



email: stott | office: BH 3532H | tel: 310-825-6871 | fax: 310-794-5056



Judea Pearl, Emeritus on recall Professor of Computer Science

PhD (1965) Polytechnic Institute of Brooklyn

NAE Member, AAAI, IEEE Fellow; AI Research Excellence Award, Allen Newell Award

Research: artificial intelligence and knowledge representation; probabilistic and causal reasoning; nonstandard logics; learning strategies

email: judea | office: BH 4515 | tel: 310-825-3243 | fax: 310-794-5057



Gerald J. Popek, Adjunct Professor of Computer Science

PhD (1973) Harvard University

Research: computer security; distributed Unix systems; file replication; mobile computing

email: popek | office: BH 3532F | tel: 310-825-6507 | fax: 310-794-5057



Miodrag Potkonjak, Professor of Computer Science

PhD (1991) University of California, Berkeley

Research: complex distributed systems; embedded systems; communication designs; computer-aided design; ad hoc sensor networks; computational security; electronic commerce; IP protection

email: miodrag | office: BH 3532G | tel: 310-825-0790 | fax: 310-794-5056



Peter Reiher, Adjunct Associate Professor of Computer Science

PhD (1987) UCLA

Research: file systems; network and computer security; distributed operating systems; ubiquitous computing; optimistic methods in computing; parallel discrete event simulation

email: reiher | office: BH 3564 | tel: 310-825-8332 | fax: 310-794-5056



Glenn Reinman, Assistant Professor of Computer Science

PhD (2001) University of California, San Diego

Research: processor architecture design and optimization; speculative execution; profile-guided optimization; finding and exploiting instruction-level parallelism

email: reinman | office: BH 4731 | tel: 310-794-9755 | fax: 310-825-2273



David A. Rennels, Associate Professor of Computer Science

PhD (1973) UCLA

Research: computer systems architecture; fault-tolerant computing; modular architectures for VLSI implementation

email: rennels | office: BH 4731G | tel: 310-825-1484 | fax: 310-825-2273



Amit Sahai, Associate Professor of Computer Science

PhD (2000) MIT

Sloan Research Fellow

Research: theoretical computer science; foundations of cryptography; computer security

email: sahai | office: BH 3731E | tel: 310-267-4982 | fax: 310-825-7578



M.Y. "Medy" Sanadidi, Adjunct Professor of Computer Science

PhD (1982) UCLA

Research: congestion control and adaptive multimedia streaming in heterogeneous networks; analytic modeling of computer and communications systems



email: medy | office: BH 3531F | tel: 310-206-0767 | fax: 310-794-5056



Majid Sarrafzadeh, Professor of Computer Science

PhD (1987) University of Illinois, Urbana-Champaign

IEEE Fellow

Research: embedded and reconfigurable computing; VLSI CAD; design and analysis of algorithms

email: majid | office: BH 3532C | tel: 310-794-4303 | fax: 310-794-5056



David Smallberg, Lecturer of Computer Science

MS (1978) UCLA

Research: computer science education; programming languages; generic programming; student software analysis

email: das | office: BH 4532G | tel: 310-267-4665 | fax: 310-794-5057



Stefano Soatto, Professor of Computer Science

PhD (1996) California Institute of Technology

David Marr Prize

Research: computer vision; nonlinear estimation; control theory

email: soatto | office: BH 3531D | tel: 310-825-4840 | fax: 310-794-5056



Yuval Tamir, Associate Professor of Computer Science

PhD (1985) University of California, Berkeley

Research: parallel and distributed systems; fault-tolerant computing; multicore architectures; virtualization; cluster computing (OS, middleware); interconnection networks and switches; reliable network services

email: tamir | office: BH 4531J | tel: 310-206-2852 | fax: 310-794-5057



Demetri Terzopoulos, Chancellor's Professor of Computer Science

PhD (1984) MIT

FRSC, ACM Fellow, IEEE Fellow, European Academy of Sciences Member; Academy Award

Research: computer graphics, computer vision, medical image analysis, computer-aided design, artificial life/intelligence

email: dt | office: BH 4531G | tel: 310-206-6946 | fax: 310-794-5057



Jacques Vidal, Emeritus Professor of Computer Science

PhD (1961) University of Paris

Research: neuromimetic information processing; pattern recognition; neural networks, fuzzy systems and genetic search; multimedia communication for engineering and science education

email: vidal | office: BH 4532H | tel: 310-825-2858 | fax: 310-794-5057



Carlo Zaniolo, N.E. Friedmann Chair in Knowledge Science, Professor of Computer Science

PhD (1976) UCLA

Research: knowledge-based systems; database systems; non-monotonic reasoning; spatio/temporal reasoning; scientific databases

email: zaniolo | office: BH 3532D | tel: 310-825-8137 | fax: 310-794-5056



Lixia Zhang, Professor of Computer Science

PhD (1989) MIT

ACM Fellow; IEEE Fellow

Research: Internet architecture and protocol designs; security in large scale and open systems

email: lixia | office: BH 3713 | tel: 310-825-2605 | fax: 310-825-7579



Song Chun Zhu, Professor of Statistics, Professor of Computer Science

PhD (1996) Harvard University

David Marr Prize

Research: computer vision; statistical modeling and computing; vision and visual arts; machine learning

email: [sczhu](#) | office: BH 9494 | tel: 310-206-8693 | fax: 310-206-8693

[home](#) | [information](#) | [people](#) | [research](#) | [academics](#) | [admissions](#) | [resources](#) | [alumni](#)

© 2006 University of California, Los Angeles

<https://web.archive.org/web/20080220222723/http://www.cs.ucla.edu/~pfal/>

Petros Faloutsos



Assistant Professor

Boelter Hall 4531-F
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095-1596



pfal@cs.ucla.edu



(310) 825-2393



(310) 795-5057

Research Interests

- Computer Graphics
- Physics-based Animation
- Robotics
- Biomechanics

[Courses](#)

[Press Coverage](#)

[Publications](#)

[Students](#)

[Research Highlights](#)

[Awards](#)

[MAGIX Lab](#)

[Curriculum Vitae](#)

[Family](#)

[Other Faloutsos on Web](#)

<https://web.archive.org/web/20080222132652/http://www.cs.ucla.edu/~pfal/papers.html>

Petros Faloutsos

Publications

Volumes

1. Graphical Models, Special Issue on SCA 2005, Volume 69, Numbers 5-6, September/November 2007.
2. "Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation", Ken Anjyo, Petros Faloutsos (eds.), ACM, New York, New York, USA, 2005, 358 pages, (ACM ISBN 1-59593-198-8).

Chapters and Contributions in books

1. "Chapter 10: Speech Motion Decomposition and Editing", Yong Cao, Petros Faloutsos and Fred Pighin. In *Data-Driven 3D Facial Animation* by Z. Deng and U. Neumann, Springer, in press, November 2007.
2. "On Power-Law Relationships of the Internet Topology", Michalis Faloutsos, Petros Faloutsos and Christos Faloutsos, contributed in Chapter 3 of *The Structure and Dynamics of Networks*, Mark Newman, Albert-Laszlo Barabasi, Duncan J. Watts, Princeton University Press, 2006, ISBN-13 978-0-691-11356-2, pp. 251- 262.

Refereed Journals

1. "Fool Me Twice: Exploring and Exploiting Error Tolerance in Physics-Based Animation", Tom Yeh, Sanjay Patel, Petros Faloutsos, Glenn Reinman, ACM Transactions on Graphics, accepted with revisions, 2007. [pdf](#)
2. "The Photon Pipeline Revisited: A Hardware Architecture to Accelerate Photon Mapping", Shawn Singh, Petros Faloutsos, The Journal of the Visual Computer, 23(7), Springer Berlin/ Heidelberg, pp. 479-492, 2007. [pdf](#)
3. "Expressive Speech-Driven Facial Animation", Yong Cao, Wen C. Tien, Petros Faloutsos, Fred Pighin, ACM Transactions on Graphics, Volume 24, Issue 4, 1283-1302, October 2005. [pdf](#)
4. "Power-laws and the AS-level Internet Topology", ACM/IEEE Transactions on Networking, Georgos Siganos, Michalis Faloutsos, Petros Faloutsos, Christos Faloutsos, Vol. 11, No. 4, August 2003, pp. 514-524. [pdf](#)
5. "The virtual stuntman: dynamic characters with a repertoire of autonomous motor skills", Petros Faloutsos, Michiel van de Panne, Demetri Terzopoulos, Computers and Graphics, Volume 25, Issue 6, December, 2001, BEST PAPER AWARD, pp. 933-953. [pdf](#)
6. "Dynamic Free-Form Deformations for Animation Synthesis", Petros Faloutsos, Michiel van de Panne, Demetri Terzopoulos, IEEE Transactions on Visualization and Computer Graphics, Vol. 3, No 3. July-September 1997, pp. 201-214. [pdf](#)

Refereed Conferences

1. "SIMD Packet Techniques for Photon Mapping", Shawn Singh, Petros Faloutsos, IEEE/EG Symposium on Interactive Ray Tracing 2007, to appear. [pdf](#)
2. "The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration", Tom Yeh, Petros Faloutsos, Sanjay Patel, Milos Ercegovac, and Glenn Reinman, The 40th Annual International Symposium on Microarchitecture (MICRO), Dec 2007, to appear. [pdf](#)
3. "On the Beat! Timing and Tension for Dynamic Characters", Brian Allen, Derek Chu, Ari Shapiro, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2007, pp. 239-247. [pdf](#)
4. "Flipping with Physics: Motion Editing for Acrobatics", Anna Majkowska, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2007, pp. 35-44. [pdf](#)
5. "Sketching facial expressions", Gabriele Nataneli and Petros Faloutsos, ACM SIGGRAPH 2007 Technical Sketches, pp. 60, 2007. [pdf](#)
6. "Assembling an Expressive Facial Animation System", Alice Wang, Michael Emmi, Petros Faloutsos, in ACM SIGGRAPH Video Game Symposium (Sandbox), 2007, pp. 21-26. [pdf](#)
7. "A Dynamic Controller Toolkit", Derek Chu, Ari Shapiro, Brian Allen, Petros Faloutsos, in ACM SIGGRAPH Video Game Symposium (Sandbox), pp. 15-20, 2007. [pdf](#)
8. "Parallax: An Architecture for Real-Time Physics", Tom Yeh, Petros Faloutsos, Sanjay Patel, Glenn Reinman, in 34th Annual International Symposium on Computer Architecture, pp. 232-243, June 2007. [pdf](#)
9. "Interactive Motion Correction and Object Manipulation", Ari Shapiro, Marcelo Kallmann, Petros Faloutsos, ACM SIGGRAPH, Symposium on Interactive 3D Graphics and Games, April 2007, pp. 137-144. [pdf](#)
10. "Enabling Real-Time Physics Simulation in Future Interactive Entertainment", Tom Yeh, Petros Faloutsos, Glenn Reinman, ACM SIGGRAPH Video Game Symposium, 2006, pp. 71-81. [pdf](#)
11. "Automatic Splicing for Hand and Body Animations", Anna Majkowska, Victor Zordan, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, pp. 309-316. [pdf](#)

- Eurographics Symposium on Computer Animation, 2000, pp. 309-310. [pdf](#)
12. "Automatic Splicing for Hand and Body Animations", Anna Majkowska, Victor Zordan, Petros Faloutsos, ACM SIGGRAPH Technical Sketch, 2006. [pdf](#)
 13. "Style Components", Ari Shapiro, Yong Cao, Petros Faloutsos, In Proceedings of Graphics Interface 2006, pp. 33-40, 2006. [pdf](#)
 14. "Photorealistic Lighting with Offset Radiance Transfer Mapping", Benjamin Sunshine-Hill, Petros Faloutsos, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2006, pp. 15-21. [pdf](#)
 15. "Planning Motions in Motion", Marcelo Kallmann, Ari Shapiro, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, Extended Abstract.
 16. "The Dynamic Controller Toolkit", Ari Shapiro, Derek Chu, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, Extended Abstract.
 17. "Sketch-based Facial Animation", Gabriele Nataneli, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, Extended Abstract.
 18. "Interactive and Reactive Dynamic Control", Ari Shapiro, Petros Faloutsos, ACM SIGGRAPH 2005, Technical Sketches, 2005.
 19. "Steps toward Intelligent Physical Control", Ari Shapiro, Petros Faloutsos, ACM SIGGRAPH/ Eurographics Symposium on Computer Animation 2005, Poster Proceedings, Extended Abstract, pp. 20-21, 2005.
 20. "Dynamic Animation and Control Environment", Ari Shapiro, Petros Faloutsos, Victor Ng-Thow-Hing, In Proceedings of Graphics Interface 2005, pp. 61-70, 2005. [pdf](#)
 21. "Real-time Speech Motion Synthesis from Recorded Motions", Yong Cao, Petros Faloutsos, Eddie Kohler, Fred Pighin, in Proceedings of ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2004, pp. 347-355. [pdf](#)
 22. "Interactive Motion Decomposition", A. Shapiro, Y. Cao, P. Faloutsos, ACM SIGGRAPH 2004 Technical Sketches, Los Angeles, CA, August 2004.
 23. "Stylistic Motion Decomposition", A. Shapiro, Y. Cao, P. Faloutsos, ACM SIGGRAPH/ Eurographics Symposium on Computer Animation, Poster Proceedings, Extended Abstract, Grenoble, France, pp. 8-9 August 2004.
 24. "Dynamic Animation and Control Environment", A. Shapiro, P. Faloutsos, Victor Ng-Thow-Hing, ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Poster Proceedings, Extended Abstract, Grenoble, France, pp. 12-13, August 2004.
 25. "Unsupervised Learning for Speech Motion Editing," Yong Cao, Petros Faloutsos, Frederic Pighin, In Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 225-231 (2003) [pdf](#)
 26. "Autonomous Reactive Control for Simulated Humanoids", Petros Faloutsos, Michiel van de Panne and Demetri Terzopoulos, In Proceedings of IEEE International Conference on Robotics and Automation (ICRA) 2003, Taiwan, pp. 917-924. [pdf](#)
 27. "Hybrid Control For Interactive Character Animation", Ari Shapiro, Frederic Pighin, and Petros Faloutsos, Pacific Graphics, 2003, pp. 455-461. [pdf](#)
 28. "Complex Character Animation That Combines Kinematic and Dynamic Control", Ari Shapiro and Petros Faloutsos, Technical Sketch, ACM SIGGRAPH 2003.
 29. "Composable Controllers for Character Animation", Petros Faloutsos, Michiel van de Panne and Demetri Terzopoulos, Los Angeles, August, SIGGRAPH 2001, pp. 251-260. [pdf](#)
 30. "Learning Controller Preconditions for Physics-based Character Animation", P. Faloutsos, M. van de Panne, D. Terzopoulos, Proceedings of the Learning Workshop, Snowbird, UT, April, 2001.
 31. "Towards Agile Animated Characters", Michiel van de Panne, Joe Laszlo, Pedro Huang, Petros Faloutsos, International Conference on Robotics and Automation 2000, San Francisco, CA, pp. 682-687. [pdf](#)
 32. "Dynamic Animation and Control Environment", Victor Ng-Thow-Hing, Petros Faloutsos, Technical Sketch, ACM SIGGRAPH 2000, New Orleans.
 33. "On Power-Law Relationships of the Internet Topology", Michalis Faloutsos, Petros Faloutsos, Christos Faloutsos, ACM SIGCOMM'99, Cambridge, Massachusetts, pp 251-262, 1999. [pdf](#)
 34. "BIVTECI: A Bibliography Visualization Tool", David Modjeska, Vassilis Tzerpos, Petros Faloutsos, Michalis Faloutsos, CASCON '96, CD-ROM proceedings, Toronto, Ontario. [pdf](#)

Refereed Posters and Presentations

1. "Environment-Based Physical Motion for Secondary Characters", Brian Allen, Sumit Jain, Petros Faloutsos, C. Karen Liu, ACM SIGGRAPH Posters, 2007.
2. "Planning Motions in Motion", Marcelo Kallmann, Ari Shapiro, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, Poster.
3. "The Dynamic Controller Toolkit", Ari Shapiro, Derek Chu, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, Poster.
4. "Sketch-based Facial Animation", Gabriele Nataneli, Petros Faloutsos, ACM SIGGRAPH / Eurographics Symposium on Computer Animation, 2006, Poster.
5. "Dynamic Animation and Controller Environment", Victor Ng-Thow-Hing, Petros Faloutsos, Graphics Interface Posters, Kingston, Ontario, 1999.
6. "Composeable Controllers for Physically-based Character Animation", Petros Faloutsos, Michiel van de Panne, Graphics Interface Posters, Kingston, Ontario, 1999.

