UNIVERSITY OF CALIFORNIA

Los Angeles

Architectural and Algorithmic Acceleration of Real-Time Physics Simulation in Interactive Entertainment

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Thomas Yen-Hsi Yeh

2007

The dissertation of Thomas Yen-Hsi Yeh is approved.

William Kaiser

Sanjay Patel

Yuval Tamir

Demetri Terzopoulos

Petros Faloutsos, Committee Co-chair

Glenn Reinman, Committee Co-chair

University of California, Los Angeles

2007

To Erica

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to extend my gratitude and appreciation to the many people in my academic and personal life who made this dissertation possible.

First and foremost, I would like to thank my research advisors, Professor Glenn Reinman and Professor Petros Faloutsos for their support, encouragement, and guidance. It has been a privilege to work with both. Their deep knowledge in the disjoint fields of computer architecture and computer graphics were essential for this work. Most importantly, I appreciate Professor Reinman's trust and patience during the search for my research topic.

I thank Professor David Patterson for introducing me to the world of computer architecture, Professor Bill Mangione-Smith for co-advising me and suggesting the exploration of real-time physics, Professor Sanjay Patel for his guidance and the opportunity to collaborate with AGEIA Technologies, Professor Yuval Tamir for providing invaluable feedback, Professor Milos Ercegovac for collaboration and guidance, and Dr. Enric Musoll for support and advice. I would also like to acknowledge professors William Kaiser and Demetri Terzopolous who served on my committee and provided insightful comments.

It has been a pleasure working with the talented and dedicated people at UCLA. I would like to acknowledge the help and support from my fellow graduate students (Dr. Anahita Shayesteh, Adam Kaplan, Dr. Eren Kursun, Dr. Yongxiang Liu, Kanit Therdsteerasukdi, Gruia Pitigoi-Aron, Shawn Singh, and Brian Allen), the assistance I received from the undergraduates (Eric Wood, Nathan Beckmann, Mishali Naik, and Paul Salzman), and the consistent help and advice from our department's graduate student advisor (Verra Morgan).

My deepest gratitude goes to my extended family (the Yehs, the Lis, the Changs,

the Chuangs, and the Jengs) and close friends for their unconditional support, understanding, and sacrifice. I am especially thankful to David, Gene, Mike, Snoopy, Lucia, Christine, Leo, Ramona, Shang, Lauren, Brandon, Kai-Sheng, Hank, David Sr., George, Amy, John, Albert, Pearl, David Jr., my grandparents and my brother David. I offer special thanks to my parents Shou-Shoung and Fumei, especially for their sacrifice in making my education in the United States possible and for their encouragement to always strive for my dreams. Finally, and most importantly, I want to thank my wife, Erica May-Chien Chang, M.D., for her love, support, and sacrifice without which I could not have accomplished this work. I dedicate this dissertation to her.

VITA

1974	Born, Taipei, Taiwan.
1993–1997	B.S., Electrical Engineering and Computer Science, University of California, Berkeley
1996	Hardware Engineer, SBE
1998	Verification Engineer, Intel Corporation
1998–1999	M.S., Computer Science, University of California, Los Angeles
1999 - 2001	Logic Design/ Architecture Research/ Marketing, Intel Corporation
2001 - 2002	Design Engineer, Xstream Logic/ Clearwater Networks
2002	Microarchitect, Sun Microsystems
2003 - 2007	Graduate Research Assistant, Teaching Assistant, Computer Science Department, University California, Los Angeles

PUBLICATIONS

Fool Me Twice: Exploring and Exploiting Error Tolerance in Physics-Based Animation T. Y. Yeh, G. Reinman, S. Patel, P. Faloutsos, ACM Transactions on Graphics (TOG) – accepted with major revisions, 2007

ParallAX: An Architecture for Real-Time Physics T. Y. Yeh, P. Faloutsos, S. Patel, G. Reinman, The 34th International Symposium on Computer Architecture (ISCA-34), June 2007

Enabling Real-Time Physics Simulation in Future Interactive Entertainment T. Y. Yeh, P. Faloutsos, G. Reinman, 2006 ACM SIGGRAPH Symposium on Videogames (Sandbox), August 2006

Fast and Fair: Data-stream Quality of Service T. Y. Yeh, G. Reinman, 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), September 2005

Redundant Arithmetic Optimizations T. Y. Yeh, H. Wang, The 6th International Euro-Par Conference on Parallel Computing (Euro-Par), August 2000

ABSTRACT OF THE DISSERTATION

Architectural and Algorithmic Acceleration of Real-Time Physics Simulation in Interactive Entertainment

by

Thomas Yen-Hsi Yeh

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2007

Professor Glenn Reinman, Co-chair

Professor Petros Faloutsos, Co-chair

Interactive entertainment (IE) applications are rapidly gaining significance from both technical and economical point of views. Future IE applications will feature on-the-fly content creation with large number of interacting objects, intelligent agents, and high-definition rendering. Application designers must provide at least 30 graphical frames per second to provide the illusion of visual continuity. While IE's real-time constraint necessitates a tremendous amount of performance, almost no academic attention in the architecture community has been directed at quantifying the needs of this emerging workload.

In this dissertation, we focus on the acceleration of one core component of this emerging workload, namely real-time physics simulation or physics based animation (PBA). Our holistic approach to acceleration spans benchmark creation, workload characterization, architectural acceleration, algorithmic acceleration, and architectural exploitation of algorithmic properties.

To represent this emerging workload, we developed PhysicsBench, a set of bench-

marks to capture the complexity and scale of PBA in IE applications. Using the PhysicsBench suite, we characterized the workload to identify its key differentiating factors. Based on the characterization, we propose ParallAX, an architecture to sustain interactive frame rates for real-time physics. The ParallAX architecture is a heterogeneous chip-multiprocessor that features aggressive coarse-grain cores and area-efficient fine-grain cores. Scaling the number of active cores per chip increases the load on the lowest-level cache. To alleviate cache thrashing, we propose the Performance Driven Adaptive Sharing (PDAS) cache design. PDAS is a scalable, multi-ported NUCA that dynamically allocates its distributed cache resources through an intelligent, realizable on-line partitioning strategy.

In addition to parallelism, the human perception error tolerance can also be leveraged for performance in PBA. Using prior studies of simpler scenes as a starting point, we extrapolate a methodology for evaluating the tolerable error of complex scenes. Leveraging the findings from these studies, we propose architectural techniques to exploit algorithmic properties of PBA, namely perceptual error tolerance and the notion of object-pairs.

To summarize, this dissertation is an in-depth study on the acceleration of real-time physics simulation. Given physics' similarity to other software components, our proposed methodologies and techniques can be applied to many areas within the IE space.

CHAPTER 1

Introduction and Motivation

1.1 The Emerging Workload of Interactive Entertainment

Interactive entertainment (IE) has grown to a substantial industry. According to the Entertainment Software Association [CS], IE software generates $10.3 billion in direct sales per year and $7.8 billion in complementary products. Video games are the predominant form of interactive entertainment, and have driven mass demand for high-performance computing. Gaming sustains the economy of scale for CPU and GPU development which finances research and development, impacting the entire computing industry. Sixty-nine percent of the heads of American households play games and the current average game player age is 33 [ESA].

Beyond pure entertainment, interactive gaming is being leveraged for use in education, training, health, and public policy [Ini]. One interesting example is the America's Army game, created by the United States Army for civilians to experience life as a soldier [Arm]. Other creative uses include medical screening, fitness promotion, and hazmat training. Recently, gaming software research has been integrated into the curriculum of various prestigious universities. Interactive entertainment is evolving into a powerful medium for future generations to experience [ESA].

Despite the social, economic, and technical importance of gaming software, there has been very little academic effort to quantify game's behavior and needs. The latest generation of game-consoles (Sony PlayStation 3 [CNE], Microsoft Xbox 360 [360],

and Nintendo Revolution [Rev]) shows a broad spectrum of designs aimed at the same workload. Differing design choices include the programming model, number of threads, type of chip-multiprocessor, order of execution, and complexity of branch prediction. Surprisingly, all three drastically different processor designs were created by the same company.

1.2 Software Components of Interactive Entertainment Applications

From a technical perspective, future games will be computationally intensive applications that involve various computation tasks [Kel]: artificial intelligence, physics simulation, motion synthesis, scene database query, networking, graphics, audio, video, I/O, OS, tactile feedback, and general purpose game engine code. The interdependencies of these components are illustrated in Figure 1.1 based on information from [Kel]. In order to provide smooth game-play, gaming hardware is required to complete all tasks under their respective real-time constraints. The diverse computational tasks that could compose future games will challenge future system architects to achieve the performance constraints of these games. As demands change, the algorithms employed in games are continuously being refined, further increasing computation demands into the foreseeable future.

Real-time physics simulation and artificial intelligence are considered the top two areas for dramatically enhancing user experience of future IE applications. Both components enable on-line content creation by dynamically generating game-play content. This characteristic offers both substantial advantages as well as disadvantages. Dynamic content allows for open-ended unique user experiences while reducing production costs in the form of programmers statically coding possible scenarios. However,

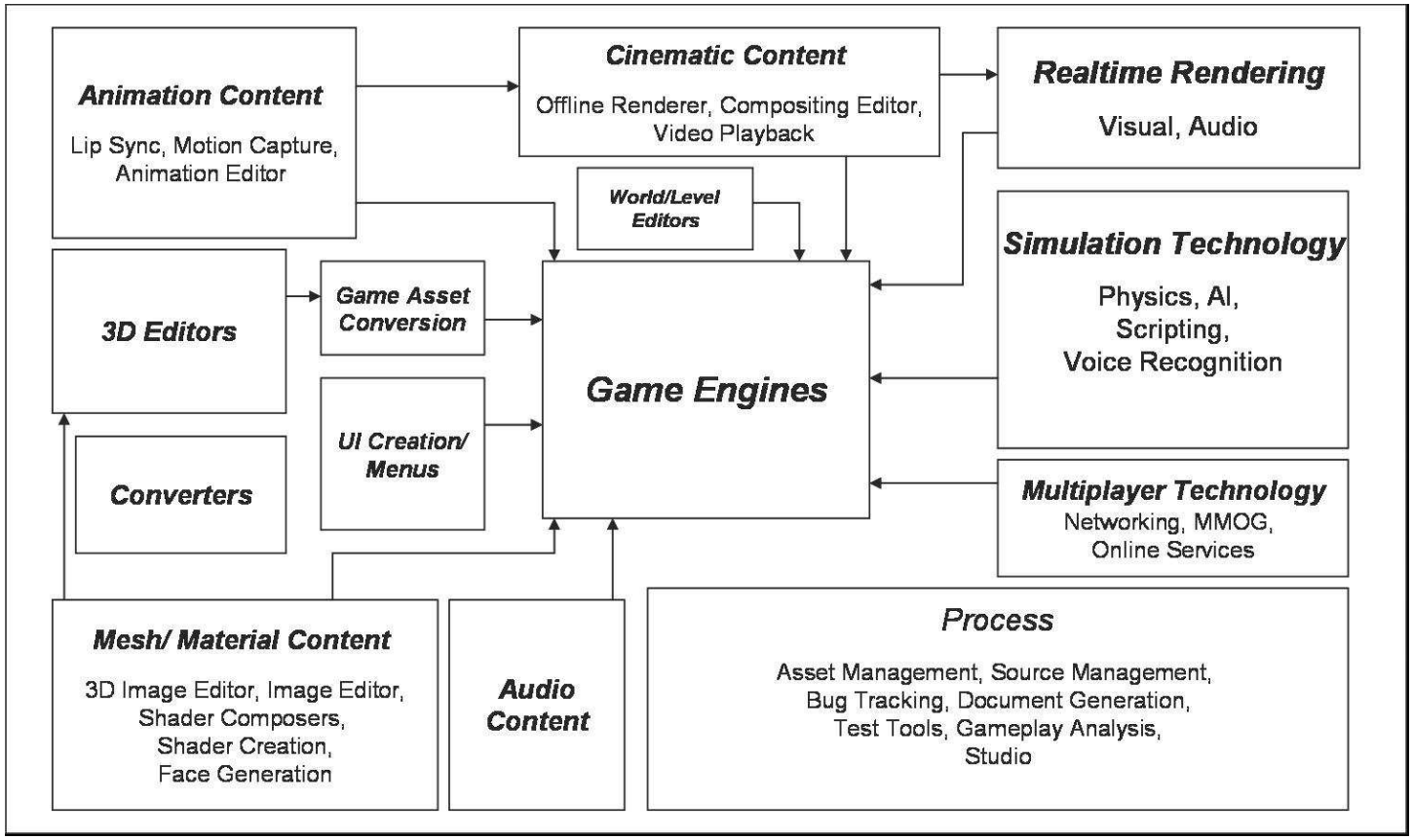


Figure 1.1: Software Components of Interactive Entertainment Applications.

it results in significant increase of hardware performance requirement as well as verification complexity.

While the laws which govern physical behavior are well understood and modeled, gaming artificial intelligence (AI) covers a wide range of tasks and is currently an active area of research for the AI community [LL00, YFR06]. The collision detection component of physics simulation is also a main component for AI tasks such as local steering [Ope]. Therefore, we focus on the software component of real-time physics simulation.

The research goal of this dissertation is to determine how to best meet the compute demands of real-time physics simulation for future gaming workloads. With proper benchmarking and characterization, we propose and evaluate novel architectural and algorithmic acceleration techniques.

1.3 Contributions

- Real-Time Physics Benchmarking: PhysicsBench 1.0 and 2.0
- Physics Workload Characterization
- ParallAX: Architecture for Physics Acceleration
- Performance-Driven Adaptive Sharing Cache
- Evaluation of Perceptual Error Tolerance for Complex Scenarios
- Precision Reduction in FPU Design for CMPs
- Fast Estimation with Error Control
- Fuzzy Value Prediction
- Object-Pair Filter

1.4 Overview

This dissertation is organized as follows. Chapter 2 presents background information on real-time physics simulation. Prior work on both software physics engines and hardware accelerators are discussed in Chapter 3, and the workload characterization is in Chapter 4. Chapter 5 presents architectural contributions for accelerating physics simulation. These include both ParallAX, a heterogeneous CMP architecture to accelerate physics simulation, and PDAS, a performance-driven adaptive sharing cache. Chapter 6 details pure algorithmic contributions in the field of real-time physics simulations. In Chapter 7, contributions in hybrid techniques which leverage algorithmic properties for architectural acceleration are presented. Finally, the conclusion and future directions are presented in Chapter 8.

CHAPTER 2

Background

Before discussing the details of our work, we describe the key background information on real-time physics simulation.

In the early days of the interactive entertainment industry, virtual characters were heavily simplified, crude polygonal models. The scenarios in which they participated were also simple, requiring them to perform small sets of simple actions. The recent advances in graphics hardware and software techniques have resulted in near cinematic quality images for entertainment applications such as *Assassin's Creed*, *Heavenly Sword*, *Motor Storm*, *Gears of War*, *World of Warcraft* and *Crysis*.

The unprecedented levels of visual quality and complexity in turn require high fidelity animation. To achieve high fidelity animation, modern interactive entertainment applications have started to incorporate new techniques into their motion synthesis engines. Among them, physics-based simulation is one of the most promising options.

2.1 Kinematics vs Physics

The current state-of-the-art in motion synthesis for interactive entertainment applications is predominantly based on *kinematic* techniques. The motion of all objects and characters in a virtual world is derived procedurally or from a convex set of parameterized recorded motions. Such techniques offer absolute control over the motion of the animated objects and are fairly efficient to compute. However, the more complex

the virtual characters are the larger the sets of recorded motions will be. For the most complex virtual characters, it is impractical to record the entire set of possible motions that their real counterparts can do.

Physics-based simulation is an alternative approach to the motion synthesis problem. It computes the motion of virtual objects by numerically simulating the laws of physics. Thus, it supports unpredictable, non-prescribed interaction between objects in the most general possible way. Physics-based simulation provides physical realism and automated motion calculation, but has greater computational cost, difficulty in object control, and potentially unstable results.

Realism: The laws of physics offer the most general constraint over the motion. Not only do they guarantee realistic motion, but they also avoid repetition. Any variation in the initial conditions (i.e. contact points) will produce a different motion. In a sense, the set of possible actions is as large as the domain of the initial conditions, and not restricted to a small set of recorded motions.

Automation: Once the equations of motion are provided for each object in a virtual world, motion can be computed automatically based on the applied forces and torques.

Control: The laws of physics specify how objects move under the influence of applied forces and torques. However, they do not specify what the forces and torques should be in order to achieve a desired action. That is a separate problem which can be very complex for dynamically balanced characters such as virtual humans. This problem is out of the scope of our work.

Stability: The numerical methods that simulation uses to solve the equations of motion can become unstable under certain circumstances. However, there are a lot of methods that have been developed to deal with this problem. Of particular interest to entertainment applications are methods that trade off accuracy for stability. This is a key issue that we exploit later on.

2.2 Types of Physical Simulation

Simulation in IE applications can be categorized based on the objects and the type of phenomena that we are most interested in simulating. They typically fall in the following 5 categories:

1. *Rigid Body*: idealization of a solid body of finite size in which deformation is neglected [Bar97]
2. *Cloth*: cloth mesh simulated as point masses connected via distance constraints [CK02, BW98]
3. *Explosion*: suspended particle explosions [FOA03]
4. Fluid: smoothed particle hydrodynamics, mass distributed around a point [FL04, MTP04]
5. Hair: geometric model of hairs using vector fields [CJY02]

This dissertation focuses on *Rigid Body*, *Cloth*, and *Explosion*.

2.3 High Level Characteristics of the Simulation Load

All types of simulation have certain characteristics that are unique to the domain of IE applications. *Efficiency* is crucial in interactive entertainment: each frame of animation must be computed at a minimum rate of approximately 30 frames per second. For a frame to be computed all the necessary components of the application must complete within a fraction of this frame rate.

Stability is also critical to creating a realistic environment. The simulation should not numerically explode under any circumstances. However, while it is important that

actions have a visually believable outcome and do not violate any constraints placed on the objects of the simulation (i.e. bones bending, walking through walls), interactive entertainment applications generally have looser requirements on accuracy than most scientific applications. Recent research in animation [HRP04b, RP03a] has actually studied and quantified errors that are visually imperceptible. For instance, length changes below 2.7% cannot be perceived by an average observer[HRP04b] while changes of over 20% are allways visible. The acceptable bounds on errors increase with scene clutter and high-speed motions[HRP04b].

The physics load of interactive entertainment applications has certain unique features. First, it seems to be *distributed*. For most scenes that depict realistic events, there are many things happening simultaneously but independently of each other.

This distributed nature of the physics load can be exploited to reduce the complexity of the underlying solvers and allows for parallel execution. Second, the physics load seems to be *sparse*. Numerical solvers and dynamic formulations can exploit sparsity to improve computational efficiency. Third, since the applications are interactive there is usually a human viewer/user involved. Therefore, the application inherently tolerates errors that the human user cannot perceive.

In summary, the physics load specifically as it applies to interactive entertainment applications seems to be *distributed*, *sparse*, and *error-tolerant*. At the same time, such applications require *efficiency*, and *stability* for which they can trade off *accuracy*. Based on these considerations, we use the *Open Dynamic Engine* [Eng] as a representative physics-based simulator for interactive entertainment applications.

2.4 Open Dynamics Engine Algorithmic Load

The Open Dynamics Engine follows a constraint-based approach for modeling articulated figures, similar to [Bar97]. ODE is designed with efficiency rather than accuracy in mind and it is particularly tuned to the characteristics of constrained rigid body dynamics simulation. A typical application that uses ODE has the following high level algorithmic structure:

1. Create a dynamics world.
2. Create bodies in the dynamics world.
3. Set the state (position and velocities) of all bodies.
4. Create the joints (constraints) that connect bodies.
5. Create a collision world and collision geometry objects.
6. While $(time < time_{max})$
   (a) Apply forces to the bodies as necessary.
   (b) Call collision detection.
   (c) Create a contact joint for every collision point, and put it in the contact joint group.
   (d) Take a forward simulation step.
   (e) Remove all joints in the contact joint group.
   (f) Advance the time: $time = time + \Delta t$
7. End.

