

# Integrated Circuits and Systems

## **Series Editor**

A. Chandrakasan  
Massachusetts Institute of Technology  
Cambridge, Massachusetts

For further volumes:  
<http://www.springer.com/series/7236>

|                                           |
|-------------------------------------------|
| Google Exhibit 1124<br>Google v. Singular |
|-------------------------------------------|

Stephen W. Keckler · Kunle Olukotun ·  
H. Peter Hofstee  
Editors

# Multicore Processors and Systems

 Springer

*Editors*

Stephen W. Keckler  
The University of Texas at Austin  
1 University Station, C0500  
Austin, TX 78712-0233  
USA  
skeckler@cs.utexas.edu

Kunle Olukotun  
Department of Electrical Engineering and  
Computer Science  
Stanford University  
Stanford, CA 94305-9030  
USA  
kunle@stanford.edu

H. Peter Hofstee  
IBM Systems and Technology Group  
11500 Burnet Rd.  
Austin, TX 78758  
USA  
hofstee@us.ibm.com

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)

# Contents

|                                                                                                                                                                                                                                                                            |     |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| <b>1 Tiled Multicore Processors</b> .....                                                                                                                                                                                                                                  | 1   |
| Michael B. Taylor, Walter Lee, Jason E. Miller, David Wentzlaff,<br>Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul R. Johnson,<br>Jason S. Kim, James Psota, Arvind Saraf, Nathan Shnidman,<br>Volker Strumpfen, Matthew I. Frank, Saman Amarasinghe, and<br>Anant Agarwal |     |
| <b>2 On-Chip Networks for Multicore Systems</b> .....                                                                                                                                                                                                                      | 35  |
| Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal                                                                                                                                                                                                                       |     |
| <b>3 Composable Multicore Chips</b> .....                                                                                                                                                                                                                                  | 73  |
| Doug Burger, Stephen W. Keckler, and Simha Sethumadhavan                                                                                                                                                                                                                   |     |
| <b>4 Speculatively Multithreaded Architectures</b> .....                                                                                                                                                                                                                   | 111 |
| Gurindar S. Sohi and T.N. Vijaykumar                                                                                                                                                                                                                                       |     |
| <b>5 Optimizing Memory Transactions for Multicore Systems</b> .....                                                                                                                                                                                                        | 145 |
| Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha                                                                                                                                                                                                                |     |
| <b>6 General-Purpose Multi-core Processors</b> .....                                                                                                                                                                                                                       | 173 |
| Chuck Moore and Pat Conway                                                                                                                                                                                                                                                 |     |
| <b>7 Throughput-Oriented Multicore Processors</b> .....                                                                                                                                                                                                                    | 205 |
| James Laudon, Robert Golla, and Greg Grohoski                                                                                                                                                                                                                              |     |
| <b>8 Stream Processors</b> .....                                                                                                                                                                                                                                           | 231 |
| Mattan Erez and William J. Dally                                                                                                                                                                                                                                           |     |
| <b>9 Heterogeneous Multi-core Processors: The Cell<br/>Broadband Engine</b> .....                                                                                                                                                                                          | 271 |
| H. Peter Hofstee                                                                                                                                                                                                                                                           |     |
| <b>Index</b> .....                                                                                                                                                                                                                                                         | 297 |

## 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

### 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 <sup>2</sup> ) | Core size (mm <sup>2</sup> ) | Normalized to 65 nm |
|--------------------------|------------|--------------|-----------------------------|------------------------------|---------------------|
| <b>Bulldozers</b>        |            |              |                             |                              |                     |
| 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                |
| <b>Chainsaws</b>         |            |              |                             |                              |                     |
| 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                 |
| <b>Termites</b>          |            |              |                             |                              |                     |
| 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  $20 \text{ mm}^2$  in 65 nm technology as “bulldozer cores,” and cores larger than  $5 \text{ mm}^2$  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  $2.4 \text{ mm}^2$  at 65 nm) is in the size range of termite cores, four can be aggregated into a logical  $9.5 \text{ mm}^2$  core, and even 16 can be aggregated to form a logical  $37.9 \text{ mm}^2$  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.

