sequence $stmts_p$. This overhead can be easily avoided by adding in an additional mono conditional of the form:

```
if (e4%__NUM_PES__)
```

3.2 Data Environment

OpenMP defines a set of constructs to describe how data is shared in parallel regions and we must capture how these are mapped into *OpenMP SIMD*.

OpenMP defines storage as being either private to a thread or shared, i.e. globally shared for all threads. The obvious translation into *OpenMP SIMD* is to map private variables to PE memory (poly) and shared to mono memory.

The remainder of this section considers the consequences of this simple translation and provides some extensions to *OpenMP* when the mapping does not quite capture the required semantics.

Sharing Attribute Rules. *OpenMP* defines a set of rules for describing sharing attributes of variables within parallel constructs. These are variables without explicit sharing attributes, i.e. the default rules, and we need to confirm their semantics for *OpenMP SIMD*. These rules apply in two cases:

1. Variables explicitly declared within the work sharing loop. For our translation the default rules work well and do not need modification.
2. Variables within called functions. *OpenMP* refers to these as being within a region but not a construct, meaning that syntactically a variable does not appear in a parallel construct but at runtime the function is called from a parallel region. The rules for data sharing are as in the explicit case but with one major addition: Formal arguments passed by reference inherit the data sharing attributes of the actual parameters.

This implies that at compile time, it is impossible for a compiler to know if a function will be called from a parallel region and thus what the data sharing attributes of its formal parameters will be. In order for the translation to *OpenMP SIMD* to work efficiently the user is required to provide this information. This leads to the introduction of one of three extensions to *OpenMP* as it stands:

```
#pragma omp private_function <name>
```

This $private\_function$ clause marks the function $< name >$ as being called from a parallel region and hence the default data sharing rules are to be applied. The default is that all functions are serial, i.e. mono, but this can be overridden via a default clause. In a typical program, parallel functions can be grouped thus reducing the amount of explicit clauses needed.

Unlike standard *OpenMP* the formal arguments to a function for the CSX architecture can be either mono or poly but not both. This leads to the observation

that data-sharing attributes of formal parameters need to be specified at compile time as they affect code generation. This is captured in *OpenMP SIMD* by introducing a second extension to *OpenMP*:

```
#pragma omp param <name> (list)
```

Where list is defined as a comma separated list, one entry per formal parameter to $< name >$; each entry can be of the form:

- private - formal is private (i.e. poly)
- shared - formal is shared (i.e mono)

Variable argument parameters and function pointers are not supported.

The final extension needed for the translation is analogous to that for a function's formal arguments but applies to the return value as it can be either mono or poly but not both:

```
#pragma omp private_return <name>
```

This *private_return* clause marks the function $< name >$ as having a private return value, i.e. poly. The default is that functions return a shared value, i.e. mono.

Any function can have any number of sharing directives applied to describe all possible calling conditions. This defines a set of 'signatures' describing the set of possible calling conditions. Given these extra annotations a compiler might choose to use a C++ [11] style name mangling scheme at function definition time and at function call sites to generate the required function signature. For example, a simple void function, void foo(void), with no parameters which can be called from both a parallel and non-parallel region could have two instances, $\_POLY\_foo$ and $\_MONO\_foo$. In a serial context, the mono version is called, and in a poly context the poly version is called. The linker will then select the appropriate versions. This scheme is also applied to a function's formal arguments to generate C++ style mangled names.

It is important to note that for this system to be sound, it must at all times be known if a piece of code is in a parallel region or not. This is where *OpenMP SIMD* differs slightly from *OpenMP* as it requires this to be enforced by the user through application of the above extensions.

Data-Sharing Attribute Clauses. *OpenMP* assumes that multiple threads execute within the same shared address space and thus inter-thread communication is straightforward. Data is sent to other threads by assigning to shared variables. *OpenMP* also provides support for a variable to be designated as private to each thread rather than shared, causing each thread to get a private copy of a particular variable for the duration of the parallel construct.

This notion of shared and private fits well within our SIMD model of OpenMP, producing the mapping of *private* to *poly* and *shared* to *mono*. The *firstprivate* and *lastprivate* clauses are also supported with extra runtime functions provided to perform the initialization and finalization.

The reduction clause describes the application of a reduction operator to be applied to a list of private variables. The reduction operator comes from the base

language and is applied to each (private) copy of the variable at the end of the parallel construct. This construct is supported by introducing a private copy of the reduction variable, initialized to the particular reduction operator's identity, at the start of the parallel construct. The reduction is then preformed across all PEs. The swazzle path is used to reduce into either the top or bottom PE, where the data can be quickly transferred into mono space and assigned to the original reduction variable. For example, consider the following reduction clause and loop:

```
#pragma omp parallel for reduction(+:acc)
for (...) { ... ;  acc += ...;  }
```

which becomes[7]:

```
poly double acc_p = 0.0;
for (...) { ... ;  acc_p += ... ; }
acc += reduce_sum(acc_p);
```

4 An Example

As an example of *OpenMP SIMD* we consider performing a Monte-Carlo simulation to calculate European option prices [12]. The intention here is to show the extensions in practice. Due to space restrictions we choose not to show the translated code but rather concentrate our discussion of the extensions themselves.

Monte-Carlo simulation problems include derivative pricing, portfolio risk management and Markov chain simulation. In financial modeling, the problem of finding the arbitrage-free value of a particular derivative can be expressed as the calculation of a given integral. As the number of dimensions grows, today's financial mathematicians use Monte-Carlo simulation methods to solve problems in a tractable manner.

Monte-Carlo simulations can be easily parallelized, since tasks are largely independent and can be partitioned among different processors, and the need for communication is limited. Due to the simple but highly computational nature of Monte-Carlo it is clear that any performance increase in their implementation can lead to substantial speed improvements for the entire system and significant advances in time to result. The following function implements Monte-Carlo option pricing in *OpenMP SIMD*:

```
void MCOption(double S, double E, double r,
              double  sigma, double  T,
          int nSimulations,
          double *Pmean, double *width) {
```

[7] The function *double reduce_sum*(*poly double*) is one of a family of reduction operators provided as part of the *OpenMP SIMD* runtime, with this function performing a double precision sum across the PEs placing the scalar result into mono.

```
  double sumx, sumx2, tmp1, tmp2, stdev;
  int i;

  sumx=0; sumx2=0;
  tmp1=(r-0.5*sigma*sigma)*T;  tmp2=sigma*sqrt(T);

  #pragma omp for reduction(+:sumx,+:sumx2)
  for ( i = 0; i< nSimulations; i++)
  {
    double Svals, Pvals;
    Svals = S* exp (tmp1 +tmp2*gaussrand());
    Pvals = Max( Svals - E, 0);
    sumx  +=  Pvals;
    sumx2 += (Pvals*Pvals);
  }
  stdev = sqrt(((double)nSimulations*sumx2-sumx*sumx)/
     ((double)nSimulations*(double)(nSimulations-1)));

  *width = exp(-r *T) * 1.96 * stdev /
                        sqrt((double)nSimulations);
  *Pmean = (exp(-r *T) * (sumx / nSimulations));
}
```

The above code calls the function *gaussrand* from within a parallel region and as a consequence the programmer must tell the compiler, through use of the *OpenMP SIMD* extensions, to produce an instance suitable for calling from within the parallel construct:

```
#pragma omp private_function gaussrand private(V1,V2,S)
#pragma omp private_return gaussrand
double gaussrand(void) {
  static double V1, V2, S; static int phase = 0;
  double X;
  ...
  return X;
}
```

Here the application of clause *private_function* expresses the requirement that *gaussrand* is intended to be called from within a parallel region or loop and *private_return* implies that the function will produce a private, i.e. poly, result. This highlights an inportant difference between the threaded approach of *OpenMP*'s out-of-step execution when compared with *OpenMP SIMD*'s lock-step execution.

Using our prototype compiler early benchmarking of the Monte-Carlo simulation for European Option pricing has shown excellent results for the OpenMP SIMD implementation running on ClearSpeed's Advance Board. These benchmarks were performed on a 2.33GHz dual-core Intel Xeon (Woodcrest, HP DL

Table 1. Monte-Carlo - ClearSpeed Board vs. host-only performance comparison

GNU C Compiler with Math Kernel Library	Intel C Compiler with Math Kernel Library	ClearSpeed Advance Board	ClearSpeed Advance Board plus Intel Math Kernel Library
36.6M	40.5M	206.5M	240M

380 G5 system) and the results are shown in Table 1. The results show a 5.15x performance improvement, for 100,000 Monte-Carlo simulations performed using a single ClearSpeed Advance board when compared to simply using an Intel Xeon processor, the Intel C compiler and Math Kernel Library (MKL). When compared against the GNU C compiler and the Intel MKL the performance improvement increases to 5.7x. However, combining both the general purpose Intel processor and MKL with the Advance board achieves the best result at 240M samples per second and a performance factor of up to 6x compared to using just the host processor, Intel C compiler and MKL.

5 Conclusion

In this paper, we have introduced the programming extensions $OpenMP\ SIMD$ for programming arbitrary-sized SIMD arrays. This language is understood by providing an alternative semantics for $OpenMP$. While there have been a number of extensions to existing programming lanuages for accessing the features of SIMD machines, including C^n and C^* [4,13], to our knowledge we are the first to propose using $OpenMP$ to program SIMD machines.

We have developed a set of prototype tools targeting the CSX array architecture and are evaluating using $OpenMP\ SIMD$ as an alternative to the explicit parallel programming language C^n. Early results indicate that it provides at least the same functionality and ability for performance.

$OpenMP$ has a relaxed-consistency shared-memory model that allows threads to have their own view of global data which only required to be synched up at specific points including the explicit flush operation. Any compiler implementing OpenMP SIMD has the ability to promote shared variables to registers and even cache them in PE memory if needed. The flexibility this model allows provides a compiler with the chance to optimize through data migration between mono memory, PE memory and host memory.

In conclusion we believe that there is an important and fascinating concept at the heart of $OpenMP\ SIMD$. Essentially we are translating runtime parallelism into data parallelism. The model can only work if this fundamental difference in the model is not observable. We believe that this is sound as long as we can only observe a program's behavior by its effect on variables. Since we believe we have correctly defined the data parallelism then there should be no observable difference in behavior.

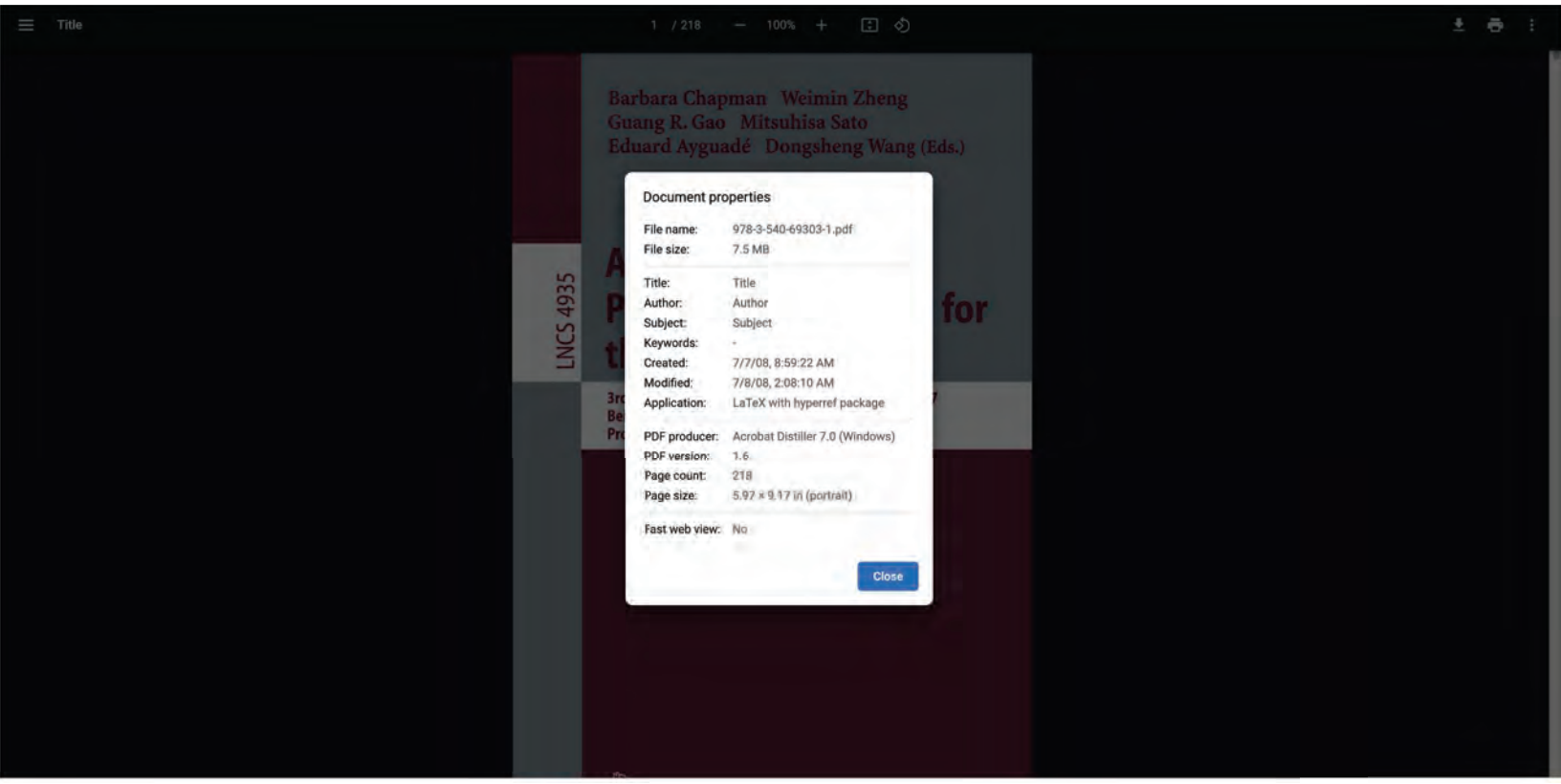
Acknowledgments

We would like to thank Ray McConnell who originally introduced us to OpenMP and David Lacey for his useful feedback on the translation.

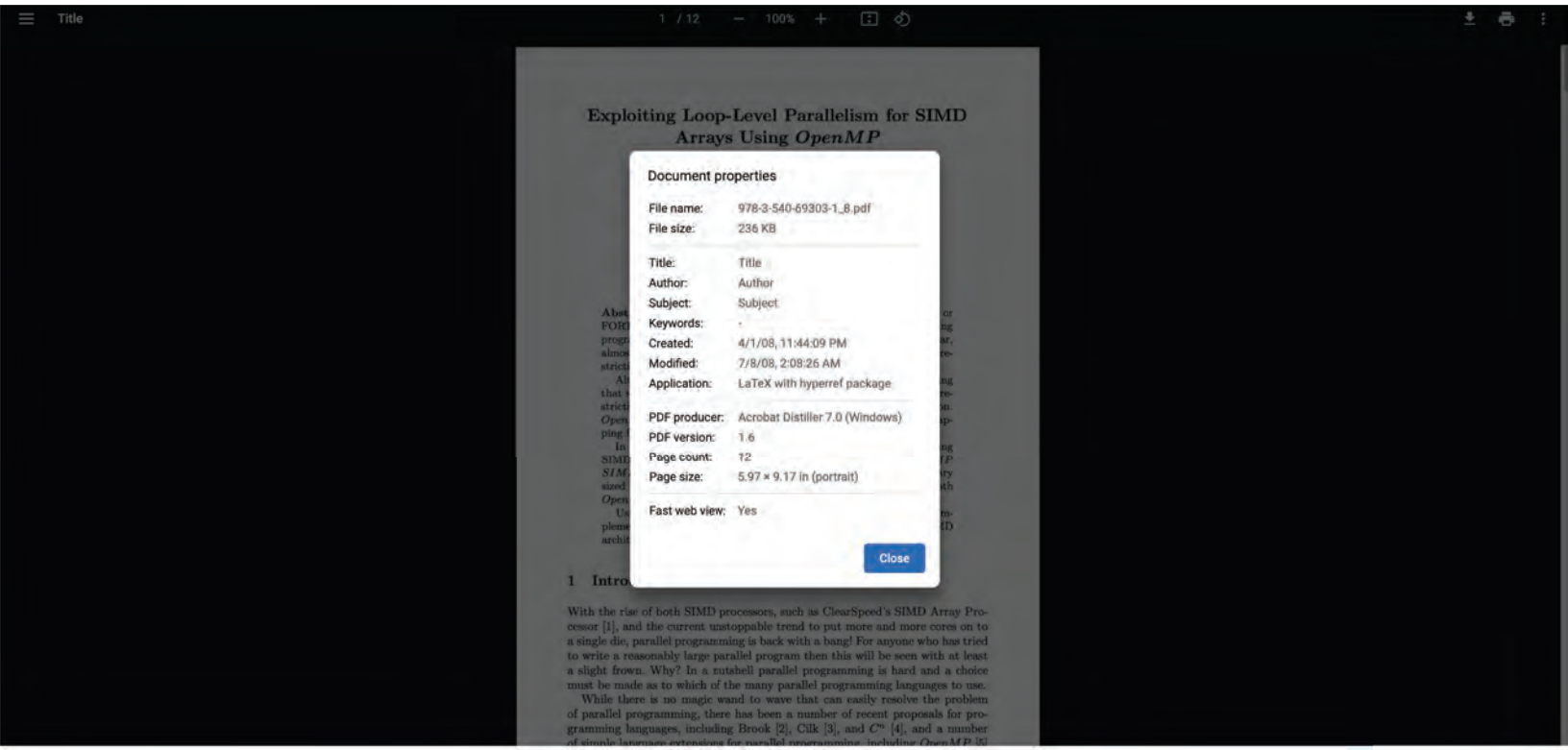
References

1. ClearSpeed Technology Plc: White paper: CSX Processor Architecture (2004)
2. Buck, I.: Brook: A Streaming Programming Language. Stanford University (2001)
3. Frigo, M.: Portable High-Performance Programs. PhD thesis, MIT Department of Electrical Engineering and Computer Science (1999)
4. Gaster, B.R., Hickey, N., Stuttard, D.: The C^n Language Specification. ClearSpeed Technology Plc. (2006)
5. OpenMP Architecture Review Board: OpenMP Application Program Interface (2005)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005. Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 519–538. ACM Press, New York (2005)
7. International Standards Organization: ISO/IEC 98899 - Programming Language C. (1999)
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Specification, 3rd edn. Addison-Wesley, Reading (2005)
9. Ramanathan, R.: Intel white paper: Extending the world's most popular processor architecture. Intel Corp. (2006)
10. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
11. Stroustrup, B.: The C++ Programming Language, 2nd edn. Addison-Wesley, Reading (2000)
12. Wilmott, P.: Quantitative Finance, 2nd edn. John Wiley & Sons Ltd., Chichester (2000)
13. Rose, J., Steele, G.: C^*: An extended C language for data parallel programming. In: Internationl Conference on Supercomputing, vol. 2, pp. 72–76 (1987)