The computational load of a simulation is defined by two main components: *Collision Detection*, (b), and the *forward dynamics step*, (d). Finer granularity distinction between computation phases will be explored later.

Collision Detection Collision detection (CD) uses geometrical approaches to identify bodies that are in contact and generate appropriate contact points. A *space* in CD contains geometric objects that represent the outline of rigid bodies [Eng]. Spaces are used to accelerate collision detection by allowing the removal of certain object pairs that would result in useless tests.

Collision detection depends significantly on the geometric properties of the objects involved. ODE supports contact between standard shapes such as boxes, spheres, and cylinders, and also arbitrary triangle meshes. The contact resolution module of ODE supports both instantaneous collisions and resting contact with friction. High speed collisions can be resolved even at coarse time steps. In such cases, the collision may produce penetrating configurations. However, a nice feature of ODE is that the penetration will be eliminated after a short number of steps. Such features make ODE especially suitable for interactive applications.

Forward Dynamics Step The simulator takes a forward step in time by computing the constraint forces that maintain the structure of the objects and that satisfy the collision constraints produced by collision detection. This is one of the most expensive part of the simulator and requires the solution of a *Linear Complementary Problem*(LCP). ODE offers two ways of solving the LCP system for the constraint forces: an accurate and expensive one based on a *big-matrix* approach (the so called *normal step*), and a less accurate approach called *quick step* that iteratively solves a number of much smaller LCP problems. Their respective complexities are $O(m^3)$ and $O(m \times i)$, where m is the total number of constraints and i is the number of iterations, typically 20. For any scene of average complexity the iterative (quick-step) approach far outperforms the big-matrix approach. IE applications' tolerance for lower accuracy is one main characteristic that we can leverage for performance. By using the *quick step*, we enable massive parallelization for the constraint solver as described in the fine-grain

parallelism section.

ODE's integrator trade-offs accuracy for efficiency and allows relatively high time steps, even in situations with multiple high speed collisions. The key parameter here is the integration time step which, for a fixed-step integrator, relates directly to the time step of the simulation Δt. Typical values range from 0.01 to 0.03.

In the physics integration computation, the concept of an island is analogous to the space in the above discussion on CD. The island concept is defined as a group of bodies that can not be pulled apart [Eng], which means that there are joints interconnecting these bodies. Each island of bodies is computed independently from other islands by the physics engine.

The computation demand is affected significantly by the number and the complexity of islands during one simulation step. The complexity of an island can be quantified by the number of objects along with the number and complexity of the interconnecting joints. The complexity of a joint is characterized by the degrees of freedom (DoF) it removes as listed in the following table:

Joint	Ball	Hinge	Slider	Contact	Universal	Fixed
DoF Removed	3	5 (4)	5	1	4	6

The formation of an island has different temporal behaviors. Some persist for a long time while others constantly change between each integration step. This behavior contributes to the variance in computation demands by the engine.

CHAPTER 3

Prior Work

There is little work directly related to interactive entertainment (IE) in the architecture community. In this chapter, prior work on IE benchmarking, hardware acceleration, and perceptual error tolerance is presented. Throughout the dissertation, discussion of contributions will include additional related work specific to each contribution.

3.1 Benchmarking

At the start of the project detailed in this dissertation, no benchmarks for real-time physics simulation existed. Since then, AGEIA Technologies joined Futuremark's 3DMark Benchmark Development Program to include two complex game-like scenarios in 3DMark06 [3DM]. While these benchmarks support multi-thread and multi-core architectures, the lack of source code hampers in depth architectural studies.

[MWG04] compared the performance counter statistics of a single second execution between two first person shooter games to music and video playback applications. This work shows the difference between gaming and multimedia applications due to game's content creation tasks, and points to chip multiprocessors (CMP) [ONH96a] as a promising approach to providing performance.

Physics Engines Physics engines are software libraries that enable rigid body dynamics simulation. [SR06] compares three physics engines, namely ODE, Newton,

and Novodex. Different engines may support different types of physics simulation and solver algorithms. Below is the list of prominent engines currently on the market:

1. Commercial: AGEIA [agea] and Havok [Hava]
2. Open-source: ODE [Eng]
3. Free, close-source: Newton [New] and Tokamak [Tok]

Both AGEIA [agea] and Havok [Havb] provide their own proprietary SDKs. Open Dynamics Engine (ODE) [Eng] 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.

3.2 Hardware Physics Accelerators

The MDGRAPE-3 chip by RIKEN [Tai04] and the PhysX chip by AGEIA [agea] are currently the only dedicated physics simulation accelerator designs. While MD-GRAPE targets computational physics, PhysX targets real-time physics for games. Both designs are placed on accelerator boards which connect to the host CPU through a system bus. PhysX's architectural design is not public, and MD-GRAPE's design is specific to computing forces for molecular dynamics and astrophysical N-body simulations with limited programmability.

Two other closely related bodies of prior work are *vector processing* [HP96] and *stream computation* [LMT04, KRD03].

Vector Processing. The massive parallelism available in real-time physics hints at the use of vector processors like VIRAM [KPP97], Tarantula [EAE02], and CODE [KP03]. VIRAM[KP02] has achieved an order of magnitude performance

improvement on certain multimedia benchmarks. However, conventional vector architectures are constrained [KP03] by limitations like the complexity of a centralized register file, the difficulty to implement precise exceptions, and the requirement of an expensive on-chip memory system. Most importantly, the physics workload requires tremendous speedup that necessitates massive parallel execution. While CODE is scalable by increasing clusters and lanes, the data shows a plateau at eight clusters with eight lanes, and the cache-less CODE can not satisfy our measured physics workload.

Stream Computation. Stream architectures (SAs) aim to enable ASIC-like performance efficiency while being programmable with a high-level language. 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 [LMT04].

The Stream Virtual Machine (SVM) architecture model logically consists of three execution engines and three storage structures. The execution engines include a control processor, kernel processor, and DMA. The storage structures are local registers, local memory, and global memory. The SVM mitigates the engineering complexity of developing new stream languages or architectures by enabling a 2-level compilation approach. Related designs in this space include IBM's Cell, GPUs, and the Xbox360 system.

IBM's Cell [Hof05] consists of one general purpose PowerPC core (PPE) and eight application specific streaming engines (SPE) – all connected by a ring of on-chip interconnect (EIB). Although the Cell's programming model is described as cellular computing, the design can be included in the broad space of streaming computation. The PPE is a 64-bit, 2-way SMT, in-order execution PowerPC design with 32KB L1 caches and a 512KB L2 cache. The SPEs are RISC cores each with 128 128-bit SIMD registers, customized SIMD instructions, and 256KB local private memory. SPEs are not

ISA compatible with conventional PowerPC cores. Heterogeneous CMP designs such as the Cell are able to target the best *task* specific performance using different cores. The PPE targets control intensive tasks such as the OS and the SPEs target compute intensive tasks. However, according to our exploration, both the PPE and SPE designs are not optimal for physics computation. Serial components' performance on the PPE will take a significant amount of each frame's time, and the SPE's complexity prevents the placement of the required number of cores to achieve 30 FPS. This may be a result of the fact that the Cell is designed to execute *all* components of a game, not just physics simulation.

The Graphics Processing Unit (GPU) [PF05] is another design point within the streaming architecture space. 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. Furthermore, the host CPU is connected to the GPU via a system bus. This communication latency is problematic for physics simulation working in a continuous feedback loop. This may be one reason why Havok's FX only enables effect physics and not game-play physics. This limitation is alleviated in the Xbox360 system [AB06], 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.

3.3 Perceptual Error Tolerance

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

3.3.1 Perceptual Believability

[OHM04] 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, visualization and non-photorealistic rendering. This dissertation focuses on the animation category. Given the comprehensive coverage of this prior survey paper, we will only present prior work most related to our work and point the reader to [OHM04] for additional information.

[BHW96] is credited with the introduction of the plausible simulation concept, and [CF00] built upon this idea to develop a scheme for sampling plausible solutions. [ODG03] 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 capture the effects of different metrics. This paper only uses simple scenarios with 2 objects colliding for clinical trials.

[HRP04a] 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.

[RP03b] is a study on the visual tolerance of ballistic motion for character animation. Errors in horizonal velocity were found to be more detectable than vertical velocity, and added accelerations were easier to detect than added deceleration.

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.

3.3.2 Simulation believability

Chapter 4 of [SR06] 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.

CHAPTER 4

Workload Characterization

This chapter covers our initial characterization of the real-time physics simulation workload.

4.1 PhysicsBench 1.0

In order to suggest architectural improvements to enable future applications' use of real-time physics simulation, the workload characterization of real-time physics engine kernel is required. Due to the lack of prior work, this involves the creation of a representative suite of benchmarks that covers a wide range of common situations in interactive entertainment applications. We have created two versions of PhysicsBench. This section covers the details and reasoning in creating PhysicsBench 1.0. Our first pass on PhysicsBench takes a bottom-up approach and focuses on parameters that affect the computation load. The latest version, PhysicsBench 2.0, is described in the next chapter.

High Level Considerations PhysicsBench covers a wide range of typical IE situations that involve object interaction. Our scenarios include fighting humans, object to human collisions, object to object collisions, exploding structures, and fairly complex battle scenes. Our benchmarks are designed to test the scalability of the simulation both in terms of the objects that interact with each other simultaneously, for example

stacking, and in independent groups, for example a large battle scene.

The benchmarks represent scenes of realistic complexity (interactions) but not necessarily realistic motions. The visual representation of the objects in a scene show the geometries used for collisions not the ones used for visual display. We are only interested in the simulation load, not the graphics load.

While the benchmarks below cover a wide range of representative scenarios, more complex situations can be constructed by mixing multiple benchmarks as shown below. Because of the distributed nature of the physics load as it applies to interactive entertainment applications, the combined computational load can be roughly extrapolated from the results of the individual scenarios.

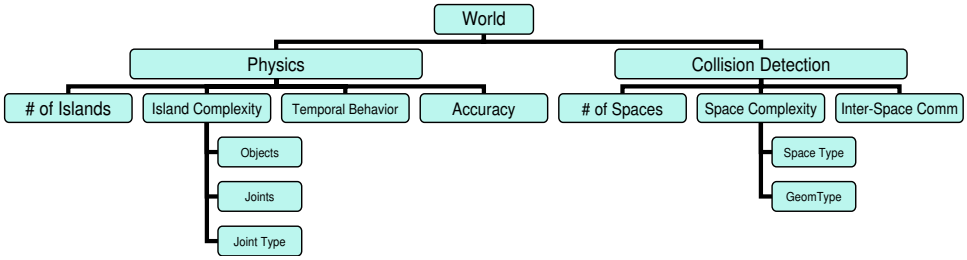


Figure 4.1: Parameters Affecting Computation Load.

Computation Load As described previously, the physics simulation engine is composed of two major dependent components: *collision detection* and *forward dynamics step*. Collision detection determines all contact points and creates joints to model the impulse forces generated. Then, the bodies along with these contact joints are computed to determine the new positions. Within each component, there are a number of factors that affect the computation load. The high-level chart 4.1 captures the most significant parameters.

Benchmarks The benchmarks involve virtual humans, cars, tanks, walls and projectiles. The virtual humans are of anthropomorphic dimensions and mass properties.

Each character consists of 16 segments (bones) connected with idealized joints that allow movement similar to their real world counterpart. The car consists of a single rigid body and four wheels that can rotate around their main axis. Four slider joints model the suspension at the wheels. The walls are modeled with blocks of light concrete. The projectiles are single bodies with spherical, cylindrical or box geometry. In all benchmarks, the simulator is configured to resolve collisions and resting contact with friction. Table 4.1 summarizes the quantitative differences between benchmarks.

Benchmark	Number of Islands (Max, Min, Avg, Dev)	Number of Spaces
2 Cars	2, 2, 2, 0	1
10 Cars	10, 10, 10, 0	1
Car Crash Sk	3, 1, 2, 0.65	1
Car Crash Wall	105, 99, 101, 1.4	1,3
Environment	337, 196, 245, 46	1,10
Car Crash Sk x100	300, 100, 220, 64	100
Battle I	120, 2, 93, 18	3
Fight	10, 7, 8, 1.1	1,10
Battle II	156, 113, 134, 18	1,15

Table 4.1: Parameters Affecting Computation Load.

The benchmarks are as follows:

- 2-Cars: Two cars driving - two cars, each with 3 wheels that are steered to run in parallel then collide. One of the cars goes over a wooden ramp.
- 10-Cars: Ten cars driving - to get a sense of how the load changes with scale, we extend the two-car scenario to ten cars.
- CrashSk: Car crashing on two people - a car with four wheels crashing into two 16-bone virtual humans.
- CrashWa: Extreme-speed Car crashing on wall, tank shooting projectiles - a high speed car (velocity 200Mph) crashing into a wall, while a tank shoots varying

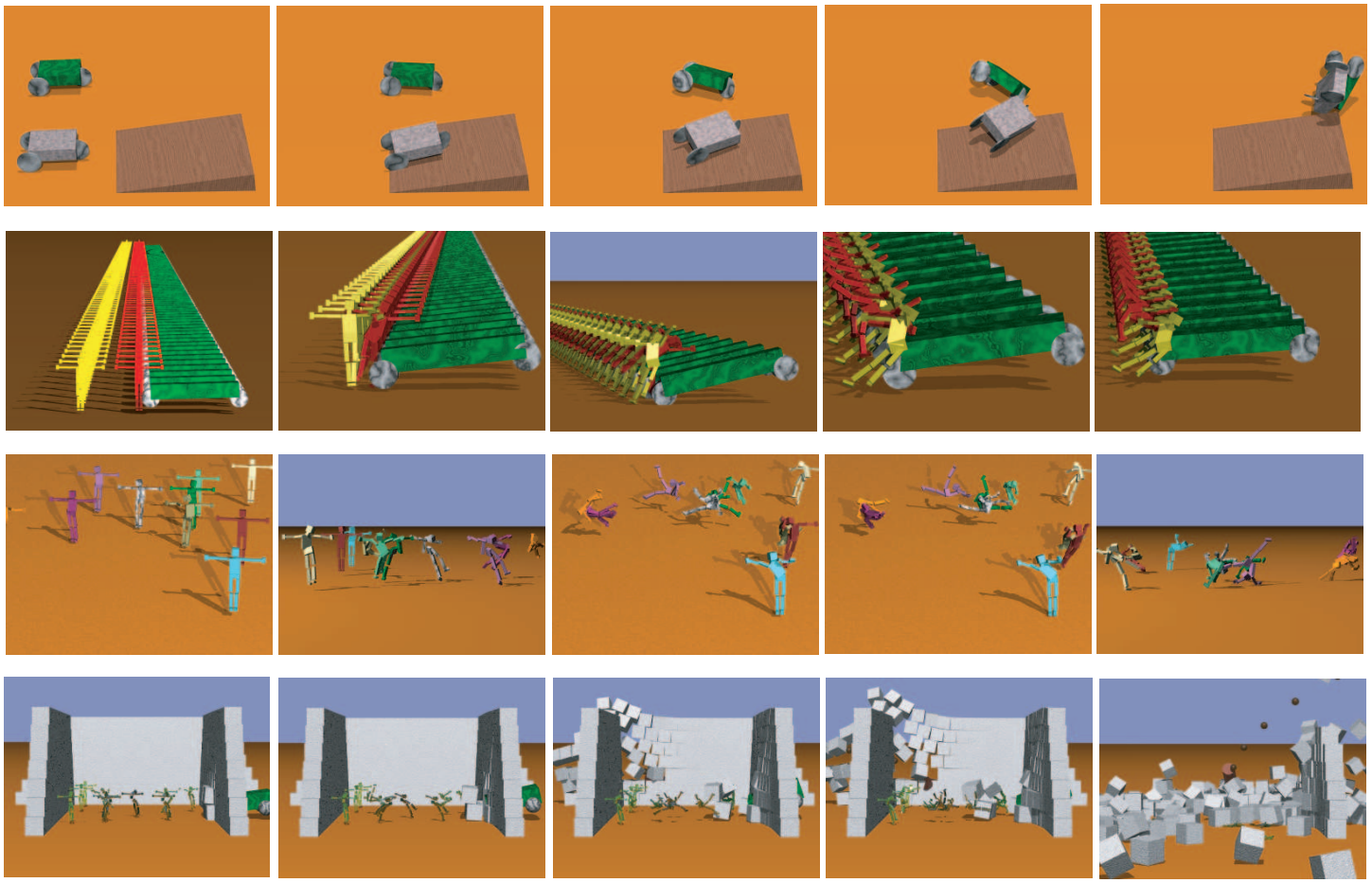


Figure 4.2: Benchmarks 2-Cars, 100CrSk, Fight, and Battle2 from top to bottom. Images in raster order.

shape projectiles towards the wall. The wall consists of a large number of blocks.

- Environ: Complex environment scene with wall, tank, car, monster, and projectiles Similar to previous benchmark. Addition of tank firing projectiles and a centipede monster.
- 100CrSk: Car crashing on two people replicated 100 times - replicate the previous scenario 100 times.
- Battle: Battle scene I - One group of 10 humanoids attacked by tank. 2 groups of 4 and 6 humanoids crashing into each other.
- Fight: Fighting Scene, 2 groups of 5 humanoids - two groups of five humanoids that come in contact in pairs and eventually form a number of piles.
- Battle2: Battle scene II - a relative complex battle scene. A tank is behind the far wall shooting projectiles in different directions. A car crashes on the right wall while two groups of five people are fighting inside the compound. The walls eventually get destroyed and fall on the people.

These scenarios can capture complex interactions. The computational load of *2-Cars*, for example, relates to a wide range of two objects interactions that arise in interactive entertainment applications. These include racing games, airplanes that crash in midair, rocket and plane collision, tank-to-tank collision and even simple ships colliding. *Fight* captures the computational complexity of a wide range of human group activities that involve progressive interaction such as action, sports games and urban simulation scenes.

4.2 Characterization

We have described the unique characteristics of real-time physics simulation. To further support our claims, we compare and contrast PhysicsBench to graphics, embedded, and scientific applications at the algorithm level. Then, we present a detailed characterization of PhysicsBench and consider possible techniques to accelerate performance. The Alpha ISA was used for the data presented in this section.

Comparison Against Other Workloads A *graphics* workload includes the computations needed to draw a single frame after all motion parameters have been computed and applied to the associated graphics primitives (object geometries). For interactive entertainment applications all geometric primitives are approximated with polygonal meshes and most often meshes of quadrilaterals or triangles. To produce the final image, all polygons go through a set of well defined stages that include: geometric transformations, lighting calculations, clipping, projections, and finally rasterization. Most of these stages perform calculations based on a polygon's vertices. Each vertex is defined by four floating point numbers. All of these stages treat each polygon independently of the others. For realistic scenes, there are thousands of polygons involved. Therefore the typical graphics load is highly parallel and pipelined. Modern graphics cards have multiple hardware pipelines capable of treating massive numbers of polygons. Certain research groups have managed to use graphics hardware to accelerate specific physics-based formulations such as computational fluid dynamics. The grid-based nature of such approaches can be supported, albeit in awkward ways, by the graphics hardware. However, this type of adaptation is not appropriate for constrained rigid body formulations.

The SPEC CPU 2000 FP suite seems similar in that it makes use of similar numerical methods, but the constraints imposed by interactive entertainment applications

along with the specific load characteristics make it a different problem. For example, the relaxed accuracy requirement allows multiple levels of approximations and optimizations, such as higher error thresholds, restricted size matrices, constraint violation, higher time-steps, inter-penetrations, approximate iterative techniques etc. The differences are reflected by our measurements.