Other publications

1. "Immersive training and mentoring for laparoscopic surgery", Vasile Nistor, Brian Allen, E. Dutson, P. Faloutsos, G. P. Carman, Proceedings of SPIE, Volume 6528, Nanosensors, Microsensors, and Biosensors and Systems 2007. [pdf](#)
2. "[CSD-TR 60026] ParallAX: An Architecture for Real-Time Physics", Thomas Y. Yeh, Petros Faloutsos, Sanjay Patel, and Glenn Reinman, UCLA, DCS, Technical Report 60026, 2007.
3. "Cache Behavior of Real-Time Physics Simulation", Thomas Y. Yeh, Petros Faloutsos, Glenn Reinman, DCS, UCLA, Technical Report 060024, 2006.
4. "Accelerating Real-Time Physics Simulation by Leveraging High-Level Information", Thomas Y. Yeh, Petros Faloutsos, Glenn Reinman, DCS, UCLA, Technical Report 060023, 2006.
5. "Composeable Controllers for Physically-based Character Animation", Petros Faloutsos, Michiel van de Panne, Demetri Terzopoulos. Technical report 418, 1999.
6. "Physics-Based Animation and Control of Flexible Characters", CSRI Technical report 326.
7. Petros Faloutsos, "Composable Controllers for Physics-Based Character Animation", Ph.D. Thesis, Department of Computer Science, University of Toronto, 2002. [pdf](#)
8. Petros Faloutsos, "Physics-Based Animation and Control of Flexible Characters", MA.Sc Thesis, Department of Computer Science, University of Toronto, 1995. [pdf](#)

Tutorials

- CASA 2003: Tutorial II: "Physics-based control of Dynamic Characters", organized by Petros Faloutsos, Victor Zordan.

Petros Faloutsos 2007-08-21

EXHIBIT B

<https://web.archive.org/web/20080221203220/http://www.cs.ucla.edu/~pfal/papers/TOG-08-Fool.pdf>

Fool Me Twice: Exploring and Exploiting Error Tolerance in Physics-Based Animation

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

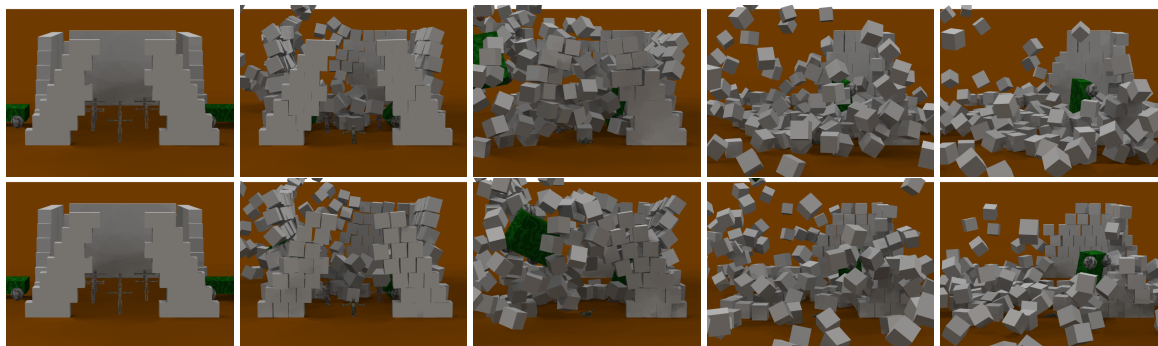


Figure 1: Snapshots of two simulation runs with the same initial conditions. The simulation results shown on top is the baseline, and the bottom row is simulation computed with 7-bit mantissa floating-point computation in Narrowphase and LCP. The results are different but both are visually correct.

Abstract

The error tolerance of human perception offers a range of opportunities to trade numerical accuracy for performance in physics-based simulation. However, most previous approaches either focus exclusively on understanding the tolerance of the human visual system or burden the application developer with case-specific implementations. In this paper, based on a detailed set of perceptual metrics, we propose a methodology to identify the maximum error tolerance of physics simulation. Then, we apply this methodology in the evaluation of two techniques. The first is the hardware optimization technique of precision reduction which reduces the size of floating point units (FPUs), allowing more of them to occupy the same silicon area. The increased number of FPUs can significantly improve the performance of future physics accelerators. A key benefit of our approach is that it is transparent to the application developer. The second is the software optimization of choosing the largest time-step for simulation.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques.

Keywords: physics-simulation, perceptual errors

1 Introduction

Physics-based animation (PBA) is becoming one of the most important elements of interactive entertainment applications, such as computer games, largely because of the automation and realism that it offers. However, the benefits of PBA come at a considerable computational cost. Furthermore, this cost grows prohibitively with the number and complexity of objects and interactions in the virtual world, making it extremely challenging to satisfy such complex worlds in real-time.

Fortunately, there is a tremendous amount of parallelism in the physical simulation of complex scenes. Exploiting this parallelism to improve the performance of PBA is an active area of research both in terms of software techniques and hardware accelerators. Commercial solutions, such as PhysX [AGEIA], GPUs [Havok Inc.], and the Cell [Hofstee 2005], have just started to address this problem.

But while parallelism can help PBA achieve a real-time frame rate, there is another avenue to help improve PBA performance which also has the potential to reduce the hardware required to exploit parallelism: *perceptual error tolerance*. There is a fundamental difference between *accuracy* and *believability* in interactive entertainment – the results of PBA do not need to be absolutely accurate, but do need to appear correct (i.e. believable) to human users. The perceptual acuity of human viewers has been studied extensively both in graphics and psychology [O’Sullivan et al. 2004; O’Sullivan and Dingliana 2001]. It has been demonstrated that there is a surprisingly large degree of error tolerance in our perceptual ability.¹

This perceptual error tolerance can be exploited by a wide spectrum of techniques ranging from high-level software techniques down to low-level hardware optimizations. At the application-level, level of detail (LOD) simulation [Reitsma and Pollard 2003a; Carlson and Hodgins 1997; McDonnell et al. 2006] can be used to handle distant objects with simpler models. At the physics engine library level, one option is to use approximate algorithms optimized for speed rather than accuracy [Seugling and Rolin. 2006]. At the compiler level, dependencies among parallel tasks could be broken to reduce

¹This is independent of a viewer’s understanding of physics [Proffitt].

synchronization overhead. At the hardware design level, floating-point precision reduction can be leveraged to reduce area, reduce energy, or improve performance for physics accelerators.

In this paper, we address the challenging problem of leveraging perceptual error tolerances to improve the performance of real-time physics simulation in interactive entertainment. The main challenge is the need to establish a set of error metrics that can measure the visual performance of a complex simulation. Prior perceptual thresholds do not scale to complex scenes. In our work we address this challenge and investigate specific hardware solutions.

Our contributions are threefold:

- We propose a methodology to evaluate physical simulation errors in complex dynamic scenes.
- We identify the maximum error that can be injected into each phase of the low-level numerical PBA computation.
- We explore hardware precision reduction and software time-step tuning to exploit error tolerance to improve PBA performance.

2 Background

In this section, we present a brief overview of physics simulation, identify its main computational phases, and categorize possible errors. We then review the latest results in perceptual error metrics and their relation to physical simulation.

2.1 Computational Phases of a typical Physics Engine

Physics simulation requires the numerical solution of a discrete approximation of the differential equations of motion of all objects in a scene. Articulations between objects, and contact configurations are most often solved with constraint based approaches such as [Baraff 1997; Muller et al. 2006; Havok Inc.]. Time is discretized with a fixed or adaptive time-step. The time-step is one of the most important parameters of the simulation and largely defines the accuracy of the simulation. For interactive applications the time step needs to be in the range of 0.01 to 0.03 simulated seconds or higher. The errors and the stability issues associated with the time-step are orthogonal to the first part of this study. The time-step is a high-level parameter that should be controlled by the developer of an application. Our work identifies both low-level trade-offs that are part of a hardware solution and as such are transparent to an application developer as well as high-level trade-offs to assist the programmer. As we demonstrate later, the errors that our low-level approach introduces in the simulation results are no greater than those introduced by the discretization alone.

Physics simulation can be described by the dataflow of computational phases shown in Figure 2. Cloth and fluid simulation are special cases of this pipeline. Below we describe the four computational phases in more detail.

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

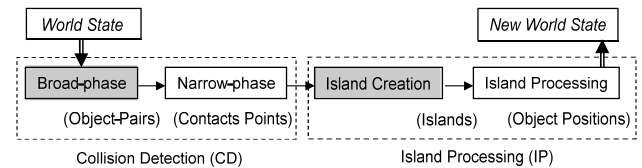


Figure 2: Physics Engine Flow. All phases are serialized with respect to each other, but **unshaded** stages can exploit parallelism within the stage.

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

Island Creation. After generating the contact joints linking interacting objects together, the engine serially steps through the list of all objects to create islands (connected components) of interacting objects. This phase is serializing in the sense that it must be completed before the next phase can begin. The full topology of the contacts isn’t known until the last pair is examined by the algorithm, and only then can the constraint solvers begin. This phase contains no floating-point operations, so it is excluded from this study.

Island Processing. For each island, given the applied forces and torques, the engine computes the resulting accelerations and integrates them to compute the new position and velocity of each object. This phase exhibits both coarse-grain (CG) and fine-grain (FG) parallelism. Each island is independent, and the constraint solver for each island contains independent iterations of work. We further split this component into two phases:

- **Island Processing** which includes constraint setup and integration (CG)
- **LCP** which includes the solving of constraint equations (FG)

2.2 Simulation Accuracy and Stability

The discrete approximation of the equations of motion introduce errors in the results of any non-trivial physics-based simulation. For the purposes of entertainment applications, we can distinguish between three kinds of errors in order of increasing importance:

- **Imperceptible.** These are errors that cannot be perceived by an average human observer.
- **Visible but bounded.** There are errors that are visible but remain bounded.
- **Catastrophic.** These errors make the simulation unstable which results in numerical explosion. In this case, the simulation often reaches a state from which it cannot recover gracefully.

To differentiate between categories of errors, we employ the conservative thresholds presented in prior work [O’Sullivan et al. 2003; Harrison et al. 2004] for simple scenarios. All errors with magnitude smaller than these thresholds are considered imperceptible. All errors exceeding these thresholds without simulation blow-up are

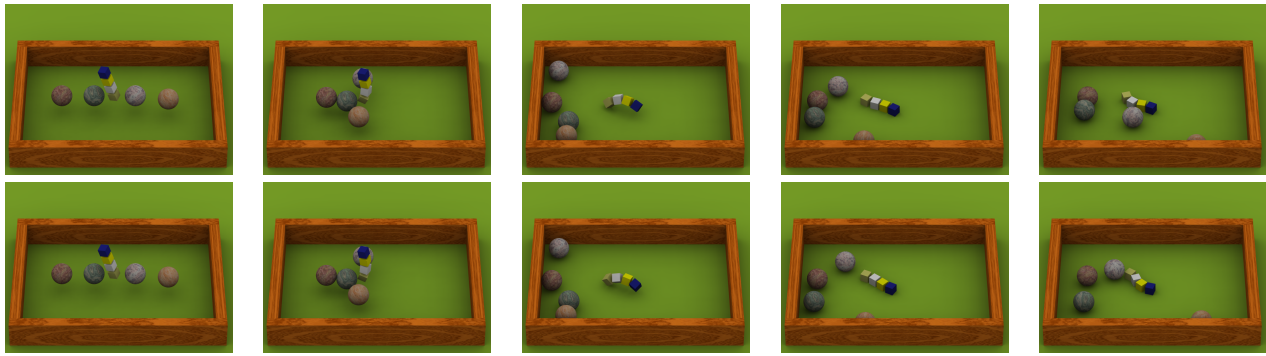


Figure 3: Snapshots of two simulation runs with the same initial conditions and different constraint ordering. The results are different but both are visually correct.

considered visible but bounded. All errors that lead to simulation blow-up are considered catastrophic.

An interesting example that demonstrates imperceptible simulation errors is shown in Figure 3. In this example four spheres and a chain of square objects are initially suspended in the air. The two spheres at the sides have horizontal velocities towards the object next to them. With the same initial conditions two different simulation runs shown in the figure result in visibly different final configurations. However, both runs appear physically correct.² This behavior is due to the *constraint reordering* that the iterative constraint solver employs to reduce bias [Smith]. Constraint reordering is a well studied technique that improves the stability of the numerical constraint solver and it is employed by commercial products such as [AGEIA]. For our purposes though, it provides an objective way to establish what physical errors are acceptable when we develop the error metrics in Section 4.

The first two categories are the basis for perceptually-based approaches such as ours. Specifically, we investigate how we can leverage the perceptual error tolerance of human observers without introducing catastrophic errors in the simulation.

The next section reviews the relevant literature in the area of perceptual error tolerance.

2.3 Perceptual Believability

[O’Sullivan et al. 2004] is a 2004 state of the art survey report on the field of perceptual adaptive techniques proposed in the graphics community. There are six main categories of such techniques: interactive graphics, image fidelity, animation, virtual environments, and visualization and non-photorealistic rendering. Our paper will focus on the animation category. Given the comprehensive coverage of this prior survey paper, we will only present prior work most related to our paper and point the reader to [O’Sullivan et al. 2004] for additional information.

[Barzel et al. 1996] is credited with the introduction of the plausible simulation concept, and [Chenney and Forsyth 2000] built upon this idea to develop a scheme for sampling plausible solutions. [O’Sullivan et al. 2003] is a recent paper upon which we base most of our perceptual metrics. For the metrics examined in this paper, the authors experimentally arrive at thresholds for high probability of user believability. Then, a probability function is developed to

²The complete animation of this experiment is included in the accompanying video.

capture the effects of different metrics. This paper only uses simple scenarios with 2 colliding objects.

[Harrison et al. 2004] is a study on the visual tolerance of lengthening or shortening of human limbs due to constraint errors produced by physics simulation. We derive the threshold for constraint error from this paper.

[Reitsma and Pollard 2003b] is a study on the visual tolerance of ballistic motion for character animation. Errors in horizontal velocity were found to be more detectable than vertical velocity; and added accelerations were easier to detect than added deceleration.

[Yeh et al. 2006] examines the techniques of fuzzy value prediction and fast estimation with error control to leverage error tolerance in interactive entertainment applications. However, this prior work does not evaluate how to measure tolerable error nor deal with maximum tolerable error. It shows absolute differences in terms of position and orientation of objects and constraints against baseline simulation without error. In fact, our work argues against such methods of measuring error using absolute position and orientation differences as demonstrated by Figure 3.

In general, prior work has focused on simple scenarios in isolation (involving 2 colliding objects, a human jumping, human arm/foot movement, etc). Isolated special cases allow us to see the effect of instantaneous phenomena, such as collisions, over time. In addition, they allow apriori knowledge of the correct motion which serves as the baseline for exact error comparisons. Complex cases do not offer that luxury. On the other hand, in complex cases, such as multiple simultaneous collisions, errors become difficult to detect and, may in fact cancel out. We are the first to address this challenging problem and provide a methodology to estimate the perceptual error tolerance of physical simulation corresponding to a complex, game-like scenario.

2.4 Simulation believability

Chapter 4 of [Seugling and Rolin. 2006] compares three physics engines, namely ODE, Newton, and Novodex, by conducting performance tests on friction, gyroscopic forces, bounce, constraints, accuracy, scalability, stability, and energy conservation. All tests show significant differences between the three engines, and the engine choice will produce different simulation results with the same initial conditions. Even without any error-injection, there is no single *correct* simulation for real-time physics simulation in games as the algorithms are optimized for speed rather than accuracy.

3 Methodology

One major challenge in exploring the tradeoff between accuracy and performance in PBA is coming up with a set of metrics that will evaluate believability. Furthermore, since some of these metrics are relative (i.e. the resultant velocity of an object involved in a particular collision), there must be a reasonable standard for comparing these metrics. In this section, we detail the set of numerical metrics we have assembled to gauge believability, along with a technique to fairly compare worlds which may have substantially diverged.

3.1 Experimental Setup

To represent in-game scenarios, we construct one complex test-case which includes stacking, highly articulated objects, and highspeed objects shown in Figure 1. The scene is composed of a building enclosed on all four sides by brickwalls with one opening. The wall sections framing the opening is unstable. Ten humans with anthropomorphic dimension, mass, and joints are stationed within the enclosed area. A cannon shoots highspeed (88 m/s) cannonballs at the building, and two cars collide into opposing walls. Assuming time starts at 0 sec, one cannonball is shot every 0.04 seconds until 0.4 seconds. The cars are accelerated to roughly 100 miles/hr (44 m/s) at time 0.12 to crash into the walls. No forces are injected after 0.4 seconds. Because we want to measure the maximum and average errors, we target the time period with the most interaction (the first 55 frames).

The described methodology has also been applied to three other scenarios with varying complexity:

1. 2 spheres colliding ([O’Sullivan et al. 2003], video)
2. 4 spheres and a chain of square objects (Figure 3, video)
3. Complex scenario without humans (not shown)

Our physics engine is a modified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [Smith]. ODE follows a constraint-based approach for modeling articulated figures, similar to [Baraff 1997; AGEIA], and it is designed for efficiency rather than accuracy. Like most commercial solutions it uses an iterative constraint solver. Our experiments use a conservative time-step of 0.01 seconds and 20 solver iterations as recommended by the ODE user-base.