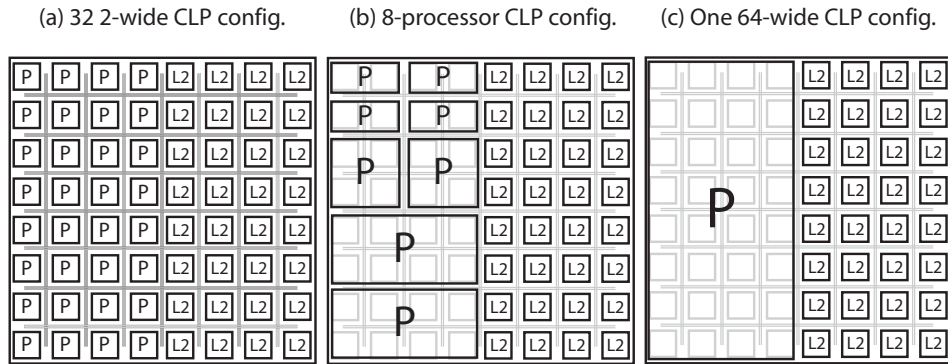


**Fig. 3.1** Three dynamically assigned CLP configurations

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  $1.4\times$  performance improvement,  $3.4\times$  performance/area improvement, and  $2.0\times$  performance<sup>2</sup>/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 executing 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.

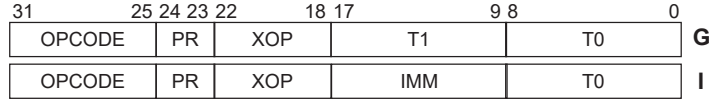
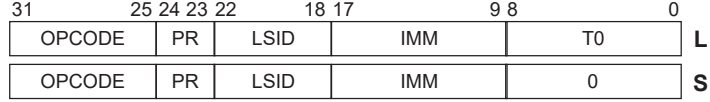
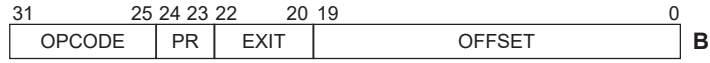
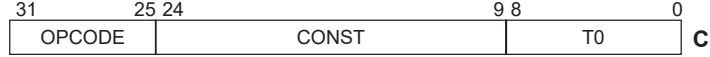
The header chunk also holds three types of control state for the block: a 32-bit “store mask” that indicates which of the possible 32 memory instructions are stores, block execution flags that indicate the execution mode of the block, and the number of instruction “body” chunks in the block. The store 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

**General Instruction Formats****Load and Store Instruction Formats****Branch Instruction Format****Constant Instruction Format****Read Instruction Format****Write Instruction Format****INSTRUCTION FIELDS**

OPCODE = Primary Opcode  
 XOP = Extended Opcode  
 PR = Predicate Field  
 IMM = Signed Immediate  
 T0 = Target 0 Specifier  
 T1 = Target 1 Specifier  
 LSID = Load/Store ID  
 EXIT = Exit Number  
 OFFSET = Branch Offset  
 CONST = 16-bit Constant  
 V = Valid Bit  
 GR = General Register Index  
 RT0 = Read Target 0 Specifier  
 RT1 = Read Target 1 Specifier