ATTACHMENT 13D


Title
1 / 218
100%
Barbara Chapman Weimin Zheng
Guang R. Gao Mitsuhisa Sato
Eduard Ayguadé Dongsheng Wang (Eds.)
LNCS 4935
for
Document properties
File name: 978-3-540-69303-1.pdf
File size: 7.5 MB
Title: Title
Author: Author
Subject: Subject
Keywords: -
Created: 7/7/08, 8:59:22 AM
Modified: 7/8/08, 2:08:10 AM
Application: LaTeX with hyperref package
PDF producer: Acrobat Distiller 7.0 (Windows)
PDF version: 1.6
Page count: 218
Page size: 5.97 × 9.17 in (portrait)
Fast web view: No
Close

ATTACHMENT 13E


Title
1 / 12
100%
Exploiting Loop-Level Parallelism for SIMD Arrays Using OpenMP
Document properties
File name: 978-3-540-69303-1_8.pdf
File size: 236 KB
Title: Title
Author: Author
Subject: Subject
Keywords: -
Created: 4/1/08, 11:44:09 PM
Modified: 7/8/08, 2:08:26 AM
Application: LaTeX with hyperref package
PDF producer: Acrobat Distiller 7.0 (Windows)
PDF version: 1.6
Page count: 12
Page size: 5.97 × 9.17 in (portrait)
Fast web view: Yes
Close
1 Intro
With the rise of both SIMD processors, such as ClearSpeed's SIMD Array Processor [1], and the current unstoppable trend to put more and more cores on to a single die, parallel programming is back with a bang! For anyone who has tried to write a reasonably large parallel program then this will be seen with at least a slight frown. Why? In a nutshell parallel programming is hard and a choice must be made as to which of the many parallel programming languages to use.
While there is no magic wand to wave that can easily resolve the problem of parallel programming, there has been a number of recent proposals for programming languages, including Brook [2], Cilk [3], and C* [4], and a number

ATTACHMENT 13F

All locations : No holdings in LDL - 40 other holdings

A Practical Programming Model for the Multi-Core Era : 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007 Proceedings / edited by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Barbara Chapman, Weiming Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, Dongsheng Wang. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2008. 1 Online-Ressource : v.: digital OCLC:315988955

Holdings Information

STATE	LOCATIONS
NC	NRC
CH	h9g
DE	defhd defro defuh dehmf dehol dehtw deibt dekbl delnd demab demia demzb deovg depot derhe dethk detul detwd deubi deubk deubo deubr deude deuld dewup dezhb du# g4w gwn6z gwpi9 gwsub h6u h9h UB# UF# uk@ ukpad
TW	twniu

CLOSE WINDOW

ATTACHMENT 14A

IEEE.org IEEE *Xplore* IEEE SA IEEE Spectrum More Sites

Cart Create Personal
Account Sign In

Browse My Settings Help

Access provided by:
University of Nebraska - Lincoln

Sign Out

Access provided by:
University of Nebraska - Lincoln

Sign Out

All

ADVANCED SEARCH

Back to Results

Conferences > Proceedings Fifth IEEE Intern...

In the development and evaluation of specialized processors for computing high-order 2-D image moments in real-time

Publisher: IEEE

Cite This PDF

<< Results

N. Roma ; L. Sousa **All Authors**

24
Full
Text Views

Alerts

Manage Content Alerts

Add to Citation Alerts

Abstract
Authors
References
Keywords
Metrics
More Like This

Downl
PDF

Abstract: Image moments are used in image analysis for object modelling, matching and representation. The computation of high-order moments is a computational intensive task that c... **View more**

More Like This

Tried and true. IV. DSP selection for real-time systems
IEEE Instrumentation & Measurement Magazine
Published: 2002

A digital signal processing chip for iterative deconvolution restoration algorithms
[1991] Conference Record. IEEE Instrumentation and Measurement Technology Conference
Published: 1991

Show More

Metadata

Abstract:
Image moments are used in image analysis for object modelling, matching and representation. The computation of high-order moments is a computational intensive task that can not be implemented in real-time with nowadays general-purpose processors. This paper proposes a set of specialised processors for generating an image moment of an arbitrary order in real time, by adopting systolic processing techniques and floating-point arithmetic units. It proposes a modular and cost effective architecture for generating image moments, with a processing time not dependent on the order of the computed moments. The architecture was implemented using different devices, such as programmable digital processors, configurable hardware logic and integrated circuits, by using

Loading [MathJax]/extensions/MathZoom.js

a 0.7 /spl mu/m CMOS process. The several implementations have shown the effectiveness of the architecture, and the obtained results allow us to compare the different solutions in terms of speed, flexibility, cost and power consumption.

Published in: Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception

Date of Conference: 11-13 September 2000

Date Added to IEEE *Xplore*: 06 August 2002

Print ISBN:0-7695-0740-9

INSPEC Accession Number: 6742322

DOI: 10.1109/CAMP.2000.875975

Publisher: IEEE

Conference Location: Padua, Italy

Authors

References

Keywords

Metrics

IEEE Personal Account

CHANGE USERNAME/PASSWORD

Purchase Details

PAYMENT OPTIONS

VIEW PURCHASED DOCUMENTS

Profile Information

COMMUNICATIONS PREFERENCES

PROFESSION AND EDUCATION

TECHNICAL INTERESTS

Need Help?

US & CANADA: +1 800 678 4333

WORLDWIDE: +1 732 981 0060

CONTACT & SUPPORT

Follow

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | IEEE Ethics Reporting | Sitemap | IEEE Privacy Policy

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved.

IEEE Account

» Change Username/Password

» Update Address

Purchase Details

» Payment Options

» Order History

» View Purchased Documents

Profile Information

» Communications Preferences

» Profession and Education

» Technical Interests

Need Help?

» **US & Canada:** +1 800 678 4333

» **Worldwide:** +1 732 981 0060

» Contact & Support

About IEEE *Xplore* | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2022 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.

Loading [MathJax]/extensions/MathZoom.js

ATTACHMENT 14B

In the Development and Evaluation of Specialized Processors for Computing High-Order 2-D Image Moments in Real-Time

Nuno Roma
Nuno.Roma@inesc.pt

Leonel Sousa
las@inesc.pt

Instituto Superior Técnico / INESC-ID
Department of Electrical Engineering
Rua Alves Redol, No. 9 - 1000-029 Lisboa - PORTUGAL
Tel. +351 21 3100000 - Fax: +351 21 3145843

Abstract

Image moments are used in image analysis for object modelling, matching and representation. The computation of high-order moments is a computational intensive task that can not be implemented in real-time with nowadays general-purpose processors. This paper proposes a set of specialised processors for generating an image moment of an arbitrary order in real time, by adopting systolic processing techniques and floating-point arithmetic units. It proposes a modular and cost effective architecture for generating image moments, with a processing time not dependent on the order of the computed moments. The architecture was implemented using different devices, such as programmable digital processors, configurable hardware logic and integrated circuits, by using a 0.7 μm CMOS process. The several implementations have shown the effectiveness of the architecture, and the obtained results allow us to compare the different solutions in terms of speed, flexibility, cost and power consumption.

Keywords: High-order image moments, Pipeline architectures, Systolic processing, Digital Signal Processors

1. Introduction

Two-dimensional object representation and recognition is an important topic in the computer vision area. Moments of the intensity function of pixels are commonly used for representing an object or an image. The $p = m + n$ order moment $(\mathcal{M}_{m,n})$ of an array of pixels with values $f(x, y)$ and size NM is defined by equation 1 [1, 2].

$$\mathcal{M}_{m,n} = \sum_{x=1}^{N} \sum_{y=1}^{M} x^m y^n f(x, y) \qquad (1)$$

Low order moments are commonly used to extract features, such as area and center of gravity, and to find the location and orientation of objects in a image [1, 2, 3, 4]. In contrast, high order moments are used for pattern recognition and image representation [5, 6].

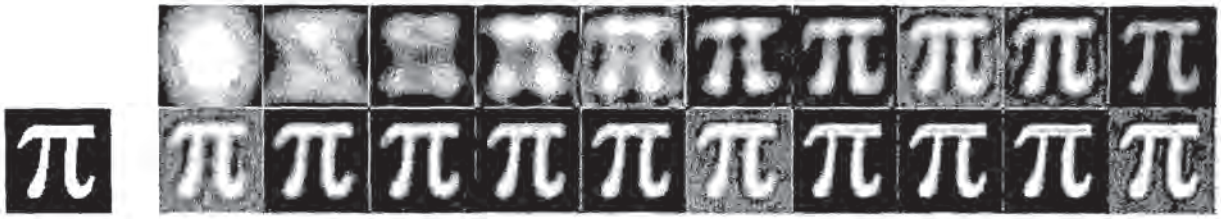
A direct computation of equation 1 requires NM additions and $(m + n)NM$ multiplications. In the last years, several algorithms and architectures have been proposed to improve the speed of moments computation. However, some of them are only valid for low-order moments and/or for binary images [7, 8, 9, 10]. Other proposals suggest architectures without practical interest due to its complexity [11]. Moreover, floating-point arithmetic has to be used to compute moments of different orders, given that the required dynamic range increases very rapidly with the moment's order. Some other proposed architectures [7] adopt wavefront processing techniques to adapt the throughput to the variable processing time exhibited by floating-point arithmetic units. However, although they eliminate the need to respect the "worst" processing period, they require additional circuits to exchange data between Processing Elements (PE), which represents a significant overhead in the hardware cost.

This paper proposes a new architecture for computing 2-D gray level image moments with any order in real-time, by adopting systolic processing techniques and floating point arithmetic. The proposed architecture is modular and scalable for higher order moments and requires a reduced amount of hardware. Moreover, the results presented in this paper demonstrate that the architecture is well suited to implement specialised processors for computing image moments based on Digital Signal Processors (DSP), Field Programmable Date Arrays (FPGA) and Application Specific Integrated Circuits (ASIC). By comparing the different implementations, it is possible to conclude that the real-time processing is always achieved. However, some advantages

0-7695-0740-9/00 $10.00 © 2000 IEEE

Attachment - Page A363

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.



(a) (b)

Figure 2. Representation and reconstruction of an image pattern using high order moments: (a) Original input image. (b) Reconstructed image with up to second-order moment through up to 40th-order moment, from top to bottom rows and from left to right.

of the specific characteristics of each of the considered processor implementations can be taken, in what concerns the operating speed, flexibility and power consumption.

This paper is organised as follows. Section 2 presents two main application examples of the use of image moments in image processing. An efficient *power core* for generating the x and y powers is proposed in section 3. In section 4 a new systolic architecture is proposed and in section 5 floating point arithmetic units are described. Section 6 presents and compares several implementations of a proposed processor based on different devices, such as: FPGAs (section 6.1), ASICs (section 6.2) and DSPs (section 6.3). Section 7 concludes the presentation.

2. 2D Image Moments

As it has been referred in the previous section, image moments are often used in computer vision to provide a set of image features for different applications, such as localization, pattern recognition and image representation [1, 2]. The zero through second order moments provide information about the area, center of mass and about the approximation of an object to an ellipse [3] (see figure 1). Moments with order lower than $p = 6$ can also be combined to form moment invariants to certain shape transformations, such as translation, rotation and scaling [4].

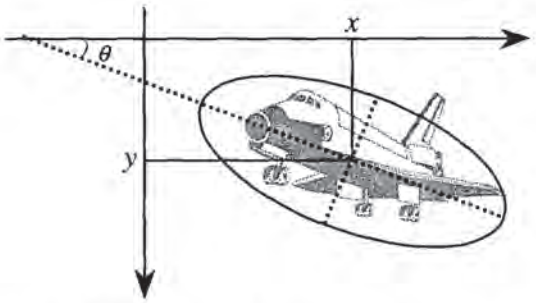


Figure 1. Definition of several localization parameters through the computation of the image ellipse.

Besides feature extraction, a significant research effort has recently been made to use high order image moments in pattern recognition and image representation applications [5, 6]. In fact, it has been shown that high order moments can provide a means of reconstructing images or patterns using a finite set of moments. This makes it possible to use fewer bytes to represent the reconstructed image than the required to represent the original image. However, since the usage of an infinite number of image moments is not feasible, this representation presents always an inherent error that is dependent on the maximum order of the set of used moments (see figure 2).

3. Power Core

To compute the values of the x^m and y^n powers with a traditional sequential architecture, m and n clock cycles are required, respectively. Such basic architecture leads to a great amount of computation time for high order moments. According to [7], a faster architecture can be obtained by applying the following decomposition:

$$m = \sum_{i=0}^{k-1} b_i^m 2^i \quad , \quad n = \sum_{i=0}^{k-1} b_i^n 2^i \qquad (2)$$

$$k = \lceil \max \{\log_2 (m+1), \log_2 (n+1)\} \rceil$$

Hence, the expressions for the powers x^m and y^n take the following form:

$$x^m = x^{b_0^m \cdot 2^0 + b_1^m \cdot 2^1 + \ldots + b_{k-1}^m \cdot 2^{k-1}} = \prod_{i=0}^{k-1} x^{b_i^m \cdot 2^i} \qquad (3)$$

$$y^n = y^{b_0^n \cdot 2^0 + b_1^n \cdot 2^1 + \ldots + b_{k-1}^n \cdot 2^{k-1}} = \prod_{i=0}^{k-1} y^{b_i^n \cdot 2^i} \qquad (4)$$

Attachment - Page A364

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

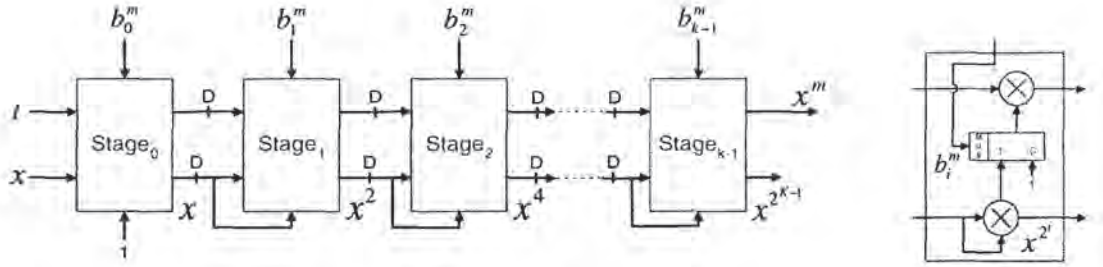


Figure 3. Power-Core module.

Thus, it is possible to conclude that, in order to compute the value of x^m, it is only necessary to perform $2k$ multiplications: k of them for computing the x^{2^i} terms and the remaining k to calculate the product of the $x^{b_i^m \cdot 2^i}$ terms. Figure 3 represents a structure with k stages to compute equation 3. If the input values were composed by the sequence $\{1, 2, 3, \ldots, N\}$, the sequence $\{1^m, 2^m, 3^m, \ldots, N^m\}$ would be obtained at the output. In this systolic architecture, the overall structure is operated in a synchronous way, with the D symbol representing a delay of one clock cycle. With this structure, it is possible to obtain one output value in each clock cycle, with a latency of k clock cycles.

4. Dedicated Pipeline Architecture for Computing an Image Moment

The computation of $\mathcal{M}_{m,n}$ given by equation 1 can be represented by a vector-matrix product followed by a vector dot product, as depicted in equation 5. In these equations, F represents the $N \times M$ image matrix, while X and Y represent the N and M sized vectors composed by the set of x^m and y^n powers, respectively. According to [7], the computation of this equation can be represented by the planar Dependence Graph (DG) presented in figure 4, corresponding to an array architecture composed by $N \times (M+1)$ PEs.

$$\begin{aligned} \mathcal{M}_{m,n} &= XFY = XG = HY \\ X &= \left[\, 1^m \;\; 2^m \;\; \cdots \;\; N^m \,\right] \\ Y &= \left[\, 1^n \;\; 2^n \;\; \cdots \;\; M^n \,\right] \\ F(x,y) &= \{f(x,y) : x = 1 \ldots N, y = 1 \ldots M\} \end{aligned} \tag{5}$$

For real images, with a great number of pixels, it is convenient to reduce the dimension of this array. This can be achieved by projecting the DG on a linear structure with a valid pair of *projection* and *schedule* vectors $(\vec{d}, \vec{s})$. The systolic structure presented in figure 5 is obtained by defining

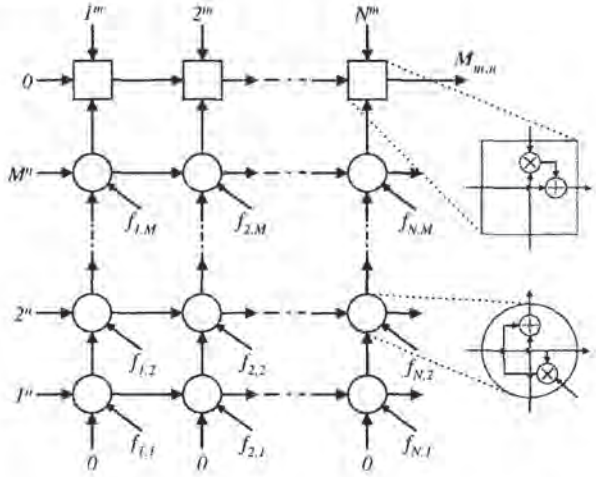


Figure 4. 2-D systolic architecture derivation: Dependence Graph (DG).

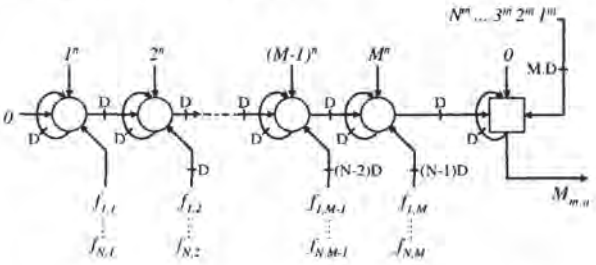


Figure 5. 2-D systolic architecture derivation: Linear Signal Flow Graph (SFG).

$\vec{d} = [1\ 0]^T$ and $\vec{s} = [1\ 1]^T$, while maintaining the same type of PEs.