Embedded application suites like MiBench [GRE01] are also quite different from PhysicsBench. One major difference is the relatively small amount of floating point instructions seen in typical embedded applications.

PhysicsBench Characterization In order to accurately characterize PhysicsBench, we present some system independent data, along with performance data from some specific systems. In particular, we evaluate state-of-the-art mobile console, desktop, and server processors.

Table 4.2 presents the architectural parameters for these classes.

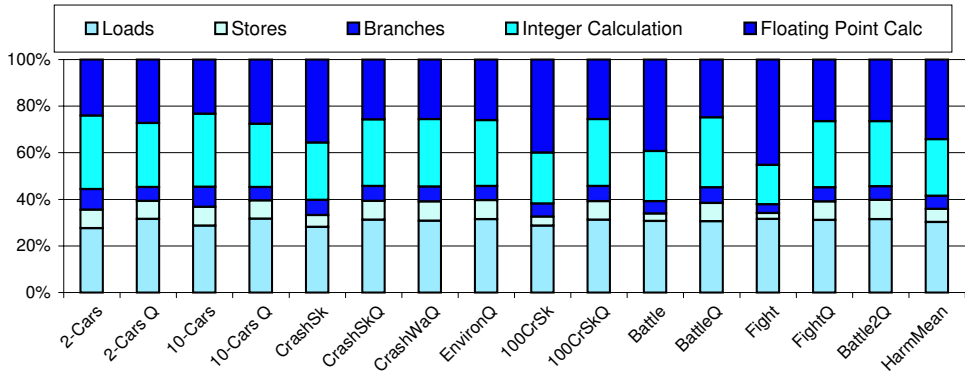


Figure 4.3: Instruction Mix for PhysicsBench.

Platform-Independent Characteristics Figure 4.3 provides the instruction mix for PhysicsBench. Despite the diverse input sets given to the physics engine, the instruction mix remains fairly uniform across benchmarks. This uniformity strongly contrasts with the diversity seen in typical general purpose, scientific, and embedded benchmark

suites. On average, PhysicsBench is composed of 34% floating point calculations, 25% integer calculations, 6% branches, 5% stores, and 30% loads. The relatively large amount of both integer and floating point calculations shows a fundamental difference between PhysicsBench and the integer heavy SPEC INT and MiBench, as well as the floating point heavy SPEC FP.

Instruction mix also depends on the algorithm used. For simple tests, ODE's quick step algorithm [Eng] executes more floating calculations and loads than the normal step. However, the the normal step executes more floating calculations in complex runs. The two algorithms are more efficient at different levels of complexity. The more accurate normal step algorithm is faster for simple islands while the quick step algorithm is faster for complex islands.

The harmonic mean for instruction per branch (IPB) across all PhysicsBench tests is 16. Similar to the instruction mix, the average behavior accurately represents all but a few outliers. *Fight Normal* shows 27 IPB while *2 Cars Normal* and *10 Cars Normal* show 11 IPB. On the other hand, SPEC CPU2000 FP shows an average IPB of 21 with many outliers ranging from 7.7 to 341.

On average, PhysicsBench's tests have 400KB of text and 370KB of data. In contrast, SPEC FP has on average 740MB of data and 980KB of Inst. The more than order of magnitude difference between the data segment sizes suggests a significant memory behavior difference between these two workloads. This will be corroborated by our performance data comparison.

Furthermore, we observe a drastic difference between the maximum stack sizes of these two workloads during run-time. The average maximum stack size for SPEC FP tests is 23KB with a maximum of 75KB, but the average maximum stack size for PhysicsBench is 1.1MB with a maximum of 2.9MB. This large stack is due to the dynamically allocated temporary structures to hold large matrices.

The combination of a small data size along with a large maximum stack size suggests that PhysicsBench's performance will heavily depend on the L1 cache's performance, while memory latency effects may be minimal. In contrast, SPEC FP's large data sizes with a small maximum stack size suggests the exact opposite behavior. This observation will also be corroborated by our performance data.

In order to focus our attention on the bottleneck, we capture the percentage of total instructions contributed by collision detection vs physics simulation. On average, collision makes up 7% of all executed instructions, ranging between 2% and 20%. Despite this, we will later demonstrate that accelerating collision detection can also yield substantial gains.

Platform Dependent Characterization Figure 4.4 presents results for the four architectures in table 4.2 on Physics Bench. We consider three metrics: IPC, Frame Rate, and the % of frames that were computed within 10% of 30 frame/sec constraint. The first two metrics are averages over all frames executed – they give some indication of how close we are getting on average to meeting the frame constraint.

The latter metric gives an indication of whether or not all frames were computed in time – ideally, we would like this metric to be as close to 100% as possible to provide stability and realism. For this initial study, we allocate 10% of each 1/30th ofa second to provide a frame to physics simulation. While this time allocation may be too conservative, the data presented can be extrapolated for larger allocation of each 1/30th of a second.

From this first set of data, we clearly see why current interactive entertainment applications rarely use realistic physics simulation to dynamically generate content. In this suite of physics-only tests, even the powerful *Desktop* and *Server* processors can only satisfy the demand of the three simplest scenarios. Surprisingly, the much lower

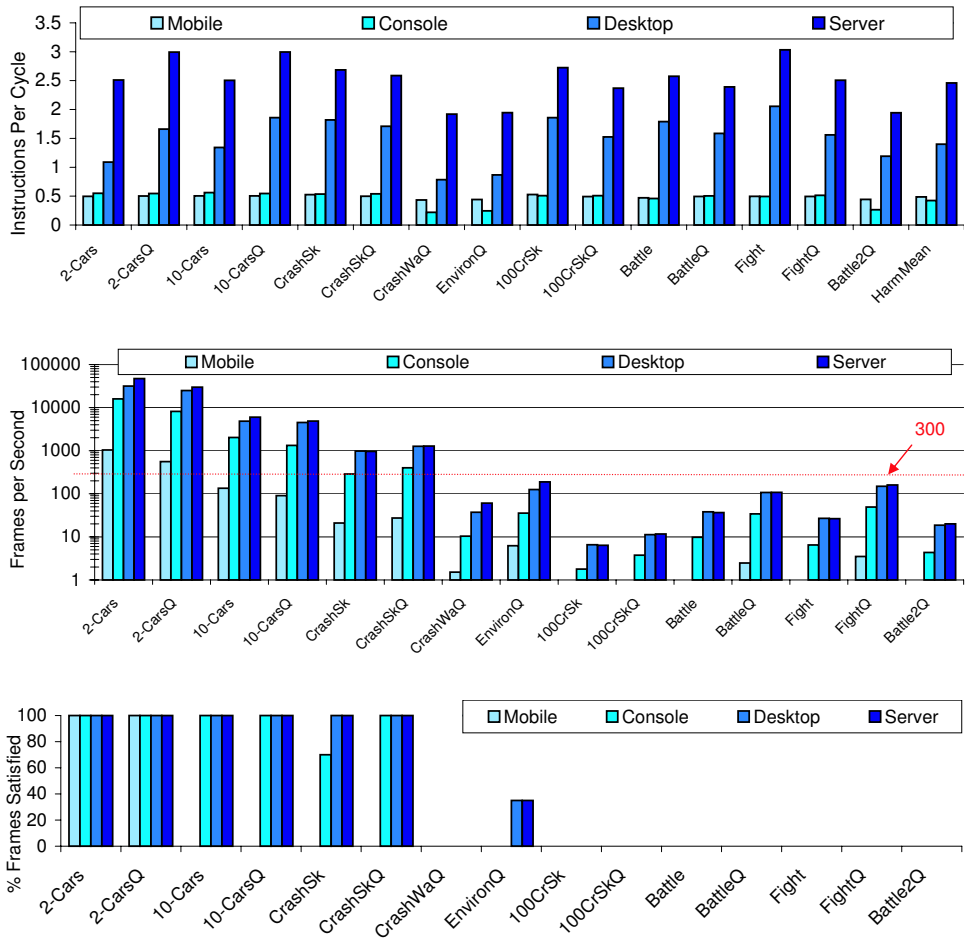


Figure 4.4: Performance of four modern architectures on PhysicsBench. [Note: Frame Rate uses log scale]

performance in-order *Console* processor is able to satisfy a similar number of frames. Due to its low clock frequency, *Mobile* is shown to be adequate only for the basic test of 2-car.

From these high-level observations, we can conclude the following: real-time physics simulation is extremely difficult to satisfy due to both the amount of computation required and the real-time constraint. Furthermore, there is a large performance requirement gap between the simple scenarios and the typical in-game scenarios.

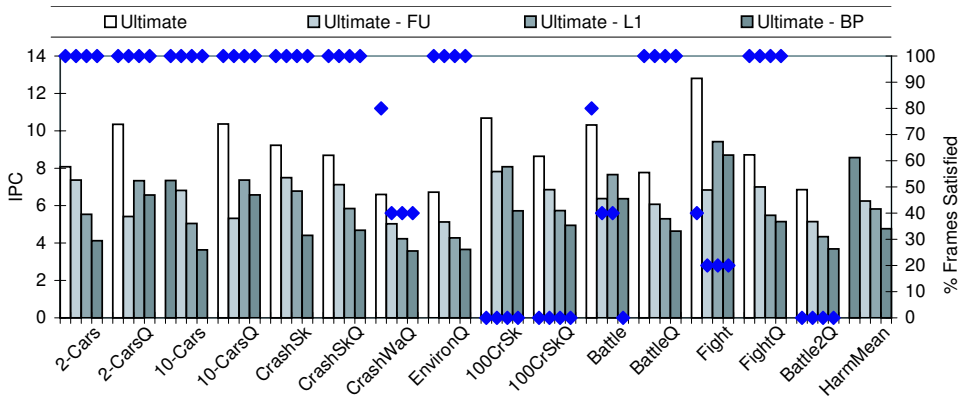


Figure 4.5: Performance of an ideal architecture on PhysicsBench.

In order to find the most useful directions of attacking the large performance gap shown above, we scale processor parameters to find the most critical bottlenecks. Due to the order of magnitude performance improvement required, we start this study with an idealized processor, the *Ultimate* core from Table 4.2.

Figure 4.5 shows the performance for *Ultimate* and when individual parameters are scaled down to more realistic conditions. The primary y-axis (bars) shows IPC and the secondary y-axis (diamonds) shows the % of frames that were computed within 10% of our 30 frame/sec constraint.

We explored scaling down each parameter of *Ultimate* independently. The suffix after the "-" indicates the one parameter being scaled (FU = functional latency, BP = branch misprediction penalty, and L1 = data cache size). We have scaled other parameters also, but only the parameters with the greatest interest and impact are presented. On average, the descending order of % performance degradation is miss penalty (BP) at 44% , L1 at 32%, and FU (functional unit latency) at 27%. First, the large effect of a more realistic branch misprediction penalty shows the large amount of instruction level parallelism being exploited by *Ultimate's* large instruction window and supporting structures. Second, the L1 is scaled to equal that of the *Desktop*. The performance

degradation supports our prior observation that the physics engine generates numerous temporary values during computation which necessitates a fast and large memory hierarchy. Finally, the effects of FU indicates that we have dependent chains of long latency operations on the critical path.

Although this extremely ideal configuration is getting closer to the goal, not all applications are being satisfied. With the major bottlenecks identified, we explore techniques to move us closer to real-time physics behavior.

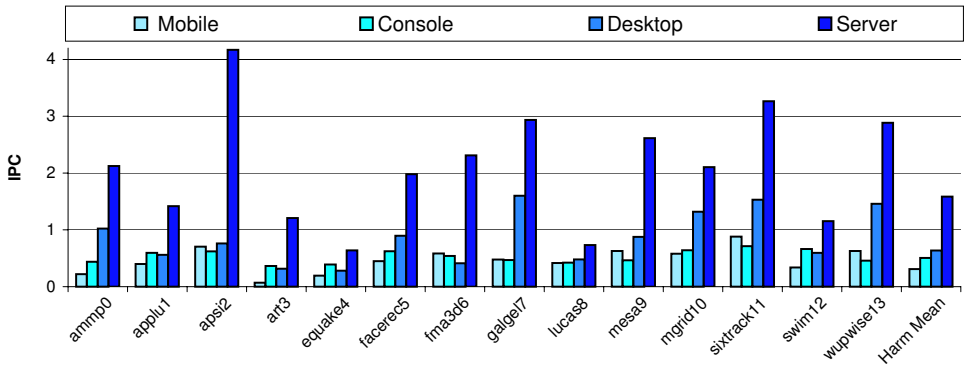


Figure 4.6: Performance of four modern architectures on SPEC FP.

As a point of comparison, we also show performance data from the SPEC FP suite. Figure 4.6 presents IPC results for the four architectures in Table 4.2 on SPEC FP. The most apparent behavior in this graph is the drastic performance difference between *Server* and all other designs. This corresponds with the fact that scientific workload is one of the major target workload for server processor designs. The much larger % of FP operations for this workload contributes to this difference among the four designs.

To find the most critical design parameters, we scale resources as in the above PhysicsBench study. Figure 4.7 presents IPC results for the *Ultimate* running SPECFP to compare against PhysicsBench. Again, each parameter of *Ultimate* is scaled one at a time. In addition to FP, BP, and L1, we also present the result when scaling memory latency to 267 cycles. On average, the descending order of % performance degradation

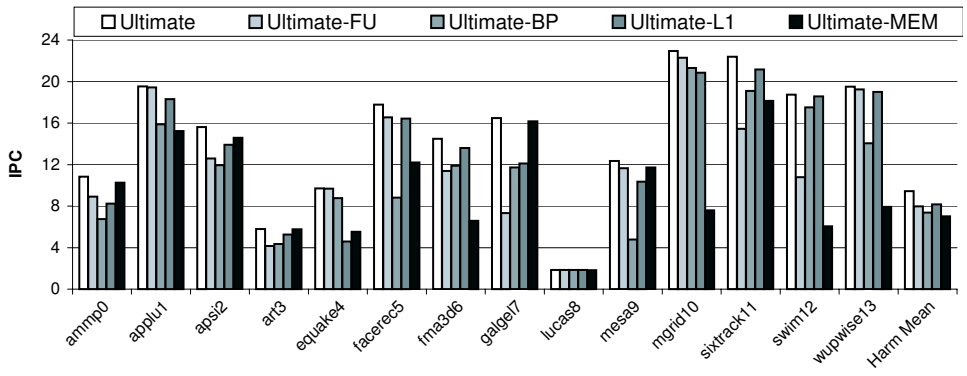


Figure 4.7: Performance of an ideal architecture on SPEC FP.

is memory latency (MEM) at 25% , branch mispredict penalty (BP) at 22%, and FU (functional unit latency) at 15%. Compared to PhysicsBench, the memory behavior is drastically different in that memory latency dominates any L1 cache behavior. This confirms our earlier workload characterization. Memory latency scaling for Physics-Bench results in $< 1\%$ IPC degradation.

	Mobile	Game Console	Desktop	Server	Ultimate
Frequency	222 MHz	3 GHz	3 GHz	2 GHz	5 GHz
Fetch, Decode, Issue	1, 1, 1	2, 2, 2	4, 8, 4	8, 8, 8	32, 32, 32
FetchQ Size FetchQ Speed	4, 1	16, 1	32, 2	32, 1	128, 2
Issue	In-order	In-order	Out-of-order	Out-of-order	Out-of-order
Issue window	8	16	32	64	512
Branch Predictor	Taken	4K Bimod, 4K 4-way BTB	8K Gshare , 4K 4-way BTB	8K Gshare, 4K 4-way BTB	16K Gshare, 32K 4-way BTB
Branch Miss Penalty	3	19	19	17	2
Inst L1 Cache Latency	8K 4 way 2 cycle	32K 4-way 4 cycle	8K 4-way 2 cycle	64K Direct Map 2 cycle	512K 4 way 1 cycle
Data L1 Cache Latency	8K 4 way 2 cycle	32K 4-way 4 cycle	16K 4-way 4 cycle	32K 2-way 4 cycle	512K 4 way 1 cycle
Inst Window, Load/Store	32,8	64,32	128,64	256,64	2048,1024
L2 Cache	None	512K 8-way 16 cycle	1M 8-way 27 cycle	2M 8-way 24 cycle	64M 16-way 12 cycle
Functional Units (int ALU, Mult) (FP, FP Mult) (Memport)	 1, 1 1, 1 1	 3, 1 2, 1 1	 6, 1 2, 1 2	 6, 1 4, 4 3	 32, 32 32, 32 32
Mem Latency	20	258	269	184	50

Table 4.2: Parameters for our architectural configurations. All architectures use a common ISA.

4.3 Parallelization

In this section we explore architectural candidates to satisfy the frame constraint of PhysicsBench 1.0.

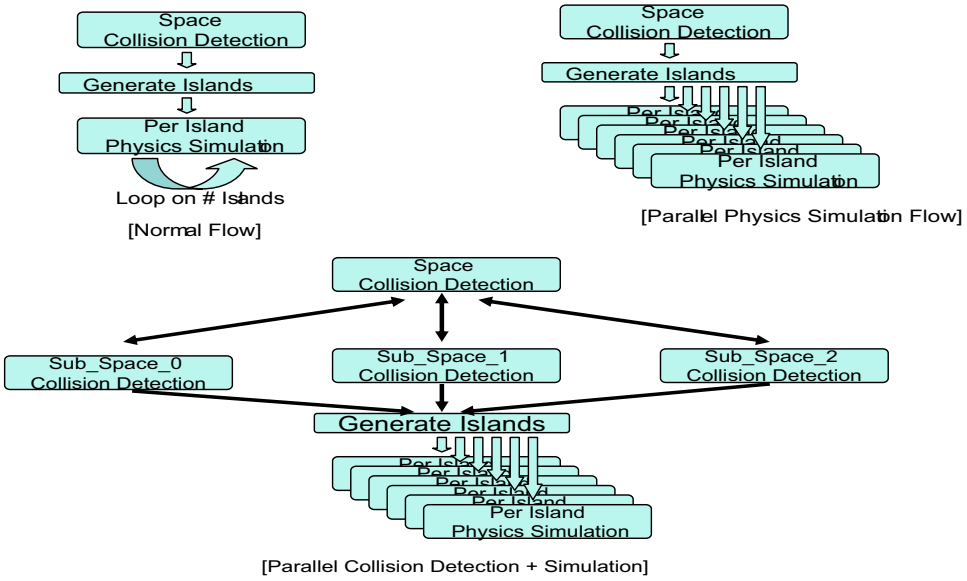


Figure 4.8: Normal, Parallel Simulation, and Parallel Collision Detection Flows.

Parallel Threads As demonstrated by Figure 4.8, the core physics simulation loop can be simplified into 3 dependent logical steps: collision detection, island creation, and per island physics simulation. Collision detection finds all object pairs that interact with one another. Then, islands are formed by interconnected objects. Finally, the engine computes the new positions for all objects at the island granularity. The physics simulation for all of the islands is done serially.

Parallel Physics Simulation From the earlier instruction mix study, we see that physics simulation (PS) code contributes the bulk of instructions executed. On average, PS contributes to 92% of the execution time across the suite using the four processors presented earlier. Therefore we first explore the limits of parallel physics

simulation by creating one thread for every island. This parallelization process involves farming off the threads to either logical or physical processors. The initial data communication requirement is dictated by the number of bodies and joints for each island spawned away from the original thread. Because every island is independent of other islands, only the final position data of objects needs to be communicated back to a central thread at the end of each simulation step.

To evaluate the potential of this optimization, we first capture the upper bound performance by simulations that assume an unlimited supply of homogeneous cores and ignores overhead from sources such as thread creation, thread migration, data migration, and setup codes. The data presented in Figure 4.9 contains both frame rate and % frames satisfied. Because tests 2-Cars and 10-Cars are easily satisfied, we remove these from future data graphs and discussions to conserve space.