3.2 Error Sampling Methodology

To evaluate the numerical error tolerance of physics simulation, we will inject errors at a per-instruction granularity. We only inject errors into floating point (FP) add, subtract, and multiply instructions, as these make up the majority of FP operations for this workload.

Our error injection technique will be fairly general, and should be representative of a range of possible scenarios where error could occur. At a high level, we will be changing the output of FP computations by some varying amount. This could reflect changes from an imprecise ALU, an algorithm that cuts corners, or a poorly synchronized set of ODE threads. The goal is to show how believable the simulation is for a particular magnitude of allowed error.

To achieve this generality, we randomly determine the amount of error injected at each instruction, but vary the absolute magnitude of allowed error for different runs. This allowed error bound is expressed as a maximum percentage change from the correct value, in either the positive or negative direction. For example, an error

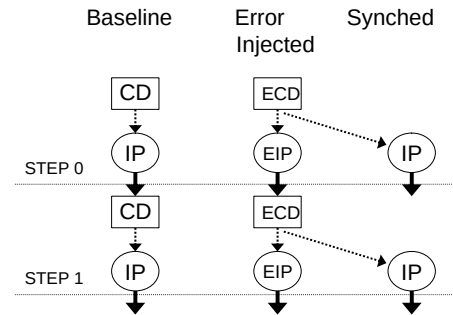


Figure 4: *Simulation Worlds.* CD = Collision Detection. IP = Island Processing. E = Error-injected

bound of 1% would mean that the correct value of an FP computation could change by any amount in the range from -1% to 1%.

A random percentage, less than the preselected max and min, is applied to the result to compute the absolute injected error. By using random error injection, we avoid biasing of injected errors. For each configuration, we average the results from 100 different simulations (each with a different random seed) to ensure that our results converge. We have verified that 100 simulations are enough to converge by comparing results with only 50 simulations – these results are identical. We evaluate the error tolerance of the entire application and each phase individually.

Both error injection and precision reduction are done by replacing every floating point add, subtract, and multiply operation by a function call which simulates either random error injection or precision reduction. This is done by modifying the source code of ODE. ODE uses its own implementation of the solvers, and the errors are introduced in every single floating point add, subtract, and multiply.

3.3 Error Metrics

Now that we have a way of injecting error, we want to be able to determine when the behavior of a simulation with error is still believable through numerical analysis. Many of the metrics we will propose are relative values, and therefore we need to have reasonable comparison points for these metrics. However, it is not sufficient to simply compare a simulation that has error injected into it with a simulation without any error. Small, but perceptually tolerable difference can result in large behavioral differences as shown in Figure 3.

To address this, we make use of three simulation worlds as shown in Figure 4: *Baseline*, *Error-Injected*, and *Synched*. All worlds are created with the same initial state, and the same set of injected forces (cannon ball shooting or cars speeding up) are applied to all worlds. *Error-injected* refers to the error injected world, where random errors within the preselected range are injected for every FP $+/-*$ instruction. *Baseline* refers to the deterministic simulation with no error injection. Finally, we have the *Synched* world – a world where the state of every object and contact is copied from the error-injected world after each simulation step’s collision detection. The island processing computation of *Synched* contains no error injection – so it is using the collisions detected by *Error-Injected* but is performing correct island processing.

We use the following seven numerical metrics:

- **Energy Difference:** difference in total energy between *baseline* and *error-injected* worlds – due to energy conservation, these should match.
- **Penetration Depth:** distance from the object’s surface to the contact point created by collision detection.
- **Constraint Violation:** distance between object position and where object is supposed to be based on statically defined joints (car’s suspension or human limbs).
- **Linear Velocity Magnitude:** difference in linear velocity magnitude for the same object between *error-injected* and *synched* worlds.
- **Angular Velocity Magnitude:** difference in angular velocity magnitude for the same object between *error-injected* and *synched* worlds.
- **Linear Velocity Angle:** angle between linear velocity vectors of the same object inside *error-injected* and *synched* worlds.
- **Gap Distance:** distance between two objects that are found to be colliding, but are not actually touching.

We can measure gap, penetration, and constraint errors directly in the *error-injected* world, but we still use *baseline* here to normalize these metrics. If penetration is equally large in the *baseline* world and *error-injected* world, then our injected error has not made things worse – perhaps the time-step is too small.

The above error metrics capture both globally conserved quantities, such as total energy, and instantaneous per-object quantities such as positions and velocities. The metrics do not include momentum because most simulators for computer games trade off momentum conservation for stability. Furthermore, the approximate collision resolution models often do not conserve momentum either.

4 Numerical Error Tolerance

In this section, we will explore the use of our error metrics in a complex game scene with a large number of objects. We will inject error into this scene for different ODE phases, and measure the response from these metrics to determine how far we can go when trading accuracy for performance.

Before delving into the details, we briefly articulate the potential outcome of error injection in different ODE phases. Because *Broadphase* is a first-pass filter on potentially colliding object-pairs, it can only create functional errors by omitting actually colliding pairs. Since *Narrowphase* will not see the omitted pairs, the omissions can lead to missed collisions, increased penetration (if collision is detected later), and, in the worst case, tunneling (collision never detected). On the other hand, poor *Broadphase* filtering can degrade performance by sending *Narrowphase* more object-pairs to process. Errors in *Narrowphase* may lead to missed collisions or different contact points. The results are similar to omission by *Broadphase*, but can also include errors in the angular component of linear velocity due to errors in contact points. Because *Island processing* sets up the constraint equations to be solved, errors here can drastically alter the motion of objects, causing movement without applied force. Errors inside the *LCP solver* will alter the applied force due to collisions. Therefore, the resulting momentum of two colliding objects may be severely increased or dampened. Since the *LCP* algorithm is designed to self-correct algorithmic errors by iterating multiple times, *LCP* will most likely be more error tolerant than *Island Processing*.

While our study focuses on rigid body simulation, we qualitatively argue that perceptual error tolerance can be exploited similarly in particle, fluid, and cloth simulation. Rigid body simulation in some sense has tighter requirements for accuracy than other types of real-time physical simulation. Unexpected behaviors (interpenetration, unexpected increases in velocity, etc) are more often visually apparent because rigid bodies are often used to model larger objects, which usually affect gameplay. In contrast, errors in a particle simulation are more likely to be tolerated by the viewer because the artifacts themselves are smaller phenomena. We conjecture that fluid and cloth simulation (particularly those that are particle-based) are likely to exhibit similar numerical tolerance, as the visual representations (such as the fluid surface or cloth mesh) represents a down sampling of the underlying physical model. That is, many particles can represent a volume of fluid, but fewer actually represent the surface. We leave the empirical evaluation for future work.

4.1 Numerical Error Analysis

For this initial error analysis, we ran a series of experiments where we injected an increasing degree of error into different phases of ODE. Figure 5 demonstrates the maximal error that results from error injection on our seven perceptual metrics. Following the error injection methodology of section 3, we progressively increased the maximum possible error in order-of-magnitude steps from 0.0001% to 100% of the correct value, labeling this range 1.E-6 to 1.E+00 along the x-axis. We show results for injecting error into each of the four ODE phases alone, and then an *All* result when injecting error into all phases of ODE.

The serial phases, *Broadphase* and *Island Processing*, exhibit the the highest and lowest per phase error tolerance respectively. Only *Broadphase* does not result in simulation blow-up as increasingly large errors are injected. *Island Processing* is the most sensitive individual phase to error. The highly parallel phases of *Narrowphase* and *LCP* show similar average sensitivity to error. We will make use of these per-phase sensitivity differences when trading accuracy for performance.

As shown by the graphs in Figure 5, most of the metrics are strongly correlated. Most metrics show a distinct flat portion and a knee where the difference starts to increase rapidly. As these errors pile up, the energy in the system grows as there are higher velocities, deeper penetrations, and more constraint violations. The exceptions are gap distance and linear velocity angle. Gap distance remains consistently small. One reason for this is the filtering that is done in collision detection. For a gap error to occur, broadphase would need to allow two objects which are not actually colliding to get to narrowphase, and narrowphase would need to mistakenly find that these objects were actually not in contact with one another. A penetration error is much easier to generate – only one of the two phases needs to incorrectly detect that two objects are not colliding.

The angle of linear velocity does not correlate with other metrics either – in fact, the measured error actually decreases with more error. The problem with this metric is that colliding objects with even very small velocities can have drastically different angles of deflection depending on the error injected in any one of the four phases. From the determination of contact points to the eventual solution of resultant velocity, errors in angle of deflection can truly propagate. However, we observe that these maximal errors in the angle of linear velocity are extremely rare and mainly occur in the bricks composing our wall that are seeing extremely small amounts of jitter. This error typically lasts only a single frame and is not readily visible.

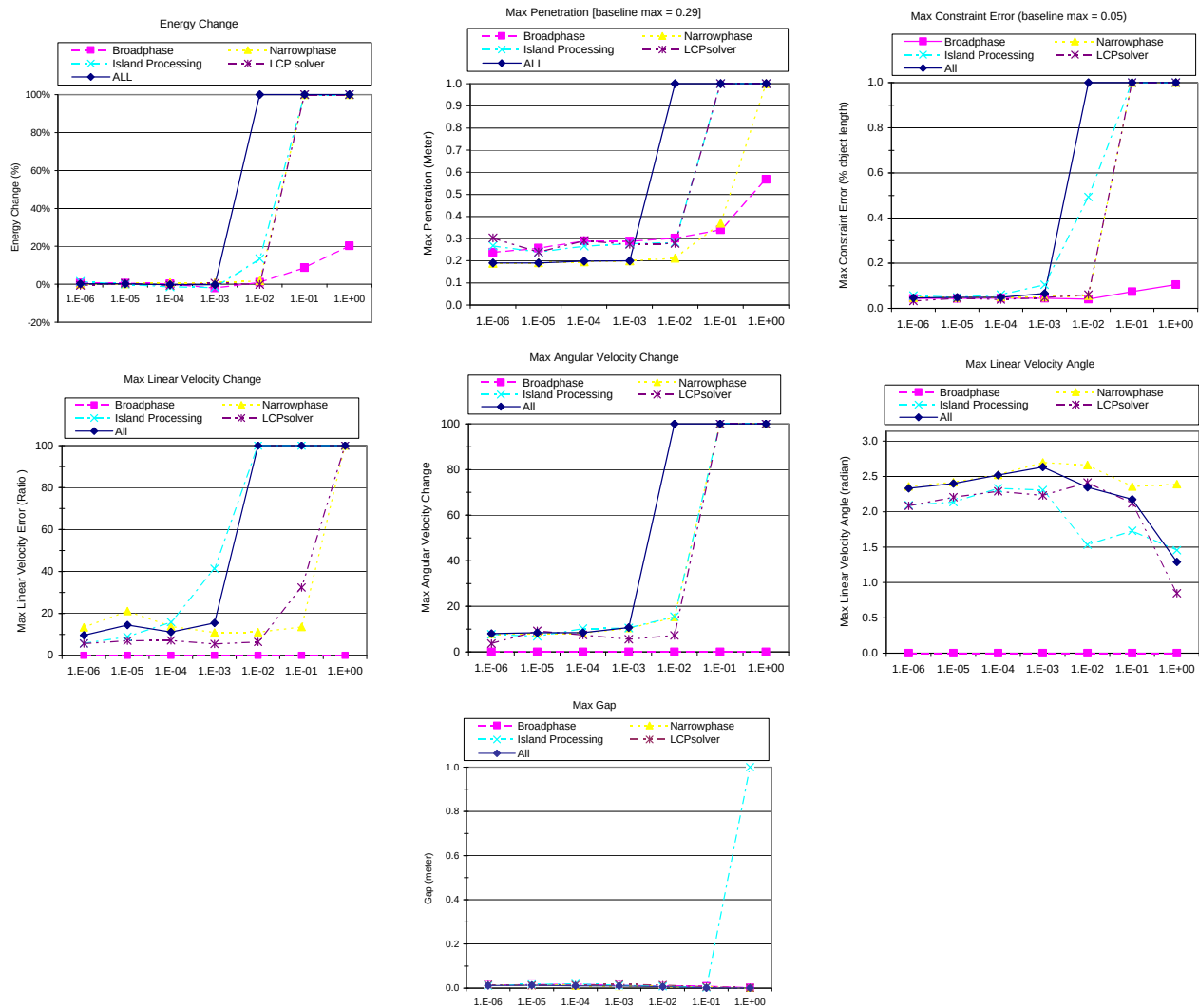


Figure 5: Perceptual Metrics Data for Error-Injection. X-axis shows the maximum possible injected error. Note: Extremely large numbers and infinity are converted to the max value of each Y-axis scale for better visualization.

While these maximal error values are interesting, as shown in figure 6, the average error in our numerical metrics and the standard deviation of errors are both extremely small. This shows that most objects in the simulation are behaving similarly to the baseline. Only the percentage change in magnitude of linear velocity has a significant average error. This is because magnitude change on extremely small velocities can result in significant percentage change.

4.2 Acceptable Error Threshold

Now that we have examined the response of PBA to injected error using our error metrics, the question still remains as to how much error we can actually tolerate and keep the simulation believable. Consider figure 5 where we show the maximum value of each metric for a given level of error. Instead of using fixed thresholds to evaluate simulation fidelity and find the largest amount of tolerable error, we argue for finding the knee in the curve where simulation differences start to diverge towards instability. The average

error is extremely low, figure 6, with the majority of errors resulting in imperceptible differences in behavior. Of the remaining errors, many are simply transient errors lasting for a frame or two. We are most concerned with the visible, persistent outliers that can eventually result in catastrophic errors. For many of the maximal error curves, there is a clear point where the slope of the curve drastically increases – these are points of catastrophic errors that are not tolerable by the system, as confirmed by a visual inspection of the results.

Error Tolerance	Broadphase	Narrowphase	Island Processing	LCP	All
(100 Samples)	[1%]	[1%]	[0.01%]	[1%]	[0.1%]

Table 1: Max Error Tolerated for Each Computation Phase

Table 1 summarizes the maximum % error tolerated by each computation phase (using 100 samples), based on finding the earliest knee where the simulation blows up over all error metric curves. It is interesting that *All* is more error tolerant than only injecting errors in *Island Processing*. The reason for this behavior is that injecting errors into more operations with *All* is similar to taking more sam-

Penetration	Broadphase		Narrowphase		Island Processing		LCPsolver		ALL	
Max Error	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev
0.0001%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.0010%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.0100%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.1000%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1.0000%	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	inf	inf
10.0000%	0.00	0.01	0.00	0.02	inf	inf	inf	inf	inf	inf
100.0000%	0.00	0.02	inf	inf	inf	inf	inf	inf	inf	inf

Constraint	Broadphase		Narrowphase		Island Processing		LCPsolver		ALL	
Max Error	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev
0.0001%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
0.0010%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
0.0100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
0.1000%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1.0000%	0%	0%	0%	0%	1%	3%	0%	0%	inf	inf
10.0000%	0%	0%	inf	inf	inf	inf	inf	inf	inf	inf
100.0000%	0%	0%	inf	inf	inf	inf	inf	inf	inf	inf

Linear Vel	Broadphase		Narrowphase		Island Processing		LCPsolver		ALL	
Max Error	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev
0.0001%	0%	0%	3%	34%	0%	8%	0%	9%	1%	18%
0.0010%	0%	0%	10%	79%	1%	14%	0%	10%	3%	38%
0.0100%	0%	0%	3%	34%	7%	36%	1%	11%	4%	25%
0.1000%	0%	0%	2%	20%	91%	245%	1%	9%	11%	49%
1.0000%	0%	0%	3%	26%	826%	2427%	1%	13%	10%	185%
10.0000%	0%	0%	12%	53%	5335%	42312%	7%	49%	inf	inf
100.0000%	0%	0%	inf	inf	inf	inf	inf	inf	inf	inf

Angular Vel	Broadphase		Narrowphase		Island Processing		LCPsolver		ALL	
Max Error	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev
0.0001%	0.00	0.00	0.00	0.09	0.00	0.09	0.00	0.04	0.00	0.09
0.0010%	0.00	0.00	0.00	0.09	0.00	0.07	0.00	0.11	0.00	0.09
0.0100%	0.00	0.00	0.00	0.09	0.00	0.11	0.00	0.09	0.00	0.10
0.1000%	0.00	0.00	0.00	0.12	0.00	0.11	0.00	0.06	0.01	0.12
1.0000%	0.00	0.00	0.01	0.18	0.04	0.23	0.00	0.08	inf	inf
10.0000%	0.00	0.00	inf	inf	inf	inf	inf	inf	inf	inf
100.0000%	0.00	0.00	inf	inf	inf	inf	inf	inf	inf	inf

Linear Vel Angle	Broadphase		Narrowphase		Island Processing		LCPsolver		ALL	
Max Error	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev
0.0001%	0.00	0.00	0.01	0.07	0.00	0.04	0.00	0.05	0.01	0.06
0.0010%	0.00	0.00	0.01	0.07	0.00	0.05	0.00	0.05	0.01	0.07
0.0100%	0.00	0.00	0.01	0.08	0.01	0.06	0.01	0.06	0.01	0.08
0.1000%	0.00	0.00	0.01	0.10	0.01	0.05	0.01	0.06	0.02	0.09
1.0000%	0.00	0.00	0.02	0.11	0.03	0.04	0.01	0.07	0.03	0.09
10.0000%	0.00	0.00	0.02	0.10	0.03	0.07	0.03	0.10	0.04	0.09
100.0000%	0.00	0.00	0.01	0.06	0.04	0.05	0.04	0.05	0.03	0.06

Gap	Broadphase		Narrowphase		Island Processing		LCPsolver		ALL	
Max Error	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev	Avg	Stddev
0.0001%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.0010%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.0100%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.1000%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1.0000%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
10.0000%	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
100.0000%	0.000	0.000	0.000	0.000	inf	inf	0.000	0.000	0.000	0.000

Figure 6: Average and Standard Deviation of Error-Injection

ples of a random variable. More samples lead to a converging mean which in our case is zero.