Although the dimension of the processor has been reduced, this structure still requires a large amount of hardware, namely, for images with median and high resolutions (e.g. N=M=512). According to [7], this problem can

Attachment - Page A365

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

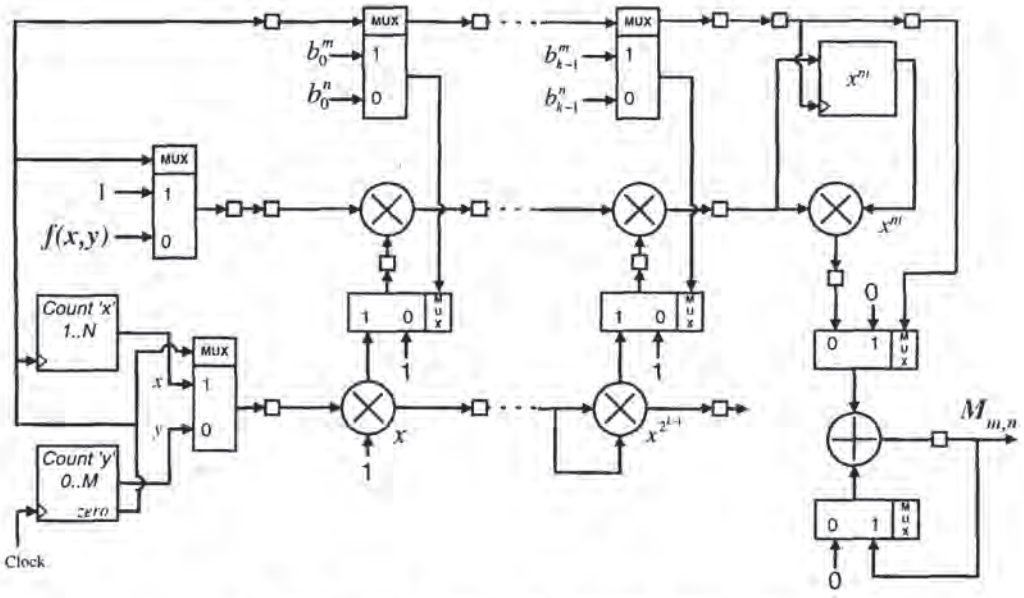


Figure 7. Block diagram of the serial processor.

be solved by regularly partitioning the Signal Flow Graph (SFG) into blocks, each consisting of a cluster of PEs [12]. Figure 6 presents the limit situation, where the set of all PEs represented by circles is mapped on a single PE, leading to a serial pipeline architecture. This configuration represents the most economical solution in what concerns hardware resources. In fact, only one multiplier is used in the PE represented by a circle and partial results are accumulated in the output PE, as depicted in figure 6.

A complete diagram of the described serial systolic processor for the computation of an image moment with any order is shown in figure 7, with the square symbol (□) representing a processing delay. In order to use a single power core to compute the powers of x^m and y^n, a register was used in the final stage of the processor to store the x^m value of the line being processed. Furthermore, in order to increase the processing frequency of the power-core, a pipeline register was introduced between the multiplier and the adder, thus reducing the overall critical path.

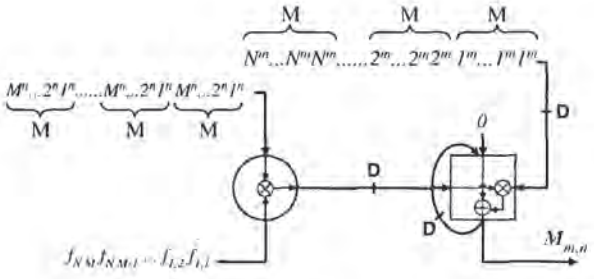


Figure 6. 2-D systolic architecture derivation: Serial pipeline structure.

The serial pipeline processor shown in figure 7 uses a total of $2k+1$ multipliers and 1 adder. The total processing time required to calculate the $\mathcal{M}_{m,n}$ moment of an $N \times M$ image is:

$$T = [N(M+1) + t_{pow} + 2] \times t_{op}, \tag{6}$$

where t_{pow} is the latency of the power-core circuit given by:

$$t_{pow} = 1 + \log_2(\lceil \max\{m+1, n+1\} \rceil) + 1 \tag{7}$$

and t_{op} is the maximum time required to perform a single floating-point multiply or addition operation. Hence, the total processing time is:

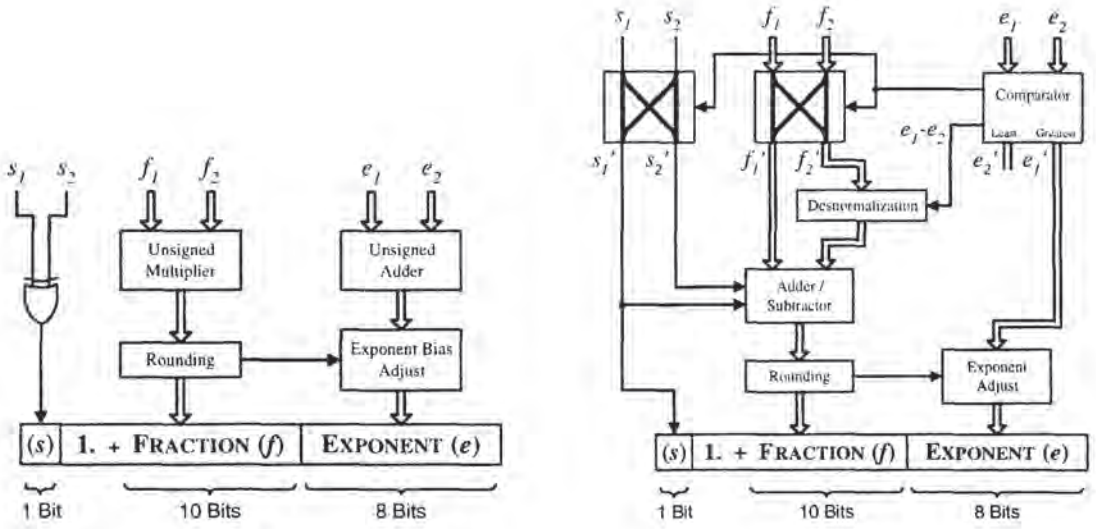
$$T = [N \times (M+1) + \log_2(\lceil \max\{m+1, n+1\} \rceil) + 4] \cdot t_{op} \tag{8}$$

5. Arithmetic Units

Due to the great number of arithmetic operations involved in the computation of the image moments, some of the most critical blocks of the proposed processor are the arithmetic units. In fact, its overall performance is remarkably dependent on the efficiency of these elements. As it was referred in the previous sections, floating-point arithmetic units are used to accommodate high-dynamic ranges required to compute high-order image moments with smaller word widths. Experimental results show that the

Attachment - Page A366

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.



(a) Multiplier **(b) Adder**

Figure 9. Floating-point arithmetic unit.

used floating-point precision, whose formats is presented in figure 8, seems to be enough (1 sign bit (s), 8 bits for the exponent (e) and 10 bits used for the fractional part (f) of the mantissa). In fact, in the present application, the sign bit could be eliminated, since image data is usually represented with unsigned values. However, it was decided to implement the arithmetic units with a signed representation, in order to extend its application to different sources of data.

The exponent is represented as a biased signed number, with a bias of 127. The mantissa is composed by a hidden bit, equal to 1, and by a fractional part, which represents a number smaller than 1. Hence, the floating-point number is represented by the value of $(-1)^s \times \left[1.f \times 2^{e-127}\right]$.

5.1. Floating-Point Multiplier

The floating-point multiplication is generally computed according to equation 9 (where $M_i = 1.f_i$ and $E_i = e_i - 127$):

$$\left[(-1)^{S_1} \times M_1 \times 2^{E_1}\right] \cdot \left[(-1)^{S_2} \times M_2 \times 2^{E_2}\right] = \quad (9)$$

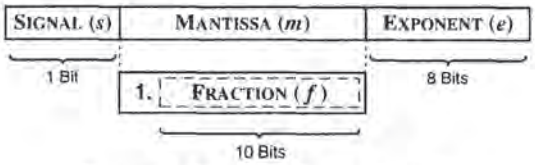


Figure 8. Floating-Point Format.

$$= (-1)^{S_2 \oplus S_2} \times (M_1 \cdot M_2) \times 2^{(E_1+E_2)}$$

The result of this operation is thus obtained by making use of an unsigned multiplier to compute the mantissa of the result, an unsigned adder to compute the exponent, and a barrel shifter to accomplish the rounding of the fractional part of the product, in order to normalise the result. Hence, this operation is performed in 4 steps, as illustrated in figure 9(a):

i) the mantissa of the operands are multiplied;

ii) the result is rounded in order to obtain a normalised 10-bit number;

iii) the new exponent is calculated by adding e_1 and e_2 and by taking in consideration the previous rounding step;

iv) since the exponents are stored with a biased format, the bias constant is subtracted from the sum of the biased exponents:
$(e_1 - 127) + (e_2 - 127) + 127 = e_1 + e_2 - 127.$

The last step can be done simultaneously with the third step. The sign value is simply the result of the XOR operation $(\oplus)$ between s_1 and s_2.

5.2. Floating-Point Adder

The floating-point addition is somewhat more complex than the multiplication because, depending on the signs of the operands, it may actually be a subtraction. Furthermore, it is necessary to start by adjusting the operands, so that they

Attachment - Page A367

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

have the same exponent when they are applied to the adder, according to the following formula, where $e_1 \geq e_2$:

$$(f_1 \times 2^{e_1}) + (f_2 \times 2^{e_2}) = \left[f_1 + f_2 \times 2^{-(e_1 - e_2)}\right] \times 2^{e_1} \quad (10)$$

This operation is performed in 5 steps, as illustrated in figure 9(b):

i) the exponents are compared and the operands are swapped so that $e_1 - e_2 \geq 0$, and the exponent of the result is temporary assumed to be e_1;

ii) the mantissa of the smaller operand is right shifted $d = e_1 - e_2$ places by using a barrel shifter;

iii) the result of this shift and the mantissa of the greater operand are applied to the adder/subtracter, taking in consideration the signs of both operands;

iv) the result of this operation is rounded and shifted until it is normalised and the exponent of the result is adjusted;

v) the sign of the result is determined.

The last step can be done simultaneously with any of the previous steps.

5.3. Floating-Point Units Implementation

The floating point units were described using synthesizable VHDL code. A standard library of arithmetic units was used to implement the adders and multipliers [13]. This library contains many units for a comprehensive set of integer arithmetic operations, with multiple structurally different implementations, allowing the adjustment of the performance of the circuit by trading circuit area versus speed. Among the several available arithmetic units, fast multipliers and adders, implemented using the carry-save technique, were selected. Furthermore, Wallace trees were adopted to speed up carry-save addition in binary adders and multipliers.

6. System Implementation

The choice of the hardware support to be used in the implementation of the described architecture is a significant aspect in what concerns the final characteristics of the system. Three main aspects are usually considered:

- *System Performance*, which characterises the operating frequency of the system, as well as its power requirements.
- *System Configurability*, characterising the processor modularity according to a set of implementation parameters, as well as its programmability.
- *Implementation Costs* of the system.

In order to obtain a detailed comparison of the characteristics of several different implementation supports, three distinct approaches were considered: Field Programmable Gate Array (FPGA), Application Specific Integrated Circuit (ASIC) and Digital Signal Processor (DSP). Each of these hardware implementations can be distinctly characterised in what concerns the three main aspects referred before. Application specific integrated circuits are often regarded as being the obvious choice, offering better conditions to provide the necessary signal processing performance required by a real-time operation. However, such dedicated hardware has the disadvantage of lacking flexibility and modularity, making it difficult to introduce later modifications on the algorithm or on its parameters. In contrast, DSP based implementations offer a programmability capability which usually makes them the optimal choice in what concerns system flexibility. However, applications with real-time demands usually require signal processing performances which cannot be provided by standard DSPs. Such applications are often implemented by using dedicated co-processors to perform the more computational intense tasks. FPGAs can be considered as a compromise between these two options. Although these configurable devices have a hardware structure more closed to the ASIC's structure, they offer a programming flexibility comparable to the one offered by DSPs.

In the following subsections it will be described and compared the several implementations of the proposed architecture. In what concerns the system programmability and parameterisation, the three referred implementations can be classified into two categories: *hardware implementations*, where VHDL hardware description language has been used in the architecture description of the FPGAs and of the ASIC, and *software implementations*, where Assembly programming language has been used to program the DSP.

6.1. FPGA based implementation

A modular processor for generating high-order image moments, based on the architecture presented in the previous sections, was physically implemented in a small number of low-cost FPGAs. Comparing figure 10 with figure 7, it can be observed that an additional pipeline register was introduced in the accumulator circuit, in order to balance the processing time of the pipeline stages. With this additional register, the odd and even partial results are independently accumulated in two registers. At the end of the image, the adder is used to compute the final result.

The processor is composed by three different types of blocks. Module A' generates x and y values and introduces, at the correct time, the pixel intensity values into the power-core. Each module of type B implements one stage of the power-core. Module A'' receives, as input, the several val-

Attachment - Page A368

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

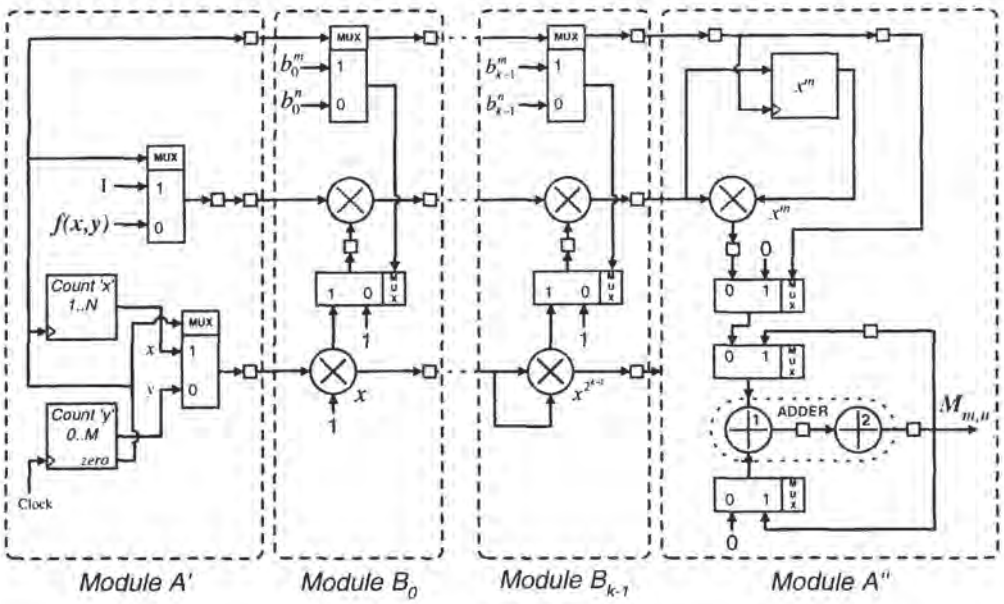


Figure 10. Processor implementation with Xilinx XC4013E FPGAs.

ues of x^m (which are sequentially stored in a register) and the pre-calculated values of $y^n \times f(x,y)$. The moment value is computed by multiplying these values along the set of N image lines: $\mathcal{M}_{m,n} = \sum_{x=1}^{N} \sum_{y=1}^{M} x^m \left(y^n f(x,y)\right)$. A single module of type A' and A'' and a set of k ($k = \lceil \max\{\log_2(m+1), \log_2(n+1)\}\rceil$) modules of type B are used to compute the 2-D image moments $\mathcal{M}_{m,n}$. As an example, only three or four modules of type B are required to compute moments of any order $p = (m+n) \leq 14$ (with $m,n \leq 7$) or $p \leq 30$ (with $m,n \leq 15$), respectively.

The processor modules were implemented using the XC4013E-PQ160 FPGA [14] and the Xilinx Foundation Series 2.1 software [15]. This configurable device is one of the most basic FPGAs of the XC4000E family, composed by 576 Configurable Logic Blocks (CLBs) and 129 I/O pads [14]. Module A' requires little hardware: just two counters and one multiplexer. For this reason, it was implemented together with module A'' ($A = A' + A''$) in a single FPGA (see figure 10). In table 1 it is presented the results of these implementations, regarding the device occupation and the maximum working frequency of the processor. The

Table 1. Implementation results of the several modules that compose the FPGA based processor.

Module	Occupation [%]	Minimum Period [ns]	Frequency [MHz]
$A = A' + A''$	78.3	99.5	10.1
B	75.3	69.4	

obtained results evidence that both types of modules A and B can be implemented in a single XC4013E-PQ160 FPGA. They occupy about 78 % of the available set of CLBs and about 98 % of the available I/O pads. In figure 11 it is presented the relation between the computed moment order (p) and the required number of XC4013E-PQ160 FPGAs. The maximum working frequency achieved is about 10 MHz, which makes the processor able to compute 2-D image moments at the video-rate required for resolutions of 640×480 pixels, as it is depicted in figure 11. The value of the work-

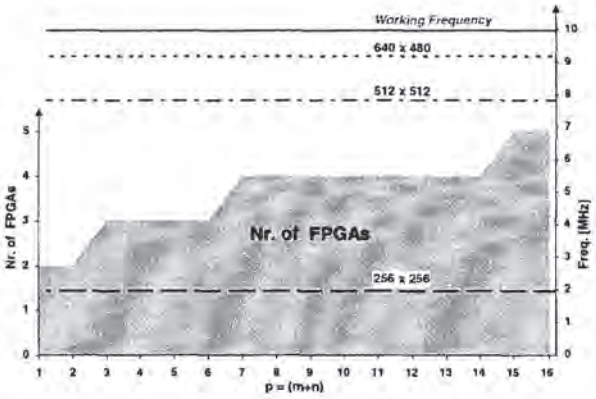


Figure 11. Number of required FPGAs to compute a p=(m+n) order moment (with m,n $\leq$ 7); required clock frequency to process several image sizes.