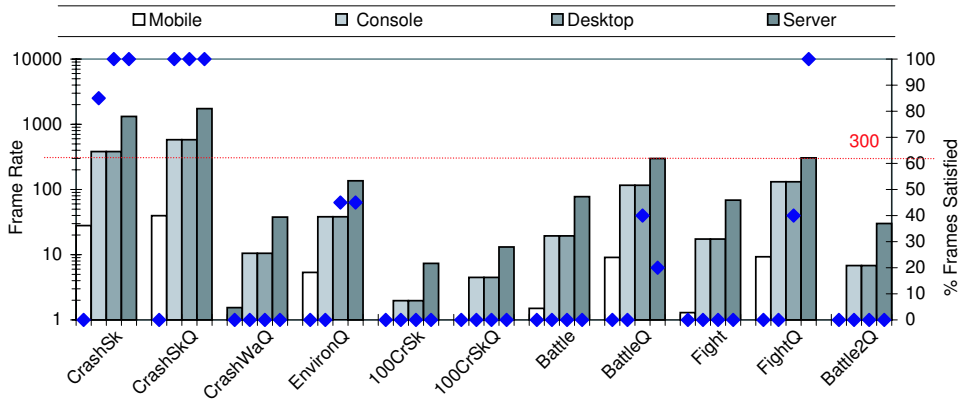


Figure 4.9: Alpha ISA Performance of Parallel Physics Simulation. [Frame Rate uses log scale]

For frames satisfied, we see that *Mobile* is now able to satisfy 100% of 10-Cars and *Server* is now satisfying 100% of FightQ. Frame rate improvement is consistent across the suite with a max of 516% and an average of 118%. However, the magnitude of speedup from parallel physics simulation varies between different tests. The tests that showed the least amount of improvement are CrashWaQ and EnvironQ. CrashWaQ and

EnvironQ contain a large complex island simulating a brick wall. This wall's bricks apply contact forces on one another, and this cannot be parallelized unless these bricks are pushed away from one another. In scenarios containing one extremely complex island, the frame rate is dictated by the processing of this island even with parallelization.

At the algorithm level, the QuickStep function used for these tests already estimates the result by processing each object independently of others. QuickStep uses an iterative approach where during each iteration (a) each body in an Island is essentially considered a free body in space and solved independently of the others (b) a constraint relaxation step progressively enforces the constraints by some small amount. The constraint satisfaction increases with the number of iterations. Fine-grain parallelization of the LCP solver will be discussed in the next chapter.

Even though our idealized coarse-grain parallel physics simulation is very effective, we are still some distance away from satisfying the demands of these benchmarks, especially for the extreme cases described. As a result of parallel physics simulation, the % of cycles taken up by collision detection becomes much more significant at 20% on average, and a maximum of 55%. Therefore, we consider performing collision detection in parallel.

Parallel Collision Detection + Physics Simulation Figure 4.8 shows our implementation of parallel collision detection through a hierarchy of collision spaces. Groups of frequently interacting objects are inserted into the same subspace, and only the subspaces are directly inserted into the root space. Note the bidirectional arrows, representing two-way communication between collision threads, interconnecting the root space with subspaces. During collision detection, the root thread handles any collisions across subspaces while each subspace handles collisions within its domain.

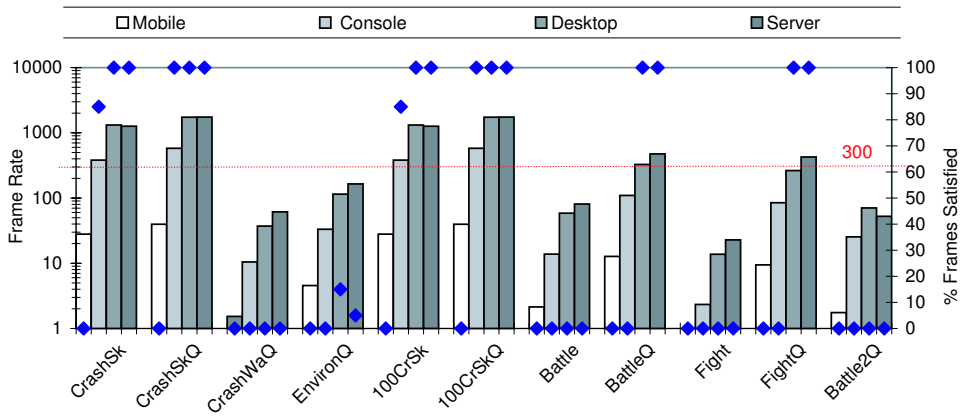


Figure 4.10: Alpha ISA Performance of Parallel Collision Detection + Physics Simulation. [Frame Rate uses log scale]

Take the 100CrSk test for example – we can create 100 subspaces to contain each trio of a car and two humanoids. Because of each trio's physical location, no interaction between subspaces happens during its execution. This allows us to completely parallelize it into a fraction of the original task with minimal overhead on the root space thread.

Frame rate and % frames satisfied are presented in Figure 4.10. In contrast to parallel physics simulation, the results show both significant improvements as well as degradations. The most apparent change is that 100CrSk and 100CrSkQ both now have more than 80% satisfaction for *Console*, *Desktop*, and *Server* processors. In addition, both FightQ and BattleQ can now be 100% satisfied by *Desktop* and *Server*.

Even without taking certain overheads into account, parallel collision detection degrades the performance of EnvironQ, Fight, and Battle. All three tests have fast changing island makeup along with collisions across the logical spaces created. This indicates a need for utilizing high-level context information to selectively enable parallel collision detection.

With both optimizations enabled, a comparison of Figure 4.10 and Figure 4.5

shows that *Desktop* and *Server* can achieve levels of user experience similar to that of the unrealistic *Ultimate* design.

Resource Requirement for Parallelization To bound the resource requirement necessary to achieve the improvements presented above, we refer back to Table 4.1 which shows the distribution of island count and space count across the benchmarks.

For parallel physics simulation, the maximum island count for each benchmark indicates the number of cores we need to achieve the improvements shown earlier. As shown in Table 4.1, the maximum island count in the suite is 337. However, we may need far fewer cores to achieve the performance shown due to simple islands which can be serially processed on one core. We present the resource requirements using optimal load balancing for the *Server* processor in Table 4.3. The data shows that most benchmarks with high island counts can actually be satisfied with less than 5 *Server* cores. The outlier, 100CarSk, can also be satisfied with a few cores, but we parallelized it into 100 worlds to show the opportunity for effective massive parallelization given non-interacting virtual spaces.

Name	2 Cars N/Q	10 Cars N/Q	CrashSk N/Q	Environ	CrashWa	100 CarSk N/Q	Battle N/Q	Fight N/Q	Battle2
Number of Cores	3 , 2	9 , 9	3 , 3	4	3	100 , 100	14 , 19	3 , 4	3

Table 4.3: Resource Requirement for Full Parallelization using *Server* Cores.

4.3.1 Real x86 Processor Evaluation

In this section, we explore the performance of PhysicsBench on a real x86 processor. For this real processor study, we used a 2.4GHz Intel P4 Xeon CPU with 512KB L2 cache, with support for SSE/2 instructions. This study is similar to section 4.3.

Real Processor Methodology PhysicsBench 1.0 includes tests using both the *big-matrix* and the iterative solvers. In this section, we focus on enabling real-time physics simulation by accelerating the iterative solvers (QuickStep), the faster of the two approaches.

The PhysicsBench suite consists of two sets of source code. The first set contains graphics code to allow for visual correctness inspection, and the second set contains only user input and physics simulation code for performance evaluation. We compiled binaries for the x86 ISA using gcc version 3.4.5 at optimization level -O2 (recommended by ODE), using single precision floating point and the following flags: -ffast-math, -mmmx, -msse2, -msse, -mfpmath=sse and -march=pentium4. These options enable full SSE support to exploit SIMD parallelism.

All benchmarks are warmed up for 3 frames to execute past setup code as well as warm up processor resources, and then we execute 5 frames. We have designed the benchmarks so that significant activity is captured within these 5 frames (i.e. the actual collision of a car and skeleton, the crumbling of a wall, etc).

We model a uniprocessor as the baseline for our study – the predominant execution hardware for current gaming platforms and the architectural target of most current physics engines, including ODE.

As described in the introduction and [Wu05], the physics engine is interdependent on other software components of the application. These include AI, game-play logic, audio, IO, and graphics rendering. We allocate 10% of each 1/30th of a second frame

for computing physics simulation. While this is a conservative estimate, behaviors for a larger time allocation can be extrapolated from the presented results.

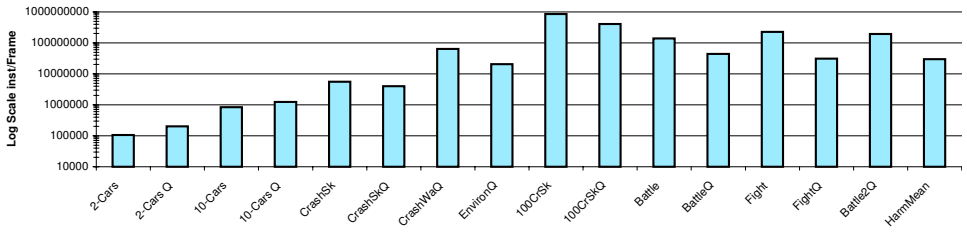


Figure 4.11: Instructions Per Frame for PhysicsBench.

We consider two performance metrics: Frame Rate (frames per second), and the % of frames that were computed within 10% of our 30 frames/sec constraint. The first metric is the harmonic mean over all frames executed, giving some indication of how close we are getting on average to meeting the frame constraint. The second metric gives an indication of whether or not all frames were computed in time. Ideally, we would like this metric to be as close to 100% as possible to provide stability and realism.

We base our performance metrics on frames rather than instructions, as frames are a more natural fit for interactive entertainment – particularly since the performance goal is measured in frames per second. We show the number of instructions per frame for each individual benchmark in Figure 4.11.

PhysicsBench Results Figure 4.12 presents results for actual runs of PhysicsBench on the architecture in section 4.3.1. It is clear why current interactive entertainment applications rarely use realistic physics simulation to dynamically generate content. In this suite of physics-only tests, our test processor can only satisfy the demands of the simple scenarios described by 2-Cars, 10-Cars, and CrashSk.

We present the resource requirements using optimal load balancing in Table 4.4. The data shows that most benchmarks with high island counts can actually be satisfied

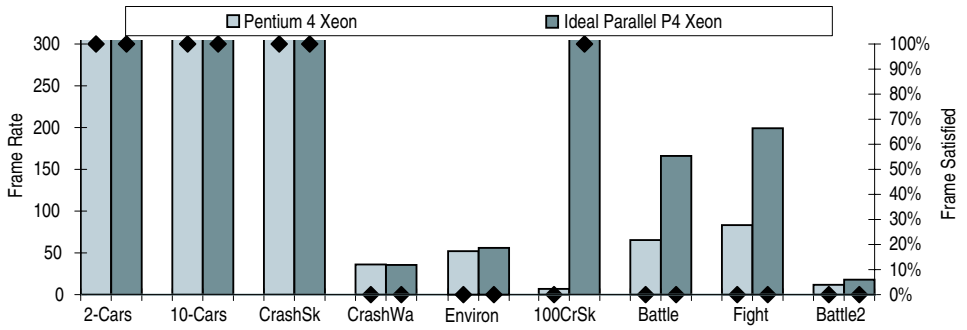


Figure 4.12: x86 ISA PhysicsBench Performance.

with less than 5 cores. The outlier, 100CarSk, can also be satisfied with a few cores, but we parallelized it into 100 worlds to show the opportunity for effective massive parallelization given non-interacting virtual spaces.

Benchmark	2-Cars	10-Cars	CrashSk	Environ	CrashWa
Number of Cores	2	9	3	4	3

Benchmark	100CrSk	Battle	Fight	Battle2
Number of Cores	100	19	4	3

Table 4.4: Resource Requirement for Full Parallelization for real x86 Processor.

4.3.2 Fine Grain Parallelism

After exploiting coarse grain parallelism for islands and spaces, benchmarks with large islands still can not be satisfied. Given that quickstep's algorithm computes forces from different constraints independently, fine grain parallelism offers additional performance. To quantify the available fine-grain parallelism, we graphed the dependency chain of dynamic instructions for the most frequently executed loop in ODE inside the LCP solver. This is done by tracking each instruction's dependencies to the last logical register producer inside the PTLsim simulator. This data revealed massive parallelism for the most compute-intensive islands. Computation for each degree of freedom re-

moved can be solved independently. Similarly, all pair-wise collision detections can be computed in parallel. The number of pairs is pruned by hierarchical collision geometries, but the worst case is NxN, where N = the number of objects.

Benchmark	Degrees of Freedom				Number of Bodies			
	< 10	11..30	31..120	> 120	< 10	11..30	31..120	> 120
Crash Sk	0%	27%	69%	4%	27%	63%	10%	0%
Crash Wa	36%	51%	0%	13%	87%	0%	0%	29%
Environment	72%	12%	3%	12%	83%	1%	10%	9%
100 Car Sk	0%	31%	62%	7%	31%	51%	18%	0%
Battle	73%	14%	13%	0%	95%	21%	6%	0%
Fight	0%	0%	75%	25%	0%	75%	25%	0%
Battle2	26%	2%	45%	28%	27%	45%	22%	7%
Average	30%	20%	38%	13%	50%	37%	13%	6%
Benchmark	Number of Joints							
	< 10	11..30	31..120	> 120				
Crash Sk	28%	65%	6%	0%				
Crash Wa	71%	0%	0%	11%				
Environment	84%	3%	3%	13%				
100 Car Sk	31%	50%	19%	0%				
Battle	69%	29%	2%	1%				
Fight	0%	58%	42%	1%				
Battle2	31%	40%	22%	25%				
Average	45%	35%	13%	7%				

Table 4.5: Distribution of factors affecting fine-grain parallelism.

Table 4.3.2 shows the distribution of factors that reflect each islands' processing demand, namely degrees of freedom (DOF) removed, number of bodies, and number of joints. The data is grouped in four clusters, and we show the percentage of islands that has a value within the range specified by the groups. DOF ranges in the thousands

for the worst case islands.

Per-Island Execution Time Breakdown Using Amdhal's law to estimate the required speedup on parallel sections, we breakdown the most demanding island's execution time into serial and parallel portions. Using a real Intel Pentium 4 processor with ideal parallelization, it takes 233 Million cycles. The execution time breakdown for the 1st captured frame for battle2 shows the following:

- 196M cycles for stepping function with 160M cycles inside constraint solver loop LCP.
- 19M cycles for Collision detection.
- 4M cycles for setting up matrix before calling stepping function.
- The rest of the execution takes 14M cycles.

Parallelizable sections total 219M cycles, and serial sections total 14M cycles. Assuming a clock rate of 5GHz and a frame rate of 300, we need to complete all execution in roughly 16.6 Million cycles. This necessitates a speedup of roughly 84X on the parallel sections. The design exploration to enable this level of parallelism is described in Section 5.2.

CHAPTER 5

Architectural Acceleration

This chapter details our major architectural contributions: PhysicsBench 2.0, ParallAX, and PDAS.

5.1 PhysicsBench 2.0

PhysicsBench 2.0 is a comprehensive set of benchmarks that capture the complexity and scale of physics simulation that might appear in future game-scenarios. Our benchmark suite can be leveraged by: (1) computer architects/researchers to explore real-time physics hardware and software designs, and (2) application designers to determine gaming platform performance bounds.

Our second-pass at physics benchmark design was guided by the approach shown in Table 5.1, and based on the set of required features demonstrated in Table 5.2. Table 5.3 explains the benchmarks while Table 5.4 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 the physics complexity of 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.

PhysicsBench 1.0 benchmarks [YFR06] are limited to small scale rigid-body interactions. PhysicsBench 2.0 has dramatically increased the types of simulations and number of interacting objects. Based on personal communications with 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.

High-Level Physical Actions	These types of actions set the focus for the benchmarks. They include continuous contact, periodic contact, high velocity impulse explosions, and deformations.
Representative Game Genre	The most popular game genres which use or have the potential to use physics include racing, sports, action, first-person shooter real-time strategy, and massive multi-player role-playing. For each benchmark, we pick a genre which we believe best illustrates a given high-level physical action. Screen-shots of upcoming next generation games were used as reference. These include Motor Storm, Battlefield 2, Cell Factor, GTA: San Andreas, Assassin's Creed, World Soccer Winning Eleven, and World of Warcraft.
Parameterization and Scaling	All benchmarks have a set of parameters that scale its computational load. For collision detection, these parameters include number, distribution and shape of objects. For forward stepping, these parameters include complexity and number of both islands and objects. For cloth simulation, the parameters include the number of vertices representing each cloth object, the number of cloth objects, and their location.

Table 5.1: Our benchmarks cover a wide range of parameterized situations within different game genres.

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 height fields or trimeshes.
Breakable Joints	Joints are broken by accumulation of force or a single strong force exceeding a predetermined threshold. Bridges, cars, and robots contain breakable joints.
Prefractured Objects	Each breakable object contains a set amount of debris that can break apart from the object. The object and geom representing each piece of debris is created at startup time and enabled once the object breaks.
Explosions	Each object is marked with an explosive flag. If an explosive object makes contact with any other object, the explosive object is replaced by a sphere representing the blast radius. The blast radius and duration are predetermined. The sphere is disabled after duration. Time bombs and cannonballs are used.
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	Soft-body 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 5.2: Features Found in Our Benchmarks.

Benchmark	Avg Inst/Frame	Description
Periodic Contact	34 million	Role-playing game genre scenario with groups of humanoids engaging in hand-to-hand combat: 30 humanoids with 3 groups of 5, 3 groups of 3, and 3 groups of 2 where all members of each group are engaged in combat with one another.
Ragdoll Effects	36 million	First-person shooter (FPS) genre scenario with 30 humanoids falling due due to impact from projectiles.
Continuous Contact	47 million	Racing genre scenario with cars driving on terrain and between obstacles: a rally race with 30 cars driving over terrain formed by height fields and trimeshes.
Breakable	256 million	FPS genre scenario with cannons shooting and bombs exploding. Three areas are each enclosed by three walls. Two bridges are in each area 30 humans are scattered in groups of 10. The wall bricks fracture into pieces due to explosions from the cannonballs. Six vehicles ram the walls and explode upon contact.
Deformable	409 million	Sports or action genre scenario with 30 uniformed players and 2 large cloth objects each in contact with one player. Each uniform is a small cloth object attached on a player.
Explosions	547 million	Real-time strategy scenario with an army fighting in an urban environment: 10 areas are enclosed on three sides by walls. 50 vehicles roam the area with 10 cannons shooting exploding projectiles. There are no breakable joints or prefractured objects.
Highspeed	518 million	Action scenario with cars crashing into walls and high-speed rockets destroying buildings: there are 10 buildings and 20 moving cars. 10 cannons shoot high-speed projectiles at the buildings. There are no explosions – just the complexity of detecting high-speed impacts.
Mix	829 million	A combination of all the features and entities used in the previous 7 benchmarks. There are 3 buildings, 6 bridges, 30 humanoids and 6 vehicles in the area. The humanoids are draped in cloth, and the buildings' openings are covered by large cloths. Height field terrain, breakable joints, prefractured objects, and exploding projectiles are all used.

Table 5.3: Our Physics Benchmarking Suite.

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

Table 5.4: Benchmark Specs.

5.2 ParrallAX: An Architecture for Real-Time Physics

Interactive entertainment (IE) applications demand high performance from all architectural components. In the future, this demand will increase even further. These applications will be composed of a diverse set of computationally demanding tasks. One critical task is modeling how objects and characters move and interact in a virtual environment. Most current IE applications make use of recorded motion clips to synthesize the motion of virtual objects (kinematics), but as these objects and their interactions scale up in complexity, 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 5.2.3 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 granularity. 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 [PF05], the Cell's SPE's paired with its PPE [Hof05], and a conventional core paired with multiple vector units [KP03]. 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 work 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.