**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

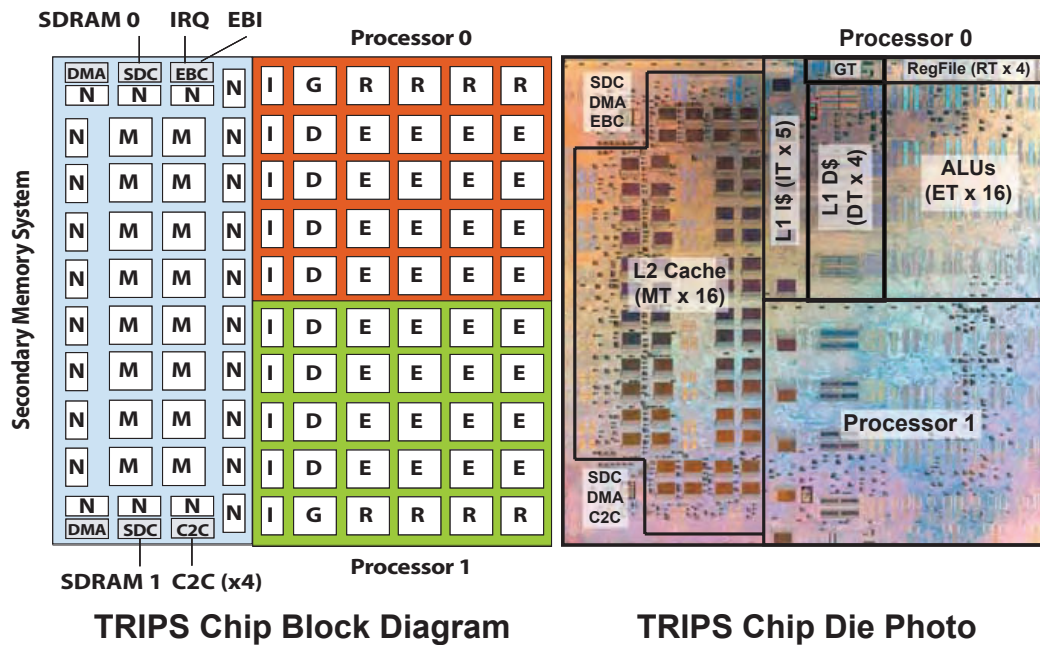


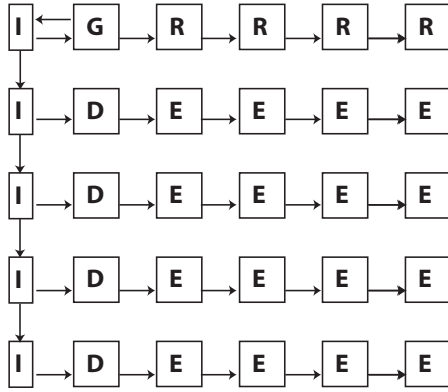
**Fig. 3.3** TRIPS prototype block diagram

TRIPS prototype. The three major components on the chip are two processors and the secondary memory system, each connected internally by one or more micromet-

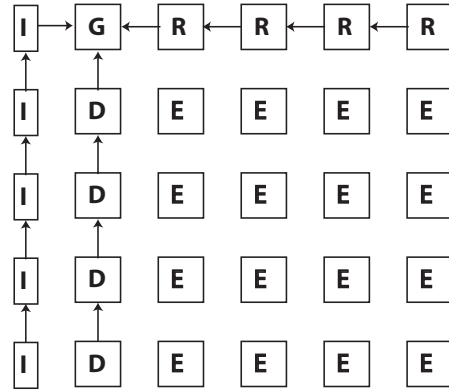
Each of the processor cores is implemented using five unique tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The main processor core micronetwork is the operand network (OPN), shown in Fig. 3.4. It connects all of the tiles except for the ITs in a two-dimensional, wormhole-routed,  $5 \times 5$  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.

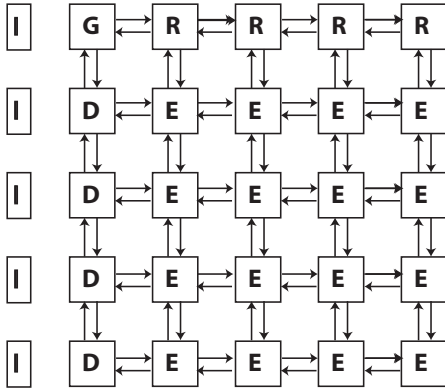
This type of tiled microarchitecture is *composable* at design time, permitting different numbers and topologies of tiles in new implementations with only moderate changes to the tile logic, and no changes to the software model. The particular

**Global dispatch network (GDN)**

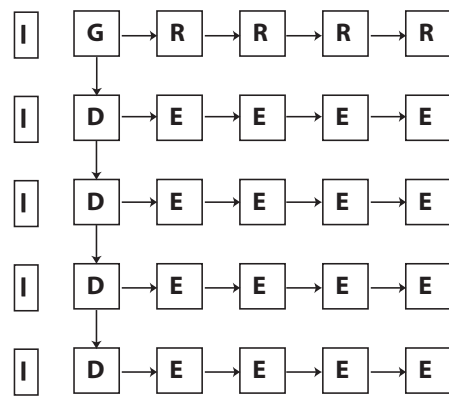
Issues block fetch command and  
dispatches instructions

**Global status network (GSN)**

Signals completion of block execution, l-cache  
miss refill, and block commit completion

**Operand network (OPN)**

Handles transport of all data operands

**Global control network (GCN)**

Issues block commit and block flush commands

**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  $4 \times 10$  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