Attachment - Page A369

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

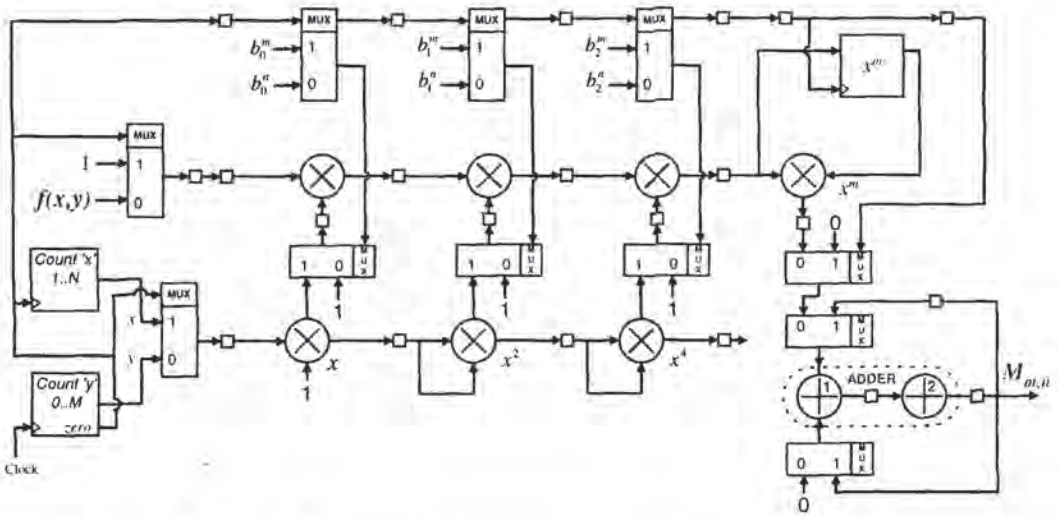


Figure 12. Processor implementation using an ASIC device.

Table 2. FPGA based processor implementation results.

Parameter	Value	
Devices Used	4	$FPGAs$
Logic Blocks	1319	$CLBs$
Power Dissip.	4.67	W
Critical Path	99.47	ns
Proc. Time	26.13	ms

ing frequency can still be increased by reducing the processing time of the arithmetic units, which can be achieved by increasing the pipeline depth.

A processor composed by four XC4013E-PQ160 devices, for generating image moments with order less than 14 (with $m, n \leq 7$), was implemented. In table 2 it is depicted its electrical characteristics, where the presented processing time refers to the computation of a 512×512 image moment. This processor was connected and fully tested with an acquisition video processing system, which accepts PAL video signals and sends the computed image moments to a Pentium Personal Computer, through one I/O parallel port.

6.2. ASIC based implementation

The system implementation supported on the hardware structure of an ASIC device is based on the architecture previously described in section 4 and presented in figure 12. Similarly to the FPGA based processor, this implementation has been carried out to obtain an efficient hardware processor capable of computing image moments with order less than 14 (with $m, n \leq 7$). As it has been done with the previously described FPGA based device, an additional pipeline register was introduced in the accumulator circuit in order to increase the working frequency of the processor. Likewise, the parameterisation of the order of the computed image moment is done simply by programming two processor registers.

The VHDL description of the circuit was synthesised with Synopsys^{TM} Design Compiler. It was implemented using the Atmel ES2 Standard Cells Library for the $0.7\ \mu m$ dual-metal CMOS process, based on a $5.0\ V$ technology [16]. This library is composed by a complete set of StdLib logic cells, whose simplest device, the inverter, is characterised by a typical propagation time of about $0.12\ ns$, occupying a total area of $273.6\ \mu m^2$.

The implementation results of the synthesised circuit are presented in table 3. By comparing these results with those obtained for the FPGA based processor (see table 2), it can be observed a reduction of about 88.5% in power consumption. Moreover, the working frequency can be increased to $23.36\ MHz$, which decreases the processing time by a factor of 2.32.

Table 3. ASIC implementation results.

Parameter	Value	
Critical Path	42.80	ns
Cells	4716	
Transistors	43894	
Area	3.26	mm^2
Power	534.4	mW
Proc. Time	11.242	ms

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

```
; x^m Power Computation (result in R0)

      ldf    0,R1          ; R1 <- x (line counter)
      ldi    @_tableX,AR0
      ldi    @powerX,R7    ; R7 <- m
L1:   ldi    511,RC        ; Repeats the following
                           ; block 511+1 times
      rptb   L2

Lx1:  lsh    -1,R7         ; Cflag <- b0
      bncd   Lx2           ; Tests bo bit
      ldi    1,R0          ; R0 <- 1
      mpyf3  R1,R1,R2      ; R2 <- x^2
      mpyf3  R2,R2,R3      ; R3 <- x^4

      mpyf   R1,R0         ; R0*=b0.x
Lx2:  lsh    -1,R7         ; Cflag <- b1
      bncd   Lx4           ; Tests b1 bit
      addi   1,R1          ; R1++ (for next cycle)
      nop
      nop

      mpyf   R2,R0         ; R0*=b1.x^2
Lx4:  lsh    -1,R7         ; Cflag <- b1
      bnc    Lx4           ; Tests b2 bit

      mpyf   R3,R0         ; R0*=b2.x^4
L2:   stf    R0,*AR0++     ; tableX[AR0++] <- x^m
```

(a)

```
; Moment Mmn computation
;  - tableX contains the x^m power values
;  - tableY contains the x^n power values
;  - frame contains the pixel value at line x and column y

      ldi    @_tableY,AR0
      ldi    @_tableX,AR1
      ldi    @_frame,AR7
      ldi    0,R2
      ldi    0,R3
      ldi    512,R6          ; repeats next block 512 times

      subi   1,R6            ; counter --
L7:   mpyf   *AR7,*AR0++,R0  ; R0 <- frame * tableY[counterY++]
      rpts   510             ; repeats next block 511 times
      mpyf3  *AR7,*AR0++,R0  ; R0 <- frame * tableY[counterY++]
   || addf   R0,R2           ; R2 += R0

      addf   R0,R2           ; R2 += R0
      mpyf3  *AR1++,R2,R1    ; R1 <- tableX[counterX++]*R2
      addf   R1,R3           ; R3 += R1

      cmpi   0,R6
      bnzd   L7
      ldi    @_tableY,AR0
      ldi    0,R2
      subi   1,R6            ; R6--
                             ; R3 = Moment Value
```

(b)

Figure 13. Assembly code: (a)-Initialisation of the table containing the x^m values; (b)-moment computation.

6.3. DSP based implementation

In order to compare the previously described dedicated *hardware implementations* with an implementation based on a digital signal processor, a *software implementation* has been performed using a standard floating-point DSP. The chosen device was the 32-bit DSP TMS320C30 from Texas Instruments, based on a 1.0 μm CMOS technology and operated using a clock frequency of 33 MHz. This processor is characterised by having a 60 ns single-cycle instruction execution time, capable of performing 33.3 MFLOPS and 16.7 MIPS [17]. This DSP exhibits greater precision than the hardware implementations previously described (the mantissa is represented with 24 bits).

In order to obtain an algorithm as efficient as possible, the computation of the values of x^m and y^n was done by making use of two distinct tables, pre-computed during an initialisation phase. This, of course, has the inherent disadvantage of loosing part of the flexibility of the proposed architecture. Nevertheless, due to the intrinsic sequential execution characteristics of the DSP (contrasting with the parallel structures of the FPGAs and ASIC), this is the most efficient method to compute these power values. Moreover, the other alternative would require the repetitive computation of the same set of values for all processed frames. Using this method, these tables are computed only once, during the initialisation phase of the processor.

Therefore, two distinct main tasks must be considered in this implementation: the initialisation of the x^m and y^n tables and the computation of the image moment. The Assembly code of these tasks (correspondent to processing of a (512×512) image) is depicted in figure 13(a) and in figure 13(b), respectively.

A total of $((512 \times 19) + 8)$ cycles are required to initialise the tables containing the x^m and y^n power values (with $m, n \leq 7$). Therefore, the initialisation task correspondent to the pre-computation of the x^m and y^n tables comprises about 19472 execution cycles, which corresponds to a initialization time of about 1.17 ms. In what concerns the computation of the moment value, it is possible to conclude that $(524 \times 512 + 7)$ execution cycles are required, which corresponds to a processing time of about 16.1 ms. Table 4 depictes the main characteristics of this DSP based implementation. It can be observed that while the processing time value is between those obtained for the other two implementations, the obtained power consumption value presents an increase by a factor of about 2.33 when compared with the ASIC based implementation. With this processor, moments with order $p \leq 14$ $(m, n \leq 7)$ can be computed for 512×512 images at a rate of about 60 frames per second.

Attachment - Page A371

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

Table 4. DSP based implementation.

Parameter	Value	
Devices Used	1	DSP
Power Dissip.	1.25	W
Clock Frequency	33.0	MHz
Execution Cycle	60	ns
Processing Time	16.1	ms

7. Discussion and Conclusions

This paper proposes a new systolic architecture for computing high order 2-D gray level image moments, $\mathcal{M}_{m,n}$, requiring only $(2k+1)$ multipliers ($k = \lceil \max\{\log_2(m+1), \log_2(n+1)\}\rceil$) and one adder. The described architecture makes use of floating-point representation to accommodate the high dynamic range of the operands with a small number of bits. As far as we know, this is the first architecture that allows the design of real-time processors for computing 2-D image moments with a number of multipliers that only increases with the logarithm of the maximum order of the moments in one dimension. Moreover, the repetitive structure of the architecture is well adapted to the development of modular processors with VLSI circuits and with configurable devices.

It has been shown the usefulness of the proposed architecture for developing real-time processors based on the different type of devices: FPGAs, ASICs and DSPs. The ASIC based implementation has proved to be the most suitable to be used in applications requiring low-power consumption, such as mobile equipment. It has also been shown that DSP based implementations are more suitable for applications where the proposed architecture is used as a dedicated co-processor in a host processing system and where power consumption is not a significant concern. The results of the FPGA implementation have shown that this configuration is well suitable only in the design and test phases of the dedicated processors, due to the high power consumption and moderate operating frequency.

References

[1] M.-K. Hu, "Visual Pattern Recognition by Moment Invariants", *IRE Transactions in Information Theory*, vol. IT-1, February 1962, pp. 179–187.

[2] R. J. Prokop and A. P. Reeves, "A Survey of Moment-Based Techniques for Unoccluded Object Representation and Recognition", *Graphical Models and Image Processing*, vol. 54, no. 5, September 1992, pp. 438–460.

[3] G. Agin, "Handbook of Industrial Robotics", chap. *Vision Systems, Addison-Wesley*, 1985.

[4] R. Haralick and L. Shapiro, "Computer and Robot Vision", *Addison-Wesley*, vol. II, chap. 18. 1993.

[5] C.-H. Teh and R. T. Chin, "On Image Analysis by the Methods of Moments", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 4, July 1988, pp. 496–513.

[6] S. X. Liao and M. Pawlak, "On Image Analysis by Moments", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 3, March 1996, pp. 254–266.

[7] C. Coelho, N. Roma, L. Sousa, "Pipeline Architectures for Computing 2-D Image Moments", *Proc. of DCIS'99*, Palma de Mallorca, Spain, November 1999.

[8] R. Andersson, "Real-Time Gray-Scale Video Processing Using a Moment-Generating Chip", *IEEE Journal of robotics and Automation*, vol. 1, no. 2, June 1985, pp. 79-85.

[9] K. Chen, "Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments", *Pattern Recognition*, vol. 23, no. 1/2, 1990, pp. 109-119.

[10] X. Jiang and H. Bunke, "Simple and Fast Computation of Moments", *Pattern Recognition*, vol. 24, no. 8, 1991, pp. 801-806.

[11] H. D. Cheng, C. Y. Wu and D. L. Hung, "VLSI for Moment Computation and its Application to Breast Cancer Detection", *Pattern Recognition*, vol. 31, no. 9, 1998, pp. 1391-1406.

[12] P. Pirsch, "Architectures for Digital Signal Processing", *John Wiley & Sons*, chap.5, 1998.

[13] R. Zimmermann, "VHDL Library of Arithmetic Units", *Forum on Design Languages*, Lausanne, September 1998.

[14] "XC4000E and XC4000X series Field Programmable Gate Array", *Xilinx Inc.*, June 1997.

[15] "Foundation Series 2.1i User Guide", *Xilinx Inc.*, 1999.

[16] "ES2 ECPD07 Library Databook", *Atmel ES2*, September 1997.

[17] "TMS320C3X User's Guide", *Texas Instruments*, June 1992.

Attachment - Page A372

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on December 05,2022 at 16:05:17 UTC from IEEE Xplore. Restrictions apply.

ATTACHMENT 14C

ieeexplore-ieee-org.libproxy.unl.edu/stamp/stamp.jsp?tp=&arnumber=875975

In the development and evaluation of specialized processors for computing high-order 2-D image m... 1 / 10 125%

Document properties

File name: getPDF.jsp
File size: 862 KB

Title: In the development and evaluation of specialized ...
Author: N. Roma,L. Sousa
Subject: Proceedings Fifth IEEE International Workshop on ...
Keywords: -
Created: 3/1/04, 9:09:12 AM
Modified: 12/5/22, 10:05:17 AM
Application: Acrobat Capture 3.0

PDF producer: Adobe PDF Library 4.0; modified using iText® 7.1....
PDF version: 1.3
Page count: 10
Page size: Varies

Fast web view: No

Close

In the Develop ...s for Computing ...ne

...AL

Image moments are u... modelling, matching and ... of high-order moments ... that can not be implem... general-purpose process... specialised processors fo... an arbitrary order in re... cessing techniques and floating-point arithmetic units. It proposes a modular and cost effective architecture for generating image moments, with a processing time not dependent on the order of the computed moments. The architecture was implemented using different devices, such as programmable digital processors, configurable hardware logic and integrated circuits, by using a 0.7 μm CMOS process. The several implementations have shown the effectiveness of the architecture, and the obtained results allow us to compare the different solutions in terms of speed, flexibility, cost

...mmonly used to extract fea... of gravity, and to find the ...ects in a image [1, 2, 3, 4]. In ...are used for pattern recogni... [5, 6].

...uation 1 requires NM addi... ...plications. In the last years, ...ectures have been proposed ...ents computation. However, some of them are only valid for low-order moments and/or for binary images [7, 8, 9, 10]. Other proposals suggest architectures without practical interest due to its complexity [11]. Moreover, floating-point arithmetic has to be used to compute moments of different orders, given that the required dynamic range increases very rapidly with the moment's order. Some other proposed architectures [7] adopt wavefront processing techniques to adapt the throughput to the variable processing time exhibited by floating-point arithmetic units. However, although they eliminate the need

ATTACHMENT 15A

WorldCat: Languages and Compilers for Parallel Computing

OCLC	299078541	**No holdings in LDL - 24 other holdings**									
Books	**Rec Stat**	c		**Entered**	20080812		**Replaced**	20220526234125.6			
Type	a	**ELvl**	M	**Srce**	d		**Audn**		**Ctrl**		**Lang**	eng
BLvl	m	**Form**	o	**Conf**	0		**Biog**		**MRec**		**Ctry**	gw
		Cont	b	**GPub**			**LitF**	0	**Indx**	0		
Desc	i	**Ills**		**Fest**	0		**DtSt**	s	**Dates**	2008,		

007 c ‡b r ‡d n ‡e n ‡g 008 ‡h m ‡i a ‡j m ‡k a ‡l a
040 Nz ‡b eng ‡e pn ‡c UX0 ‡d OCLCQ ‡d CEF ‡d OCLCQ ‡d A7U ‡d OCLCQ ‡d OCLCO ‡d OCLCQ ‡d DEBSZ ‡d OCLCQ ‡d SHS ‡d NJR ‡d OCLCF ‡d WYU ‡d QE2 ‡d CNTRU ‡d OCLCQ ‡d BWN ‡d ZHM ‡d OCLCQ ‡d UKAHL ‡d OCLCO
019 402311510 ‡a 1044191037 ‡a 1084940692 ‡a 1110294644 ‡a 1110929598
020 9783540852612
020 3540852611
020 ‡z 9783540852605
024 7_ 10.1007/978-3-540-85261-2 ‡2 doi
050 _4 QA76.58 ‡b .W656 2007 eb
055 _3 QA75 ‡b .L38 no.5234
082 04 005.13
084 54.51 ‡2 bcl
084 54.53 ‡2 bcl
090
049 LDLL
100 1_ Hutchison, David.
245 10 Languages and Compilers for Parallel Computing : ‡b 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers / ‡c edited by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Vikram Adve, María Jesús Garzarán, Paul Petersen.
260 Berlin, Heidelberg : ‡b Springer-Verlag Berlin Heidelberg, ‡c 2008.
300 1 online resource
336 text ‡b txt ‡2 rdacontent
337 computer ‡b c ‡2 rdamedia
338 online resource ‡b cr ‡2 rdacarrier
490 1_ Lecture Notes in Computer Science, ‡x 0302-9743 ; ‡v 5234
505 0_ Reliability -- Compiler-Enhanced Incremental Checkpointing -- Techniques for Efficient Software Checking -- Languages -- Revisiting SIMD Programming -- Multidimensional Blocking in UPC -- An Experimental Evaluation of the New OpenMP Tasking Model -- Language Extensions in Support of Compiler Parallelization -- Parallel Compiler Technology I -- Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers -- Iteration Disambiguation for Parallelism Identification in Time-Sliced Applications -- A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor -- Pillar: A Parallel Implementation Language -- Associative Parallel Containers in STAPL -- Explicit Dependence Metadata in an Active Visual Effects Library -- Supporting Huge Address Spaces in a Virtual Machine for Java on a Cluster -- Modeling Relations between Inputs and Dynamic Behavior for General Programs -- Evaluation of RDMA Opportunities in an Object-Oriented DSM -- Automatic Communication Performance Debugging in PGAS Languages -- Parallel Compiler Technology II -- Exploiting SIMD Parallelism with the CGiS Compiler Framework -- Critical Block Scheduling: A Thread-Level Parallelizing Mechanism for a Heterogeneous Chip Multiprocessor Architecture -- Languages II -- Capsules: Expressing Composable Computations in a Parallel Programming Model -- Communicating Multiprocessor-Tasks -- An Effective Automated Approach to