This section presents the following contributions:

- Identifying 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.
- 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 5.2.3.1 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 5.2.4.1 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 5.2.4.2 and 5.2.5.2 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.

A bulk of the prior work has been described in section 3.2.

The rest of this section is organized as follows. Section 5.2.1 describes the modified physics engine and the associated computational load. In section 5.1 we propose a set of future-thinking benchmarks that represent a wide range of physical actions and entertainment scenarios. Section 5.2.2 details the experimental setup. Section 5.2.3 explores the performance of this suite on conventional architectures and threading methodologies. In section 5.2.4 we outline the proposed physics architecture, and in section 5.2.5 we explore its design space. we conclude in section 5.2.6.

5.2.1 Physics Simulation and Workload

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

5.2.1.1 Our Physics Engine

Our physics engine is a heavily modified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [Eng]. ODE follows a constraint-based approach for modeling articulated figures, similar to [Bar97, MHH06], and it is designed for efficiency rather than accuracy. Our implementation supports more complex physical functions, including cloth simulation, prefractured objects, and explosions. We have parallelized it using POSIX threads and a work-queue model with persistent worker threads. POSIX Threads minimize thread overhead, while persistent threads eliminate thread creation and destruction costs.

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 prefractured 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 heart (forward simulation step) of the simulation loop lies the constraint solver which is typically implemented with an iterative relaxation method. The simulator provides two key parameters that can be used to trade off accuracy for efficiency, the time-step Δt and the number of iterations n that the constraint solver performs per time-step. The time-step defines the amount of simulated time that separates successive executions of the simulation loop and therefore defines how many times the loop will execute per simulated second. The number of solver iterations controls how many relaxation steps the solver takes in a single simulation step.

For our benchmarks, the 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 also use 20 solver iterations as recommended by [Eng].

5.2.1.2 Computational Phases of The Physics Workload

The steps in the above algorithm form a data-flow of computational phases shown in Figure 5.1. Simulations such as cloth and fluid are specializations of this. However,

because cloth simulation presents a conceptually and numerically different load than rigid-body simulation we considered it as a separate phase in our study. Below we describe the 5 computational phases in more detail. In this paper, we will use the terms *phase* and *task* interchangeably.

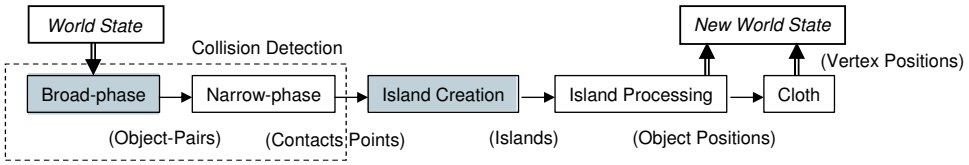


Figure 5.1: Physics Engine Flow. All phases are serialized with respect to each other, but **unshaded** stages can exploit parallelism within the stage.

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. ODE originally uses a single joint-group where all contact joints are stored, enforcing an artificial dependency across all object-pairs. We added a data structure for each thread to store created contacts, allowing all object-pairs to be processed in parallel. 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 granularity. 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[Jak01]. An extension of this approach that handles more general constraints has been proposed by AGEIA[MHH06]. 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.

5.2.2 Experimental Setup

We used a Simics-based [MCE] full-system execution-driven simulator. Both the cache and processor timing simulators are from the GEMS tool-set [MSB]. All L2 caches are based on 1MB 4-way banks, and Table 5.5 shows the configuration parameters used for the coarse-grain cores in our simulations. Fine-grain cores are described in section 5.2.5.

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 5.5: Coarse-grain Core Design.

PhysicsBench 2.0 benchmarks are executed for 3 frames of physics simulation due to very long simulation times – the average number of instructions in a frame is shown in table 5.3. 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 so that most of the activity happen in the first 10 frames. The frames 5-7 are executed, and the worst-case frame in terms of 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 within Simics are SPARC ISA processors running Solaris 10 in single user mode. All benchmark binaries executed for performance simulation are compiled with gcc 4.1.1 for the SPARC ISA using the following optimization flags to enable O2 optimizations, SPARC's SIMD instructions, 32-bit pointers, and the POSIX threads library: [-mcpu=v9 -mtune=ultrasparc3 -mvis -m32 -pthreads -O2]. The SPARC binaries used for performance simulation do not include any graphical display code. For visual verification, the benchmarks with visual display code are compiled for the x86 ISA and executed on real x86 machines. All instrumentation for separating computation phases uses Simics' MAGIC instruction, which is not counted when calculating execution time.

5.2.2.1 Interconnect Models

For the on-chip 2D mesh interconnect, we used the data in Table I from [SEW06] 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) [HTX] or PCI Express (PCIe) [PCI] to connect the discrete chips. On the fine-grain chip, the same 2D mesh described above connects all cores to the I/O.

5.2.3 Performance Demands of Real-Time Physics Workload

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

Figure 5.2(a) shows the total execution time 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* can be completed within a frame's worth of time and the most complicated benchmark *Mix* requires over an order of magnitude performance improvement to reach 30 FPS. Execution time is broken down into contributions by the phases described in section 5.2.1.

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 5.7(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 to satisfy 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).

5.2.3.1 Working Set Analysis: Serial and Parallel Phases

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 5.2

(b) shows execution time for the serial phases as the L2 cache is scaled from 1MB to 32MB by incrementing the number of 4-way 1MB banks. The banks are configured with a point-to-point network and use a 2-level directory based MOESI coherency protocol [MSB].

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 5.3 (a) and 5.4 (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 5.2 (b).

Figures 5.3(b), 5.4(b), and 5.5(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 5.4. The increased amount of data required for *Explosions* and *Highspeed* results in their higher sensitivity to L2 cache size.

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 5.3(a) and 5.4(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 readable but not modifiable 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.

5.2.3.2 Exploiting Coarse-Grain Thread-Level Parallelism

Figure 5.5 (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 observations made earlier: (12MB to-

tal – organized into three 4MB partitions [STW92, RAJ00b, YR]: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 5.1. 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 DMA transfers.

The L2 is composed of 1MB 4-way cache banks, and the partitioning granularity is done per cache way [CJD00]. Because serial stages are more sensitive to load latency, the 4MB cache partitions designated 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.

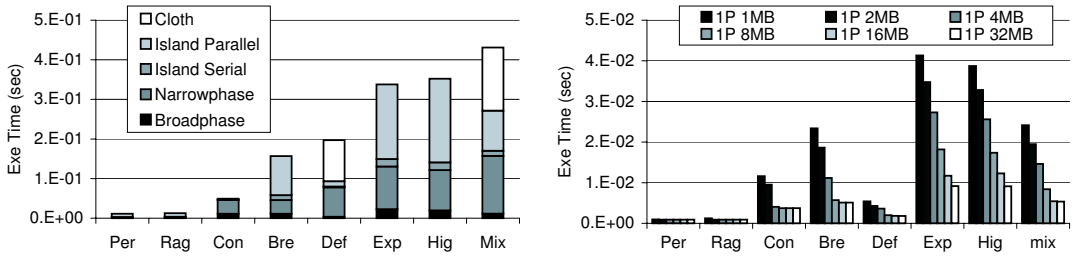
Figure 5.5 (b) clearly shows that the performance improvement starts to plateau at 4 cores. On average, scaling from one to two cores improves performance by 53% and scaling from two to four cores improves performance by 29%. Figure 5.6 (a) shows the execution time breakdown for a four core processor 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 5.6 (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.

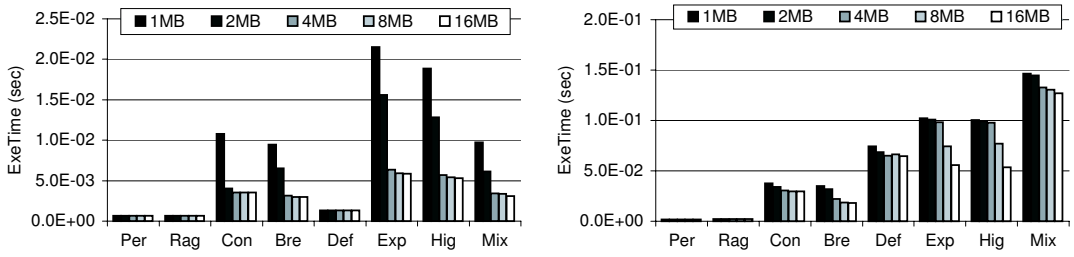
Assuming that this behavior can be alleviated by a custom-tuned operating system and programmer defined memory mapping/management, CG scaling may continue past four threads. However, even under ideal conditions (removal of OS overhead and cache contention, unlimited number of cores, and ideal load balancing), CG parallelism will not be sufficient to achieve interactive frame-rates. Figure 5.7(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 data-path 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 data-path 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 5.2.5 that the use of complex cores results in prohibitively large die areas.



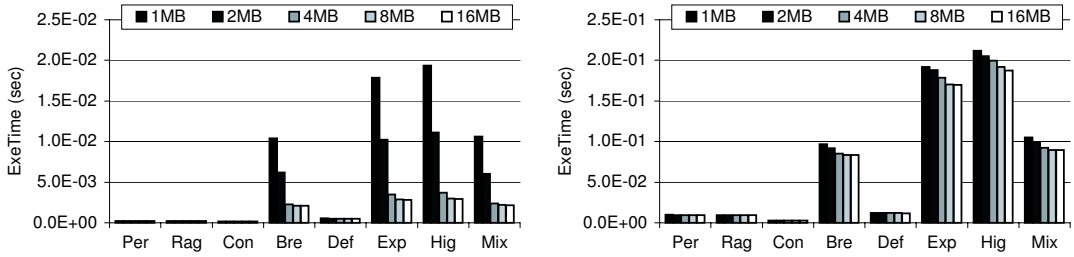
(a) (b)

Figure 5.2: (a) Execution Time Breakdown of 1 Core + 1MB L2 — (b) Single Core Execution of Serial Parts with Different L2 Sizes.



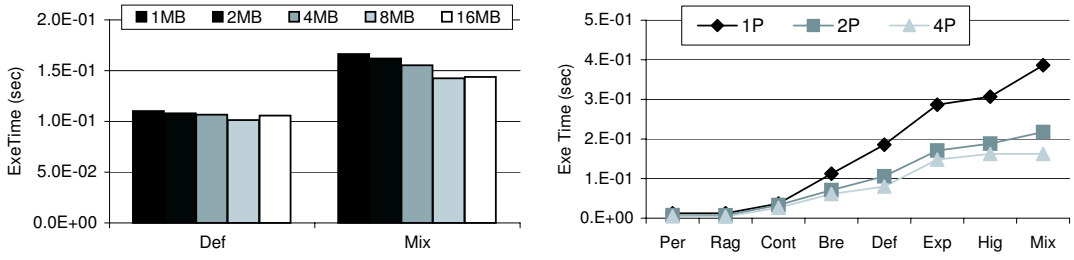
(a) (b)

Figure 5.3: (a) Performance of Broadphase with dedicated L2 — (b) Performance of Narrowphase with dedicated L2.



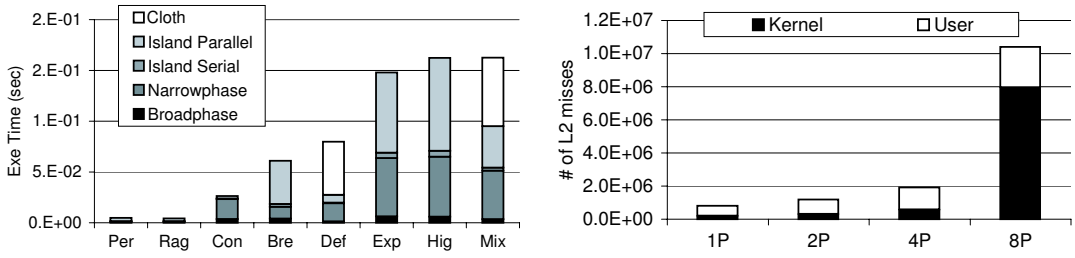
(a) (b)

Figure 5.4: (a) Performance of Island Creation with dedicated L2 — (b) Performance of Island Processing with dedicated L2.



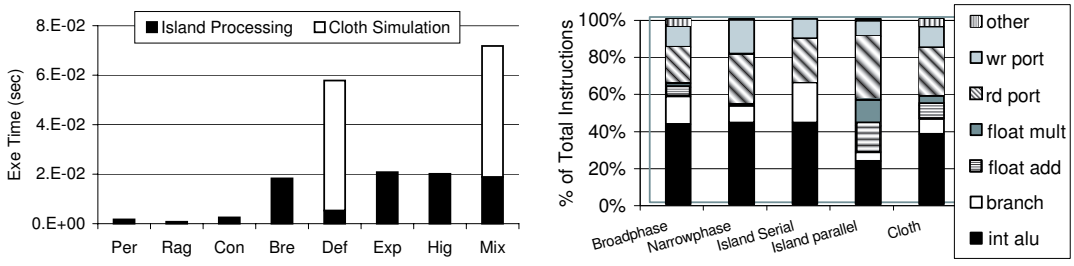
(a) (b)

Figure 5.5: (a) Performance of Narrowphase with dedicated L2 — (b) Performance with Processor Scaling.



(a) (b)

Figure 5.6: (a) Execution Time Breakdown of 4 Core + 12MB L2 — (b) L2 Miss Breakdown with Thread Scaling.



(a) (b)

Figure 5.7: (a) Limit of Coarse-grain Parallelism. — (b) Instruction Mix for all 5 Phases.

5.2.4 ParallAX Architecture

Figure 5.8 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 5.2.4.1), (2) the L2 partitioning strategy that has already been addressed in section 5.2.3.1, and (3) an exploration of interconnect sensitivity (sections 5.2.4.2 and 5.2.5.2). The two models of Figure 5.8 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 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 simulation 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:

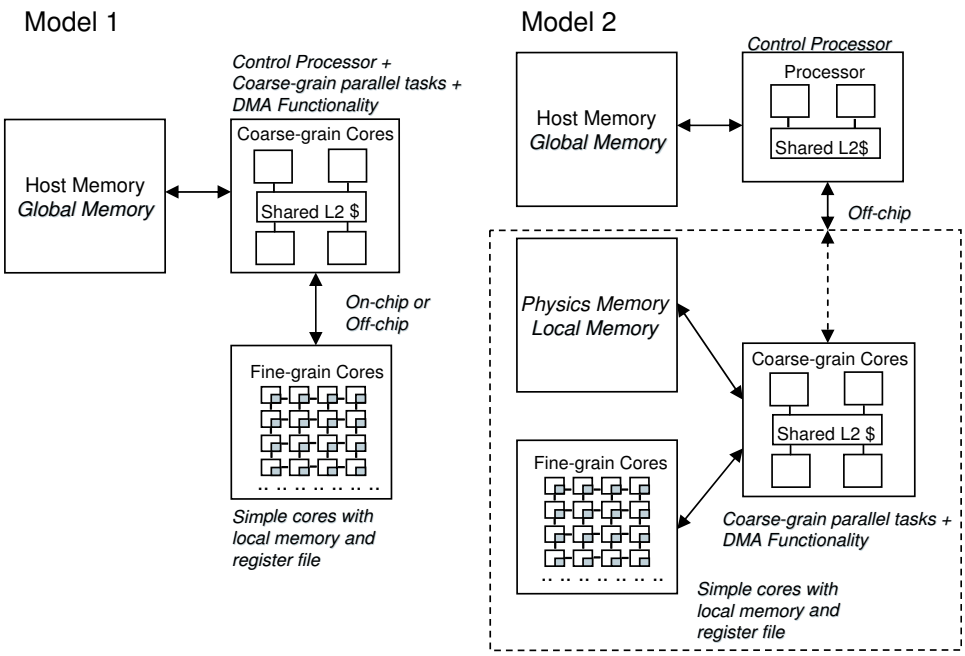


Figure 5.8: ParallAX - Parallel Physics Accelerator.

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

These granularity 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 granularity strike a good balance between the lowest level of parallelism (ILP) vs. completely decoupled parallelism (TLP). We refer the reader back to Table 5.4 and Figure 5.11 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 compu-

tation. 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 [MGR05], physics computation is inherently different from graphics where little pipelining of the phases described in section 5.2.1 can occur. There is additional overhead involved in pushing the data to the FG cores and retrieving the data back after computation completes.

5.2.4.1 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 of these sets 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 tasks to map to FG cores, 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.

We plan to study in detail the load balancing, dynamic work allocation, task stealing (provided by languages like Cilk), etc. in follow up studies.

5.2.4.2 Communication Latency Between Coarse-Grain and Fine-Grain Cores

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 sufficient application parallelism to overlap communication and computation. Looser coupling of cores allows the use of less expensive interconnect and the more flexible placement of cores (i.e. off-chip). And looser coupling facilitates the flexible mapping of FG cores to CG cores.

Our goal is to completely overlap all communication latency with computation except for the initial and final set of communications from CG to FG cores (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 we assume the same for FG cores. The 2D mesh combines simplicity, area effective design, and power efficiency. Its latency and power consumption are slightly worse than a 2D torus, but

its design simplicity should translate 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.

5.2.4.3 Programming Model

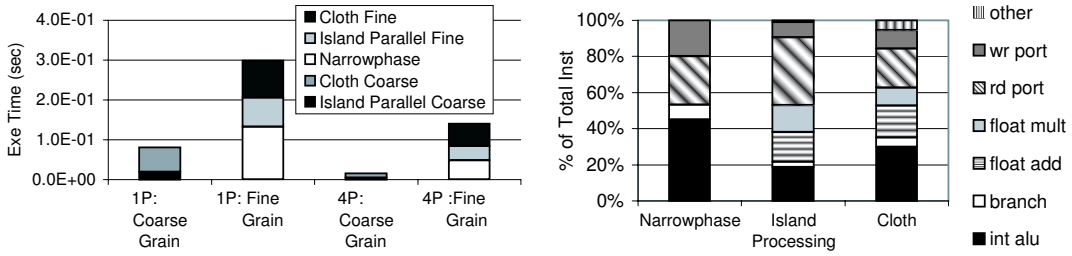
The orchestration of the FG cores need not require ISA modifications. The memory locations of instructions and data structure need to be remapped into the local memory space of the FG cores, and FG kernel functions need to be inlined. All of this can leverage existing compilers by adding a new back-end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

Additional code to do data packing (before sending data from FG to CG cores) and data scattering (before sending data from CG to FG cores) is also be 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.

5.2.5 Architectural 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 of PhysicsBench 2.0, including memory requirements.



(a) (b)

Figure 5.9: (a) Coarse-grain vs Fine-grain Execution Time. — (b) Instruction Mix of Fine-grain Kernels.

5.2.5.1 Characterizing PhysicsBench 2.0 Fine-Grain Computation

Figure 5.9 (a) shows the breakdown of *Mix*'s execution time into serial, CG parallel, and FG parallel components. The first set of two 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.

PhysicsBench 2.0 Kernel Characterization Figure 5.9 (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 square-root instructions.

Memory Required for Instruction 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*.)

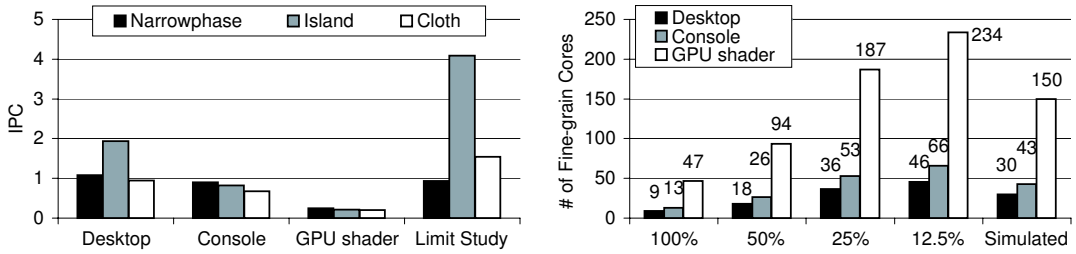
For data memory, we statistically sampled the total unique data read from memory in each kernel for 100 iterations – we found this to be 1,668B for *NarrowPhase*, 604B

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 instruction and data memory reuse exploration, the final FG core design contains a total of 8KB local memory.