To further support the choice made in Table 1, we consider four approaches: confirming our thresholds visually, comparing our errors to a very simple scenario with clear error thresholds [O’Sullivan et al. 2003], comparing the magnitude of our observed error to constraint reordering, and examining the effect of the time-step on this error.

4.2.1 Threshold Evaluation 1

First we visually investigate the differences in our thresholds. The initial pass of our visual inspection involved watching each error-injected scene in real-time to see how believable the behavior looked, including the presence of jitter, unrealistic deflections, etc. This highly subjective test confirmed our thresholds, and only experiments with error rates above our thresholds had clear visual errors – bricks moving on their own, human figures flying apart, and other chaotic developments.

Second, we specifically flagged errors that exceeded the thresholds from [O’Sullivan et al. 2003]. By using their thresholds, we could filter out errors that were certainly imperceptible. Objects that were not flagged were made translucent, and objects that had been flagged as having an error were colored according to the type of error. We analyzed this behavior by slowly walking frame by frame through the simulation. The errors we saw for error injection at or below the thresholds of table 1 were transient – typically lasting only a single frame – and we would not have been able to distinguish them were it not for the coloring of objects and our pausing

at each frame to inspect the object positions.³

4.2.2 Threshold Evaluation 2

We recreated the experiment used in [O’Sullivan et al. 2003] (but with ODE) to generate the thresholds for perceptual metrics. This scenario places two spheres on a 2-D plane: one is stationary and the other has an initial velocity that results in a collision with the stationary sphere. No gravity is used, and the spheres are placed two meters above the ground plane. When evaluating this simple scenario, the thresholds shown in the rightmost column of table 2, extrapolated from prior work [O’Sullivan et al. 2003; Harrison et al. 2004], are sufficient. They are conservative enough to flag catastrophic errors such as tunneling, large penetration, and movement without collision.

However, when applying this method to a complicated game-like scenario, these thresholds become far too conservative. Even the authors of [O’Sullivan et al. 2003] point out that thresholds for simple scenarios may not generalize to more complex animations. In a chaotic environment with many collisions, it has been shown that human perceptual acuity will become even worse - particularly in areas of the simulation that are not the current focal point.

But since these metrics work for this simple example, we will use the error rates from table 1 to inject error into this simple example. Using the thresholds from [O’Sullivan et al. 2003], we were unable to find any errors that exceeded these thresholds – the error rates that we experimentally found in the previous section show no perceptible error for this simple example.

4.2.3 Threshold Evaluation 3

Interestingly, when we enable reordering of constraints in ODE, most of our perceptual metrics will exceed the thresholds from [O’Sullivan et al. 2003], compared to a run without reordering. There is no particular ordering which will generate an absolutely correct simulation when using an iterative solver such as ODE’s quickstep. Changes in the order in which constraints are solved can result in simulation differences. The ordering present in the deterministic, baseline simulation is arbitrarily determined by the order in which objects were created during initialization. The same exact set of initial conditions with a different initialization order will result in constraint reordering relative to the original baseline.

Therefore, to understand this inherent variance in the ODE engine, we have again colored objects with errors and analyzed each frame of our simulation. The variance seen when enabling/disabling reordering results in errors that are either imperceptible or transient. Based on this analysis, we will use the magnitude of difference generated by reordering as a comparison point for the errors we have experimentally found tolerable in table 1. Our goal is to leverage a similar level of variance as what is observed for random reordering when measuring the impact of error in PBA.

Table 2 shows the maximum errors in our perceptual metrics when enabling reordering versus a baseline simulation of a complex scene without any reordering. As a further reference point, we also include the perceptual thresholds from the simple simulation described in [O’Sullivan et al. 2003]. For the simple simulation, some of our metrics were not used, and we mark these as not available (NA). For baseline simulation, only absolute metrics (i.e. those that require no comparison point like gap and penetration) are shown,

³One such experiment is shown in the accompanying video.

and relative metrics that would ordinarily be compared against the baseline itself (i.e. linear velocity) are also marked *NA*.

Perceptual Metrics	Random Reordering	Baseline	Simple Threshold
Energy (% change)	-1%	NA	NA
Penetration (meters)	0.25	0.29	NA
Constraint Error (ratio)	0.05	0.05	0.03/0.2
Linear Vel (ratio)	5.27	NA	-0.4/+0.5
Angular Vel (radians/sec)	4.48	NA	-20/+60
Linear Vel Angle (radians)	1.72	NA	-0.87/+1.05
Gap (meters)	0.01	0.01	0.001

Table 2: *Perceptual Metric Data for Random Reordering, Baseline, and Simple Simulations*

The first thing to notice from these results is that the magnitude of maximal error for a complex scene (reordering and baseline) can be much more than the simple scenario. The second thing to notice is that despite some large variances in velocity and angle of deflection, energy is still relatively unaffected. This indicates that these errors are not catastrophic, and do not cause the simulation to blow up. It is also interesting to notice the magnitude of penetration errors. Penetration can be controlled by using a smaller time step, but by observing the amount of penetration from a baseline at a given time step, we can ensure that it does not get any worse from introduced error.

The sheer magnitude of the errors from reordering relative to the very conservative thresholds from prior work [O’Sullivan et al. 2003] demonstrates that these thresholds are not useful as a means of guiding the tradeoff between accuracy and performance in large, complex simulations. Furthermore, the similarity in magnitude of the errors we found to be tolerable and the errors from reordering establish credibility for the results in table 1.

4.2.4 Threshold Evaluation 4

Some metrics, such as penetration and gap, are dependent on the time-step of the simulation – if objects are moving rapidly enough and the time-step is too large, large penetrations can occur even without injecting any error. To demonstrate the impact of the time-step on our error metrics, we reduce the time step to 0.001. The maximum penetration and gap at this time-step when injecting a maximum 0.1% error into all phases of PBA reduce to 0.007 meters and 0.005 meters respectively. The same metrics for a run without any error injection reduce to 0.000 meters and 0.009 meters respectively. Both metrics see a comparable reduction when shrinking the time-step, which demonstrates that the larger magnitude penetration errors in figure 5 are a function of the time-step and not the error injected.

5 Precision Reduction

Now that we can evaluate our perceptual metrics for complex scenarios, we apply our findings to the hardware-optimization technique of precision reduction.

5.1 Prior work

The IEEE 754 standard [Goldberg. 1991] defines the single precision floating-point format as shown in figure 7. The first bit is the sign bit, the next eight bits are the exponent bits, and the next 23 bits are the mantissa (fraction) bits. There is an implicit 1 at the

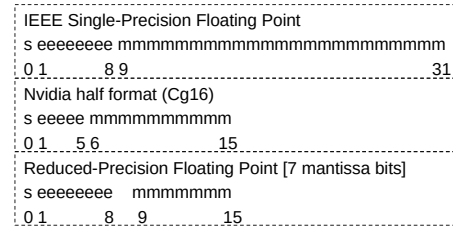


Figure 7: *Floating-point Representation Formats (s = sign, e = exponent, and m = mantissa)*

beginning of the mantissa, so logically there are 24 mantissa bits. When reducing a X number of mantissa bits, we remove the least-significant X bits. There will never be a case where all information is lost since there is always an implicit 1.

For graphics processing, Nvidia has proposed the 16-bit “half” format (Cg16) [Mark et al. 2003] as shown in Figure 7. There is one sign bit, and both the exponent and mantissa bits are reduced. The Cg16 format has been used for several embedded applications [Etiemble et al. 2005; Lacassagne et al. 2005; Eilert et al. 2004] outside of graphics. Reducing the exponent bits reduces the range of representable values. While this is tolerable for media applications, physics simulation is highly sensitive and quickly explodes based on our simulations.

Our methodology for precision reduction follows two prior papers [Samani et al. 1993; Fang et al. 2002]. Both prior work emulate variable-precision floating point arithmetic by using new C++ classes.

Prior work [Goddeke et al. 2006] has also compared the use of double precision solvers with native, emulated, and mixed-precision double precision support. They conclude that a mixed precision approach gains performance by saving area while maintaining accuracy. Our work further pushes the boundary of this area by reducing precision below that of IEEE single-precision configured at a per-phase granularity.

5.2 Methodology

We apply precision reduction first to both input values, then to the result of operating on these precision-reduced inputs. This allows for more accurate modeling than [Samani et al. 1993] and is comparable to [Fang et al. 2002]. Two rounding modes are supported (round to nearest and truncation). We use function calls instead of overloading arithmetic operations with a new class. This enables us to separately tune the precision of code segments (such as computation-phase) instead of the entire application. As the data will show, there is a significant difference between the two granularities. We support the most frequently executed FP operations for physics processing which have dedicated hardware support: +, -, and *. Denormal handling is unchanged, so denormal values are not impacted by precision reduction.

Our precision reduction analysis will focus on mantissa bit reduction because preliminary analysis of exponent bit reductions shows high sensitivity (no tolerance for even a single bit of reduction). A single exponent bit reduction can cause up to orders of magnitude errors being injected. If we follow the IEEE convention of biased exponents, using 7 instead of 8 bits reduces the exponent range from [-126,127] to [-63,64]. This can result in large % errors as compared to the small % errors injected due to precision

reduction.

5.3 Per Phase Precision Analysis

The following section analyzes the precision reduction tolerance of each phase as well as *Narrowphase* + *LCP*. The reason for examining this combination of phases instead of all phases is because of their highly parallel nature, described in section 2.4. The goal of precision reduction is to reduce the die area of FPUs – and the fine-grain parallelism in these two phases would be exploited by small, relatively simple cores with FPUs in physics accelerators such as PhysX or in GPUs. The FPUs in such simple cores would occupy a considerable amount of overall core area.

Broadphase and *Island Processing* do not have vast amounts of fine-grain parallelism, and would therefore need a smaller number of more complex cores to ensure high performance. Therefore it is unlikely that the FPU would occupy a sufficiently large fraction of the complex core to make precision reduction worthwhile. However, we still present results here for all four phases for completeness.

Based on the error tolerance shown in Table 1, we can numerically estimate the minimum number of mantissa bits for each phase. When using the IEEE single-precision format with an implicit 1, the maximum numerical error from using an X -bit mantissa with rounding is $2^{-(X+1)}$ and with truncation is 2^{-X} . Rounding allows for both positive and negative errors while truncation only allows negative errors.

Since base 2 numbers do not neatly map to the base 10 values shown in Table 1, we present a range of possible minimum mantissa bits in table 3 for rounding and truncation. Now that we have an estimate on how far we can take precision reduction, we evaluate the actual simulation results to confirm our estimation.

Mantissa Bits Derived	Broadphase	Narrowphase	Island Processing	LCP
[round, truncate]	[5-6, 6-7]	[5-6, 6-7]	[12-13, 13-14]	[5-6, 6-7]

Table 3: Numerically-derived Min Mantissa Precision Tolerated for Each Computation Phase

The top left graph of Figure 8 shows the actual simulation error generated from precision reduction with rounding. This data corresponds to the numerical analysis of $2^{-(X+1)}$.

Following the same methodology we used in the section 4, we summarize the per-phase minimum precision required in Table 4 based on data in Figure 8. When comparing Table 4 and Table 3, we see that the actual simulation is more tolerant than the stricter numerically-derived thresholds. This gives further confidence in our numerical error tolerance thresholds.

We also studied the effects of truncation instead of rounding when reducing precision (data not shown due to space constraints). Truncation further optimizes area by removing the adder used for rounding. The metrics point to a minimum of 7 bits for the mantissa when executing both *Narrowphase* and *LCP* with reduced precision. The perceptual believability of 7-bit mantissa simulation is supported by the fact that the average and standard deviation are well below the very conservative thresholds of [O’Sullivan et al. 2003; Harrison et al. 2004].

Mantissa Bits Simulated	Broadphase	Narrowphase	Island Processing	LCP	Narrow + LCP
[round, truncate]	[3, 4]	[4, 7]	[7, 8]	[4, 6]	[5, 7]

Table 4: Simulation-based Min Mantissa Precision Tolerated for Each Computation Phase

5.4 Hardware Area and Performance Improvement

For commercial physics accelerators such as PhysX and GPUs, simple cores make up the bulk of the die area. In 90nm process technology, a simple processor core’s area (assuming a $3mm^2$ core) would have the following breakdown: 30% memory area ($0.9mm^2$), 50% logic and flops ($1.5mm^2$), 10% register file ($0.3mm^2$), and 10% overhead ($0.3mm^2$).

Reducing the mantissa from 24 (23 + implicit 1) bits to 8 (7 + implicit 1) bits reduces the logic cost from $1.5mm^2$ to about $0.5mm^2$. The precision-reduced core is 33% smaller than the IEEE single-precision core. This estimate is based on results from a high-level synthesis tool tuned for generating area-efficient floating point datapaths.

This 33% reduction in core area can be leveraged to increase the number of cores in the same silicon area, reduce the overall silicon area required for the cores, add performance-enhancing reduced-precision SIMD hardware to each core, reduce energy by turning off logic to handle unnecessary bits, or increase core clock frequency, just to name a few options. A hybrid chip design that has support for low-precision FP (at a higher throughput) and single-precision FP (at a lower throughput) is a good way to exploit perceptual tolerance because the developer can choose between high-throughput and high-accuracy. Components like the LCP solver and Narrow-phase (which dominate the run time of PBA) would greatly benefit from the higher throughput, and as we have shown, they are very tolerant of reduced accuracy.

While the exact precision reduction tolerance may vary across different physics engines, this study shows the potential for leveraging precision reduction for hardware optimizations such as area reduction, SIMD hardware, energy reduction, or increase core clock frequency.

6 Simulation Time-Step

As described in Section 2.4, the simulation time-step largely defines the accuracy of simulation. In this section, we apply a similar methodology to study the effects of scaling the time-step. We restrict the data shown here to % energy difference as it has been shown to be the main indication of simulation stability. Because time advances in different granularities for these simulations, user input needs to be injected at a common granularity where all simulations see the inputs at the exact same time. The % energy difference also needs to be computed at a common interval for all time-steps. Because of this restriction, we show results for time-steps of 0.001, 0.002, 0.003, 0.004, 0.005, 0.01, 0.015, 0.02, 0.03, and 0.06. The common interval used is 0.06 second.

As shown on Figure 9, the maximum time-step (out of the ones simulated) for the scenario we tested is 0.03 second. Although the appropriate time-step may depend on details of the exact scenario, this methodology can be leverage to tune this physics engine parameter for optimal performance.