		Specialization of Code -- Flow-Sensitive Loop-Variant Variable Classification in Linear Time -- Using ZBDDs in Points-to Analysis.
520		This book constitutes the thoroughly refereed post-conference proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing, LCPC 2007, held in Urbana, IL, USA, in October 2007. The 23 revised full papers presented were carefully reviewed and selected from 49 submissions. The papers are organized in topical sections on reliability, languages, parallel compiler technology, libraries, run-time systems and performance analysis, and general compiler techniques.
504		Includes bibliographical references and author index.
650	_0	Computer Communication Networks.
650	_0	Computer science.
650	_0	Data structures (Computer science)
650	_0	Electronic data processing.
650	_6	Informatique. ‡0 (CaQQLa)201-0063036
650	_6	Structures de données (Informatique) ‡0 (CaQQLa)201-0051595
650	_7	data processing. ‡2 aat ‡0 (CStmoGRI)aat300054636
650	_7	computer science. ‡2 aat ‡0 (CStmoGRI)aat300054575
650	_7	Computer science. ‡2 fast ‡0 (OCoLC)fst00872451
650	_7	Data structures (Computer science) ‡2 fast ‡0 (OCoLC)fst00887978
650	_7	Electronic data processing. ‡2 fast ‡0 (OCoLC)fst00906956
655	_4	Electronic books.
700	1_	Adve, Vikram.
700	1_	Garzarán, María Jesús.
700	1_	Kanade, Takeo.
700	1_	Kittler, Josef.
700	1_	Kleinberg, Jon M.
700	1_	Mattern, Friedemann.
700	1_	Mitchell, John C.
700	1_	Naor, Moni.
700	1_	Nierstrasz, Oscar.
700	1_	Pandu Rangan, C.
700	1_	Petersen, Paul.
700	1_	Steffen, Bernhard.
700	1_	Sudan, Madhu.
700	1_	Terzopoulos, Demetri.
700	1_	Tygar, Doug.
700	1_	Vardi, Moshe Y.
700	1_	Weikum, Gerhard.
830	_0	Lecture notes in computer science ; ‡v 5234. ‡x 0302-9743
856	40	‡3 SpringerLink ‡u https://doi.org/10.1007/978-3-540-85261-2
856	40	‡3 SpringerLink ‡u https://link.springer.com/book/10.1007/978-3-540-85261-2
856	40	‡3 SpringerLink ‡u http://www.springerlink.com/openurl.asp?genre=issue&issn=302-9743&volume=5234
856	4_	‡3 Cover ‡u http://swbplus.bsz-bw.de/bsz286098229cov.htm ‡v 20151204153501
856	40	‡u https://www.vlebooks.com/vleweb/product/openreader?id=none&isbn=9783540852612
938		Askews and Holts Library Services ‡b ASKH ‡n AH26901101
029	0_	NZ1 ‡b 12798388
029	1_	DEBSZ ‡b 286098229

006 fields for Computer Files	**Computer File**					
Audn	**Form** o	**File** d	**GPub**			
Action Status	Delete Holdings _	Export _	Label _	Produce _	Update Holdings _	Validate _

ATTACHMENT 15B

Springer Link

Search Log in

Download book PDF



International Workshop on Languages and Compilers for Parallel Computing

LCPC 2007: **Languages and Compilers for Parallel Computing** pp 32–46

Revisiting SIMD Programming

Anton Lokhmotov, Benedict R. Gaster, Alan Mycroft, Neil Hickey & David Stuttard

Conference paper

403 Accesses | **1** Citations

Part of the Lecture Notes in Computer Science book series (LNTCS,volume 5234)

Abstract

Massively parallel SIMD array architectures are making their way into embedded processors. In these architectures, a number of identical processing elements having small private storage and using asynchronous I/O for accessing large shared memory executes the same instruction in lockstep.

In this paper, we outline a simple extension to the C language, called C^n, used for programming a commercial SIMD array architecture. The design of

C^n is based on the concept of the SIMD array type architecture and revisits first principles of designing efficient and portable parallel programming languages. C^n has a low level of abstraction and can also be seen as an intermediate language in the compilation from higher level parallel languages to machine code.

Keywords

Processing Element

Single Instruction Multiple Data

Intermediate Language

Processing Element Array

Single Instruction Multiple Data Architecture

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

Download conference paper PDF

References

1. Parhami, B.: SIMD machines: do they have a significant future? SIGARCH Comput. Archit. News 23(4), 19–22 (1995)

2. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A.: The Illiac IV computer. IEEE Trans. Computers C-17(8), 746–757 (1968)

3. ClearSpeed Technology: The CSX architecture, http://www.clearspeed.com/

4. Snyder, L.: The design and development of ZPL. In: Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III), pp. 8–37. ACM Press, New York (2007)

5. Slotnick, D.: The conception and development of parallel processors—a personal memoir. IEEE Annals of the History of Computing 4(1), 20–30 (1982)

6. Wilkes, M.V.: The lure of parallelism and its problems. In: Computer Perspectives. Morgan Kaufmann, San Francisco (1995)

7. Snyder, L.: Type architecture, shared memory and the corollary of modest potential. Annual Review of Computer Science 1, 289–317 (1986)

8. Kennedy, K., Koelbel, C., Zima, H.: The rise and fall of High Performance Fortran: an historical object lesson. In: Proc. of the third ACM SIGPLAN

conference on History of programming languages (HOPL III), pp. 7–22. ACM Press, New York (2007)

9. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages – C (1999)

10. Lawrie, D.H., Layman, T., Baer, D., Randal, J.M.: Glypnir—a programming language for Illiac IV. Commun. ACM 18(3), 157–164 (1975)

11. Stevens Jr., K.: CFD—a Fortran-like language for the Illiac IV. SIGPLAN Not. 10(3), 72–76 (1975)

12. Perrott, R.H.: A language for array and vector processors. ACM Trans. Program. Lang. Syst. 1(2), 177–195 (1979)

13. Li, K.C., Schwetman, H.: Vector C: a vector processing language. Journal of Parallel and Distributed Computing 2(2), 132–169 (1985)

14. Knobe, K., Lukas, J.D., Steele Jr., G.L.: Data optimization: allocation of arrays to reduce communication on SIMD machines. J. Parallel Distrib. Comput. 8(2), 102–118 (1990)

15. Weiss, M.: Strip mining on SIMD architectures. In: Proc. of the 5th International Conference on Supercomputing (ICS), pp. 234–243. ACM Press, New York (1991)

16. Rose, J.R., Steele Jr., G.L.: C*: An extended C language for data parallel programming. In: Proc. of the 2nd International Conference on Supercomputing (ICS), vol. 2, pp. 2–16 (1987)

17. MasPar Computer Corporation: MasPar Programming Language (ANSI C compatible MPL) Reference Manual (1992)

18. Kyo, S., Okazaki, S., Arai, T.: An integrated memory array processor for embedded image recognition systems. IEEE Trans. Computers 56(5), 622–634 (2007)

19. Christy, P.: Software to support massively parallel computing on the MasPar MP-1. In: Proc. of the 35th IEEE Computer Society International Conference (Compcon Spring), pp. 29–33 (1990)

20. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)

21. Freescale Semiconductor: AltiVec technology programming interface manual (1999)

22. Bradley, C., Gaster, B.R.: Exploiting loop-level parallelism for SIMD arrays using OpenMP. In: Proc. of the 3rd International Workshop on OpenMP (IWOPM) (2007)

23. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 325–335. ACM Press, New York (2006)

24. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Proc. of the 2006 ACM/IEEE Conference on Supercomputing (SC), pp. 83–92. ACM Press, New York (2006)

25. ACE Associated Compiler Experts: The CoSy compiler development system, http://www.ace.nl/

26. Kim, H.: Region-based register allocation for EPIC architectures. PhD thesis, Department of Computer Science, New York University (2001)

27. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641. Springer, Heidelberg (2007)

Author information

Authors and Affiliations

Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK

Anton Lokhmotov & Alan Mycroft

ClearSpeed Technology, , 3110 Great Western Court, Bristol, BS34 8HP, UK

Benedict R. Gaster, Neil Hickey & David Stuttard

Editor information

Editors and Affiliations

Rights and permissions

Reprints and Permissions

Copyright information

© 2008 Springer-Verlag Berlin Heidelberg

About this paper

Cite this paper

Lokhmotov, A., Gaster, B.R., Mycroft, A., Hickey, N., Stuttard, D. (2008). Revisiting SIMD Programming. In: Adve, V., Garzarán, M.J., Petersen, P. (eds) Languages and Compilers for Parallel Computing. LCPC 2007. Lecture Notes in Computer Science, vol 5234. Springer, Berlin, Heidelberg. https://doi-org.libproxy.unl.edu/10.1007/978-3-540-85261-2_3

.RIS .ENW .BIB

DOI
https://doi-org.libproxy.unl.edu/10.1007/978-3-540-85261-2_3

Publisher Name	Print ISBN	Online ISBN
Springer, Berlin, Heidelberg	978-3-540-85260-5	978-3-540-85261-2

eBook Packages
Computer Science
Computer Science (R0)

Share this paper

Anyone you share the following link with will be able to read this content:

Get shareable link

Provided by the Springer Nature SharedIt content-sharing initiative

Not logged in - 129.93.169.156
Big Ten Academic Alliance Formerly (CIC) (3000133814) - University of Nebraska-Lincoln (3000134173)
SPRINGER NATURE

© 2022 Springer Nature Switzerland AG. Part of Springer Nature.

ATTACHMENT 15C

Vikram Adve
María Jesús Garzarán
Paul Petersen (Eds.)

Languages and Compilers for Parallel Computing

LNCS 5234

20th International Workshop, LCPC 2007
Urbana, IL, USA, October 2007
Revised Selected Papers

Springer

Attachment - Page A389

Lecture Notes in Computer Science 5234

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Alfred Kobsa
University of California, Irvine, CA, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Vikram Adve
María Jesús Garzarán
Paul Petersen (Eds.)

Languages and Compilers for Parallel Computing

20th International Workshop, LCPC 2007
Urbana, IL, USA, October 11-13, 2007
Revised Selected Papers



Attachment - Page A391

Volume Editors

Vikram Adve
María Jesús Garzarán
University of Illinois at Urbana-Champaign
Department of Computer Science
Thomas M. Siebel Center for Computer Science
201 N. Goodwin Ave, Urbana, IL, 61801, USA
E-mail: {vadve,garzaran}@cs.uiuc.edu

Paul Petersen
Intel Corporation
1906 Fox Drive, Champaign, IL, 61820, USA
E-mail: paul.petersen@intel.com

Library of Congress Control Number: 2008932585

CR Subject Classification (1998): D.1.3, C.2.4, D.4.2, H.3.4, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-85260-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-85260-5 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12455096 06/3180 5 4 3 2 1 0

Preface

It is our pleasure to present the papers from the 20th International Workshop on Languages and Compilers for Parallel Computing! For the past 19 years, this workshop has been one of the primary venues for presenting and learning about a wide range of current research in parallel computing. We believe that tradition has continued in this, the 20th year of the workshop.

This year, we received 49 paper submissions from 10 countries. About a quarter of the papers (12 out of 49) included authors from industry. We selected 23 papers to be presented at the workshop, for an acceptance rate of 47%, which was similar to that of the last two years. Each paper received at least three reviews, with about two-thirds of the papers getting four or more reviews each. Most papers also received at least one review from an external reviewer. The committee held a full-day teleconference to discuss the reviews and select papers. Program Committee members who had a conflict with a paper left the call when that paper was being discussed. There were seven submissions that included Program Committee members as co-authors. These papers were evaluated more stringently and four of seven were accepted.

The workshop this year also included two exciting special events. First, David Kirk, Chief Scientist of nVidia and a member of the National Academy of Engineering, gave a keynote talk on using highly multithreaded graphics processors for accelerating general-purpose parallel computing applications. Kirk and nVidia have led the drive to make the high parallelism in graphics processors more easily accessible for a wide range of applications beyond traditional graphics processing, and this talk gave LCPC attendees a valuable perspective on the potential of this work.

Second, a special panel was held on Friday morning to commemorate the 20th year of LCPC. This panel, organized and moderated by Steering Committee Chair David Padua, was scheduled for an entire session to allow seven leaders in parallel computing to give their perspective on how the field has evolved over the past 20 years, and what the major challenges are for the future. The panel included a number of luminaries in parallel computing – Arvind (MIT), David Kuck (Intel), Monica Lam (Stanford University), Alexandru Nicolau (University of California Irvine), Keshav Pingali (University of Texas, Austin), Burton Smith (Microsoft Research), and Michael Wolfe (The Portland Group). Our thanks to David Padua for organizing this panel.

We would like to thank the many people who contributed valuable time and effort to making LCPC 2007 a success. Most importantly, we want to thank all the members of the community who submitted papers to the workshop. This workshop is, in the end, an (incomplete) representation of the results of their hard work. Second, the Program Committee worked hard to review 12–13 papers each and to participate in the all-day Program Committee teleconference. The quality of

the technical program owes much to their effort. In his role as Steering Committee Chair, David Padua provided valuable guidance and support. He also lent his stack of 19 previous LCPC proceedings to mark the occasion!

Finally, Sheila Clark put in tremendous effort behind the scenes to manage many organizational details for the workshop and we would not have been able to pull it off without all her help.

October 2007

Vikram Adve
María Jesús Garzarán
Paul Petersen

Attachment - Page A394

Organization

LCPC 2007 was organized by the Steering and Program Committees.

General/Program Co-chairs

Vikram Adve	University of Illinois at Urbana-Champaign
María Jesús Garzarán	University of Illinois at Urbana-Champaign
Paul Petersen	Intel Corporation

Program Committee

Vikram Adve	University of Illinois at Urbana-Champaign
Gheorghe Almási	IBM Research
José Nelson Amaral	University of Alberta, Canada
Eduard Ayguadé	Universitat Politécnica de Catalunya, Spain
Gerald Baumgartner	Louisiana State University
Călin Caşcaval	IBM Research
María Jesús Garzarán	University of Illinois at Urbana-Champaign
Sam Midkiff	Purdue University
Paul Petersen	Intel Corporation
J. Ramanujam	Louisiana State University
P. Sadayappan	Ohio State University
Peng Wu	IBM Research

Steering Committee

Rudolf Eigenmann	Purdue University
Alex Nicolau	UC Irvine
David Padua	University of Illinois at Urbana-Champaign
Lawrence Rauchwerger	Texas A&M University

Sponsoring Institution

University of Illinois at Urbana-Champaign
Siebel Center for Computer Science
Urbana, Illinois, October 11–13, 2007

Referees

V. Adve
G. Agrawal
G. Almasi
J.N. Amaral
E. Ayguadé
M. Baskaran
G. Baumgartner
P. Berube
R. Bocchino
U. Bondhugula
J. Brodman
C. Caşcaval
L. Ceze
A. Chakravarti
A. Cohen
G. Dozsa
A. Eichenberger
R. Eigenmann
B. Fraguela
F. Franchetti
R. Garg
M.J. Garzarán
R. Ghiya
M. Girkar
M. Gonzalez
A. Gotsman
J. Guo
S. Hack
J. Hoeflinger
A. Kejariwal
G. Kondrak
D. Koppelman
S. Krishnamoorthy
K. Laufer
O. Lhotak
X. Martorell
L. Meadows
J. Mellor-Crummey
S. Midkiff
R. Niewiadomski
D. Nikolopoulos
R. Nishtala
S. Pande
S. Parthasarathy
P. Petersen
C. Pickett
K. Pingali
J. Ramanujam
A. Ramirez
A. Rountev
R. Rugina
P. Sadayappan
E. Salami
R. Schreiber
P. Sdayappan
A. Sidelnik
K. Strauss
Z. Sura
D. Tarditi
X. Tian
P. Unnikrishnan
N. Vasilache
R. Wilhelm
P. Wu
Y. Zheng
P. Zhao

LCPC at Illinois

1989



2007



Keynote Presentation: NVIDIA CUDA Software and GPU Parallel Computing Architecture

David Kirk, Chief Scientist, nVidia Corp.

Abstract. In the past, graphics processors were special-purpose hardwired application accelerators, suitable only for conventional rasterization-style graphics applications. Modern GPUs are now fully programmable, massively parallel floating point processors. This talk describes NVIDIA's massively multithreaded computing architecture and CUDA software for GPU computing. The architecture is a scalable, highly parallel architecture that delivers high throughput for data-intensive processing. Although not truly general-purpose processors, GPUs can now be used for a wide variety of compute-intensive applications beyond graphics.

Panel I: How Is Multicore Programming Different from Traditional Parallel Programming?

Panelists:
Arch Robison (Intel)
Călin Caşcaval (IBM Research)
Wen-mei Hwu (University of Illinois at Urbana-Champaign)
Hironori Kasahara (Waseda University, Japan) and
Gudula Rünger (Chemnitz University of Technology, Germany)
Moderator:
Vikram Adve (University of Illinois at Urbana-Champaign)

Panel II: What Have We Learned After 20 LCPCs?

Panelists:
David Kuck (Intel)
Arvind (MIT)
Monica Lam (Stanford University)
Alexandru Nicolau (University of California Irvine)
Keshav Pingali (University of Texas, Austin)
Burton Smith (Microsoft Research) and
Michael Wolfe (the Portland Group)
Moderator:
David Padua (University of Illinois at Urbana-Champaign)

Table of Contents

Reliability

Compiler-Enhanced Incremental Checkpointing 1
Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Radu Rugina

Techniques for Efficient Software Checking 16
Jing Yu, María Jesús Garzarán, and Marc Snir

Languages

Revisiting SIMD Programming 32
Anton Lokhmotov, Benedict R. Gaster, Alan Mycroft, Neil Hickey, and David Stuttard

Multidimensional Blocking in UPC 47
Christopher Barton, Călin Caşcaval, George Almasi, Rahul Garg, José Nelson Amaral, and Montse Farreras

An Experimental Evaluation of the New OpenMP Tasking Model 63
Eduard Ayguadé, Alejandro Duran, Jay Hoeflinger, Federico Massaioli, and Xavier Teruel

Language Extensions in Support of Compiler Parallelization 78
Jun Shirako, Hironori Kasahara, and Vivek Sarkar

Parallel Compiler Technology I

Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers 95
Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao

Iteration Disambiguation for Parallelism Identification in Time-Sliced Applications 110
Shane Ryoo, Christopher I. Rodrigues, and Wen-mei W. Hwu

A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor 125
Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien

Pillar: A Parallel Implementation Language 141
Todd Anderson, Neal Glew, Peng Guo, Brian T. Lewis, Wei Liu, Zhanglin Liu, Leaf Petersen, Mohan Rajagopalan, James M. Stichnoth, Gansha Wu, and Dan Zhang

Libraries

Associative Parallel Containers in STAPL 156
Gabriel Tanase, Chidambareswaran Raman, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger

Explicit Dependence Metadata in an Active Visual Effects Library 172
Jay L.T. Cornwall, Paul H.J. Kelly, Phil Parsonage, and Bruno Nicoletti

Run-Time Systems and Performance Analysis

Supporting Huge Address Spaces in a Virtual Machine for Java on a Cluster 187
Ronald Veldema and Michael Philippsen

Modeling Relations between Inputs and Dynamic Behavior for General Programs 202
Xipeng Shen and Feng Mao

Evaluation of RDMA Opportunities in an Object-Oriented DSM 217
Ronald Veldema and Michael Philippsen

Automatic Communication Performance Debugging in PGAS Languages 232
Jimmy Su and Katherine Yelick

Parallel Compiler Technology II

Exploiting SIMD Parallelism with the CGiS Compiler Framework 246
Nicolas Fritz, Philipp Lucas, and Reinhard Wilhelm

Critical Block Scheduling: A Thread-Level Parallelizing Mechanism for a Heterogeneous Chip Multiprocessor Architecture 261
Slo-Li Chu

Languages II

Capsules: Expressing Composable Computations in a Parallel Programming Model 276
Hasnain A. Mandviwala, Umakishore Ramachandran, and Kathleen Knobe

Communicating Multiprocessor-Tasks 292
Jörg Dümmler, Thomas Rauber, and Gudula Rünger

General Compiler Techniques

An Effective Automated Approach to Specialization of Code 308
Minhaj Ahmad Khan, H.-P. Charles, and D. Barthou

Flow-Sensitive Loop-Variant Variable Classification in Linear Time 323
Yixin Shou, Robert van Engelen, and Johnnie Birch

Using ZBDDs in Points-to Analysis 338
Ondřej Lhoták, Stephen Curial, and José Nelson Amaral

Author Index 353

Compiler-Enhanced Incremental Checkpointing

Greg Bronevetsky[1], Daniel Marques[2],
Keshav Pingali[2], and Radu Rugina[3]

[1] Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA
greg@bronevetsky.com
[2] Department of Computer Sciences,
The University of Texas at Austin,
Austin, TX 78712, USA
daniel@ices.utexas.edu, pingali@cs.utexas.edu
[3] Department of Computer Science,
Cornell University,
Ithaca, NY 14850, USA
rugina@cs.cornell.edu

Abstract. As modern supercomputing systems reach the peta-flop performance range, they grow in both size and complexity. This makes them increasingly vulnerable to failures from a variety of causes. Checkpointing is a popular technique for tolerating such failures in that it allows applications to periodically save their state and restart the computation after a failure. Although a variety of automated system-level checkpointing solutions are currently available to HPC users, manual application-level checkpointing remains by far the most popular approach because of its superior performance. This paper focuses on improving the performance of automated checkpointing via a compiler analysis for incremental checkpointing. This analysis is shown to significantly reduce checkpoint sizes (upto 78%) and to enable asynchronous checkpointing.