5.2.5.2 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 5.6 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.



(a) (b)

Figure 5.10: (a) IPC of Different Fine-grain Core Types. — (b) Number of Fine-grain Cores Required per Type to Achieve 30 FPS.

Desktop	Design based on Intel Core Duo core with 32-entry Instruction Window, 96-entry Reorder Buffer, 17KB YAGS branch predictor, 4-wide 14-cycle pipeline.
Console	Design based on IBM Cell's core with 8-entry Instruction Window, 32-entry Reorder Buffer, 17KB YAGS branch predictor, 2-wide 12-cycle pipeline.
GPU Shader	Design based on GPU shaders with 1-entry Instruction Window, 32-entry Reorder Buffer, 1KB YAGS branch predictor, 1-wide 8-cycle pipeline.
Limit Study	Unrealistic core with 128-entry Instruction Window, 512-entry Reorder Buffer, 64KB YAGS branch predictor, 128-wide 14-cycle pipeline.

Table 5.6: Our Fine-Grain Core Designs.

Figure 5.10 (a) shows the performance, measured in IPC, on the three kernels. *Island* and *Cloth* have bursty amounts of ILP, as confirmed by their source code and the drastic decrease in IPC from the desktop-class to the console-class cores. The limit study core 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.

Number of Finegrain Cores Required Using the average IPC data from Figure 5.10(a) along with the total number of instructions in FG computation, we show the number of cores required for each design to reach 30 FPS for the most demanding benchmark, *Mix*. This calculation assumes 100% utilization of FG cores during each of the parallel sections, which we address in the next section. We also assume that we can send enough tasks to the FG cores to effectively hide any communication latency,

except the startup and post-process communication costs between CG to FG cores.

The first four sets of data in Figure 5.10(b) show the requirement if a given % of the total frame time is available for FG computation. Our four-core CG simulation results (Figure 5.9(a)) left 32% of the frame's 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 [ST], IBM Cell [Hof05], and Nvidia G80 [Som], we derive area estimates for each core type using 90nm technology. The required interconnect area is derived from Table III of [SEW06]. The area estimates for 30 desktop, 43 console, and 150 shader cores are 1388 mm^2, 926 mm^2, and 591 mm^2 respectively. This argues for the simplest cores as the most area-efficient alternative and points to a severe need of area optimization, which will be one 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 (due to more required cores) than an architecture that

can more flexibly and efficiently leverage core resources.

Available 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 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 5.7 shows the amount of parallel FG tasks required to hide communication latency for different core types and interconnect technologies.

(Narrowphase, Island Processing, 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 5.7: Number of Fine-Grain Tasks Required to Hide Communication.

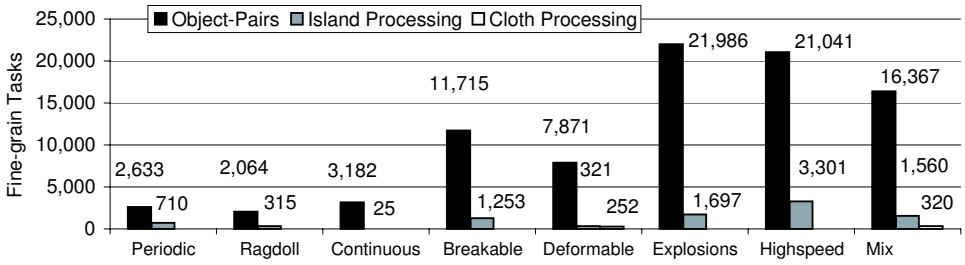


Figure 5.11: Average Number of Available Fine-grain Parallel Tasks.

With four threads, Figure 5.11 shows all benchmarks contain enough parallel FG tasks to hide on-chip interconnect latency to the number of FG cores indicated on Figure 5.10 (b) except *Island Processing* for *Continuous* and *Deformable*, and *Cloth* for *Deformable*. Both benchmarks contain no large islands with more than 25 parallel

FG tasks. Figure 5.6 (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.

5.2.5.3 Implementation Alternatives

As described in section 3, our design is within the space of streaming computation. Model 1 in Figure 5.8 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 granularity 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 5.8 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 5.2.3. 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 [LMT04], 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. 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 have a sound attached to it or may alert an AI character.

5.2.6 Summary

We have proposed ParallAX, an architecture that features aggressive coarse-grain cores with sufficient, partitioned, cache space to handle both the serial and coarse-grain parallel components of physics simulation – combined with a set of fine-grain cores to exploit the massive fine-grain parallelism available for certain components of the computation. Fine-grain cores should be flexibly mapped to coarse-grain 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.

5.3 Performance-Driven Adaptive Sharing Cache

5.3.1 Introduction and Motivation

Chip Multi-Processors (CMPs) [ONH96b] make efficient use of a growing transistor budget by placing multiple processor cores on die to exploit thread level parallelism. High performance embedded processors in different application domains such as Sun's Niagara [P], IBM's Xenon [Sto], and Broadcom's BCM1480 [Bro], and IBM's Cell [Hof05] have lead CMP development with increasing number of cores on-die. However, the CMP architecture is fundamentally pin limited with respect to the optimal number of on-die cores as shown by Huh et al [HBK01].

This limitation exposes the impact of the lowest level on-chip cache on per-thread as well as total throughput. There are two main issues to resolve. The first is satisfying the aggregate demand for cache bandwidth and cache capacity. The second is dealing with inter-thread interference.

Cache thrashing due to inter-thread interference can result in simultaneously active threads degrading each other's performance. Conventional cache implementations can not adequately cope with this interference as shown by Hily and Seznec in [HS98]. To complicate things, cache requirement changes dynamically with phase transitions, context switches, power saving features, and assignments to asymmetric cores. This non-deterministic side-effect of resource sharing is especially problematic for embedded applications requiring certain quality-of-service.

One approach to mitigating the impact of limited off-chip bandwidth is to scale the size of a physically shared L2 to meet the demands of an increasing number of cores on a die. While a large shared L2 provides flexible use of storage, latency and power consumption worsens, particularly with aggressive porting or banking in the face of poor wire scaling [AHK00].

At the other extreme, private caches can provide more bandwidth and reduced latency at the cost of lower utilization. Although prior work [Mic04] enables sharing of private resources, this scheme is only applicable with one active thread on the entire chip.

A compromise between the two previous organizations, non-uniform access cache (NUCA) [KBK02] is one approach to scaling large caches. However, upstream migrations have been shown to be essential to prior NUCA's performance. With multiple threads competing for space, migrations put a great deal of pressure on internal cache bandwidth as well as increasing cache power consumption.

In this work, our goal is to provide data-stream quality of service for all active cores on a CMP. This translates to maximizing overall throughput while guaranteeing fair progress to threads competing for shared resources, namely L2 space, L2 bandwidth, and memory bandwidth. Applications with lower degrees of parallelism and larger memory footprints should not be unduly impacted by the partitioning scheme.

To this end, we physically distribute shared cache resources such that different cache clusters are directly connected to different cores, similar to physically private caches. This organization scales cache bandwidth with increasing number of cores, and the smaller independent cache clusters provide low access latencies and low power.

On top of this design, we explore a dynamic adaptive allocation scheme to handle varying application loads and sudden changes in thread composition. Cache footprints vary both within an application and across different applications. Our approach pinpoints each thread's minimum allocation for acceptable performance and the additional space required for significant improvement at a phase granularity. This data allows precise management of all cache resource to achieve our goal.

All prior work in cache partitioning for CMPs have assumed a physically shared cache. Miss-rate has been the dominant parameter for tuning, and internal migrations

have not been taken into account. It is the performance of a given application, and not the relative cache miss rate that should guide any policy considering fairness.

This work makes the following contributions:

- Distributed NUCA L2 design for CMPs.
- Realizable, on-line strategy for distributed cache partitioning based on actual performance.
- Bounds per-core degradation while optimizing throughput for CMPs.
- Handles dynamic performance changing events.
- Comparison to prior work in terms of performance, cache migrations, and miss rate.

The remainder of the section is organized as follows. We first review related work in this area. Our architecture is described, then the simulation methodology and results are presented. Finally, we conclude.

5.3.2 Related Work

In this Section, we consider prior work on CMPs, NUCA caches, and cache partitioning related to our research.

To increase transistor utilization and reduce inter-core communication, CMP architectures such as the HYDRA project [ONH96b], IBM's Xenon [Sto], and Broadcom's BCM1480 [Bro] communicate through a shared L2 cache. Piranha [BGM00] and Sun's Niagara [P] are examples of the logically shared, physically interleaved L2 cache. Cyclops [CCC02] is a software-controlled, distributed L2 cache targeting array-based computation. With memory space constrained to embedded DRAMs, Cyclops encodes the cache location of logical address in unused address bits. Liu et al. [LSK04] partition the L2 cache by core ID instead of memory address. These partitions are physical banks of the shared cache. The sequential lookups of local and remote partitions increase both hit and miss latencies. All of these designs differ from the independent, hardware-controlled cache clusters of PDAS.

Pursuing higher cache utilization, Michaud proposes execution migration to effectively share private L2 caches on a CMP [Mic04]. The proposed method is active when there is only 1 active thread. With PDAS, we enable all active threads to utilize any available resources by connecting the private L2 caches.

Most recently, Beckmann and Wood examined the applicability of TLC in the CMP environment [BW04]. Similar to our distributed L2 design, [BW04] shows a NUCA design for the 8-core CMP in the study. This paper concludes with a hybrid approach combining prefetching, TLC, and migration to tolerate wire latency for CMP caches. This work examined parallel workloads sharing data between threads. The multi-programmed workload we examine poses more pressure on the memory hierarchy. In addition, both inter-thread interference and internal migrations are not addressed.

To mitigate wire delay impact on access time [Boh95], DNUCA [KBK02] pioneered a non-uniform L2 cache architecture to provide varying access latencies for different parts of a single large cache. NuRAPID [CPV03] improved upon DNUCA by significantly reducing the number of migrations. [CPV03] shows drastic power savings and internal network simplification as a result of migration reduction. PDAS contributes comparable migration reduction for multi-threaded scenarios.

While DNUCA and NuRAPID focus only on the performance and power running a single application, we target an architecture that optimizes performance and power for both single and multi-threaded scenarios.

Cache partitioning has been used to mitigate inter-thread interference. Recent work [SDR02, SRD04] uses marginal gain calculations, expected reduction in miss-rate, to allocate cache space. Ranganathan et al. [RAJ00a] propose reconfigurable caches that can be divided and allocated to different processor activities. Liu et al. [LSK04] describe a cache partitioning scheme targeting off-chip access reduction for parallel benchmarks. No control of the partition scheme was presented. Each repartition requires a system call to the OS. Upstream migrations are enabled as with DNUCA and NuRAPID designs. Kim et al.[KCS04] partitioned a conventional shared cache for CMPs using different fairness metrics based on cache miss behavior. The best dynamic algorithm requires a static profile of each thread's entire execution. Existing cache partitioning work only uses the cache miss behavior to guide allocation. As will be discussed , miss rate does not always correlate well with actual performance.

Our one-pass L2 miss-rate measuring mechanism is related to prior work on one-pass cache simulation such as [SA95] or [LNM04]. The simulation methodology in [SA95] is emulated in hardware by [SDR02] to estimate multiple marginal gains in miss rate. PDAS differs from previous work by using accurate, on-line measured L2 miss rates.

Our cluster level associative partitioning is similar to [Alb99] and [CJD00]. While [Alb99] disables certain ways to save power and [CJD00] partitions the ways to prevent inter-thread collisions, PDAS partitions the cache to achieve both.

5.3.3 Distributed L2 Cache

The lowest-level cache in a CMP architecture should provide flexible storage for threads with different memory requirements, high bandwidth to support requests of simultaneously active cores, and low latency for high performance.

A simple approach, the *private* cache architecture, associates a separate L2 cache with each core. Each core only checks its local cache for hits and snoops other caches for shared data. Private L2 caches provide dedicated ports for each core with low latency and low power, but they lack the flexibility of shared cache resources. However, private caches are fair in that every core gets the same amount of cache space. We will use the private cache as our baseline in this work.

A single large conventional cache can be dynamically shared among cores, but can exhibit long access latency when multi-porting or banking to provide bandwidth. NUCA designs provide a gradient of access latencies based on physical locality to address this problem. On cache hits, cache lines are migrated to lower latency portions of the cache to provide faster hits on lines that exhibit good locality. If multiple cores share a common interface to a NUCA cache, it will be difficult to scale the cache bandwidth. With competing threads, the lower latency portions will experience significant thrashing.

5.3.3.1 PDAS

To address both cache bandwidth and resource interference in multi-threaded workloads, we propose the *Performance-Driven Adaptive Sharing* (PDAS) cache architecture. PDAS consists of an intelligent dynamic partition controller and a distributed cluster of caches as shown in Figure 5.12. We will discuss physical design details first, then consider the partitioning strategy.

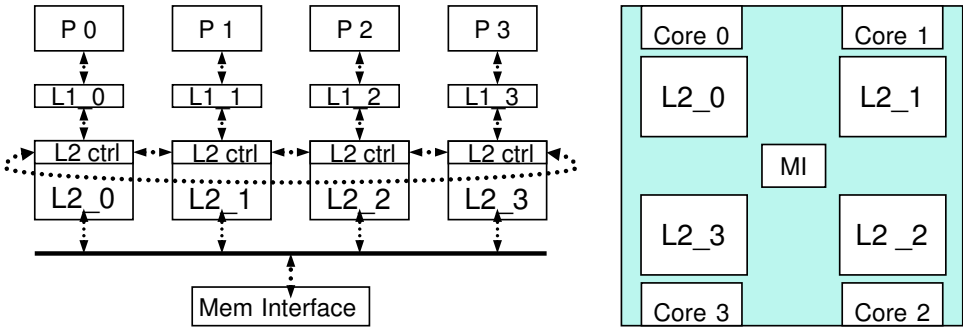


Figure 5.12: The Proposed PDAS CMP Memory Hierarchy and High Level Floorplan.

Each core has an L2 cache physically associated with it. Unlike the private cache, any core can access any of the L2 caches associated with other cores. From the point of view of each core, the distributed caches form a non-uniform cache hierarchy. Access latency depends on the distance between the requesting core and destination cache – Figure 5.12 shows that L2_0's access latency from core 0 differs from that from core 2 due to the physical layout.

Each PDAS cache cluster is hardware-controlled and fully independent with its own tag array. The closest cache to each core is assigned to the core as its primary cache. By forcing cores to access their primary caches first, each processor core sees the fastest access time for the most recently used data. In addition, interference between newly loaded data from all active threads is completely removed.

PDAS leverages CMP's flexibility in physical placement to closely connect these L2 caches. Each cache cluster communicates to its nearest two neighbors through the interconnected L2 controllers. This forms a bidirectional ring across all cache resources.

5.3.3.2 PDAS Access

Since data could reside in any cache cluster, all distributed tag arrays must be checked before determining a real L2 miss. This operation effectively implements the snooping mechanism across conventional private caches. While this work focuses on multi-programmed workloads, the underlying architecture can be utilized for cooperative multi-threading. Shared data may not be duplicated with PDAS, and inter-thread interference is removed.

Each request is initially forwarded to the two nearest neighbors of the originating core. These cores in turn forward it to their neighbor along the same direction. The originating core location is used to tag the request, and this tag persists with the request as it traverses the network to allow search termination. We effectively implement a serial tag-data access that is commonly used with large caches for energy efficiency.

Remote requests to a particular cache are serviced with the same priority as local tag lookups. If there is a hit in the cache for a remote core, data is sent back through the interconnect between L2 controllers.

Each memory reference generates requests to both neighbors as soon as it misses in the L1. This remote request is forwarded to the farthest neighbor in parallel with tag accesses. The initial memory reference creates an entry in the memory interface (MI) with 2 bit-vectors. Each bit-vector has length n, where n is the number of PDAS cache clusters. Bit-vector 1 indicates the hit/miss status, and bit-vector 0 indicates valid results. If any tag lookup hits, it invalidates the entry in the MI. If any tag lookup misses, it sets its corresponding bit for both the hit/miss and the valid bit-vectors. Only valid MI entries with all bits set in both bit-vectors are issued to memory.

Invalidations from the memory bus are also routed to all tag arrays in the same manner.

If a core is constrained to its primary cache, PDAS allow its primary cache misses to be issued as soon as possible. The tag checks still occur for coherency. If a hit occurs after the memory request has been sent, the request will be dropped upon return and data is forwarded as any other requests. This optimization is important for applications that are sensitive to memory latency, but do not have a large L2 cache footprint.

5.3.3.3 Cache Line Migration

From the perspective of a single thread, the cache line migration policy would keep the most recently used blocks physically close. Cache lines that move among clusters to maintain LRU ordering are said to migrate between caches. Clusters physically closer to the executing core are considered *upstream* in the migration order when compared to clusters that are further away from the core (i.e. *downstream* caches). One approach to handling migration is to force evicted lines from a given cluster downstream (i.e. physically away from a given core). We will assume this simple greedy downstream strategy in this work, as it provides the best single thread performance among the strategies we explored.

Previously described mechanisms were chosen for a simple, fast implementation of this distributed cache. In depth analysis on the communication strategy is beyond the scope of this work.

5.3.4 Limiting Data Migration Among Clusters

Given the physical distributed cache design, inter-thread thrashing effects still need to be addressed to bound per-core performance degradation. While memory intensive application can starve other threads from keeping state in the L2 cache, contention and migration congest the interconnect that allows cache lines to migrate.

To ensure fairness among threads, we aim to guarantee a minimal level of performance to each thread. While we tune based on known application characteristics, a natural lower bound for evaluation is the private cache. This approach is the simplest fair share policy – all threads have the same amount of space with the same relative latency.

Assuming equal priority, we would like to maximize the throughput achieved by our CMP architecture after ensuring the per-core minimums. Not every application needs all of its primary cache to achieve the minimum performance. The remaining space after ensuring per-core performance should be allocated to increase the overall throughput.

Each application has some minimum working set [DS02] that is required for a certain performance level during a given application phase. It is not always possible to satisfy all applications' working sets for their current phase, and the goal is to make intelligent allocations.

5.3.4.1 Cache Partitioning

We will explore cache partitioning [STW92] as one possible approach to this. Our cache partitioning approach needs to accomplish the following: 1) Guarantee a minimum level of performance per thread. 2) Maximize achievable throughput by intelligent allocation. 3) Minimize migrations per memory access. 4) Operate at phase

rather than application granularity to better tune the resources to application demand. This is a very difficult set of constraints to solve statically – Sherwood et al. [SSC03] demonstrated that cache behavior can vary significantly across application phases. 5) Handle arbitrary threads running together at any particular phase. 6) Efficiently adapt to dynamic events such as new threads, phase changes, etc. without undue impact. 7) Low complexity. To reduce inter-core communication, we would like the scheme to dynamically partition cache resources with little interaction and coordinated arbitration.

In this work, we assume an 8-way 1MB cache at each cluster with a way selection mechanism, similar to the one found in [CJD00] and [Alb99]. We partition the cache into 256KB chunks – allowing each pair of ways to be assigned. Other cache and partition sizes are feasible in future studies.