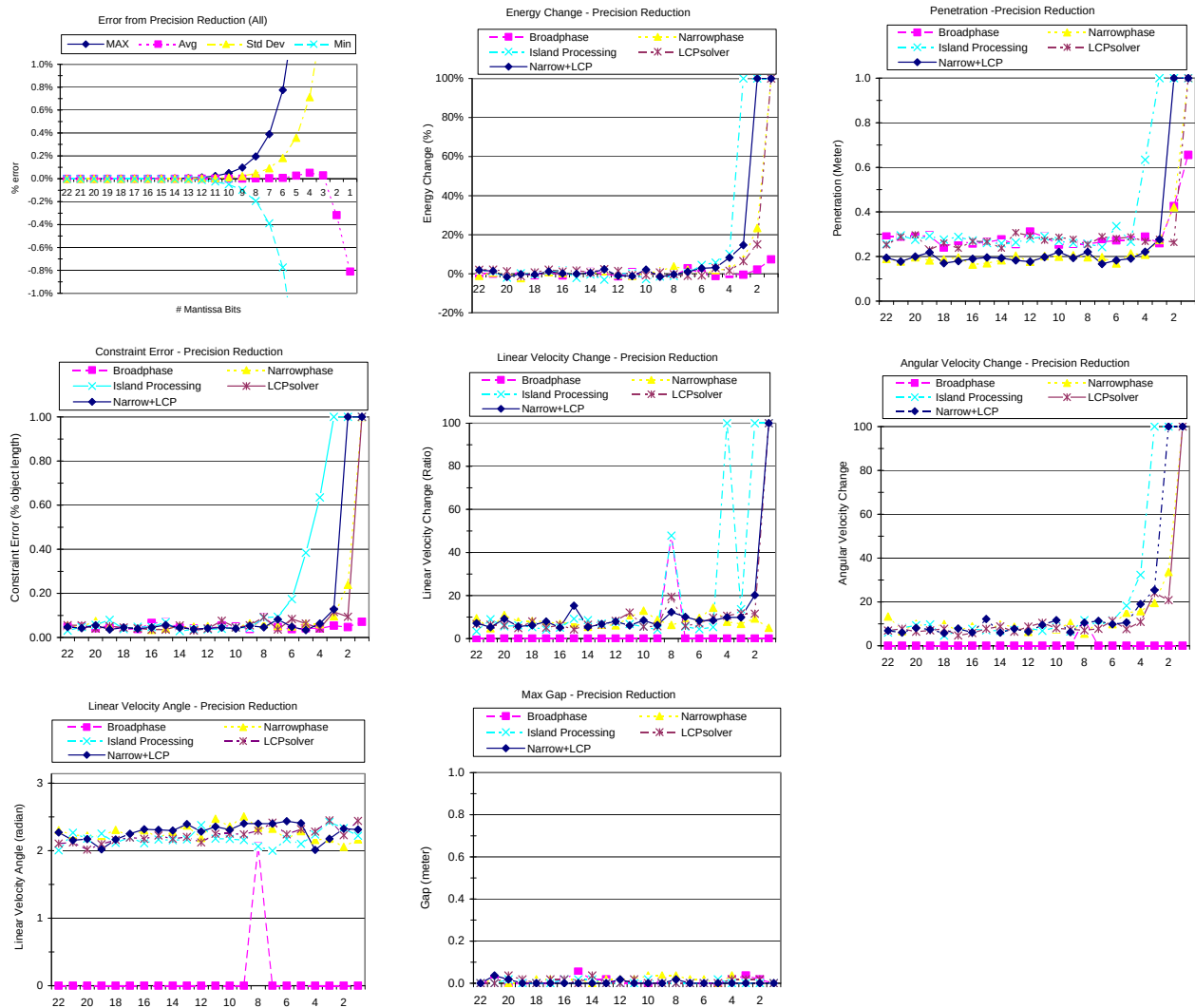


Figure 8: Perceptual Metrics Data for Precision Reduction. X-axis shows the number of mantissa bits used. Note: Extremely large numbers and infinity are converted to the max value of each Y-axis scale for better visualization.

7 Conclusion

We have addressed the challenging problem of identifying the maximum error tolerance of physical simulation as it applies to interactive entertainment applications. We have proposed a set of numerical metrics to gauge the believability of a simulation, explored the maximal amount of generalized error injection for a complex physical scenario, and examined two ways of exploiting this error tolerance: reducing precision and tuning time-step.

References

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

BARAFF, D. 1997. *Physically Based Modeling: Principles and Practice*. SIGGRAPH Online Course Notes.

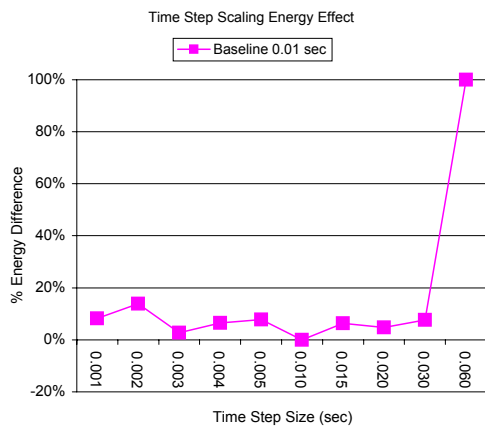


Figure 9: Effect on Energy with Time-Step Scaling

- BARZEL, R., HUGHES, J., AND WOOD, D. 1996. Plausible motion simulation for computer graphics animation. In *Computer Animation and Simulation*.
- CARLSON, D., AND HODGINS, J. 1997. Simulation levels of detail for real-time animation. In *Proceedings of Graphics interface*.
- CHENNEY, S., AND FORSYTH, D. 2000. Sampling plausible solutions to multi-body constraint problems. In *ACM SIGGRAPH*.
- EILERT, J., EHLIAR, A., AND LIU, D. 2004. Using low precision floating point numbers to reduce memory cost for mp3 decoding. In *IEEE 6th Workshop on Multimedia Signal Processing*.
- ETIEMBLE, D., BOUAZIZ, S., AND LACASSAGNE, L. 2005. Customizing 16-bit floating point instructions on a nios ii processor for fpga image and media processing. In *3rd Workshop on Embedded Systems for Real-Time Multimedia*.
- FANG, F., CHEN, T., AND RUTENBAR, R. 2002. Floating-point bit-width optimization for low-power signal processing applications. In *International Conference on Acoustic, Speech, and Signal Processing*.
- GODDEKE, D., STRZODKA, R., AND TUREK, S. 2006. Performance and accuracy of hardware-oriented native-, emulated-, and mixed-precision solvers in fem simulations. In *International Journal of Parallel, Emergent and Distributed Systems*.
- GOLDBERG., D. 1991. What every computer scientist should know about floating-point arithmetic. In *ACM Computing Surveys (CSUR)*.
- HARRISON, J., RENSINK, R. A., AND VAN DE PANNE, M. 2004. Obscuring length changes during animated motion. In *ACM Transactions on Graphics, SIGGRAPH 2004*.
- HAVOK INC. Havokfx.
- HOFSTEE, P. 2005. Power efficient architecture and the cell processor. In *HPCA11*.
- LACASSAGNE, L., ETIEMBLE, D., AND OULD KABLIA, S. 2005. 16-bit floating point instructions for embedded multimedia applications. In *Seventh International Workshop on Computer Architecture for Machine Perception*.
- MARK, W., GLANVILLE, R., AKELEY, K., AND KILGARD, M. 2003. Cg: a system for programming graphics hardware in a c-like language. In *ACM SIGGRAPH*.
- MCDONNELL, R., DOBBYN, S., COLLINS, S., AND O'SULLIVAN, C. 2006. Perceptual evaluation of lod clothing for virtual humans. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 117–126.
- MULLER, M., HEIDELBERGER, B., HENNIX, M., AND RATCLIFF, J. 2006. Position based dynamics. In *Proceedings of the 3rd Workshop in Virtual Reality Interactions and Physical Simulation*.
- O'SULLIVAN, C., AND DINGLIANA, J. 2001. Collisions and perception. In *ACM Transactions on Graphics*.
- O'SULLIVAN, C., DINGLIANA, J., GIANG, T., AND KAISER, M. 2003. Evaluating the visual fidelity of physically based animations. In *ACM Transactions on Graphics, SIGGRAPH 2003*.
- O'SULLIVAN, C., HOWLETT, S., MCDONNELL, R., MORVAN, Y., AND O'CONOR, K. 2004. Perceptually adaptive graphics. In *Eurographics 2004, State of the Art Report*.
- PROFFIT, D. Viewing animations: what people see and understand and what they don't. Keynote address, ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2006.
- REITSMA, P. S. A., AND POLLARD, N. S. 2003. Perceptual metrics for character animation: sensitivity to errors in ballistic motion. *ACM Transactions on Graphics* 22, 3, 537–542.
- REITSMA, P., AND POLLARD, N. 2003. Perceptual metrics for character animation: Sensitivity to errors in ballistic motion. In *ACM Transactions on Graphics*.
- SAMANI, D. M., ELLINGER, J., POWERS, E. J., AND SWARTZLANDER, E. E. J. 1993. Simulation of variable precision ieee floating point using c++ and its application in digital signal processor design. In *Proceedings of the 36th Midwest Symposium on Circuits and System*.
- SEUGLING, A., AND ROLIN., M. 2006. Evaluation of physics engines and implementation of a physics module ina 3d-authoring tool. In *Master's Thesis*.
- SMITH, R. Open dynamics engine.
- YEH, T. Y., FALOUTSOS, P., AND REINMAN, G. 2006. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Video Game Symposium*.

<https://web.archive.org/web/20080221201327/http://www.cs.ucla.edu/~pfal/papers/MICRO-07-art.pdf>

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

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

Abstract

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

1 Introduction and Motivation

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

With such massive parallelism, chip multiprocessor [22] (CMP) designs with large numbers of simple cores, such as ParallAX [32], can achieve desirable frame rates. In these

designs, the more cores that can be packed into a single piece of silicon, the more parallelism that can be exploited. However, physics simulation uses extensive floating-point (FP) calculation, and could require dedicated floating-point unit (FPU) resources at each simple core. If these resources could be effectively shared across multiple cores, the area savings would result in greater core density and therefore greater exploitation of parallelism. However, contention for shared FP resources can impact performance depending on the degree of sharing.

Fortunately, PBA offers more than just massive parallelism to architects as a means of achieving high performance. In particular it can be *error tolerant* – interactive entertainment requires perceptually believable results rather than numerically correct results. The user must be able to suspend disbelief and become immersed in the virtual world, but human perceptual tolerance to small errors is quite high [7, 23]. While the amount of error tolerance in PBA varies with the individual phase of the simulation and the scenario being modeled, prior work [34] has demonstrated that bounding the difference in total energy during a simulation can reliably ensure believability. We propose to leverage this observation to dynamically tune the precision of FP operations in PBA to provide just enough precision to approximately conserve simulation energy, and therefore ensure believability.

Our paper makes the following contributions:

- We propose dynamic FP precision tuning in PBA to leverage perceptual error tolerance.
- We identify and leverage three main benefits of precision reduction:
 1. *Increased Number of Trivial Operations:* Precision reduction enables new conditions and increased chances that a FP computation will not require the use of an FPU – such operations are trivial (e.g., adding shifted out operand, multiplication by one).

2. *Value Locality*: Precision reduction improves the locality that exists (a) among similar objects in similar scenarios and (b) across iterations during the relaxation of constraints.
 3. *Reduced Precision FPU*: Smaller, faster, and less power hungry FPUs can be used to replace full precision FPUs. However, the varying precision requirement necessitates occasional access to full precision FPUs.
- We propose and investigate the hierarchical sharing of FPU resources among simple cores in PBA. The L1 FPU designs leverage the benefits of precision reduction to execute FP operations locally, and the full-precision L2 FPUs are shared among multiple cores for area reduction. Area savings from FPU sharing can translate directly to performance by adding more simple cores. Per phase performance is increased by up to 55% while per phase energy is simultaneously reduced by up to 50% when compared to a baseline without FPU sharing.

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

2 Background and Related Work

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

2.1 Real-Time Physics for Interactive Entertainment

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

Physics Acceleration. Parallax [32] is a heterogeneous architecture for physics acceleration that combines a smaller set of more powerful cores, for simulation regions with limited or coarse-grain parallelism, with a much larger set of simple cores, for regions with massive parallelism. [15] characterizes physics-based simulation for computer animation and visual special effects used for motion pictures. Simulations with a 64-core chip multiprocessor (CMP) show very high on-die and main memory

bandwidth requirements, and most modules have little inter-thread communication.

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

2.2 Conjoined Cores

Kumar et. al [18] proposed having two adjacent cores in a CMP share a single floating-point unit (FPU), crossbar port, instruction cache, and data cache for area savings. They looked at a simple policy where the cores take turns using the shared resources every other cycle, and a more intelligent policy where either core can use a resource in any cycle, but the arbitration priority among the cores switches from cycle to cycle for fairness. Recent work [28] has extended this idea to share other core structures among more than two cores. Sun's Niagara [16] shares a single FPU among eight cores. The cores access the FPU via a shared bus – the latency seen for an individual FP operation is 40 cycles [29]. In this paper, we will use conjoin to describe FPU sharing.

2.3 Floating-Point Optimization

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

Not all FP operations need a functional unit to execute – for certain input operands, FP instructions can be made

trivial. For example, adding any number to zero is a trivial operation: the result is simply the original number. Early studies of trivial instructions [25, 26] focused on FP benchmarks (SPEC 92 and Perfect Club) using a set of 8 trivial computations: multiplication (by 0, 1, and -1), division (X/Y with $X = 0, Y \neq 0$, and square root (of 0 and 1). Most trivial operations were FP multiplies, and the percentage of trivial operations per program ranged from near zero to 7%. They only saw performance improvements in the range of 2-4% for a processor configuration similar to modern cores.

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

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

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

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

Another technique related to both trivialization and memoization is value prediction. Last value prediction [19] reduces instruction latency and breaks data dependencies. Yeh et al. [33] proposed fuzzy value prediction where a certain amount of error can be tolerated without recovery. However, value prediction requires speculative execution to be effective. Therefore, these techniques will likely not be cost effective for the simple, in-order fine-grain cores we are targeting in this paper. We refer to [32] for details on performance-area tradeoff of more complex cores vs the simple ones.

3 Methodology

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

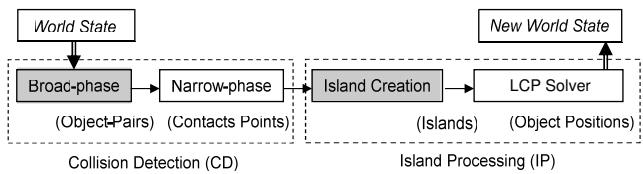


Figure 1. Physics engine flow. All phases are serialized with respect to each other. Unshaded stages can exploit parallelism.

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

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

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

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

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

4 Precision Reduction

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

4.1 Error Tolerance of FP Operations

Real-time physics for interactive entertainment must be believable, but may sacrifice some accuracy. As discussed in section 2.1, the total energy in the simulation is one way of measuring the amount of tolerable error. By using the law of energy conservation, the application can compute the energy difference between successive simulation steps to determine whether the simulation is diverging towards instability. It should be noted that this energy conservation takes into account externally injected energy by the player or the game scenario. This energy difference can be compared against empirically-derived thresholds to ensure believability. We added the dynamic monitoring of simulation energy to the physics engine. This additional code is added to the end of the simulation loop after integration. As each object is moved to the new position, its energy can be calculated and stored. The accumulation of energies can be decoupled from the simulation loop calculation and is performance insensitive. With one 32-bit floating point energy value per object, this does not present a huge bandwidth bottleneck. The overhead for computing energy in ODE scales linearly with the number of objects and particles. Our code to compute energy for each object requires 67 instructions, and each particle requires roughly 27 instructions. For the most complicated benchmark in PhysicsBench, *Mix*, this overhead translates to less than 0.3% increase in dynamic instruction count.

4.1.1 Minimum Required Precision

One way to leverage the error tolerance of real-time physics is to reduce the amount of precision used in floating-point (FP) operations. The minimum required number of mantissa bits before the error becomes discernable varies de-

Benchmark	LCP			Narrowphase		
	RN	J	T	RN	J	T
Breakable	8	17	13	17	10 (21)	23
Continuous	4	4	4	9	9 (9)	9
Deformable	3	4	8	9	9 (9)	9
Everything	10	10	23	18	10 (17)	19
Explosions	11	13	9	21	14 (14)	13
Highspeed	3	3	8	9	9 (9)	9
Periodic	13	14	23	22	21 (23)	23
Ragdoll	5	5	9	9	9 (21)	9

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

pending on the particular phase of physics simulation and the particular physics scenario being modeled. Table 1 demonstrates this variety across all simulated PhysBench scenarios and across the two different fine-grain phases of ODE. These results are based on 30 frames of simulation following the methodology proposed in [34], and each phase is evaluated independently of the other. We evaluate three different rounding modes for reducing precision: round-to-nearest, jamming, and truncation. Denormal handling remains unchanged.

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

To obtain the benefits of both, we look towards jamming, a rounding technique originally proposed by Burks et al. [8] and used by Fang et al. [11]. Jamming performs a bit-level *OR* of the least significant bit (LSB) and three subsequent guard bits (GBs). The result is placed in the LSB. If LSB is one then the GBs are ignored. If the LSB is zero, then any one in the GBs will result in rounding the LSB to one. This allows for very simple and fast logic, and the mean of injected error is 0 with no biasing.

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

While independently exploring the precision at each phase gives some intuition for the acceptable levels of precision required, we will be adaptively tuning the precision of each phase simultaneously, and therefore the error injected in one phase will impact the precision tolerance of the other phase. Since 31% and 13% of dynamics instructions

on average are FP for LCP and narrow-phase respectively, we fix LCP's required precision at the minimum found during independent exploration and evaluate the new level of precision required for narrow-phase. Dynamically precision tuning results for narrow-phase are shown in parenthesis. For some benchmarks, narrow-phase's precision requirement is higher when considering precision reduction for both phases.

4.2 Dynamic Adaptation of Floating Point Precision

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

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

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

Based on dynamic information, the precision is tuned to prevent severe degradation in simulation quality. Leveraging the results from [34], we select the energy difference between successive steps as the dynamic information for tuning. Based on statically determined thresholds of percent energy difference, the application monitors its own simulation quality. We use a threshold of 10% energy difference based on the results from [34] and our observation of the energy data. When the threshold is violated, the application throttles up the significant precision (to full precision) to prevent simulation blow-up. As noted earlier, this energy difference takes externally injected energy into account. Therefore, the total energy across successive frames may be drastically different without triggering precision adjustment. After the simulation stabilizes, the precision is incrementally reduced after every simulation step until it reaches the minimum value stored in the control register.