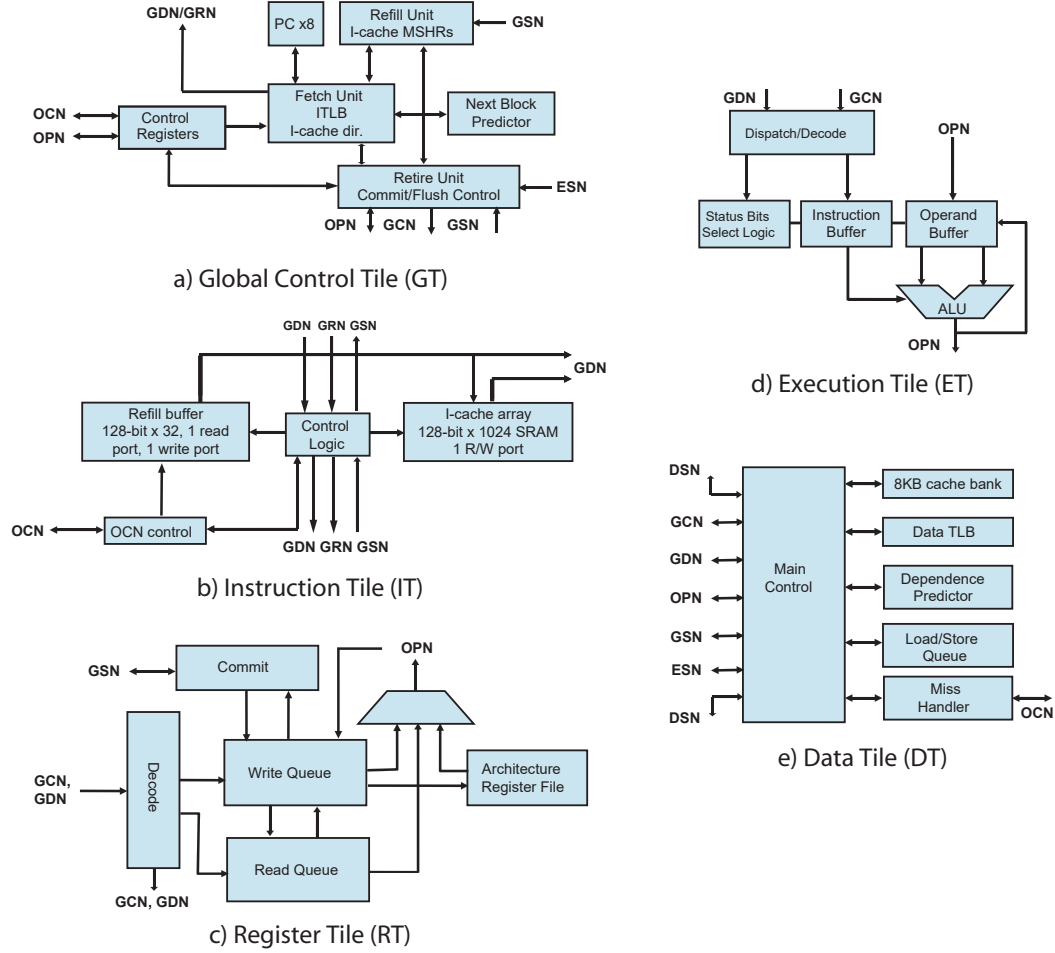


**Fig. 3.5** TRIPS block-level tile microarchitectures

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.<sup>1</sup> 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,

---

<sup>1</sup>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

```

R[0]      read R4      N[1,L] N[2,L]
N[0]      movi #0      N[1,R]
N[1]      teq          N[2,P] N[3,P]
N[2]      p_f          muli #4 N[32,L]
N[3]      p_t          null    N[34,L] N[34,R]
N[32]     lw #8        N[33,L]  LSID=0
N[33]     mov          N[34,L] N[34,R]
N[34]     sw #0        LSID=1
N[35]     callo       $func1

```

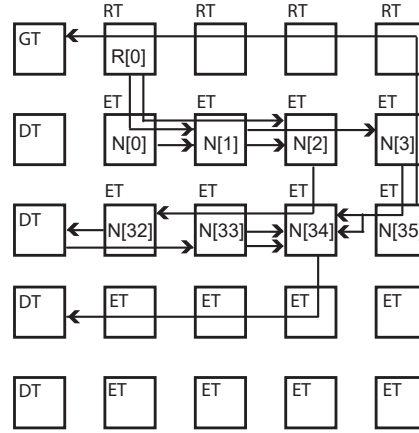


Fig. 3.6 TRIPS program and program execution

are not part of the instructions. Block execution begins when the read instruction  $R[0]$  is issued to  $RT_0$ , triggering delivery of  $R_4$  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,