1 Introduction

The dramatic growth in supercomputing system capability from the tera-flop to the peta-flop range has resulted in a dramatic increase in system complexity. While efforts have been made to limit the complexity of the Operating System used by these machines, their component counts have continued to grow. Even as systems like BlueGene/L [3] and the upcoming RoadRunner grow to more than 100,000 processors and tens of TBs of RAM, future designs promise to exceed these limits by large margins. While large supercomputers are made from high-quality components, increasing components counts make them vulnerable to faults, including hardware breakdowns [11] and soft errors [6].

Checkpointing is a popular technique for tolerating failures. The state of the application is periodically saved to reliable storage and on failure, the application rolls back to a prior state. However, automated checkpointing can be very

V. Adve, M.J. Garzarán, and P. Petersen (Eds.): LCPC 2007, LNCS 5234, pp. 1–15, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Revisiting SIMD Programming

Anton Lokhmotov[1,⋆], Benedict R. Gaster[2],
Alan Mycroft[1], Neil Hickey[2], and David Stuttard[2]

[1] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
[2] ClearSpeed Technology
3110 Great Western Court, Bristol, BS34 8HP, UK

Abstract. Massively parallel SIMD array architectures are making their way into embedded processors. In these architectures, a number of identical processing elements having small private storage and using asynchronous I/O for accessing large shared memory executes the same instruction in lockstep.

In this paper, we outline a simple extension to the C language, called C^n, used for programming a commercial SIMD array architecture. The design of C^n is based on the concept of the SIMD array type architecture and revisits first principles of designing efficient and portable parallel programming languages. C^n has a low level of abstraction and can also be seen as an intermediate language in the compilation from higher level parallel languages to machine code.

1 Introduction

Massively parallel SIMD array architectures are no longer merely the province of large supercomputer systems but are making their way into embedded processors. What seemed a cautious extrapolation a decade ago ("even a parallel SIMD coprocessor embedded in a single-user workstation may not be such a far-fetched idea" [1]) has now become a reality.

In the 40 years since the design of Illiac IV [2], the first large-scale array computer consisting of 64 processing elements (PEs), the progress of VLSI technology allows to pack even more PEs into a single-chip microprocessor that in most respects can be considered "embedded". For example, ClearSpeed's CSX600 array processor consisting of 96 PEs has excellent performance per unit power by delivering more than 3 GFLOPS per watt [3].

The CSX is a SIMD array architecture with a control unit and a number of processing elements, with each PE having relatively small private storage and using asynchronous I/O mechanisms to access large shared memory.

In this paper, we describe a data-parallel extension to the C language, called C^n, for programming the CSX architecture. In C^n, parallelism is mainly expressed at the type level rather than at the code level. Essentially, C^n introduces a new multiplicity type qualifier `poly` which implies that each PE has its own copy of a value of that type. For example, the definition `poly int X;` implies that, on the CSX600 with 96 PEs, there

⋆ This author gratefully acknowledges the financial support by a TNK-BP Cambridge Kapitza Scholarship and by an Overseas Research Students Award.

V. Adve, M.J. Garzarán, and P. Petersen (Eds.): LCPC 2007, LNCS 5234, pp. 32–46, 2008.
© Springer-Verlag Berlin Heidelberg 2008

exist 96 copies of integer variable `X`, each having the same address within its PE's local storage.

The multiplicity is also manifested in conditional statements. For example, the following code assigns zero to (a copy of) `X` on every even PE (the runtime function `get_penum()` returns the ordinal number of a PE):

```
if(get_penum()%2 == 0) X = 0;
```

On every odd PE, the assignment is not executed (this is equivalent to issuing a NOP instruction, as the SIMD array operates in lock-step).

We describe designing C^n as an efficient *and* portable language. Efficiency and portability often conflict with each other (especially in parallel languages [4]). We argue that the CSX architecture (§2) is representative of its class and thus can be considered as a *SIMD array type architecture* (§3). Core C^n operations (§4) can be thought of as cheap (in the spirit of C), while more expensive operations are relegated to the standard library. We compare and contrast C^n design decisions with similar approaches (§5), and then argue that C^n can be seen as an *intermediate* language in the compilation from higher level parallel languages (§6). We briefly discuss the C^n compiler implementation (§7) and outline future work in conclusion (§8).

2 CSX Architecture

This section outlines ClearSpeed's CSX architecture [3], which largely honours the classical SIMD array organisation pioneered by the Solomon/Illiac IV designs [5], albeit embodied in a single chip.

2.1 CSX Family

The CSX architecture is a family of processors based on ClearSpeed's multi-threaded array processor (MTAP) core. The architecture has been developed for high rate processing. CSX processors can be used as application accelerators, alongside general-purpose processors such as those from Intel and AMD.

The MTAP consists of execution units and a control unit. One part of the processor forms the *mono* execution unit, dedicated to processing scalar (or mono) data. Another part forms the *poly* execution unit, which processes parallel (or poly) data, and may consist of tens, hundreds or even thousands of identical processing element (PE) cores. This array of PE cores operates in a synchronous, Single Instruction Multiple Data (SIMD) manner, where every enabled PE core executes the same instruction on its local data.

The control unit fetches instructions from a *single* instruction stream, decodes and dispatches them to the execution units or I/O controllers. Instructions for the mono and poly execution units are handled similarly, except for conditional execution. The mono unit uses conditional jumps to branch around code like a standard RISC architecture. This affects both mono and poly operations. The poly unit uses an *enable register* to control execution of each PE. If one or more of the bits of that PE enable register is zero, then the PE core is *disabled* and most instructions it receives will be ignored. The enable register is a stack, and a new bit, specifying the result of a test, can be pushed

onto the top of the stack allowing nested predicated execution. The bit can later be popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops efficient.

In order to provide fast access to the data being processed, each PE core has its own local memory and register file. Each PE core can directly access only its own storage. (Instructions for the poly execution unit having a mono register operand indirectly access the mono register file, as a mono value gets broadcast to each PE.) Data is transferred between PE (poly) memory and the poly register file via load/store instructions. The mono unit has direct access to main (mono) memory. It also uses load/store instructions to transfer data between mono memory and the mono register file. Programmed I/O (PIO) extends the load/store model: it is used for transfers of data between mono memory and poly memory.

2.2 CSX600 Processor

The CSX600 is the first product in the CSX family. The processor is optimised for intensive double-precision floating-point computations, providing sustained 33 GFLOPS of performance on DGEMM (double precision matrix multiply), while dissipating an average of 10 watts. The poly execution unit is a linear array of 96 PE cores, with 6KB SRAM and a superscalar 64-bit FPU on each PE core. The PE cores are able to communicate with each other via what is known as *swazzle path* that connects each PE with its left and right neighbours. Further details can be found in white papers [3].

2.3 Acceleration Example

A C^n implementation of Monte-Carlo simulations for computing European option pricing with double precision performs 100,000 Monte-Carlo simulations at the rate of 206.5M samples per second on a ClearSpeed Advance board having two CSX600 chips. In comparison, an optimised C program using the Intel Math Kernel Library (MKL) and compiled with the Intel C Compiler achieves the rate of 40.5M samples per second on a 2.33GHz dual-core Intel Xeon (Woodcrest, HP DL380 G5 system). Combining both the Intel processor and the ClearSpeed board achieves the rate of 240M samples per second, which is almost 6 times the performance of the host processor alone.

3 C^n Design Goals and Choices

The key design goals of C^n were efficiency and portability. We first discuss these goals in a broader context of programming languages for sequential and parallel computers (§3.1), and then discuss how they affected the design choices of C^n (§3.2).

3.1 Efficiency and Portability

Program efficiency and portability are two common tenets of high-level programming languages. Efficiency means that it is possible to write a compiler generating code that is “almost” as fast as code written in assembly. Portability means that software can be adapted to run on different target systems.

Languages like C have been successful largely because of their efficiency and portability on von Neumann machines, having a single processor and uniform random access memory. Efficiency comes from the programmer's clear understanding of the performance implications of algorithm selection and coding style. Portability is achieved because languages like C hide most features of physical machines, such as instruction set, addressing modes, register file, *etc.* Moreover, the hidden features apparently have a negligible effect on performance, so porting often maintains efficiency [4].

Efficiency and portability are even more desired when programming parallel systems. Performance is the most compelling argument for parallel computing; and given the amount of human effort required to develop an efficient parallel program, the resulting program should better have a long useful life [6].

In the world of parallelism, however, no single model accurately abstracts the variability of parallel systems. While it is possible, for example, to program distributed memory machines using a shared memory model, programmers having limited control over data distribution and communication are unlikely to write efficient programs. So in this world, sadly, efficiency and portability are no longer close friends.

Snyder introduced [7] the notion of a *type architecture*—a machine model abstracting the performance-important features of a family of physical machines, in the same way as the RAM model abstracts von Neumann machines. He proposed the Candidate Type Architecture (CTA) which effectively abstracts MIMD machines (multicomputers) with unspecified interconnect topology. The key CTA abstraction is that accessing another processor's memory is significantly more expensive than accessing local memory (typically by 2–5 orders of magnitude) [4].

3.2 C^n as a Language for the SIMD Array Type Architecture

Our key observation is that the CSX architecture can be considered a *SIMD array type architecture* (SATA), as it respects classical organisation and has typical costs of communicating data between main (mono) and local (poly) memory. Designing a core language based on the type-architecture facilities should provide both efficiency and portability [7]. To match the programmer's intuition, core language operations are cheap, while operations relegated to the standard library are more expensive. For example, arithmetic operations are cheap, while reduction operations (using the inter-PE communication) are somewhat more expensive, albeit still cheaper than data transfer operations between mono and poly memories.

Efficiency mandates only providing language constructs that can be reliably implemented on the SATA using standard compiler technology. History shows that languages that are difficult to implement are also rarely successful (HPF is a dismal example [8]).

Portability is important because the CSX architecture family (§2.1) does not fix the PE array size. Also, as the number of PE cores increases, other (than linear) interconnect topologies could be introduced.

Designing C^n as a C extension provides a known starting point for a large community of C programmers. In addition, software development tools can be written by making (small) modifications to existing ones, rather than from scratch.

4 C^n Outline

In this section we outline the most salient features of C^n. We give further rationale behind some design decisions in §5.

4.1 Types

C^n extends C with two additional keywords that can be used as part of the declaration qualifier for declarators, *i.e.* logically amend the type system [9]. These keywords are *multiplicity qualifiers* and allow the programmer to specify a memory space in which a declared object resides. The new keywords are:

- **`mono`**: for declaring an object in the mono domain (*i.e.* one copy exists in main memory);
- **`poly`**: for declaring an object in the poly domain (*i.e.* one copy per PE in its local memory).

The default multiplicity is **`mono`**. Wherever a multiplicity qualifier may be used, an implicit **`mono`** is assumed, unless an explicit **`poly`** is provided. A consequence of this is that all C programs are valid C^n programs, with the same semantics. Thus, C^n is a superset of C (but see §5.2).

Basic types. C^n supports the same basic types as C. They can be used together with a multiplicity qualifier to produce declarations (only one of the qualifiers can be used in a basic type declaration). Some example declarations follow:

```
poly int i; // multiple copies in PE local (poly) memory
mono unsigned cu; // a single copy in main (mono) memory
unsigned long cs; // a single copy in main (mono) memory
```

Pointer types. The pointer types in C^n follow similar rules to those in C. Pointer declarations consist of a base type and a pointer. The declaration on the left of the asterisk represents the base type (the type of the object that the pointer points to). The declaration on the right of the asterisk represents the pointer object itself. It is possible to specify the multiplicity of either of these entities in the same way as **`const`** and **`volatile`** work in C. For example:

```
poly int * poly sam; // poly pointer to poly int
poly int * frodo;    // mono pointer to poly int
int * poly bilbo;    // poly pointer to mono int
```

Thus, there are four different ways of declaring pointers with multiplicity qualifiers:

- mono pointer to mono object (*e.g.* **`mono int * mono`**);
- mono pointer to poly object (*e.g.* **`poly int * mono`**);
- poly pointer to mono data (*e.g.* **`mono int * poly`**);
- poly pointer to poly data (*e.g.* **`poly int * poly`**).

Note that in the case of a poly pointer, multiple copies of the pointer exist, potentially pointing to different locations.

As in C, pointers are used to access memory. The compiler allocates named poly objects to the same address within each PE local memory. Thus, taking the address of a named object (whether mono or poly) always yields a mono pointer.

Array types. The syntax for array declaration in C^n is similar to that in C. It is possible to use a multiplicity qualifier in an array declaration. Consider the declaration `poly int A[42];`. The multiplicity qualifier applies only to the base type of the array (*i.e.* to the type of array elements). This declaration will reserve a poly space for 42 integers at the same location on each PE. Similar to C, we can say that the array name is coerced to the address of its first element, which is a mono pointer to poly object (*e.g.* `poly int * mono`).

There are some additional restrictions when dealing with arrays, specifically with array subscripting, discussed in §4.3.

Aggregate types. Similar to C, C^n distinguishes between declaration and definition of objects. A declaration is where an object type is specified but no object of that type is created, *e.g.* a struct type declaration. A definition is where an object of a particular type is created, *e.g.* a variable definition. Structs and unions in C^n match their C equivalents, the only difference being the type of fields one can specify inside a struct or union type declaration.

Standard C allows essentially any declaration as a field, with the exception of storage class qualifiers. C^n allows the same with the additional restriction that fields cannot have multiplicity qualifiers. This is because a struct type declaration just specifies the structure of memory. Memory is not allocated until an instance of that struct type is created. Thus, putting a poly field in a struct declaration and then defining an instance of that struct in the mono domain would result in having contradictory multiplicity specifications. (Similarly for a mono field in a struct instance defined in the poly domain.) For example:

```
// legal struct declaration
struct _A {
    int a;
    float b;
};
poly struct _A kaiser;

// illegal struct declaration
struct _B {
    poly int a;   // illegal use of multiplicity
    mono float b; // illegal use of multiplicity
};
mono struct _B king; // where should king.a go?
poly struct _B tsar; // where should tsar.b go?
```

Multiplicity qualifiers, however, can be used on the base type of pointers (otherwise pointers to poly data could not be declared as fields). For example:

```
union _C { // define a union
    poly int * a;
    mono int * poly * b;
};
poly union _C fred;
mono union _C barney;
```

In the declaration of `fred`, the field `a` is created as a poly pointer to a poly int (**`poly int * poly`**) and `b` is created as a poly pointer to a poly pointer to a mono int (**`mono int * poly * poly`**). In the declaration of `barney`, `a` is created as a mono pointer to a poly int (**`poly int * mono`**) and `b` is created as a mono pointer to a poly pointer to a mono int (**`mono int * poly * mono`**).

4.2 Expressions

C^n supports all the operators of C. Mono and poly objects can usually be used interchangeably in expressions (but see §4.3 for exceptions).

Note that the result of any expression involving a mono and a poly object invariably has a poly type. Thus, mono objects are promoted to the poly domain in mixed expressions. In the following example,

```
poly int x; int y; x = x + y;
```

where `y` is promoted to **`poly int`** before being added to (every copy of) `x`. (Technically, the promotion is done by broadcasting values from the mono register file to the poly execution unit.)

4.3 Assignment Statements

Assignment within a domain (*i.e.* poly to poly, or mono to mono) is always legal and has the obvious behaviour.

A mono value can also be assigned to a poly variable. In this case the same value is assigned to every copy of the variable on each PE. For example, **`poly int`** `x = 1;` results in (every copy of) `x` having the value of 1. (Again, the architecture supports such assignments by broadcasting the mono value to the poly unit.)

It is not obvious, however, what should happen when assigning a poly to a mono, *i.e.* when taking data from multiple copies of the poly variable and storing it in the single mono variable. Therefore direct assignment from the poly domain to the mono domain is disallowed in C^n.

Note that certain C^n expressions are disallowed, as otherwise they would require an implicit data transfer from mono to poly memory. One such expression is dereferencing of a poly pointer to mono data (*e.g.* **`mono int * poly`**). Attempting to dereference such a pointer would result in poly data, but the data is stored in the mono domain and would therefore need to be copied to poly memory. Since broadcasting can only send a single mono value to the poly unit at a time, such a copy would involve expensive programmed I/O mechanisms. Therefore, allowing such dereferencing would conflict with the design goal that the core language operations should be cheap. Thus, the compiler reports dereferencing a poly pointer to mono data as an error.