There can also be cases where the simulation blows up without warning. In such extreme cases, functional correctness is maintained by re-executing the previous simulation

step at full precision. We leave the full study on software algorithms for controlling re-execution to future work.

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

4.3 Opportunities Enabled by Dynamic Precision Reduction

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

4.3.1 Trivial FP Operations

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

FP Operation	Representation	Trivial when
Add	$X + Y$	$X=0$ or $Y=0$
Subtract	$X - Y$	$X=0$ or $Y=0$
Multiply	$X * Y$	$X=0$ or ± 1 , or $Y=0$ or ± 1
Divide	X / Y	$X=0$ or $Y=\pm 1$

Table 2. Conventional trivial cases

Conditions for Increased Trivialization
Small mass difference between objects
Zero linear and angular velocities before collision
Small size difference between objects
Simple object shapes
Use of ground and gravity
Higher amount of articulation (human vs box)

Table 3. Factors increasing trivialization

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

1. *Add/Subtract*: If the magnitude of exponent difference between the operands is greater than the number of

valid mantissa bits plus one, then the operation becomes trivial: the result will simply be the larger of the two operands. This is effectively addition/subtraction by zero. The addition of one take into account the implicit one of FP representation. Full precision of the non-trivial operand can be used to minimize injected error.

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

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

4.3.2 Trivialization Details

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

In *LCP*, the core computation involves multiplications of 6-element matrices such as constraint forces, jacobian matrices, and inverse jacobians. For each matrix, three elements refer to the linear components of each axis with val-

ues from the normal vector at the contact point. The other three elements refer to angular components of each axis computed as the cross-product of the contact normal with the vector defined by the contact point and the first object's point of reference.

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

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

Benchmark	Trivial (Add,Mult)		Memo (Add,Mult)	
	23-bit	Reduced	23-bit	Reduced
Bre	36, 34	48, 41	0, 2	1, 8
Con	49, 43	71, 62	0, 1	8, 38
Def	32, 31	61, 64	0, 2	7, 35
Eve	35, 33	43, 38	0, 3	1, 6
Exp	28, 25	38, 29	0, 7	1, 10
Hig	27, 23	54, 49	0, 8	11, 51
Per	32, 32	34, 34	0, 0	0, 0
Rag	34, 33	52, 53	0, 0	2, 28

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

4.3.3 Locality of FP Operations

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

When memoizing precision reduced values, however, there is certainly the immediate gain from the need to store fewer bits per entry. But, there is also an increase in coverage – with reduced precision, there are fewer possible

values and combinations to memoize. Values that are very close will become one value with reduced precision, potentially combining multiple table entries into one. By leveraging reduced precision, we see that the memoization hit rate for FP multiply increases to the level seen for SPEC benchmarks. All memoization results are shown in Table 4. Trivializable operations are filtered from accessing the memoization tables.

4.3.4 Lookup Table

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

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

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

For FP add lookup, the smaller operand is first shifted based on the exponent difference. This operation only requires a small and fast 5-bit shifter. This means that the implicit 1 of the smaller operand will be seen. Then, the concatenation of the larger operand and the shifted operand is used as the index. In FP add lookup entries, an additional bit is stored to indicate the need to increment the exponent by 1 for normalization of a carry out. Since we allocate 8 bits per entry, there is room for such annotation. One corner case for FP add is when the two operands have the same exponent. In this case, indexing with the default scheme will result in aliasing with a smaller operand which has been shifted. One solution is to first detect this by an exponent

difference of zero, then appropriately handle the most significant bit after the leading one. This solution does not require any changes to the default scheme.

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

All data presented is generated using Cacti 3.0 [30]. For the memoization tables, we use 64 address bits and 32 data bits. For the lookup table, we use 11 address bits and 8 data bits – and disregard the contribution of tag area. For benchmarks requiring precision less than 6 bits, the 2K-entry lookup table (entries are 1B each) achieves a better hit-rate than using the two (one for each operation type) 256 entry 16-way memoization tables (entries at 12B each) we simulate. With the lookup table, the area requirement is reduced by 77%. We assume a single cycle latency for lookup. We implement the lookup table in dedicated scratch-pad memory. When the required precision exceeds five bits, the lookup table is no longer useful and the memory can be used for other purposes. Detailed evaluation of this additional opportunity is left for future work.

Table Type	Latency (ns)	Energy (nJ)	Area (mm^2)
Lookup	0.40	0.03	0.08
Memo	0.88	0.73	0.35

Table 5. Lookup vs memoization table

5 FPU Sharing

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

We use simple, in-order shader-class cores with the parameters shown in Table 6 in this study, and assume a core area excluding the floating-point unit (FPU) of $2mm^2$ in 90nm technology. Published data on FPU area ranges widely within the same technology due to parameters such as the design flow, target latency, area allocation, and circuit optimization. To make this exploration more gener-

ally applicable, we explore four FPU designs with varying area requirements: $1.5mm^2$, $1.0mm^2$, $0.75mm^2$, and $0.375mm^2$. These area estimates are based respectively on results from a high-level synthesis tool tuned for generating area-efficient floating point data-paths, and published data from [18], [10], and [21].

Processor Pipeline	1-wide, 5-stages, in-order execution
Functional Units	1 int, 1 fp, 1 ld/st
Window/Scheduler	8,4
Branch Predictor	384B YAGS + 4-entry RAS
Local Inst Memory	4KB, 1-cycle
Local Data Memory	4KB, 1-cycle
Technology	90 nm
Clock Frequency	1GHz
FP Latencies	fpALU (4) fpMult (4) fpDiv (20)
INT Latencies	iALU (1) iMult (6) iDiv (40)

Table 6. Fine-grain shader core design

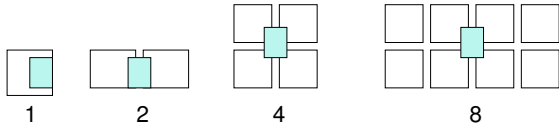


Figure 2. Layout for one, two, four, and eight cores sharing an FPU.

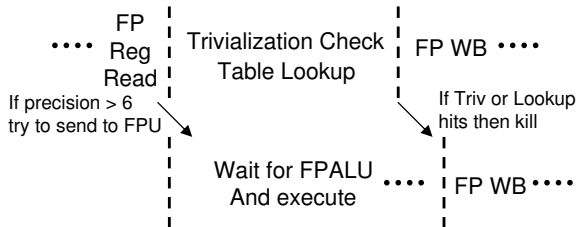


Figure 3. Pipeline for FPU sharing.

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

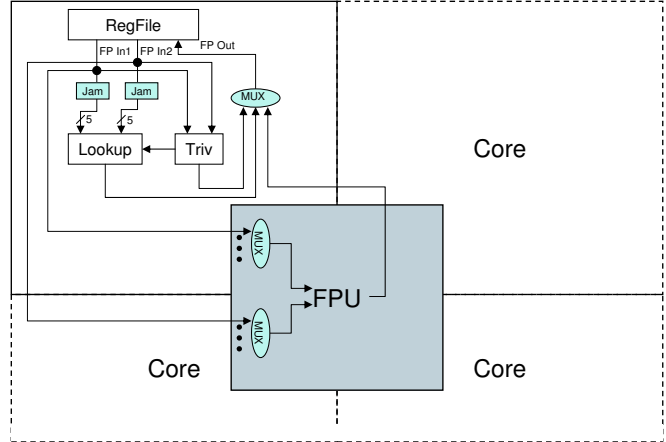


Figure 4. Four cores sharing a single FPU – each core has a trivialization and memoization unit.

For sharing among 2 cores, 4 cores and 8 cores, we assume 0-cycle, 1-cycle, and 2-cycle latency penalties. Based on mirroring of cores as described in [17, 18], sharing the FPU among 2 cores can be accomplished with no added latency. For the 4 core and 8 core sharing, we add conservative latency estimates for the additional wire delay from each core to the FPU. Using a $2mm \times 2mm$ aspect ratio for the cores, we assume a 2mm wire length for 4-core sharing and a 4mm wire length for 8-core sharing. Based on the 90nm information for the narrowest wire width in [9], the one-way added delays are 0.074 ns and 0.296 ns respectively. In section 5.2, we will evaluate the performance sensitivity to added FPU latency.

5.1 Hierarchical FPUs

To attack the overhead of sharing, we propose a hierarchical FPU (HFPU) architecture. A small, simple L1 FPU is placed local to each core for low latency and high throughput, and the full precision FPUs are shared at the L2 level to maximize area efficiency and to satisfy high-precision execution. This is similar in concept to a cache hierarchy, but we are sharing execution resources instead of storage resource. If the required level of precision for a particular operation exceeds the capability of the L1 FPU, then the operation must be performed in the L2 FPU. While it may be possible to use software support [12] to execute in the L1 FPU, we do not consider this alternative in this study.

We propose and evaluate the following L1 FPU design alternatives. All L1 designs contain logic to handle simple non-arithmetic FP operations. If a given operation exceeds the capability of the L1 FPU, it will be executed in the L2

FPU. We list the alternatives by increasing complexity of the L1 FPU design.

1. *Conventional Trivialization*: The L1 FPU only contains conventional trivialization logic – no precision reduction.
2. *Reduced Precision Trivialization*: The L1 FPU uses our reduced precision trivialization logic. Additional exponent logic is required.
3. *Lookup Table + Reduced Precision Trivialization*: The L1 FPU has our reduced precision trivialization logic, exponent logic, and a local 2K-entry lookup table. FP add and multiply instructions using mantissa precision of less than six bits make use of the lookup table. Figure 4 demonstrates a version of this design where four cores share a single L2 FPU. Each core has an L1 FPU with trivialization and lookup table hardware. The pipeline change required for this FPU sharing is illustrated in figure 3.
4. *mini-FPU + Reduced Precision Trivialization*: In addition to our reduced precision trivialization logic, this L1 FPU has a *mini-FPU*: an FPU design that uses a 14-bit mantissa and 8-bit exponents. FP add and multiply operations with mantissa precision of less than 15 bits can execute in this mini-FPU. The 14-bit precision is chosen based on the (benchmark coverage)/(area) ratio derived from Table 1. The intent of this design is to provide potentially higher coverage at the L1 FPU, but at a greater area cost.

Required Mechanism	Latency (cycles)
Trivialization or Look-up Table	1
mini-FPU	3
Full FPU Arbitration	0-1 for 2-core sharing 0-3 for 4-core sharing 0-7 for 8-core sharing
Interconnect Overhead	0 for 2-core sharing 1 for 4-core sharing 2 for 8-core sharing
Full FPU Latency	4 for fpALU and fpMult 20 for fpDiv

Table 7. Variable FP Latency

5.1.1 Variable FP Latency

All components of HFPU’s variable FP latency are listed in Table 7. If an operation can be satisfied by trivialization or look-up table, it takes one cycle to complete. If the mini-FPU is available and the operation can be satisfied by it, the operation takes three cycles. If an operation requires the

full FPU, the exact latency is determined by accumulating the required latencies of arbitration, interconnect overhead, and full FPU operation.

While the latency for each operation is variable, the latency of a non-trivial operation is known at issue time. This is accomplished by using a local counter to indicate current round-robin arbitration overhead. The interconnect overhead and FPU latencies are fixed.

For the simple in-order, single-issue cores we modeled, there is no dynamic instruction scheduler. Instructions are dispatched in program order, so instruction scheduling is done at compile time. SESC uses the MIPS-ISA, and we compile with the -mips2 option without targeting a specific microarchitecture. We do not leverage the fact that FP op might be single cycle, and the same binary is used for all simulations. If the operation is satisfied by the trivial or look-up table logic, then the operation completes in 1 cycle. If not, the pipeline stalls until the operation is completed. Since the cores are in-order single-issue cores, the performance impact of these stalls is not significant. In the case of the look-up table, if the required precision can be satisfied by the look-up table, 100% of operations sent to the look-up table will be satisfied.

For more complex cores with dynamic instruction schedulers, different implementations are possible:

1. Optimistically assumes 1 cycle latency. If the operation is not satisfied by the trivial or look-up table logic, the pipeline stalls until the operation is completed. Once an operation is detected to require the full FPU, the actual number of required cycles is known.
2. Pessimistically assumes the use of the full FPU. The required number of cycles is known at beginning of issue. This scheduling policy eliminates stalls/flushes/replays and retains the energy savings shown in the paper.

5.2 Results

For our performance simulations, we assume a baseline Parallax architecture with 128 simple (fine-grain) cores, as described in Table 6, connected by a mesh interconnect. Four different FPU designs with varying area requirements (1.5 mm^2 , 1.0 mm^2 , 0.75 mm^2 , and 0.375 mm^2) are evaluated. Area estimates include per core area (2 mm^2 each), per core mesh interconnect routers (0.19 mm^2 each) [31], the full-precision FPUs, and additional logic and structures for different L1 FPU designs. The details of each configuration are shown in Table 8.

For the 14-bit mantissa mini-FPU, we assume an area requirement of 60% that of the full precision 23-bit mantissa FPU. This ratio is based on the estimates of both prior work on reduced precision FPU [11] and results from a high-level

synthesis tool tuned for generating area-efficient floating point data-paths. All area data is for 90nm technology.

Architecture	Area Overhead Per Core (mm^2)	Avg Per Core IPC 4 Cores Per L2-FPU Narrowphase, LCP
Baseline (Conjoin)	—	0.347, 0.293
Conv Triv	0.0023	0.376, 0.319
Reduced Triv	0.0079	0.377, 0.334
Reduced Triv Lookup Table	0.0079 + 0.080	0.377, 0.357
Reduced Triv mini-FPU (14bit)	0.0079 + ($0.6 \times$ FP Area)	0.382, 0.364

Table 8. Evaluated designs. Conjoin refers to baseline with FPU sharing. Area overhead per core = additional area added to the 2 mm^2 core.

First, we consider the lower area cost alternatives, and then we will address the mini-FPU. Figures 5(a) (*LCP*) and 5(b) (*Narrow-phase*) demonstrate the aggregate improvement in throughput over the baseline configuration with no FPU sharing for LCP and narrow-phase respectively. The four clusters along the x-axis represent the different FPU area sizes we evaluate, and each cluster has four data points for each architecture representing sharing a single L2 FPU among one, two, four, and eight cores.

Note that any area saved by FPU sharing is used to add more fine-grain cores, taking into account the interconnect and core area as above. Therefore, two competing trends will influence performance: (1) an increase in the number of cores will mean more exploitation of parallelism and (2) an increase in FPU sharing may mean worse per-core performance due to the overhead of sharing. Since our baseline has 128 cores, the total area available varies with the assumed size of the FPU. For the four FPU sizes we evaluate, the total die area is: 472 mm^2 for the 1.5 mm^2 FPU, 408 mm^2 for the 1.0 mm^2 FPU, 376 mm^2 for the 0.75 mm^2 FPU, and 328 mm^2 for the 0.375 mm^2 FPU. Figure 6 shows the total number of cores available to each configuration in the same die area as that of the appropriate baseline configuration with 128 cores.

The data across the two phases from figures 5(a) and 5(b) shows two attractive configurations: *Reduced Precision Trivialization* is better for *Narrow-phase* and *Lookup Table + Reduced Precision Trivialization* is better for *LCP*. As shown by Table 8, the per core IPCs of the two configurations for *Narrow-phase* are the same. The slight degradation with Lookup is due to the lookup table area and the fact that *Narrow-phase* always requires more than five bits of precision – therefore the lookup table is not used. To choose the better design, we also evaluate the energy and power impact of these two HFPU designs.

To estimate the total dynamic energy consumed by FP operations, we leverage the energy data for different sub-units of the FPU presented in [10]. For configurations with trivialization, all FP operations are charged the trivialization logic energy. Non-trivial operations are then charged for the FPU energy. The lookup table is activated when the required precision falls below six bits. In these cases, all FP operations are charged the trivialization plus the lookup energies. The diamonds (secondary y-axis) of Figure 6(b) show a clear energy advantage to using the lookup table for *LCP*.

While performance is improved by sharing FPUs, increased per-FPU utilization could worsen the existing *hot spot* power density problem of FPUs [17]. To gauge the per-FPU utilization, we show the % of FP operations that are trivialized by trivialization or table lookup in the bars (primary y-axis) of Figure 6(b). The HFPU design trivializes 53% of FP operations in the FP intensive phase of *LCP*. This suggests that the HFPU can lower per-FPU utilization when sharing one FPU between two cores as compared to the unshared baseline. For sharing among *four* cores with HFPU, per-FPU utilization will be similar to that of the prior work on conjoined CMPs [18] sharing only among *two* cores.

Next we compare the mini-FPU design to our best performing low-overhead L1 FPU. Figures 7(a) and 7(b) demonstrates the improvement in throughput over the 128-core baseline case for these designs – with figure 6 demonstrating the number of cores possible with each architecture. The mini-FPU design provides some improvement over the baseline, but the area overhead of the L1 FPU reduces the performance of this approach in comparison with the *Lookup Table + Reduced Precision Trivialization* L1 FPU.