We further assume that cache partitioning is done every 100 million cycles by our global arbitration unit. The partitioning is based on the demand of each active thread in the CMP. Conservatively, we allocate 10,000 of these cycles (0.01%) to the arbitration unit to perform any intelligent decisions and reconfigure the caches. The remaining 99.99% of the 100 million cycle interval is used to gather statistics for our arbitration unit.

As shown in [SPH02], application phases for the SPEC2K suite are on the order of 10's of billions of instructions, which takes at least 2.5 billion cycles with an ideal IPC of 4. By partitioning at the much shorter granularity of 100 million cycles, we avoid frequent measurements/reconfigurations.

5.3.4.2 Partitioning Algorithm

We first describe our generalized partitioning algorithm that optimizes for performance. We will later describe two different approaches to approximate this perfor-

mance.

At a high level, the partitioning process can be described by four steps. First, each core executes with different amount of private space to find the minimum space required for acceptable performance. Second, each core measures its miss rates using different sizes of the cache. Third, all cores arbitrate using the data obtained in steps one and two. Finally, all cores are allocated cache space. All threads' minimum cache resources for acceptable performance are allocated first. Then, the remaining resources are given to threads which can reach their minimum requirement for improvement in a greedy fashion.

With our granularity of 256KB and a total cache space of 4MB, there will be 16 different cache partitionings that are possible for each thread. Each core will *independently* determine five values from these 16 possible partitions:

Guaranteed Partition The minimum cache partitioning required to achieve within Gthresh % of the performance of a 1MB 8-way set associative cache. This is the performance of the private cache that we would like to guarantee to every application.

Minimum Extra Partition The minimum cache partitioning beyond 1MB where the performance improves beyond Ithresh %. This is the additional cache space we need to provide measurable improvement to the application. If no such partitioning exists, this is set to the Guaranteed Partition.

Maximum Extra Partition The maximum cache partition that continues to provide at least an Athresh % cumulative performance improvement over the Minimum Extra Partition. This is the most cache space out of the entire 4MB that we would allocate this thread to obtain within Athresh % of the benefit of the entire 4MB cache. Within the architectural space studied here, any further space would

not impact performance. If no such partitioning exists, this is set to the max of Guaranteed Partition and Minimum Extra Partition.

Expected Min Gain The expected performance improvement of the Minimum Extra Partition over the Guaranteed Partition.

Expected Max Gain The expected performance improvement of the Maximum Extra Partition over the Minimum Extra Partition.

All thresholds (Gthresh, Ithresh, and Athresh) can be tuned by the designer based on acceptable degradation and perceived significant improvement. For example, one can increase Gthresh to increase the potential for free space used to increase overall throughput. In this study, we set all thresholds to 3% for all simulations. This value was selected to co-optimize fairness and throughput.

If the Maximum Extra Partition is the same as the Guaranteed Partition, the application does not benefit with more cache state (at least up to the available 4MB) and should not receive any additional cache space. If the Minimum Extra Partition is not the same as the Maximum Extra Partition, then there is likely some benefit to giving space between the Minimum Extra Partition and Maximum Extra Partition.

These five values from each thread are all that will be sent to the global arbitration unit, which can reside in the Memory Interface. The unit will always provide at least the thread's Guaranteed Partition. This approach provides fairness by bounding per-core degradation. The remaining allocation will be made to maximize the throughput of the processor. This allocation problem is actually NP-Complete and can be reduced to the knapsack problem [CLR92]. We adopt the following heuristic to approximate the solution. The specific cache-way allocations are done with physical locality in mind based on core ID (not shown).

Variables:

```
Total_Space = number of available distributed L2 cache partitions.
Allocated_Space[i] = each thread i's number of allocated partitions.
Under_Consideration[i] and Under_Consideration_Cost[i] are possible
allocation moves for each thread – benefit and space required.
Guaranteed_Partition[i], Minimum_Extra_Partition[i],
Maximum_Extra_Partition[i], Expected_Min_Gain[i],
and Expected_Max_Gain[i] are as described above.

Total_Space = 16;      # 16 different chunks of space to allocate

 # First allocate the guaranteed partitions
for (i=0; i<num_threads; i++) {
        Total_Space -= Guaranteed_Partition[i];
        Allocated_Space[i] += Guaranteed_Partition[i];
        Under_Consideration[i] = Expected_Min_Gain[i];
        Under_Consideration_Cost[i] = Minimum_Extra_Partition[i];
}

 # Allocate remaining space to thread with highest potential gain
while (1) {
        Find thread i with greatest nonzero Under_Consideration[i]
        and Under_Consideration_Cost[i] < Total_Space;
            if no such i exists, break;
        Total_Space -= Under_Consideration_Cost[i];
        Allocated_Space[i] += Under_Consideration_Cost[i];
```

```
    if (Allocated_Space[i] == Maximum_Extra_Partition[i]) {
        Under_Consideration[i]=0;
    } else {
        Under_Consideration[i]=Expected_Max_Gain[i];
        if (Maximum_Extra_Partitioning[i]-
        Minimum_Extra_Partitioning[i] >
        Total_Space)
            Under_Consideration_Cost[i]=Total_Space;
        else
            Under_Consideration_Cost[i]=
            Maximum_Extra_Partitioning[i]-
            Minimum_Extra_Partitioning[i];
    }
}
```

Our partition strategy differs from all other partition work in that it is optimized for a distributed cache, based on performance improvement, minimizes migrations, and adapts to dynamic performance changing events. By reducing migrations, PDAS's partition strategy naturally aims for the best performance/power configuration. Furthermore, PDAS's allocation copes with the distributed nature of our LRU state.

5.3.4.3 Miss Rate

Our first dynamic partitioning approach uses miss rate as the metric to guide the allocation of cache space – miss rate will be the sole metric of performance to guide our algorithm.

To dynamically capture the potential miss rate at different partitions, we make use of an additional 4MB L2 tag array at each core, which we will refer to as the *scratch pad array*. The scratch pad array has no corresponding data component, and is *only* used for estimating the reduction in miss rate through the LRU stack. All L2 requests from the core use the additional scratch pad array as well as the distributed L2 cache, updating the array based on the address request stream. Cores only access their own scratch pad array – the state contained therein can be different from what is in the distributed L2 cache. There are sixteen counters associated with the 4MB 32-way tag array, one counter for each possible partition. Each counter tracks the hits that would have been seen by a given partitioning strategy. For example, the 1MB 8-way partition counter would track the hits to the first 8 ways of the 4MB 32-way tag array. The 1.25MB 10-way partition counter would track the hits to the first 10 ways of the tag array. We can provide the expected miss rate from using any partition.

CACTI 3.0 [SJ01] indicates that the structure adds approximately 10% overhead to the overall cache area. This overhead increases as the number of cores/clusters scales, but does not impact the access time to each cache. The use of partial tags with half of the address bits reduces the area impact to 5% of the overall cache area – similar to Nurapid's pointer overhead. To scale the number of cache clusters, this structure can be shared between multiple cores that take turns to explore. Another alternative would be approximating with prior marginal gains work.

5.3.4.4 Weighted Miss Rate

Our second approach is a realistic heuristic that factors IPC impact into the first miss rate approach.

Prior work [SDR02] has suggested that measuring the L2 miss rate is sufficient to determine whether or not sharing further cache resources would be beneficial. How-

ever, this approach alone can miss some opportunities for performance or power savings.

In [SDR02], the cache partition scheme is guided by a greedy policy using marginal gain. Marginal gain gives an indication of the miss rate reduction for each line allocated to a given thread. From our simulations, several benchmarks pose problems for miss rate based methods. For example,*gcc* improves IPC by 17% with a miss rate reduction of only 3%. Programs like *lucas* retain the same IPC even as the miss rate is reduced by 18%. Finally, benchmarks like *mcf* require a minimum amount of cache space before improving either miss rate or IPC. L2 miss rate based methods cannot capture this range of behavior, when miss rate and performance are not strongly correlated.

This problem becomes more evident with heterogeneous cores on a CMP. As each core can have different resources to tolerate memory latency, the impact of miss rate on a given core can be radically different. Power and thermal throttling done at a per core granularity will also affect the impact of miss rate on performance.

Rather than exploring the entire space of possible partitions (as would be required for an oracle IPC approach), we measure the IPC difference between two points in the partitioning space and use that to weight the complete exploration of miss rate from section 5.3.4.3.

Each thread will have two performance registers to track IPC. If these registers are cleared, the thread will first be restricted to 256KB of the primary cache (the remainder of the cache can still be available to other threads) and the IPC for that cache partitioning interval will be recorded in the first performance registers. In the next partitioning interval, the thread will be restricted to 1MB of the primary cache and this interval's IPC will be recorded in the second performance registers.

[SSC03] observe that application phases are predictable and exhibit excellent lo-

cality. Although this work used the SPEC2K suite, some embedded applications will show even more regularity in behavior. We propose the use of phase detection and tracking hardware from [SSC03] to determine when we are entering a different phase. We will have a small hardware structure to track performance register values for different phases. If a thread enters a new phase that it has not seen before, its performance registers are cleared and exploration will begin for the next two partitioning intervals. If a thread enters a previously seen phase and the performance register values are stored, we will simply restore these values into the performance registers.

These two performance registers are used to determine the IPC improvement from 256KB to 1MB. This value, along with the change in miss rate from 256KB to 1MB, allows us to form a ratio of IPC improvement per L2 miss rate reduction. If the transition from 256KB to 1MB sees less than a 3% improvement in IPC, a default ratio of 3/24 is used. This approach promotes fairness for applications which see no benefit until a certain cache size is provided (e.g. *mcf*). The default ratio implies that we expect a 3% performance gain if the miss rate can be reduced by 24%, a conservative policy that avoids overzealous sharing. The 3% and 24% values chosen for this study could be tuned to the specific application target mix of a given processor.

It is possible that the 256KB to 1MB transition is not representative of the performance jump beyond 1MB. In these cases, the performance registers are adjusted to reflect the correct potential gain seen through the actual partitioning. This refinement may take a few partitioning intervals to completely settle, but the feedback loop provides better correlation with actual performance gain. This allows the above default ratio to be optimistic without wasting resources.

This ratio estimates the effect of L2 miss rate reduction on IPC for each phase of each thread. To estimate the potential IPC of a given partitioning, we multiply the measured L2 miss rate reduction by this ratio. This has been shown by our data to

accurately predict significant improvement with a threshold of 3%.

The observed performance gain may differ from the expected performance gain for a number of reasons, including a performance drop from power/thermal throttling (i.e. turning off a performance enhancing structure) or an incorrect phase detection. For phase mis-detections, we would clear the two performance registers. This triggers an exploration if the phase is new, or it will just restore the cached performance register values for a known phase. If the phase was correctly detected and the expected performance was not achieved (e.g. due to power throttling), we would clear the two performance registers and the cached performance register values for the current phase, effectively forcing our architecture to explore as if the phase was not seen before.

5.3.5 Example

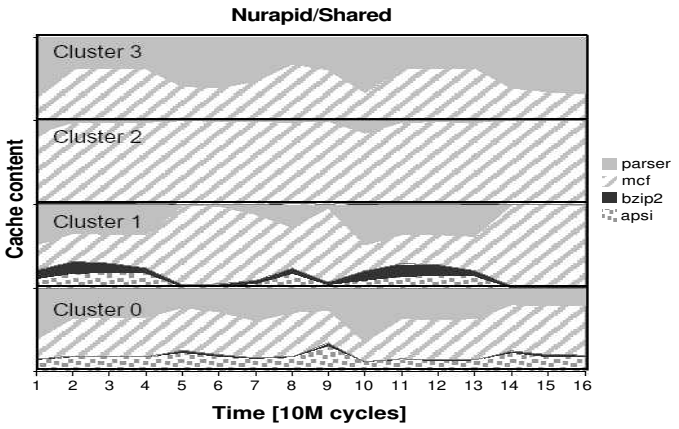


Figure 5.13: Cache content of Nurapid.

To better illustrate the problems we are attacking, we present the following example of running *apsi*, *bzip2*, *mcf*, and *parser* together on both PDAS and Nurapid. Our full simulation methodology is presented in Section 5.3.6. For their current phase of execution, the benchmarks want 768KB, 256KB, 3.25MB, and 512KB respectively to attain close to maximal performance (i.e. if they had all 4MB of L2 cache). Clearly, there is no way to satisfy all of these given the total amount of cache available. In this case, all applications but *mcf* are given their fair share, and *mcf* is relegated to the remaining space – and effectively prevented from disrupting the performance of the other applications.

Figures 5.13 and 5.14 show the cache contents over time (in 10 million cycle increments) for Nurapid and PDAs respectively. In this example, *apsi* goes through a phase transition, and will go from wanting 768KB of L2 cache to wanting 256KB of L2 cache. The four L2 caches are labeled "Cluster 0" through "Cluster 3" and correspond to the cores running *apsi*, *bzip2*, *mcf*, and *parser* in that order.

The most apparent differences between the two cache content graphs are the sta-

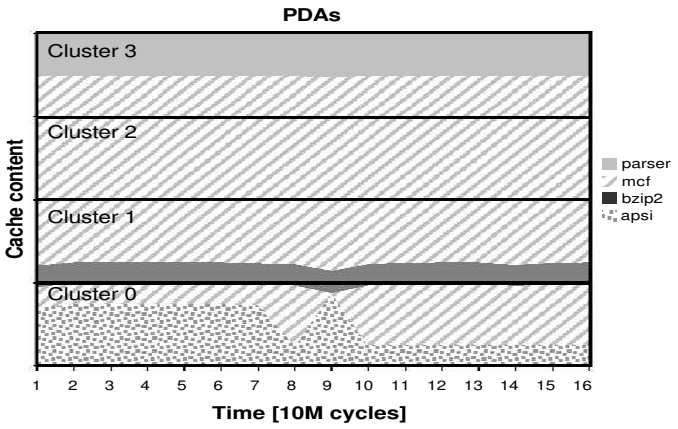


Figure 5.14: Cache content of PDAS.

bility, location, and amount of each thread's data. With Nurapid, we see that *apsi*, *mcf*, and *parser* spread data across different regions of the cache. For all applications, we see constant fluctuations in each cluster as threads contend for space with different L2 miss frequencies.

Note the exploration that *apsi* performs to determine how much cache space to use in the second phase. We assume that this is the first time this phase has been seen by *apsi* - and therefore at time interval 8 *apsi* begins its exploration by first using only 256KB of space in its local L2 cluster. This provides an opportunity for *mcf* to grab more cache space in Cluster 0 during that interval. In the next interval, *apsi* explores using the full 1MB of its local cluster. After these two exploration intervals, and using the miss rate data gathered with its replicated 4MB tag array, *apsi* concludes that it only needs 256KB of space for this phase. Note that the other applications did not need to participate in this exploration, but only saw the impact from the global arbiter granting or taking away space during *apsi's* exploration.

Figure 5.15 presents weighted IPC for both PDAS and Nurapid as they proceed through the 16 time intervals. For PDAS, all applications perform better than 97%

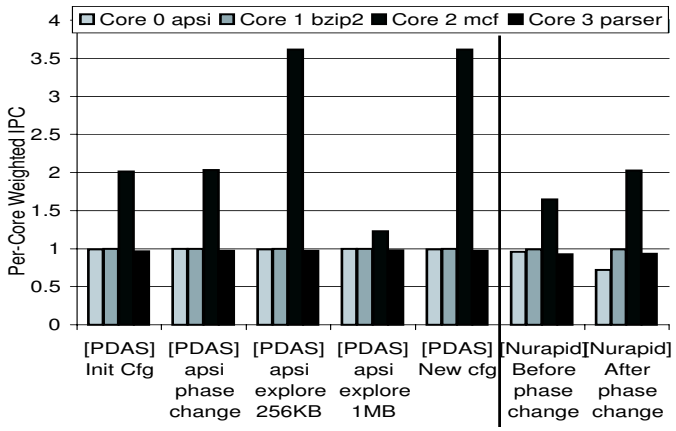


Figure 5.15: Per-core IPC weighted by ST IPC with 1MB cache.

of their single thread IPCs to meet our goal of bounding per-core degradation. This behavior is consistent in all of our simulations, but space constraint prevents us from showing such details in the results section. This ability to bound per-core degradation is feasible for CMPs sharing the L2 cache, but may be more complicated for SMT processors that share execution as well as cache resources.

The first cluster of four bars is the initial configuration of PDAS. The second cluster is the same configuration at the time of the phase change in *apsi*. In this case, *apsi* simply has more space than it needs. The next two clusters detail the IPC during the two intervals of exploration. Note the improvement seen by *mcf* in the first exploration interval. The next cluster details the eventual redistribution of cache state after exploration. The last two clusters detail the IPC seen by Nurapid before and after the phase change.

Both *apsi* and *bzip2* obtain less space with Nurapid than PDAS. While *bzip2* performs similarly, *apsi* suffers a degradation with Nurapid due to a lack of space and poor data placement in Cluster 1 (which is remote relative to *apsi*'s core placement. With Nurapid, *parser* obtains more cache space overall than the PDAS allocation but

performs worse. Because parser does not need more than 512KB of space, the extra space afforded by Nurapid does not improve performance – however, it does see an improvement in miss rate, which demonstrates the need to temper partitioning decisions with actual IPC estimation.

5.3.6 Methodology

Chip Level Parameters	
CMP	4 cores
Private L2s	4X 1MB, 8-way, 64B blocks, 20 cycle hit, LRU 1 port, seqeuntial access
Shared L2	4MB, 32-way, 64B blocks, 51 cycle hit, LRU 4 ports, sequential access
Nurapid L2	4MB, 32-way, 64B blocks, LRU, 4 ports, 4 1MB d-groups sequential access, Hit latencies 34, 40, 40, 46
Ideal L2	4MB, 32-way, 64B blocks, 20 cycle hit, LRU 4 ports, sequential access
PDAS L2	4X 1MB, 8-way, 64B blocks, LRU, 1 port sequential access, Hit latencies 20, 26, 26, 32

Core Level Parameters	
RUU	512 entries
LSQ	256 entries
Inst Queue	32 entries
Width	8-wide
Branch predictor	16K-entry gshare, 2K-entry 4-way BTB, 64-entry RAS
Functional Units	2 INT ALU, 1 INT Mult, 1 Mem, 1 FP ALU, 1 FP Mult
L1 i-cache	16K, 4-way, 32B blocks, 1 cycle hit, LRU, 1 port
L1 d-cache	16K, 4-way, 32B blocks, 1 cycle hit, LRU, 1 port
Memory latency	150 cycles + 2 cycles per 32 Bytes, 1 shared port
Mispredict penalty	15 cycles
Technology	0.10um
Cycle Time	0.25ns

PDAS-MR	Guarantee	Min	Max	Emin	Emax
art	8	10	24	23	26
facerec	4	12	22	3	15
gcc	4	26	26	3	0
lucas	6	6	6	0	0
mgrid	6	6	6	0	0
parser	6	12	22	4	7
sixtrack	2	12	16	4	4

PDAS-IPC	Guarantee	Min	Max	Emin	Emax
art	6	10	24	9	9
facerec	4	14	24	3	8
gcc	4	24	30	3	3
lucas	2	2	2	0	0
mgrid	2	2	2	0	0
parser	6	12	22	3	7
sixtrack	4	2	2	0	0

Figure 5.16: Processor and PDAS Parameters.

Table 5.16 shows the core level and the chip level parameters for the simulated chip-multiprocessor. We model four processor cores on a chip with 4MB of total L2 cache implemented in 0.10um technology with a target cycle time of 0.25ns.

Table 5.16 shows the five values obtained through our partitioning algorithm for seven of the benchmarks we examined. The top right table shows values for PDAS with miss rate (MR) guiding performance and the lower right table shows values for PDAS with IPC guiding performance. The numbers for Guarantee, Min, and Max indicate the number of cache ways. The numbers for Emin and Emax show the percentage of expected improvement.