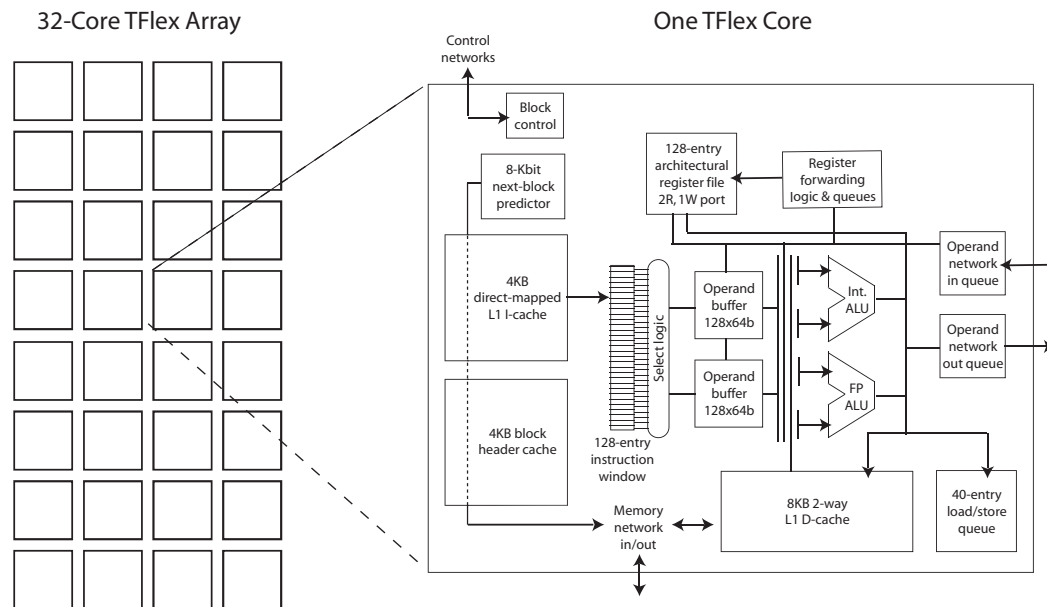


**Fig. 3.7** TFlex core microarchitecture

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  $2.4 \text{ mm}^2$  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.