Physics requires varying amount of numerical accuracy across different phases and scenarios – a clear differentiating factor between the error tolerance of physics and media processing. In the worst case, full precision is required. Therefore the mini-FPU cannot replace a full precision FPU in a physics processor architecture – there must be the ability to do full-precision FP. This makes the mini-FPU design inherently more costly, and argues for possibly sharing L1 FPUs among cores as well. We look at the case where the mini-FPU is shared among either two or four cores. We limit our exploration to configurations where the L2 FPU is shared by at least as many cores as the L1 FPU. For the Figures, *mini-FPU* refers to no mini-FPU sharing. *mini-FPU 2* shared one mini-FPU among 2 cores, and *mini-FPU 4* shared one mini-FPU among 4 cores. As before, the x-axis (*Cores per FPU*) refers to the degree of L2 FPU sharing.

When looking at the per core throughput in Table 8, the 14-bit mantissa mini-FPU has the highest IPC. This is a result of a 1-cycle reduction in latency and broad coverage of benchmarks (14-bit mantissa satisfies 7/8 benchmarks for

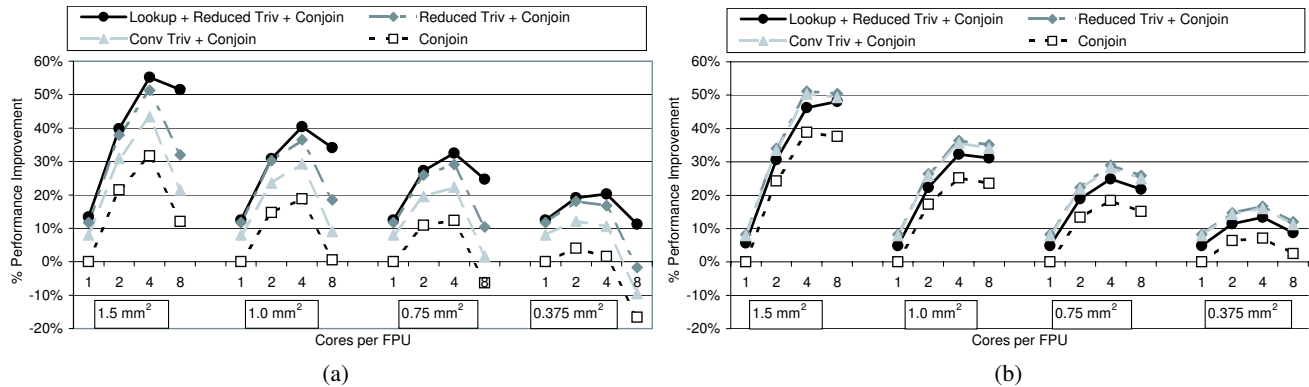


Figure 5. (a) HFPU LCP performance — (b) HFPU narrow-phase performance

LCP and 4/8 benchmarks for *Narrow-phase*). However, the mini-FPU simply cannot pack as many cores into a given silicon area due to its area overhead (as shown in figure 6), resulting in a lower overall throughput. The mini-FPU designs only become more attractive for the most aggressive FPU design (0.375mm^2) at the highest level of sharing.

As shown by the performance and energy data, the best HFPU design we examined when considering both performance and energy is to share one full-precision FPU among 4 cores using the *Lookup Table + Reduced Precision Trivialization* L1 FPU. On average, HFPU improves *LCP* performance over the baseline by 55%, 40%, 33%, and 20% for the four FPU designs. On average, HFPU improves *Narrow-phase* performance over the baseline by 46%, 32%, 25%, and 13%. Total dynamic energy expended for FP operations is reduced by 50% for *LCP* and 27% for *Narrow-phase*.

There are area and timing overheads when sharing a resource like the FPU among multiple cores regardless of the L1 FPU design – these come from the arbitration logic required to choose which core will access the shared resource and the interconnect cost to connect the cores and resource together. We evaluated the performance sensitivity to increased latency from sharing in Figure 8. The baseline for these figures is the performance of the *Lookup Table + Reduced Precision Trivialization* sharing one FPU among two cores. *LCP* (a) is shown to be more sensitive than *Narrow-phase* (b) in this figure. For *LCP*, the performance of the more aggressively designed FPUs suffers when the latency increases beyond a single cycle.

6 Summary and Future Work

In this study, we exploit the inherent perceptual error tolerance of physics-based animation to dynamically adapt the required precision for floating-point computation. Then, we identify and leverage the benefits made possible by dynamic

precision tuning to design a hierarchical floating-point unit (HFPU). This consists of an area-efficient L1 FPU and a full-precision L2 FPU that is shared among a number of cores in the Parallax architecture. Our best performing L1 FPU design consists of our enhanced trivialization logic and a mantissa lookup table – an area efficient design that allows more cores to be packed into a particular silicon area. For the range of FPU designs we consider, this L1 FPU coupled with an L2 FPU that is shared among four cores provides an improvement in overall throughput up to 55% and reduces energy expended in the FPU by up to 50%.

7 Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. Intel Corp., Microsoft Corp., AMD/ATI Corp., and AGEIA Technologies Corp. helped us with their generous support through equipment and software grants. This work was partially supported by NSF grants CCF-0429983 and CCF-0133997, and by SRC grant 1317. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or SRC.

References

- [1] Open dynamics engine. <http://www.ode.org/ode-latest-userguide.html>.
- [2] Physicsbench. <http://sourceforge.net/projects/physicsbench/>.
- [3] AGEIA. Physx product overview. www.ageia.com.
- [4] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. In *IEEE Transactions on Computers*, 2005.
- [5] E. Attoofian and A. Baniassadi. Improving energy-efficiency by bypassing trivial computations. In *The 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

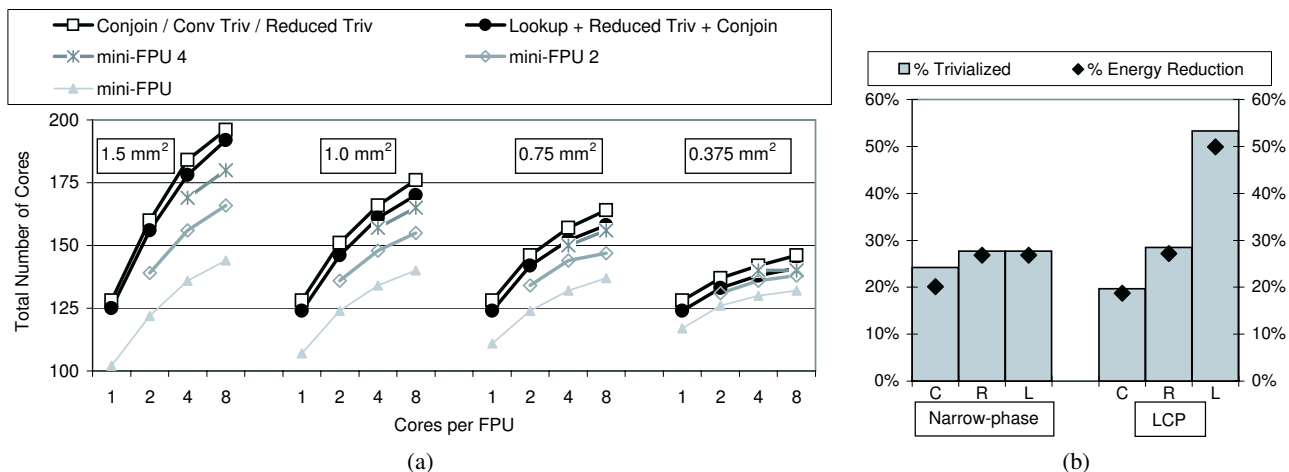


Figure 6. (a) Total number of cores in the same die area as 128-core baseline. The total area used: 472 mm² for 1.5 mm² FPU, 408 mm² for 1.0 mm² FPU, 376 mm² for 0.75 mm² FPU, and 328 mm² for 0.375 mm² FPU. — (b) FP computation % trivialized and energy reduction. C = Conv Triv, R = Reduced Triv, and L = Lookup Table + Reduced Triv.

- [6] D. Baraff. *Physically Based Modeling: Principals and Practice*. SIGGRAPH Online Course Notes, 1997.
- [7] R. Barzel, J. Hughes, and D. Wood. Plausible motion simulation for computer graphics animation. In *Computer Animation and Simulation*, 1996.
- [8] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronics computing instrument. *Computer Structures: Reading and Examples*, McGraw-Hill Inc., 1971.
- [9] G. Chen, H. Chen, M. Haurylau, N. A. Nelson, D. H. Albonesi, P. M. Fauchet, and E. G. Friedman. On-chip copper-based vs. optical interconnects: Delay uncertainty, latency, power, and bandwidth density comparative predictions. In *IEEE International Interconnect Technology Conference*, 2006.
- [10] D. Citron and D. G. Feitelson. Look it up or do the math: An energy, area, and timing analysis of instruction reuse and memoization. In *Workshop on Power-Aware Computer Systems*, 2003.
- [11] F. Fang, T. Chen, and R. Rutenbar. Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform. *EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication*, 2002.
- [12] D. Goddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in fem simulations. In *International Journal of Parallel, Emergent and Distributed Systems*, 2006.
- [13] Havok. <http://www.havok.com/content/view/187/77>.
- [14] P. Hofstee. Power efficient architecture and the cell processor. In *HPCA11*, 2005.
- [15] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y. Chen. Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors. In *34th Annual International Symposium on Computer Architecture*, 2007.
- [16] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara, a 32-way multithreaded sparc processor. In *IEEE Micro*, 2005.
- [17] K. Krewell. Ultraspac iv mirrors predecessor. In *Microprocessor Report*, 2003.
- [18] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *The 37th International Symposium on Microarchitecture (MICRO)*, 2004.
- [19] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.
- [20] M. Matthias, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. In *Proceedings of the 3rd Workshop in Virtual Reality Interactions and Physical Simulation*, 2006.
- [21] H. Oh, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong. A fully pipelined single-precision floating-point unit in the synergistic processor element of a cell processor. In *IEEE Journal of Solid-State Circuits*, 2006.
- [22] K. Olukoton, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII*, 1996.
- [23] C. O'Sullivan, S. Howlett, R. McDonnell, Y. Morvan, and K. O'Connor. Perceptually adaptive graphics. In *Eurographics 2004, State of the Art Report*, 2004.
- [24] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. <http://sesc.sourceforge.net>.

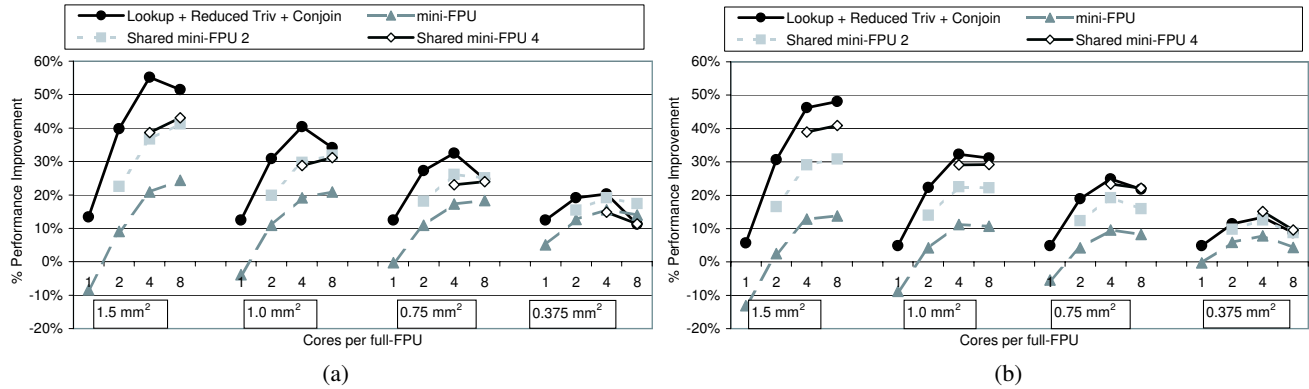


Figure 7. (a) mini-FPU LCP performance — (b) mini-FPU narrow-phase performance

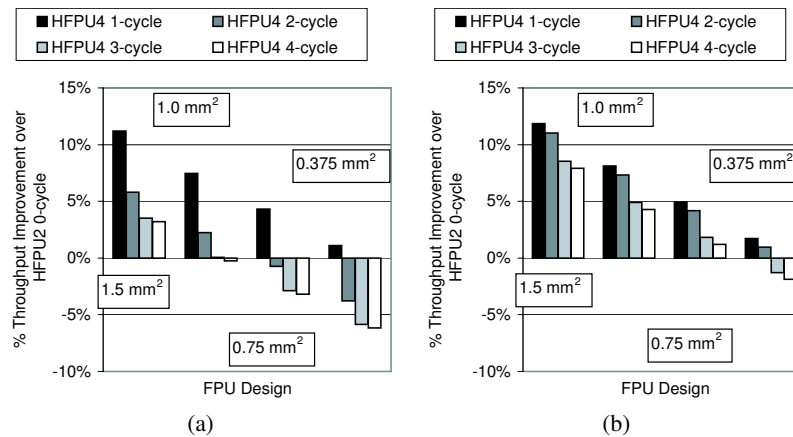


Figure 8. (a) LCP latency sensitivity — (b) Narrow-phase latency sensitivity

- [25] S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. In *IEEE Symposium on Computer Arithmetic*, 1993.
- [26] S. E. Richardson. Exploiting trivial and redundant computation. 1993.
- [27] A. Seugling and M. Rolin. Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. In *Master's Thesis*, 2006.
- [28] A. Shayesteh, G. Reinman, N. Jouppi, S. Sair, and T. Sherwood. Improving the performance and power efficiency of shared helpers in cmps. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.
- [29] D. Sheahan. Developing and tuning applications on ultra-sparc t1 chip multithreading systems. In *Sun Blue Prints Online*, 2007.
- [30] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Compaq WRL 2001/2, 2001.
- [31] V. Soteriou, N. Easley, H. Wang, B. Li, and L. Peh. Polaris: A system-level roadmap for on-chip interconnection networks. In *Proceedings of the 24th International Conference on Computer Design*, 2006.
- [32] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman. Parallax: An architecture for real-time physics. In *The 34th International Symposium on Computer Architecture (ISCA)*, 2007.
- [33] T. Y. Yeh, P. Faloutsos, and G. Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Video Game Symposium*, 2006.
- [34] T. Y. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. In *2007 ACM Transactions on Graphics (TOG) – accepted with major revisions*. <http://www.cs.ucla.edu/tomyeh/tog07.pdf>.
- [35] J. Yi and D. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *IEEE International Conference on Computer Design: VLSI in computers and Processors (ICCD)*, 2002.

<https://web.archive.org/web/20080221202312/http://www.cs.ucla.edu/~pfal/papers/ISCA-07-ParallAX.pdf>

ParallAX: An Architecture for Real-Time Physics

Thomas Y. Yeh[†] Petros Faloutsos[†] Sanjay J. Patel[‡] Glenn Reinman[†]

[†]Computer Science Department, UCLA, {tomyeh, pfal, reinman}@cs.ucla.edu

[‡]AGEIA Technologies, sjp@ageia.com

ABSTRACT

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

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

Categories and Subject Descriptors:

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

General Terms: Design, Performance.

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

1. INTRODUCTION AND MOTIVATION

Interactive entertainment (IE) applications demand high performance from all architectural components. In the future, this demand will increase even further. These applications will be composed of a diverse set of computationally demanding tasks. One critical task is modeling how objects and characters move and interact in a virtual en-

vironment. Most current IE applications make use of pre-recorded motion clips to synthesize the motion of virtual objects (kinematics), but as these objects and their interactions scale up in complexity, pre-recording has become impractical. *Physics-based simulation* has emerged as an attractive alternative to kinematics, providing high levels of physical realism through motion calculation. Future applications targeting a true immersive experience will demand this as a cornerstone of their design.

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

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

In this paper, we make the following contributions:

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-706-3/07/0006 ...\$5.00.

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

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

2. RELATED WORK

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

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

Stream architectures (SAs) aim to enable ASIC-like performance efficiency with high-level language programmability. Stream programs express computation as a signal flow

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

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

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

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

3. PHYSICS SIMULATION WORKLOAD

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

3.1 Our Physics Engine

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

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

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

- ii. *If an explosive object makes contact with another object, create a sphere representing blast radius.*
 - iii. *If a body makes contact with a cloth's bounding volume, insert body on cloth's contact list.*
 - iv. *If a pre-fractured object is in contact with a blast volume (sphere) break object into debris.*
 - (d) Form groups (islands) of objects interconnected with joints, i.e. find the connected components.
 - (e) Forward simulation step: For each island compute the applied loads and the new positions/velocities of each object.
 - (f) Check all breakable joints: if joint's applied load has exceeded a threshold, break the joint.
 - (g) *Process all cloth objects by taking a forward simulation step.*
 - (h) Advance the time: $time = time + \Delta t$
3. End.