Since, following C, C^n treats `x[i]` as equivalent to `*(x + i)`, it follows that indexing a mono array with a poly expression is also forbidden, because it would implicitly dereference a poly pointer to mono data.

In all the cases when data transfers between mono and poly memories are required, the programmer has to use memory copy routines from the standard C^n library.

4.4 Reduction Operations

Many parallel algorithms require reducing a vector of values to a single scalar value. In addition to the core operations defined above, the C^n standard library provides `sum`, `times`, and bit-wise operations defined for basic types for reducing a poly value into a mono value. For example, on an array of n PEs,

```
poly float x; mono float y; ...
y = reduce_mono_sum(x);
```

means

```
y = x(0) + x(1) + ... + x(n-1);
```

where `x(i)` refers to the value of `x` on ith PE.

For some algorithms another form of reduction is useful: logically, a poly value is reduced to an intermediate mono value which is then broadcast into a result poly value. Thus,

```
poly float x, z; ...
z = reduce_poly_sum(x);
```

means

```
z(0) = ... = z(n-1) = x(0) + x(1) + ... + x(n-1);
```

Both forms can be efficiently implemented on the CSX using the inter-PE swazzle path. The order in which the result is evaluated is unspecified.

4.5 Control Statements

The basic control flow constructs in C^n are the same as in C. Conditional expressions, however, can be of mono or poly domain. Consider the **`if`** statement:

```
if(expression) { statement-list }
```

A mono expression for the condition affects both the mono and poly execution units. If the expression is false, the statement list will be skipped entirely, and execution will continue after the **`if`** statement.

A poly expression for the condition can be true on some PEs and false on others. This is where the PE enable state (described in §2.1) comes in: all the PEs for which the condition is false are disabled for the duration of executing the statement list. The statement list is executed (even if all PEs are disabled by the condition), but any poly statements (*e.g.* assignments to poly variables) have an affect only on the enabled PEs. Mono statements inside a poly **`if`**, however, get executed irrespective of the conditions. Consider the following example:

```
poly int foo = 0; mono int bar = 1;
if(get_penum()%2 == 0) { // disable all odd PEs
  foo = 1; // foo is 1 on even and 0 on odd PEs
  bar = 2; // bar is 2
}
```

Effectively, a poly condition is invisible to any mono code inside that **if** statement.[1] This language design choice may seem counterintuitive and is indeed a controversial point in the design of SIMD languages, to which we return in §5.2.

A poly condition in an **if..else** statement implies that for the **if**-clause all the PEs on which the condition evaluates to true are enabled, and the others are disabled. Then, for the **else**-clause, the enable state of all the PEs is inverted: those PEs that were enabled by the condition are disabled and vice-versa.

Conditional statements can be nested just as in C. Poly statements are only executed on PEs when all the nested conditional expressions evaluate to true.

These rules, of course, dictate different compilation of conditional statements. Mono conditional statements result in generating branch instructions, while poly conditional statements result in generating poly instructions enabling and disabling PEs (enable stack operations on the CSX; see §7 for more details).

Similar principles apply to loop constructs **for**, **while** and **do..while**. A loop with a poly control expression executes until the loop condition is false on every PE. Note that a poly loop can iterate zero times, so in that case, unlike the **if** statement, even mono statements in its body will not be executed.

4.6 Functions

Multiplicity qualifiers can be specified for the return type of a function, as well as the types of any arguments. We refer to a function as being mono or poly according to whether its return type is mono or poly.

A return statement from a mono function behaves exactly as expected by transferring control to the point after the call statement. A poly function does not actually return control until the end of the function body. A return from a poly function works by disabling the PEs which execute it. Other PEs execute the rest of the function code. When the function returns, all PEs have their enable state restored to what it was on entry. Note that all mono code in the function is always executed (unless branched over by mono conditional statements).

5 C^n Design Rationale

5.1 Low-Level Abstraction

Early SIMD programming languages for the Illiac IV computer included Glypnir [10], with syntax based on Algol 60, and CFD [11], based on Fortran. The main design goal of these languages was “to produce a useful, reliable, and efficient programming

[1] It way be worth noting that the block structure of code is still relevant. So, for example, any mono declarations within the body of a poly conditional statement are local to that body.

tool with a high probability of success" [10]. Given the state of compiler technology in the early 1970s, the languages could not *both* be machine independent *and* satisfy this goal.[2] In addition to explicit means for specifying storage allocation and program control as in C^n, these languages even provided means for accessing subwords and registers.

Many vector processing languages have appeared since then, providing higher levels of abstraction for array-based operations (*e.g.* Actus [12], Vector C [13], High Performance Fortran [8], *etc.*). Unfortunately, such languages present greater complexity for the implementors because of data alignment, storage allocation and communication issues on SIMD machines (*e.g.* see [14,15]).

C^n builds on the C strength of "solving" most implementation efficiency problems by leaving them to the programmers. While not the most desirable solution, it relieves the programmers from solving the same problems using assembly language. In §6 we argue that C^n can be seen as an intermediate representation for compiling from higher level parallel languages.

5.2 C^n and Other SIMD Dialects of C

C^n is by no means the first C dialect for SIMD programming. Notable examples include C* [16] for programming the Connection Machine models, MPL [17] for the MasPar models, and 1DC (one-dimensional C) [18] for the IMAP models.

All these languages intentionally reflect their respective architectures. The unifying theme is the use of a keyword to specify multiple instance variables (**`poly`** in C*, **`plural`** in MPL, **`sep`** in 1DC). The language differences stem from the differences between the architectures and design goals.

Communication. While the **`poly`** keyword implies the physical data distribution, C* enshrines the viewpoint that no parallel version of C can abandon the uniform address space model without giving up its claim to be a superset of C. Uniform memory means that a PE can have a pointer `p` into the memory of another PE. For example, the statement `*p = x;` means "send message `x` to a location pointed to by `p`". Thus, C* relies on pointers for interprocessor communication.

This C* feature is underpinned by Connection Machine's key capability allowing any PE to establish a connection with any other PE in the machine (via the global routing mechanism or local meshes). Still, the programmer relies on the compiler writer's ability to implement pointer-based communication efficiently. Perhaps, this is the reason why, in contrast, MasPar's MPL provides explicit communication operators, although the MasPar has connection capabilities similar to the Connection Machine.

The IMAP architectures have a linear nearest neighbour interconnect, as does the CSX. 1DC supports special operators for nearest neighbour communication, while C^n relegates communication functions to the C^n standard library.

Dereferencing pointers. Even rejecting the pointer-based communication in C* does not mean that the compiler will be able to optimise poorly written code; in particular,

[2] Indeed, the compiler for Tranquil—the first Illiac IV language—was abandoned because of implementation difficulties and lack of resources [10].

code that makes a heavy use of costly DMA transfers between mono and poly memories. The C^n ban on dereferencing poly pointers to mono data (which would transfer only several bytes at a time) aims to shut the door in front of the abusing programmers. In contrast, MPL does not sacrifice convenience for efficiency and allows all pointer operations.

Poly conditional statements. In §4.5, we discussed the behaviour of poly **`if`** statements in C^n: even if the poly condition is false on every PE, the statement list is executed regardless, including all mono statements in the list. This model of execution is easy to implement on the CSX by inserting operations on the hardware enable stack (see §7). The same model was used in MPL [19].

The designers of C* followed the other route by adopting the "Rule of Local Support" [16, §6.2]: if the poly condition is false on every PE, then the statement list is not executed at all. The Rule of Local Support required extra implementation trouble but preserved the usual model of a **`while`** loop in terms of **`if`** and **`goto`** statements. The C* designers, nevertheless, admitted that their rule occasionally also caused inconvenience and confusion.

Deciding on whether to preserve or to change the semantics of familiar statements when extending a sequential language for parallelism is hard, and may even drive the designers to a thought that designing a language from scratch is a better idea (*e.g.* see the history of ZPL [4]). In the case of **`if`** statements, the solution does not need to be that radical and could merely come as using different keywords in a more general language supporting both execution models (for example, **`ifp`** for the C^n/MPL model and **`ifm`** for the C* model).

6 C^n as Intermediate Language

SIMD array supercomputers went out of fashion in the early 1990s when clusters of commodity computers proved to be more cost effective. Ironically, vector-style instructions reemerged in commodity processors under the name of SIMD (or multimedia) extensions, such as Intel MMX/SSE and PowerPC AltiVec.

The principal difference between "real" (array) SIMD and vector extensions is that the latter operate on data in long registers and do not have local memory attached to each PE as in SIMD arrays. In terms of the C^n language, this means that it would be inefficient (if not impossible) to use pointers to poly data when programming in C^n for vector extensions. This is because it would not be straightforward to compile C^n programs expressing memory operations that are not supported by the hardware.

We argue, however, that the C^n language can be regarded as a "portable assembly" language for both SIMD and vector architectures. Given an architecture description, high-level languages providing more abstract array-based operations can be translated to C^n and then efficiently mapped to machine code by the C^n compiler.

For example, consider a Fortran 90 style statement `B[0:959] = A[0:959]+42;` where `A[]` and `B[]` are arrays of floats. This statement loads 960 values from `A`, adds 42 to each, and stores 960 results to `B`. Suppose that the target architecture is a vector machine with 96 elements per vector register, hence the statement parallelism needs to be folded 10 times to fit the hardware by *strip mining* [20,15].

For a vector machine, the declaration `poly float va, vb;` can be thought of as defining two vector registers in a similar way as `vector float vc;` can be used to define a (4-element) vector in the AltiVec API [21]. The vector statement can then be strip-mined *horizontally* [15], resulting in:

```
const poly int me = get_penum(); // get PE number
mono float * poly pa = A + me;
mono float * poly pb = B + me;
for(int i = 0; i < 10; i++, pa += 96, pb += 96) {
  va  = *pa; // load 96-element vector
  vb  = va + 42;
  *pb = vb; // store 96-element vector
}
```

(Note that here we have lifted the C^n ban on dereferencing poly pointers to mono data, which is the artefact of distributed memory SIMD arrays.)

The same strategy works on a SIMD machine having 96 PEs: 10 vector indirect loads (one on each iteration) are replaced with 10 DMA transfers from mono to poly memory, 10 vector stores with 10 DMA transfers from poly to mono memory; each transfer moves a single float to/from a PE. It is more efficient, however, to strip-mine *vertically* [15], which in pseudo-vector notation can be written as:

```
poly float pA[10], pB[10];
pA[0:9] = A[10*me:10*me+9]; // DMA mono to poly
pB[0:9] = pA[0:9] + 42;
B[10*me:10*me+9] = pB[0:9]; // DMA poly to mono
```

This requires only two DMA transfers of 10 floats each. Given high DMA start-up costs, vertical strip-mining is several times more efficient. Hence, the compiler should favour the second form on SIMD machines. (This is a trivial example of the decisions we referred to in §5.1 that the compiler needs to make to efficiently compile high-level vector abstractions).

To summarise, C^n code can be seen as an intermediate language, from which target code can be generated. Once a C^n compiler is written and well-tuned, the quality of target code should principally depend on the ability of the (machine-description driven) front-end to generate efficient intermediate code from a higher level parallel language. Even if the front-end generates suboptimal intermediate code, the performance-conscious programmer still has a fallback to the intermediate language, rather than to assembly.

C^n has already been targeted from a subset of OpenMP [22]. We believe that C^n can also be targeted from high-level data parallel libraries such as MSR Accelerator [23] and memory hierarchy oriented languages such as Stanford Sequoia [24].

7 C^n Compiler Implementation

ClearSpeed has developed a C^n optimising compiler for the CSX architecture using the CoSy compiler development framework [25]. Small modifications to the front-end

were needed to support the multiplicity qualifiers in the source code. Each type is supplemented with a flag indicating whether it is mono or poly. A special phase was written to recognise poly conditional statements and transform them into a form of predicated execution. For example, assuming that both `x` and `y` are in the poly domain,

```
if (y > 0) {
  x = y;
}
```

becomes

```
enablestate = push(y > 0, enablestate);
x ?= y;
enablestate = pop(enablestate);
```

where the predicated assignment operator `?=` assigns its *rhs* expression to the *lhs* location only on those PEs where all the bits of the hardware enable stack (represented by the variable `enablestate`) are 1.

Predicated execution requires careful rethinking of standard optimisations. For example, the standard register allocation algorithm via coloring has to recognise that liveness of a virtual poly register is not simply implied by its def-use chain but is also a function of the enable state. This is not, however, a new problem (*e.g.* see [26]).

7.1 C^n Compiler Performance

ClearSpeed has developed a number of interesting applications in C^n, including functions from the molecular dynamics simulation package AMBER, Monte-Carlo option pricing simulations, implementations of 2D and 3D FFTs, and others.

The C^n design choice of only including features that can be efficiently implemented using standard compiler technology pays off handsomely, since the implementors can concentrate their efforts on optimisations that the programmer expects the compiler to get right. For example, Fig. 1 shows on 20 benchmarks the performance improvement achieved by the version 3.0 of the C^n compiler over the version 2.51.

Much of the improvement comes from the work on the register allocator and other optimisations becoming aware of poly variables. Particular optimisations, *e.g.* loop-invariant code motion, have proved to benefit from the target processor's ability to issue *forced* poly instructions which are executed regardless of the enable state.

Code for the `amber` benchmark generated by the current version of the compiler performs within 20% of hand-coded assembly. (No other similar data is available for comparison, because it makes little sense to re-implement assembly programs in C^n, other than to improve portability.)

8 Future Work and Conclusion

Programming in C^n tends to require the programmer to restructure programs to expose the parallelism lost when writing in a sequential language. Essentially, the C^n programmer indicates a parallel loop. We believe it is possible to ease the programmer's task of annotating sequential C programs (but expressing parallel algorithms) with the **`poly`** qualifier, if the programmer can assert certain properties.

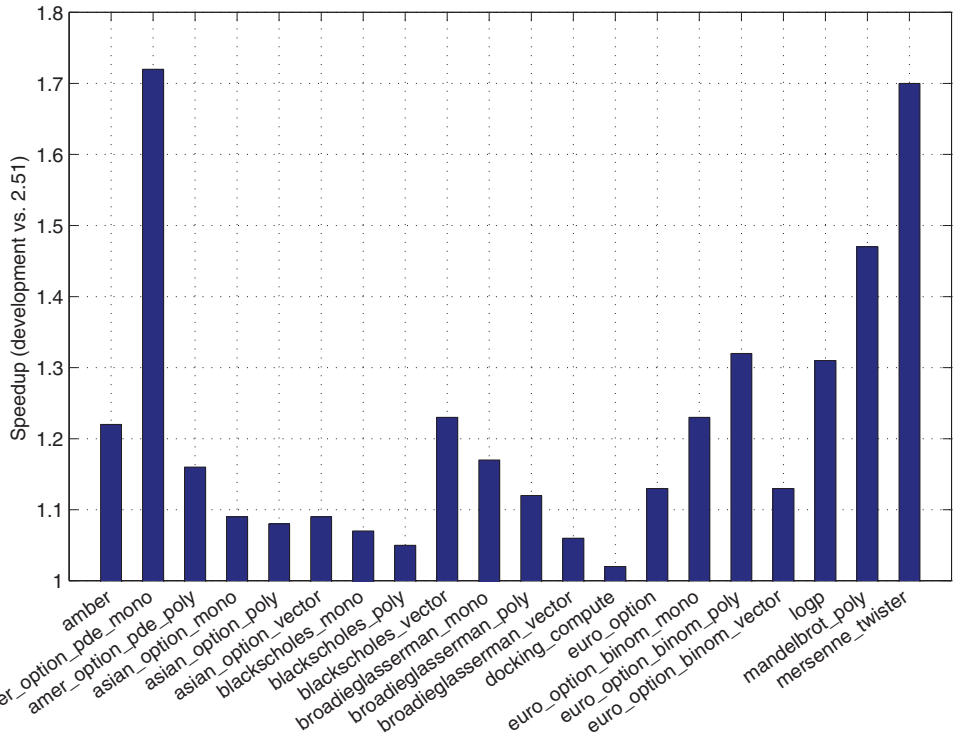


Fig. 1. Performance improvement of code generated by the compiler version 3.0 over code generated by the compiler version 2.51

We are developing an auto-parallelising compiler module converting C code to valid C^n code using programmer's annotations based on the concept of delayed writes [27]. The resulting C^n code is then fed into the optimisation and code generation phases of the C^n compiler.

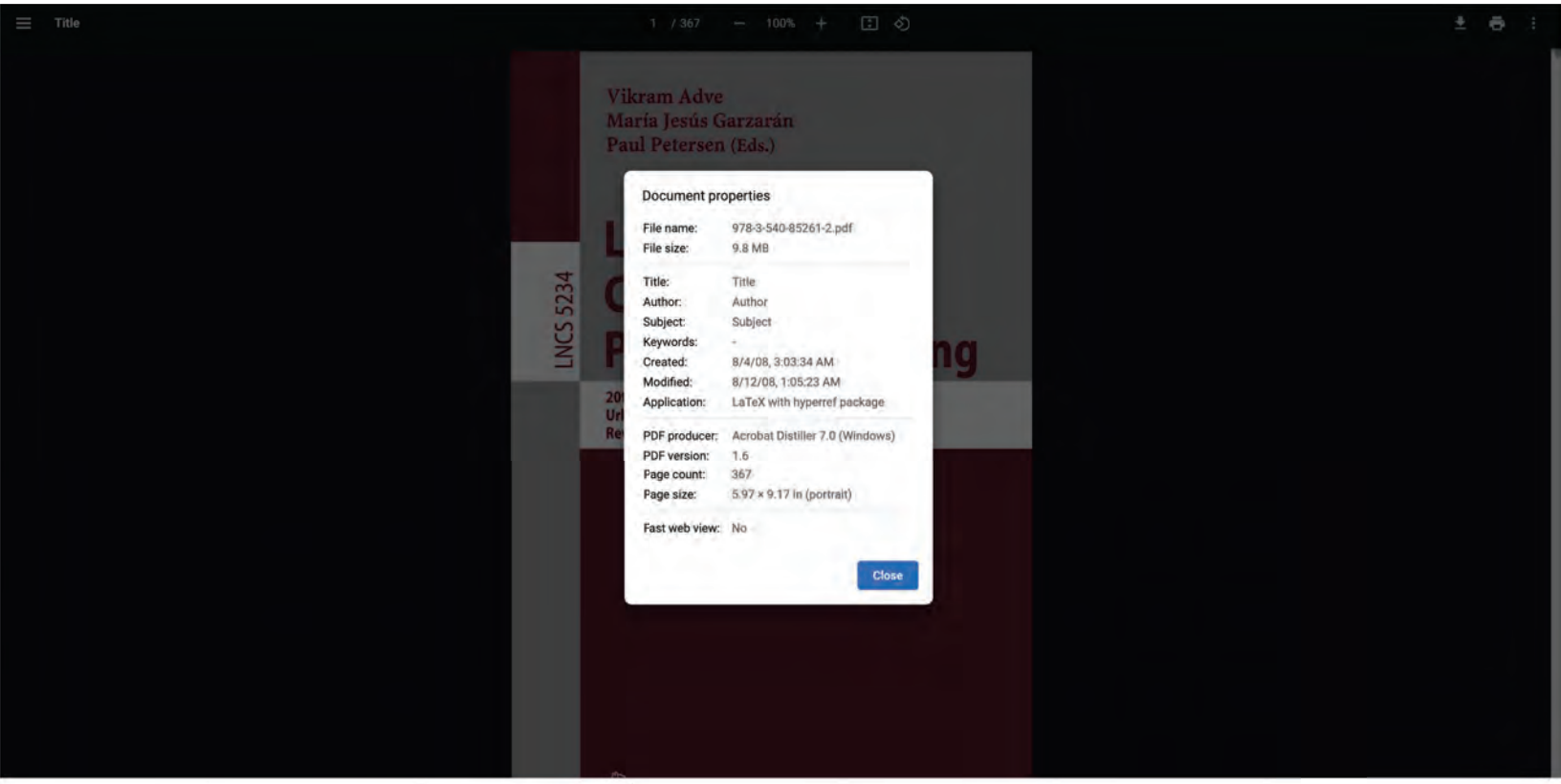
ClearSpeed has developed a set of production quality tools targeting the CSX array architecture, including an optimising compiler, assembler, linker and debugger, that make software development in C^n a viable alternative to hand-coding in assembly as loss in performance is far outweighed by the advantages of high-level language programming.