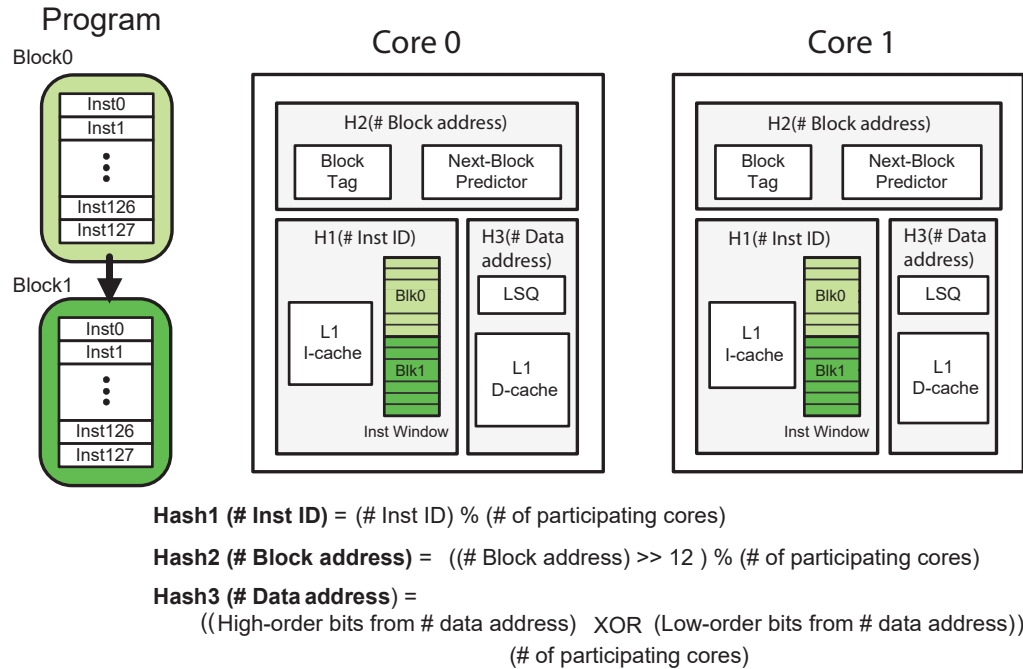


**Fig. 3.8** Interleaving of different microarchitectural structures for a two-core processor

*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

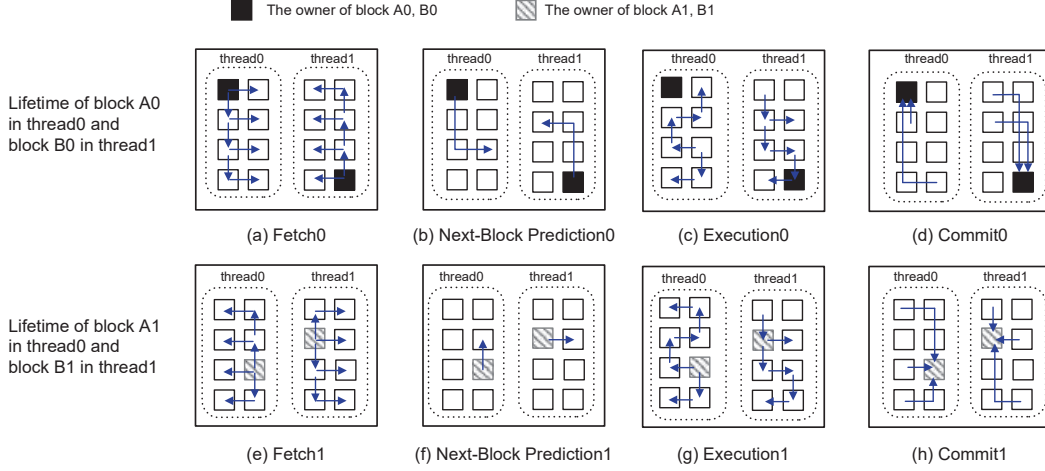


**Fig. 3.9** TFlex execution stages: execution of two successive blocks (A0, A1) and (B0,B1) from two different threads executing simultaneously on a 16-core TFlex CLP with each thread running on eight cores

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.