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

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

3.2 Computation Phases of Physics Workload

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

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

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

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

then can the constraint solvers begin. Practical techniques for parallel island generation are not commonly available.

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

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

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

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

4. FUTURE PHYSICS BENCHMARKS

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

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

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

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

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

5. EXPERIMENTAL SETUP

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

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

Table 1: Coarse-grain Core Design

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

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

5.1 Interconnect Models

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

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

6. PERFORMANCE DEMANDS

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

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

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

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

6.1 Working Set Analysis

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

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

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

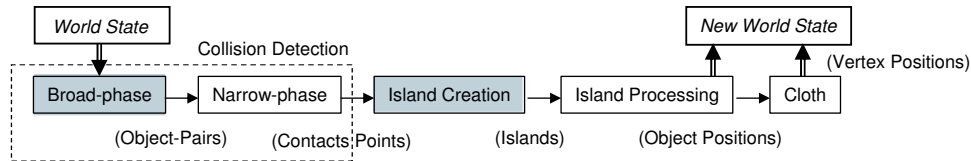


Figure 1: Physics Engine Flow.

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

Table 2: Features Found in Our Benchmarks

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

Table 3: Our Physics Benchmarking Suite

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

Table 4: Benchmark Specs

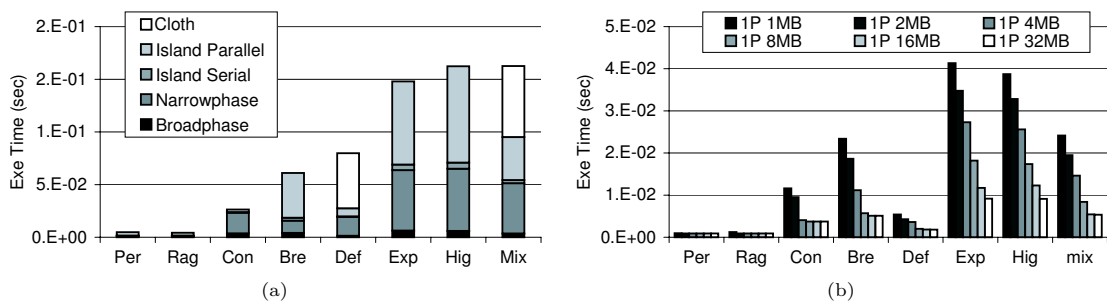


Figure 2: (a) Execution Time Breakdown of 1 Core + 1MB L2 — (b) Single Core Execution of Serial Parts

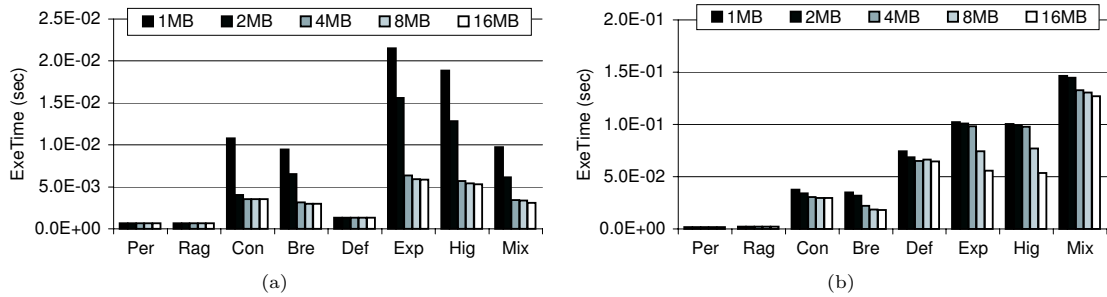


Figure 3: (a) Broadphase Performance [dedicated L2] — (b) Narrowphase Performance [dedicated L2]

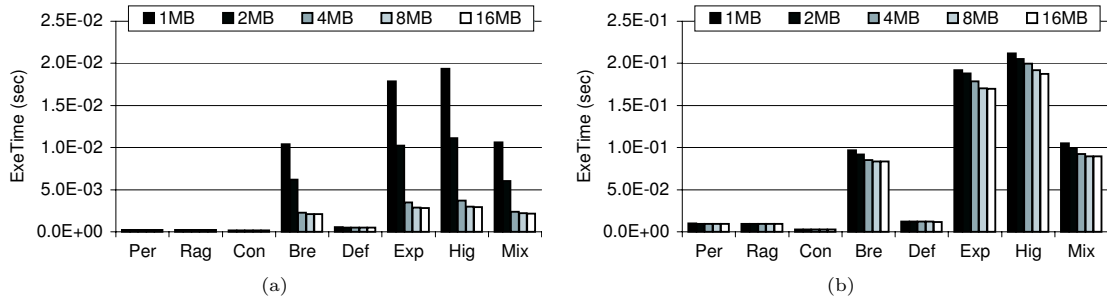


Figure 4: (a) Island Creation Performance [dedicated L2] — (b) Island Processing Performance [dedicated L2]

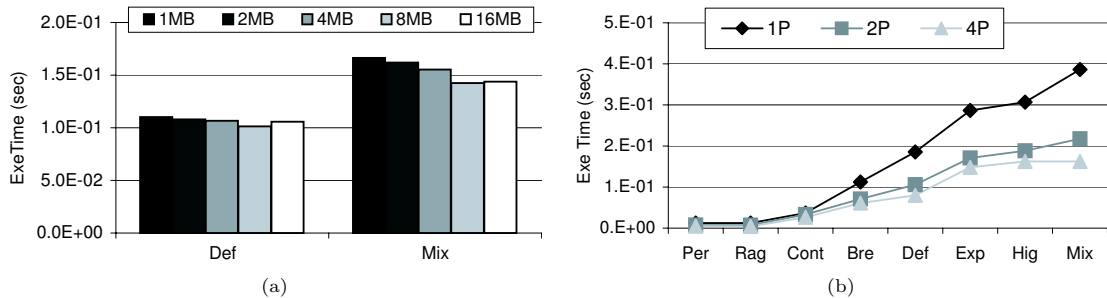


Figure 5: (a) Narrowphase Performance [dedicated L2] — (b) Performance with Processor Scaling

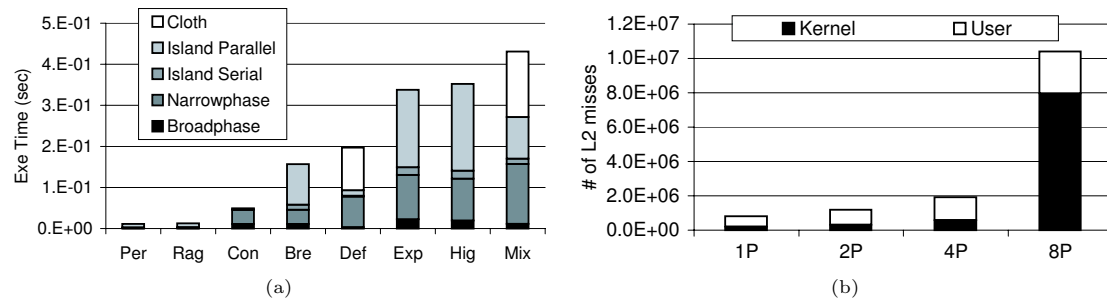


Figure 6: (a) Execution Time Breakdown (4 Core + 12MB L2) — (b) L2 Miss with Thread Scaling

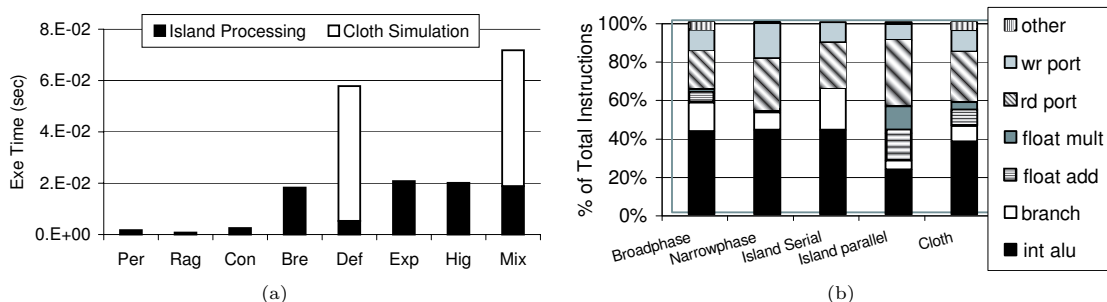


Figure 7: (a) Limit of Coarse-grain Parallelism — (b) Instruction Mix for all 5 Phases

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

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

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

6.2 Exploiting Coarse-Grain Parallelism

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

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

Figure 5 (b) clearly shows that the performance improvement starts to plateau at 4 cores. On average, scaling from one to two cores improves performance by 53% and scaling from two to four cores improves performance by 29%. Figure 6 (a) shows the execution time breakdown for a four core processor with 12MB total L2 cache space. The performance has improved by roughly 3X, but we still need an

additional 5X improvement to satisfy all benchmarks. Performance starts to degrade at eight cores (not shown). This degradation is surprising, but can be attributed to two main causes: an increased working set in the L2 cache and operating system overhead. For all parallel phases, additional threads consume more cache space by simultaneously accessing data that had not needed to co-exist in the cache. The large increase in L2 misses is shown in Figure 6 (b).

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

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

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

7. PARALLAX ARCHITECTURE

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

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

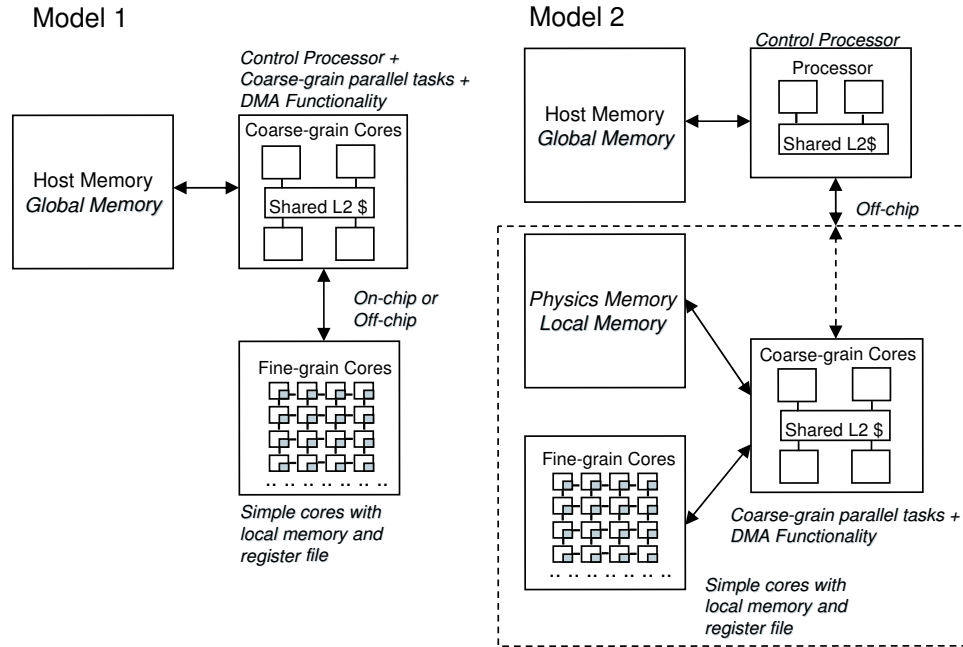


Figure 8: ParallAX - Parallel Physics Accelerator

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

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

FG parallelism refers to breaking computation down into:

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

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

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

overhead involved in pushing the data to the FG cores and retrieving the data back after computation completes.

7.1 Mapping Fine-Grain to Coarse-Grain Cores

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

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

7.2 CG to FG Communication Latency

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

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

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

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

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

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

7.3 Programming Model

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

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

The control packet, in conjunction with the previously described arbiters, sets up the flow of data packets to FG cores. Once the a full set of data is received on a FG core, the kernel id chooses the kernel to execute (kernel code already resides in FG cores). The iteration count indicates the number of iterations to execute (the code assumes all FG tasks start from iteration 0 and the data has been packed accordingly).

The task id uniquely identifies the CG thread which submitted the FG request, and the data set id uniquely identifies each FG core. This information is tracked on the CG core to identify the results returned back from FG cores. Each CG core has a network interface module to send, receive, and buffer these packets. Each arbiter tracks FG core activity by examining data packet headers.

8. DESIGN EXPLORATION

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

8.1 Characterizing Fine-Grain Computation

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

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

8.1.1 Kernel Characterization

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

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

8.1.2 Memory Required for Inst and Data Storage

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

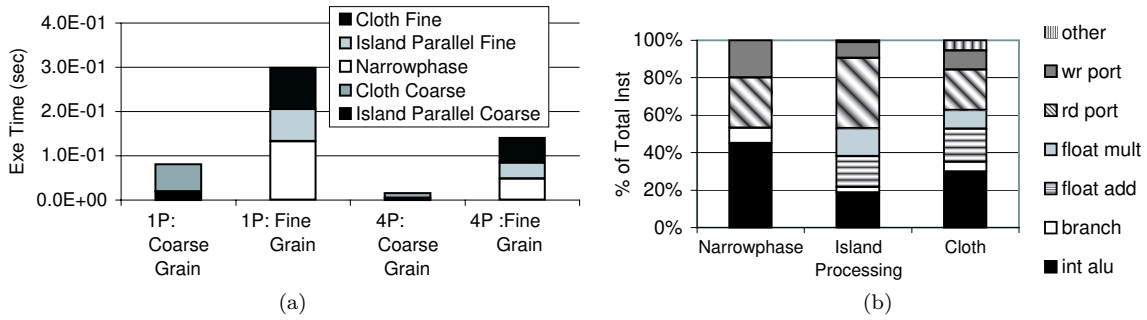


Figure 9: (a) Coarse-grain vs Fine-grain Execution Time — (b) Instruction Mix of Fine-grain Kernels

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

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

8.2 Fine-grain Core Design and Requirements

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

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

Table 5: Our Fine-Grain Core Designs

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

8.2.1 Number of Finegrain Cores Required

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

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

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

Area Estimation:

By using published die areas and photos of *single* cores from Intel Core Duo 2 [22], IBM Cell [10], and Nvidia G80 [23], we derive area estimates for each core type using 90nm technology. The required interconnect area is derived from Table III of [24]. The area estimates for 30 desktop, 43 console, and 150 shader cores are 1388 mm², 926 mm², and 591 mm² respectively. This argues for the simplest cores as the most area-efficient alternative and points to a severe need of area optimization, which will be a main thrust of our future work.

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

8.2.2 Parallelism to Hide Interconnect Latency

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

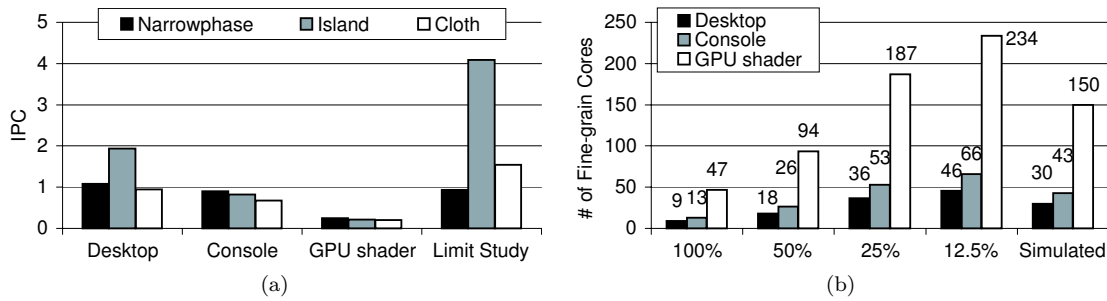


Figure 10: (a) IPC of Fine-grain Core Types — (b) Number of Fine-grain Cores to Achieve 30 FPS

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

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

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

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

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

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

Although cloth is shown to have the least amount of FG parallelism for the next generation of games, the number of

vertices used to model each cloth will likely scale up when creating realistic cloth movement further into the future.

8.3 Implementation Alternatives

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

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

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

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

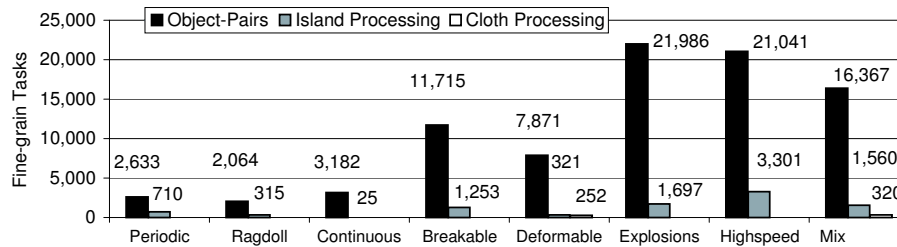


Figure 11: Average Available Fine-grain Tasks

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

9. SUMMARY

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

10. ACKNOWLEDGEMENTS

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

11. REFERENCES

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