References

1. Parhami, B.: SIMD machines: do they have a significant future? SIGARCH Comput. Archit. News 23(4), 19–22 (1995)
2. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A.: The Illiac IV computer. IEEE Trans. Computers C-17(8), 746–757 (1968)
3. ClearSpeed Technology: The CSX architecture, `http://www.clearspeed.com/`
4. Snyder, L.: The design and development of ZPL. In: Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III), pp. 8–1–8–37. ACM Press, New York (2007)
5. Slotnick, D.: The conception and development of parallel processors—a personal memoir. IEEE Annals of the History of Computing 4(1), 20–30 (1982)

6. Wilkes, M.V.: The lure of parallelism and its problems. In: Computer Perspectives. Morgan Kaufmann, San Francisco (1995)
7. Snyder, L.: Type architecture, shared memory and the corollary of modest potential. Annual Review of Computer Science 1, 289–317 (1986)
8. Kennedy, K., Koelbel, C., Zima, H.: The rise and fall of High Performance Fortran: an historical object lesson. In: Proc. of the third ACM SIGPLAN conference on History of programming languages (HOPL III), pp. 7–1–7–22. ACM Press, New York (2007)
9. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages – C (1999)
10. Lawrie, D.H., Layman, T., Baer, D., Randal, J.M.: Glypnir—a programming language for Illiac IV. Commun. ACM 18(3), 157–164 (1975)
11. Stevens Jr., K.: CFD—a Fortran-like language for the Illiac IV. SIGPLAN Not. 10(3), 72–76 (1975)
12. Perrott, R.H.: A language for array and vector processors. ACM Trans. Program. Lang. Syst. 1(2), 177–195 (1979)
13. Li, K.C., Schwetman, H.: Vector C: a vector processing language. Journal of Parallel and Distributed Computing 2(2), 132–169 (1985)
14. Knobe, K., Lukas, J.D., Steele Jr., G.L.: Data optimization: allocation of arrays to reduce communication on SIMD machines. J. Parallel Distrib. Comput. 8(2), 102–118 (1990)
15. Weiss, M.: Strip mining on SIMD architectures. In: Proc. of the 5th International Conference on Supercomputing (ICS), pp. 234–243. ACM Press, New York (1991)
16. Rose, J.R., Steele Jr., G.L.: C*: An extended C language for data parallel programming. In: Proc. of the 2nd International Conference on Supercomputing (ICS), vol. 2, pp. 2–16 (1987)
17. MasPar Computer Corporation: MasPar Programming Language (ANSI C compatible MPL) Reference Manual (1992)
18. Kyo, S., Okazaki, S., Arai, T.: An integrated memory array processor for embedded image recognition systems. IEEE Trans. Computers 56(5), 622–634 (2007)
19. Christy, P.: Software to support massively parallel computing on the MasPar MP-1. In: Proc. of the 35th IEEE Computer Society International Conference (Compcon Spring), pp. 29–33 (1990)
20. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
21. Freescale Semiconductor: AltiVec technology programming interface manual (1999)
22. Bradley, C., Gaster, B.R.: Exploiting loop-level parallelism for SIMD arrays using OpenMP. In: Proc. of the 3rd International Workshop on OpenMP (IWOPM) (2007)
23. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program GPUs for general-purpose uses. In: Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 325–335. ACM Press, New York (2006)
24. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Proc. of the 2006 ACM/IEEE Conference on Supercomputing (SC), pp. 83–92. ACM Press, New York (2006)
25. ACE Associated Compiler Experts: The CoSy compiler development system, `http://www.ace.nl/`
26. Kim, H.: Region-based register allocation for EPIC architectures. PhD thesis, Department of Computer Science, New York University (2001)
27. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641. Springer, Heidelberg (2007)

ATTACHMENT 15D


Title
1 / 367
100%
Vikram Adve
María Jesús Garzarán
Paul Petersen (Eds.)
LNCS 5234
Document properties
File name: 978-3-540-85261-2.pdf
File size: 9.8 MB
Title: Title
Author: Author
Subject: Subject
Keywords: -
Created: 8/4/08, 3:03:34 AM
Modified: 8/12/08, 1:05:23 AM
Application: LaTeX with hyperref package
PDF producer: Acrobat Distiller 7.0 (Windows)
PDF version: 1.6
Page count: 367
Page size: 5.97 × 9.17 in (portrait)
Fast web view: No
Close

ATTACHMENT 15E

All locations : No holdings in LDL - 24 other holdings

Hutchison, David. Languages and Compilers for Parallel Computing : 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers / edited by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Vikram Adve, María Jesús Garzarán, Paul Petersen. Berlin, Heidelberg : Springer-Verlag Berlin Heidelberg, 2008. 1 online resource OCLC:299078541

Holdings Information

STATE	LOCATIONS
NH	NHM
NJ	NJR
NY	COO zcy
OH	OCWMS
SC	SHC
BC	VT2
NL	QE2
ON	CEF
DE	defro dehmf dehtw demab detwd du#
GB	bwn dct KIJ ukahl
MY	mypmp
NZ	UX0 UX1 Z5A
SG	sintu

CLOSE WINDOW

EXHIBIT B

Mary K. Bolin, PhD
2433 U Street
Lincoln, Nebraska 68503
mary@marykbolin.com

Mary K. Bolin, PhD

Professor Emeritus
University Libraries
University of Nebraska—Lincoln
mbolin2@unl.edu

Lecturer,
School of Information
San José State University
mary.bolin@sjsu.edu

EDUCATION

PhD, Educational Studies (Educational Leadership and Higher Education), 2007, University of Nebraska

Dissertation: *A Typology and Discourse Analysis of the Status and Appointments of Librarians at Land Grant Universities* (available at digitalcommons.unl.edu)

MA, English, 1999, University of Idaho
Thesis: *Grace: A Contrastive Analysis of a Biblical Semantic Field* (available at digitalcommons.unl.edu)

MSLS, 1981, University of Kentucky

BA, Linguistics, 1976, University of Nebraska—Lincoln

WORK EXPERIENCE

Center for Digital Research in the Humanities. Fellow, 2019- Emeritus, 2020-

University of Nebraska--Lincoln Libraries: Professor and Catalog and Metadata Librarian, Digital Initiatives and Special Collections, 2015-2020. Emeritus, 2020-

University of Nebraska—Lincoln Libraries: Professor and Chair, Discovery and Resource Management, 2004-2015; Coordinator of Collection Development, 2012-2015

University of Idaho Library: Head, Technical Services Department, 1993-2003. Promoted to Professor in 1999

University of Idaho Library: Head, Cataloging Department, 1986-1993. Promoted to Associate Professor and tenured, 1991

University of Georgia Libraries: Head, Records Maintenance Section, Cataloging Department, 1981-1986

Teaching:
San José State University School of Information:

LIBR 281, Metadata. 2009-
LIBR 248, Introduction to Cataloging and Classification. 2013-
LIBR 202, Information Retrieval. 2008-2013

American Library Association

Cataloging for Non-Catalogers, Online Continuing Education Course, 2017-
Intermediate Cataloging for Non-Catalogers, Online Continuing Education Course, 2018-

Lyrasis

Webinar on BibFrame, taught twice a year, 2020=

Independent Study in Idaho

Library Science C415, Technical Services for Small Libraries, 2004-2013
Library Science C416, Library Services to Rural Libraries, 2004-2013
Library Science C422, Use of the School Library, 1999-2013
Library Science C425/525, Organization and Management of Small Libraries, 1988-2013
Library Science C427, Library or Media Center Practicum, 1995-2013

Gateway PhD Program (San Jose State University and Queensland University of Technology). SJSU supervisor for two PhD students, 2015-

Gateway PhD Program (San Jose State University and Manchester Metropolitan University). Associate Coordinator, 2021-

Consulting
Expert witness for multiple attorneys employed by Wolf, Greenfield & Sacks, P.C., Hughes Hubbard & Reed LLP, and other firms. 2017-

Editing
Language and academic editing for EditSprings, a service for international scholars, 2020-

SCHOLARSHIP

Refereed Publications:

(Publications available from digitalcommons.unl.edu)

Bolin, Mary K. (2022). "Academic Libraries and Writing Centers: Collaborations at US Public Research Universities." *School of Information Student Research Journal, 11*(2). Retrieved from https://scholarworks.sjsu.edu/ischoolsrj/vol11/iss2/2

Bolin, Mary K. (Forthcoming 2022). *Organizational Transformation in Academic Libraries: Discourse, Process, Product*. Oxford, UK: Chandos/Elsevier.

Bolin, Mary K. (Forthcoming). "Metadata, Catalog Systems, Linked Data, and the Evolving ILS." In: Sandra Hirsh (Ed.), *Information Services Today: An Introduction.* 3rd ed. New York: Rowman and Littlefield.

Pytlik Zillig, Brian, and Mary K. Bolin. (Forthcoming) "Visualization of Semantic Primes in Texts from Early Modern English Drama." *Journal of the Japanese Association for the Digital Humanities.*

Wintermute, Harriet E., and Bolin, Mary K. (2018). "Metadata Generators for Backlog Reduction: Metadata Maker Streamlines Cataloging and Facilitates Transition. *Technical Services Quarterly 35*(2) doi: https://doi.org/10.1080/07317131.2018.1422879

Bolin, Mary K. (2018). "Metadata, Catalog Systems, Linked Data, and the Evolving ILS." In: Sandra Hirsh (Ed.), *Information Services Today: An Introduction.* 2nd ed. New York: Rowman and Littlefield.

Bolin Mary K. (2017). *The 21-century Academic Library: Global Patterns of Organization and Discourse*. Oxford: Chandos Publishing. (Chandos Information Professional Series)

Bolin, Mary K. (2017). Linguistics and LIS: A Research Agenda." *SLIS Student Research Journal*, 7:1.
Retrieved from http://scholarworks.sjsu.edu/slissrj/vol7/iss1/2

Bolin, Mary K. (2014). "Open Access Web Resources for Library Continuing Education and Training." In S. Hines (Ed.), *Revolutionizing the Development of Library and Information Professionals: Planning for the Future* (pp. 107- 124). Hershey, PA: . doi:10.4018/978-1-4666-4675-9.ch008.

Bolin, Mary K. (2014). "The Language of Academic Librarianship: The Discourse of Promotion and Tenure. "*Advances in Library Administration and Organization 32*.

Bolin, Mary K. "Librarian Status at US Research Universities: Extending the Typology." *Journal of Academic Librarianship* 34:4 (September 2008)

Bolin, Mary K. "A Typology of Librarian Status at Land Grant Universities." *Journal of Academic Librarianship*, 34: 3 (May 2008)220:230.

Bolin, Mary K. "The Library and the Computer Center: Organizational Patterns at Land Grant Universities." *Journal of Academic Librarianship*, 31:1 (2005)

Eckwright, Gail Z. and Mary K. Bolin. "The Hybrid Librarian: The Affinity of Collection Management with Technical Services and the Organizational Benefits of an Individualized Assignment." *Journal of Academic Librarianship*, 27:6 (2001)452:457.

Bolin, Mary K. "Catalog Design, Catalog Maintenance, Catalog Governance," L*ibrary Collections, Acquisitions, and Technical Services.* 24: (2000)53-63.

Bolin, Mary K. "The Collegial Environment and the Functional Organization," *Journal of Library Administration*. 29:2(1999):49-61.

Bolin, Mary K. "Cataloging Environmental Impact Statements and Related Documents," *Resource Sharing and Library Networks*. 14:1(1999): 21-40.

Bolin, Mary K. "Waiting for the Electrician or Someone like Him: Client-Centered Technical Services" Proceedings of the Association of College and Research Libraries National Conference, Nashville, April 1997.
< http://www.ala.org/Content/ContentGroups/ACRL1/Nashville_1997_Papers/Bolin.htm >

Grants:

Electronic Theses and Dissertations. Seed Grant from University of Idaho Office of Research, 2002. Co PI with Gail Z. Eckwright. $6500.

Interoperability of Metadata for Thematic Research Collections. Grant to the Center for Digital Research in the Humanities from the Institute of Museum and Library Services 2005-2008. Project Team Member.
http://unlcms.unl.edu/cas/center-for-digital-research-in-the-humanities/projects/pages/iom_files/IOM_Final_Narrative_Report_July_2008.pdf

Other Publications:

Library Philosophy & Practice. Peer-reviewed electronic library science journal. Mary K. Bolin and Gail Z. Eckwright, founders and editors. http://digitalcommons.unl.edu/libphilprac

Library Science C415, Technical Services for Small Libraries, through Independent Study in Idaho. 2004.

Library Science C416, Library Services to Rural Libraries, through Independent Study in Idaho. 2004.

Library Science C422, Use of the School Library, through Independent Study in Idaho. Revised 1999.

Library Science C425/525, Organization and Management of Small Libraries through Independent Study in Idaho. Revised 2005.

Library Science C427, Library or Media Center Practicum, Independent Study in Idaho. Revised 1999.

Presentations:

Bolin, Mary K. “BibFrame and the MARC Cataloging Environment: an Introduction.” Lyrasis Webinar, June 16, 2020, August 17, 2021.. Recording available http://lyrasis.adobeconnect.com/paufo9e0dm30/

Pytlik Zillig, Brian, and Mary K. Bolin. “A Little More than Kin, a Little Less than Kind”: the Semantic Field Grace in Early Modern English" Poster Session, Text Encoding Initiative Meeting, Graz, Austria, September 18, 2019

Pytlik Zillig, Brian, and Mary K. Bolin. “Shakespeare and the Enumeration of Semantic Primes.” Research Paper Presentation, Text Encoding Initiative Meeting, Tokyo, Japan, September 13, 2018

"Library Philosophy and Practice: Creating and Maintaining an Open-Access E-Journal," Presentation to UNCL meeting (libraries at campuses of the University of Nebraska), Lincoln, Nebraska, March 20, 2008.

Presentation to "Thinking outside the Borders" participants on the open-access electronic journal Library Philosophy and Practice and its role in the international library community. University of Nebraska—Lincoln Libraries, October 24, 2007

Panel discussion on open access electronic journals. GWLA directors' meeting. Linda K. Hall Library, Kansas City, Missouri. April 2007.

"Waiting for the Electrician or Someone Like Him: Client-Centered Technical Services." Association of College and Research Libraries National Conference, Nashville, April 5, 1997.

" 'Some Problems of an Editor': the Idaho Librarian from Manuscript to Published Issue," Joint Annual Conference of the Idaho Library Association and the Washington Library Association, Spokane, Washington, April 6, 1995.

"Automating Idaho's Libraries: a Historical Perspective." Idaho Library Association Annual Conference, Idaho Falls, October 4, 1990.

"Small Library Management." Idaho Library Association Spring Conference, Lewiston, Idaho, May 13, 1989.

"No More 'Good Workers': Excellence and Academic Library Classified Staff." Association of College and Research Libraries 5th Annual Conference, Cincinnati, April 7, 1989.

"Communication to and from Cataloging Departments." Heads of Cataloging Discussion Group, American Library Association Midwinter Meeting, San Antonio, January 1988.

Discussion Leader at the LAMA (Library Administration and Management) Management Exchange. American Library Association Annual Meeting, San Francisco, June 1987.

"Staff Turnover." Heads of Cataloging Discussion Group, American Library Association Midwinter Meeting, Chicago, January 1987.

SELECTED PROFESSIONAL SERVICE

Reviewer. *Resources for College Libraries*. Linguistics bibliography, Spring 2020.

Peer reviewer for ALISE National Conference presentations, Spring 2020.

Peer reviewer for *Journal of Academic Librarianship*, 2012-

Peer reviewer for refereed journal, *Western Nigerian Journal of Medical Sciences* (http://www.wnjms.org/), a publication of Ladoke Akintola University of Technology (LAUTECH) Teaching Hospital, 2008-

PhD Reviewer for students from Aligarh Muslim University, India, 2010-

Outside reviewer for numerous library faculty promotion and tenure cases in US universities.

University of Nebraska--Lincoln

Chair, University Libraries Faculty, 2018

Faculty Senate Academic Rights and Responsibilities Committee, 2016- Chair, 2018-19

Faculty Senate – 2004-2008; 2008-2011

Faculty Senate Executive Committee 2004-2008

Academic Planning Committee (APC), 2005-2008; Vice Chair, 2006-2007; Chair, 2007-2008

Aesthetic Review Committee 2006-2007

APC monitor for academic program reviews: Department of Political Science, 2005; Department of Educational Psychology 2006

Pacific Northwest Library Association:

Editor, *PNLA Quarterly*, December 2002-December 2013

Idaho Library Association:

Editor, *The Idaho Librarian*, September 1990-January 1996