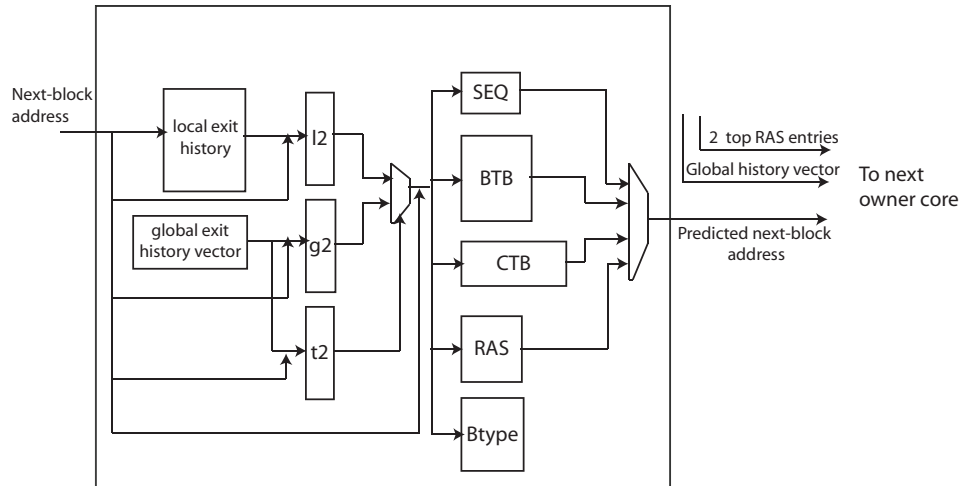


**Fig. 3.10** TFlex next-block predictor

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

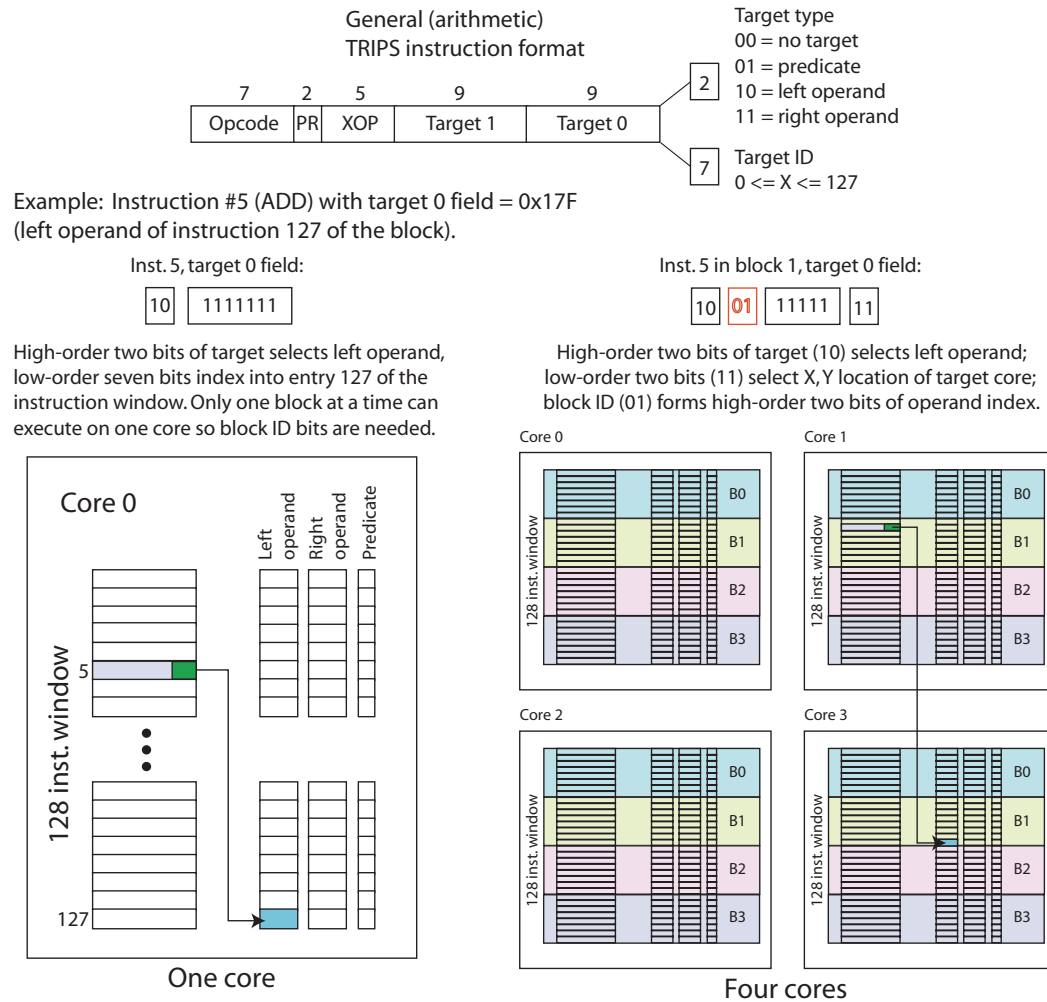


**Fig. 3.11** Block mapping for one-core and four-core processors

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 dependencies 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

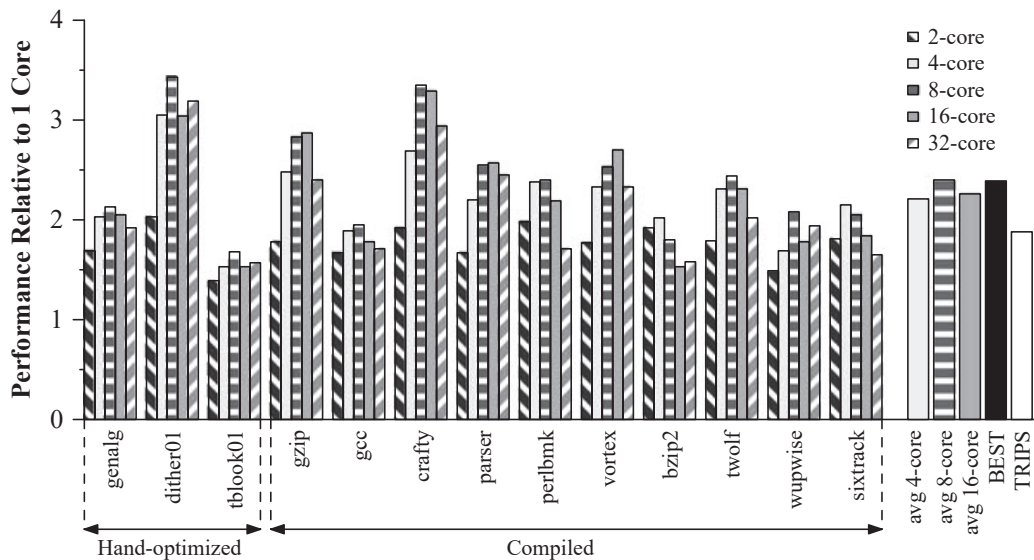


**Fig. 3.12** Low ILP benchmarks: normalized performance of applications running on 2–32 CLP cores

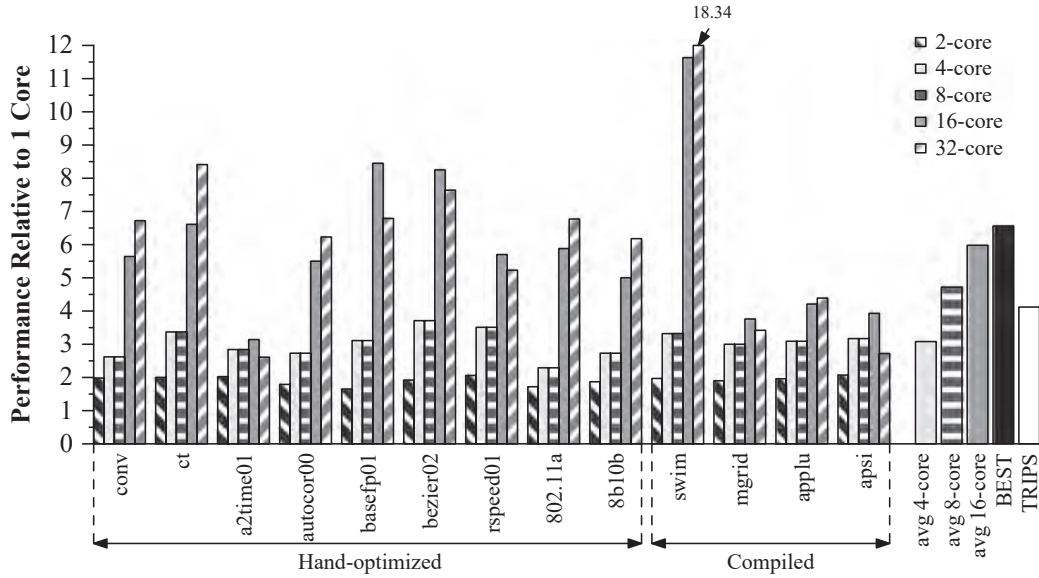


**Fig. 3.13** High ILP benchmarks: normalized performance of applications running on 2–32 CLP cores

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.

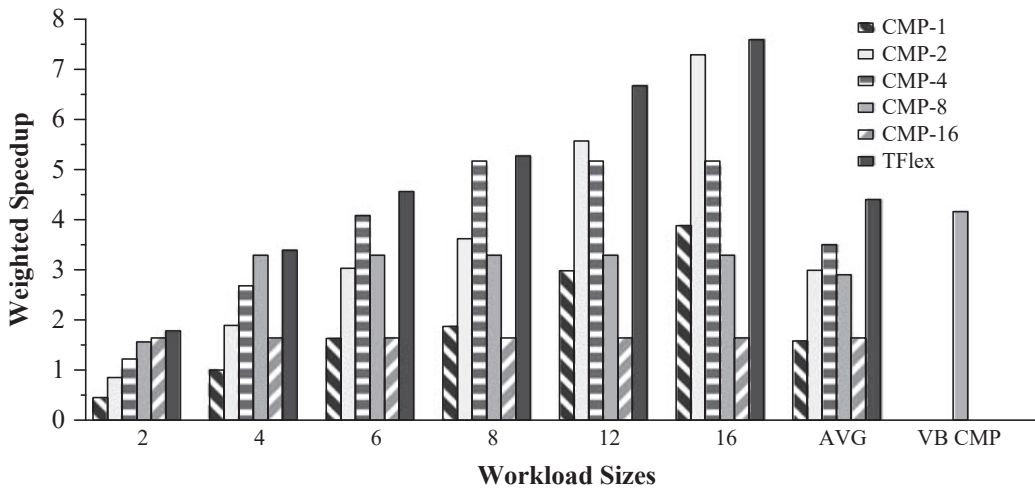


**Fig. 3.14** Weighted speedup comparison between TFlex and fixed-granularity CMPs

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

| Workload size | Cores (%) |    |    |    |    |    |
|---------------|-----------|----|----|----|----|----|
|               | 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 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<sup>2</sup>) estimate for eight-core TFlex and area of a single TRIPS core

| Structures                               | eight TFlex cores (16-issue)                |      | Single TRIPS processor (16-issue)            |      |
|------------------------------------------|---------------------------------------------|------|----------------------------------------------|------|
|                                          | 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<sup>2</sup>/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. Robotmili, 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 clearspeak. *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. Snaveley 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. Strumpfen, 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.