Each out-of-order processor core is 6-wide with 16KB 4-way instruction and data caches. We show the parameters of different L2 configurations that we compare

against. All cache latencies are based on CACTI 3.0 [SJ01] and all L2 caches are serial access (tag first, then data) and non-banked. All (except PDAS) are optimistically modeled with 4 dedicated ports while using single-port latencies. Dedicated ports is only realistic for private L2 caches and PDAS, but we simulate four dedicated ports for all configurations for a rigorous comparison. Shared, NuRapid, and Fast Shared see 23 cycles of L2 tag latency. Private and PDAS see 8 cycles of L2 tag latency.

We consider the following L2 cache organizations:

Private: The private cache of section 5.3.3. Each 1MB 8-way L2 is assigned to one core. Contention from snooping across the private caches is not modelled which produces a slighly optimistic baseline.

Shared: A realistic, conventional 4MB 32-way cache shared by all cores. The longer hit latency of this cache reflects the 4X increase in size and associativity. (4 dedicated ports, single-port latency).

Nurapid: Optimal configuration of the non-uniform access L2 cache as presented in [CPV03]. Nurapid was chosen over DNUCA [KBK02] for its improved performance and reduced migrations. We extended Nurapid such that missed references are placed into its primary data array, the closest D-group to the core. Overall, Nurapid contains one shared tag array with 4 ports and 4 1-port D-groups (single-port latency used). For array access latencies, we use the tag access latency of a 4MB 32-way set associative cache and the data access latency of a 1MB 8-way set associative cache. For each core, the primary D-group requires 34 cycles. The one-hop latency between each D-group is 3 cycles, the same as PDAS, and we assume migration latency to be the sum of access and one-hop delays.

Fast Shared: This models an unrealistic conventional 4MB 32-way cache. We set the

hit latency and tag latency to be the same as *Private*, 20 and 7 cycles respectively – effectively providing the low access latency of *Private* with the dynamic resource sharing of *Shared*.

PDAS: Our proposed cache architecture on a fully distributed L2 cache. PDAS contains four independent 1MB 8-way set associative cache clusters connected by request and reply FIFOs. Each cluster is designated as the primary L2 cluster for its nearest processor core. The hit latency depends on the location of hit and request. For each core, the primary cluster requires 20 cycles. The 2 nearby neighbors require 26 cycles, and the farthest takes 32 cycles. The one-way communication delay between neighboring clusters is 3 cycles. On a migration, we accurately model the latencies of reading the evicted line and sending it through the network. We do not allow migrated lines to service any memory request until placed into the destination cluster. We also model contention for each L2 cluster between requests from local and remote sources. The longer access time of *Nurapid* as compared with *PDAS* is due to the 4MB tag access, and the difference would be larger if using multi-porting latencies for Nurapid. The two versions, PDAS-MR and PDAS-IPC, are controlled by using L2 miss rates or estimated IPC gain respectively.

For PDAS, we fully model the contention from remote requests at each cache cluster. The direction of downstream migration for PDAS on the bidirectional ring topology is currently based on a per cycle toggling counter. For Nurapid, we model it in one direction only as in [CPV03].

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [BA97], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions. Simulation is execution-driven, including execution down any speculative path until the detection of a fault,

TLB miss, or branch mis-prediction.

We have made extensive modifications to SimpleScalar, allowing it to support a CMP design with non-cooperative execution threads – each application only uses a single thread context, and all threads have their own virtual memory space. We have modified the memory hierarchy to support Nurapid and PDAS caches, and to support multiple threads.

We used the SPEC2000 suite for evaluation due to the general applicability of our work to different computing domains spanning embedded, desktop, and server applications. Future work will utilize the new benchmarks we are creating for interactive entertainment applications.

We categorized the benchmarks into 2 sets. The categorization is based on whether the application sees measurable performance improvement (3%) with more than 1MB of L2 cache.

The group of 9 applications which see improvement, *High*, includes ammp, art, facerec, galgel, gcc, mcf, parser, twolf, and vpr. The other 15 benchmarks which see insufficient improvement makes up the *Low* group. Multi-threaded tests were generated equally for all permutations of the workloads. 12 2-Thread tests include 4 tests with 2 *High* benchmarks, 4 tests with 1 *High* and 1 *Low*, and 4 tests with 2 *Low*. 12 3-Thread tests include 3 tests with 3 *High* benchmarks, 3 tests with 2 *High* and 1 *Low*, 3 tests with 1 *High* and 2 *Low*, and 3 tests with 3 *Low*. 10 4-Thread tests include 2 tests for each permutation.

Per-thread performance metrics are measured for execution up to a maximum per-thread instruction count of 100 million. All completed threads continue execution past this point while other threads execute. This prevents freeing of resources when certain threads complete earlier than others.

The programs were compiled on a DEC Alpha AXP-21164 processor using the

DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). We model the top execution phases as described in [SSC03]. All benchmarks were simulated using the *ref* inputs.

We execute each phase with no cache warm-up and assume that the cache requirement explorations have already been done for the given application phase. We use the same thresholds throughout all simulations.

5.3.7 Results

In this section, we analyze the simulation results for different cache configurations with one, two, three, and four thread workloads. The configuration labels are described in section 5.3.6. All IPC and migration results for multi-thread runs are ordered by the group permutations described in section 5.3.6. Migrations only apply to Nurapid and PDAS.

For MT tests, the core assignment of applications follows from left to right. For example, ammp-facerec-art-parser is a test with ammp on core 0, facerec on core 1, art on core 2, and parser on core 3.

Because our goals for PDAS include per thread fairness and maximum total throughput, we want to measure the performance impact to all threads. Similar to the weighted IPC used in [ST00], the IPC data for MT runs are obtained by weighting per thread IPCs by that thread's performance using the private 1MB cache.

$$\textit{Weighted Speedup in interval t: } WS(t) = \sum_{i=0}^{\#cores-1} \frac{IPC\_thread_i}{private\_IPC\_thread_i}$$

5.3.7.1 Single Thread Results

Figure 5.17 shows representative samples of the *High* and *Low* groups of applications. The first five applications see benefit with $>$ 1MB of L2, and the second five applications do not. This distinction is clear when comparing the performance of Private and Fast Shared. Both have the same latencies while Fast Shared has 4MB of cache space.

Low applications require low access latency. All five *Low* applications suffer performance loss when comparing Private with Shared or Nurapid. *High* applications require relatively more L2 space. However, the performance gap between Shared, Nurapid, and Fast Shared indicates sensitivity to latency as well.

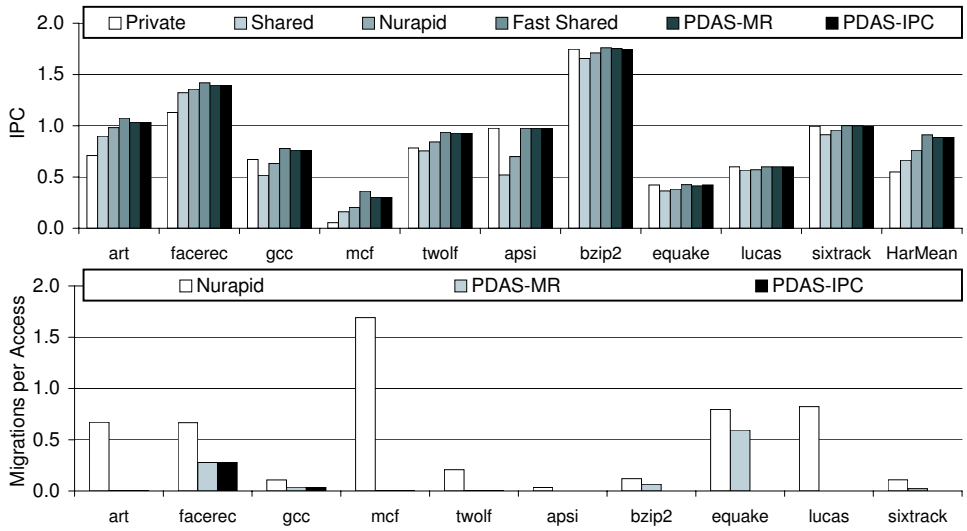


Figure 5.17: Single Thread IPC and Migration per Access. We show the harmonic mean across all benchmarks.

PDAS achieves the best of both latency and capacity, shown by only 1% degradation versus the Fast Shared across all 24 applications. On average, PDAS improves upon Private by 26%, Shared by 27%, and Nurapid by 13%.

When comparing the non-uniform access caches of Nurapid and PDAS, it is important to compare the migrations required for the achieved performance. The magnitude of migrations reflect the internal bandwidth and power requirements. Each downstream migration requires a read and write of a cache line, and each upstream migration requires two reads and two writes.

Even with a single thread, PDAS is able to reduce migrations on average by 82%. This reduction translates to an 82% reduction in data array access as well as traffic on the internal interconnect. Nurapid claims a 77% L2 cache energy reduction from a 61% migration reduction when compared to DNUCA. However, our architecture does require extra tag structures for miss-rate measurements. Future work will more accurately assess this power trade-off.

The migration reduction comes from four sources. First, all demand miss data is brought into the primary cache array for each core (also done for Nurapid). Second, PDAS does not enable upstream migration. This feature accounts for the significant performance boost for DNUCA and NuRAPID (approximately 10% for our NuRAPID implementation) at the cost of substantial migrations. Each upstream migration requires two reads and two writes. The third source of migration reduction is even migration distribution in different directions. Due to the circular connection network, we can distribute migrations in 2 directions. In our current design, directions are the same for all cores and the directions change every clock cycle. Finally, applications that are classified as *Low* because they need more than 4MB of cache space are effectively sequestered to a single cache and their evictions are not migrated all around the distributed caches.

With respect to migrations in PDAS, the LRU stat is distorted as migrated lines become the most recently used line for the new core location. This distortion of the LRU history is further compounded by the fact that PDAS distributes migrations in different directions, as compared with the single static direction of DNUCA and NURAPID.

5.3.7.2 Two Thread Results

Figure 5.18 shows the Weighted Speedup, as described in the beginning of this section, of different cache configurations. On average, PDAS-IPC improves over Private by 26%, Shared by 39%, Nurapid by 30%, Fast Shared by 20% and PDAS-MR by 1%. For *ammp-art* and *art-twolf*, the Fast Shared cache outperforms PDAS. Fast Shared is able to further reduce the miss rate for *ammp* or *twolf* in this case because of the granularity of our partitioning strategy. For *ammp* and *art*, PDAS is able to reduce art's miss-rate from 49% to 1% by allowing it to use remote space. At the same time, the migration/access ratio is reduced to essentially zero from Nurapid's 0.6 migra-

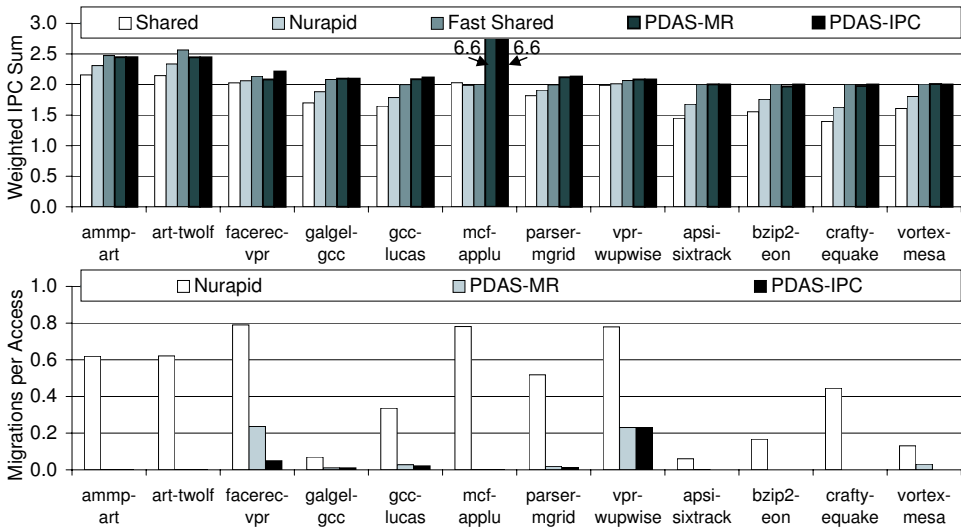


Figure 5.18: 2-Thread Weighted Speedup and Migration per Access.

tions/access.

Similar to the ST IPC results, tests with all *Low* benchmarks show the same performance for Private, Fast Shared, and PDAS while Shared and Nurapid sees degradation. This same behavior is observed in results with more threads.

On average, PDAS-IPC reduces migrations per access by 95% as compared to Nurapid. While the percent reduction stays close to 95% for 3T and 4T tests, the number of cache accesses reduced increases dramatically as more cores become active. While PDAS-MR and PDAS-IPC show similar performance numbers, PDAS-IPC is able to reduce migrations by 27% on average. Since the migration comparisons stay relatively constant with 3T and 4T workloads, we will not show any more migration details from this point.

Table 5.8 shows the L2 miss rate for some of the program mixes we considered, along with data for private 4MB caches (i.e. each core has its own 4MB cache). For *facere-vpr*, PDAS shows the benefit of its intelligent allocation scheme. The miss

Program	Private		Shared		PDAS-IPC		Private 4MB	
Mix	T0	T1	T0	T1	T0	T1	T0	T1
facerec-vpr	0.35	0.35	0.28	0.22	0.17	0.37	0.17	0.13
gcc-lucas	0.05	0.33	0.05	0.33	0.03	0.51	0.02	0.33
parser-mgrid	0.16	0.20	0.18	0.20	0.05	0.37	0.05	0.20

Table 5.8: Selected 2-Thread L2 Miss Rates.

rate of *vpr* is marginally increased to enable a much higher reduction in *facerec*'s miss rate. *Vpr* gains very little performance with increased cache space, while facerec sees substantial raw IPC gains. This is all done while guaranteeing *vpr*'s performance does not degrade beyond the performance threshold we introduced in section 5.3.4. This partitioning cannot be obtained by prior work that either balances the effect of sharing on miss rates or marginal gain. This further explains how PDAS-IPC is able to outperform PDAS-MR for *facerec-vpr*. Similar behavior can be observed for *gcc-lucas* and *parser-mgrid*.

5.3.7.3 Three and Four Thread Results

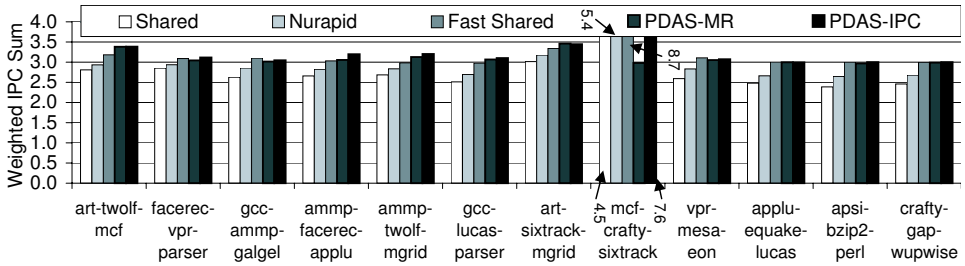


Figure 5.19: 3-Thread Weighted Speedup.

With three active threads, PDAS-IPC improves performance over Private by 17%, Shared by 23%, Nurapid by 14%, Fast Shared by 1%, and PDAS-MR by 14%. With all four cores active, PDAS-IPC improves performance over Private by 25%, Shared by 44%, Nurapid by 35%, Fast Shared by 24%, and PDAS-MR by 13%.

Program	Private			Shared			PDAS-IPC			Private 4MB		
Mix	T0	T1	T2	T0	T1	T2	T0	T1	T2	T0	T1	T2
art-sixtrack-mgrid	0.49	0.13	0.20	0.07	0.14	0.20	0.00	0.14	0.37	0.00	0.06	0.20
facerec-vpr-parser	0.35	0.35	0.16	0.31	0.27	0.18	0.20	0.37	0.18	0.17	0.13	0.05
gcc-lucas-parser	0.05	0.33	0.16	0.05	0.33	0.19	0.05	0.51	0.06	0.02	0.33	0.05

Table 5.9: Selected 3-Thread L2 Miss Rates.

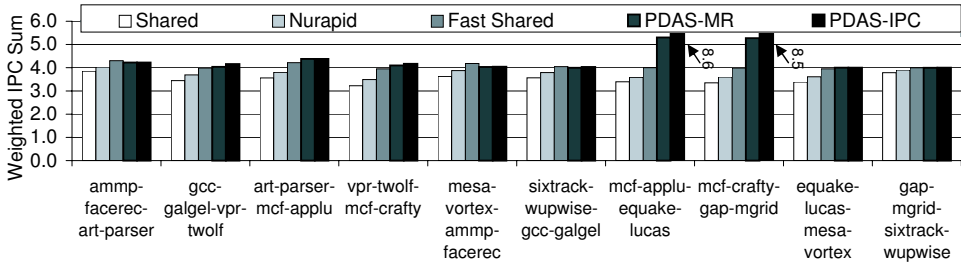


Figure 5.20: 4-Thread Weighted Speedup.

Migrations per access at both scenarios look similar to the data shown for two active threads.

With three or four active threads, Private clearly outperforms the Shared and NuRAPID caches, as both are hindered by thread interference as well as increased access latencies. Since we are weighting relative to Private, the weighted speedup for Private is equal to the number of active threads (1 for ST, 2 for 2 threads, 3 for 3 threads, and 4 for 4 threads).

With the increased load of three or four threads, PDAS needs more accurate information to effectively allocate space so that there is enough free space for throughput improvement while bounding per thread degradation. This is the main reason for the increased difference between PDAS-MR and PDAS-IPC. One example is *mcf-crafty-sixtrack*. PDAS-MR sees miss rate reduction for *sixtrack* beyond 256KB and even up to 1.5MB, but this never results in a performance improvement for this benchmark. Also, the miss-rate data predicts *mcf* will improve with less space than what is actually

required (i.e. too small of a Minimum Extra Partition). The combination of inaccuracies results in PDAS-MR's inferior partition decision. Similar behavior is observed for 4-thread cases.

Table 5.9 illustrates some miss rate trade-offs discovered by PDAS-IPC. For *art-sixtrack-mgrid*, PDAS takes space from *sixtrack* and *mgrid* to satisfy *art*'s demand. *Sixtrack* and *mgrid* achieve their guaranteed min performance while allowing *art* to boost the overall throughput.

Overall, PDAS remains the performance leader while minimizing the requirements on internal bandwidth and cache power usage through migration reduction. In addition, PDAS effectively bounds per thread IPC degradation to Gthresh as described in section 5.3.4.

5.3.8 Summary

Multi-core architectures continue to thrive in computer architecture research, as indicated by the multitude of both academic and industry projects. More active cores per chip increases the load on the lowest-level cache.

PDAS is a scalable, multi-ported NUCA that dynamically allocates its distributed cache resources through an intelligent, realizable on-line partitioning strategy. We achieve improved partitions by taking actual performance into account rather than just the miss rate. PDAS guarantees a minimum performance bound for each core while pursuing high throughput. For a less contended two thread workload, we are able to achieve a 20% performance improvement over a comparably sized shared cache with idealized latency. For more contended workloads of three and four threads, our performance driven partitioning enables PDAS to maintain performance advantage over either a private or idealistic shared L2. Miss rate guided partitioning results in performance closer to that of private L2s. Performance driven paritioning further reduces migrations as compared to miss rate guided partitioning. We see a 27% drop in migrations for two threaded workloads when comparing our approach with and without consideration for IPC.

This cache architecture enables further scaling of thread-level parallelism in future designs. Future directions will include a comprehensive power study and an exploration of both SMT and cooperative multi-threaded workloads.

CHAPTER 6

Algorithmic Acceleration

This chapter describes our algorithmic contributions in real-time physics simulation.

6.1 Perceptual Error Tolerance

6.1.1 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 [agea], GPUs [Havb], and the Cell [Hof05], 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 funda-

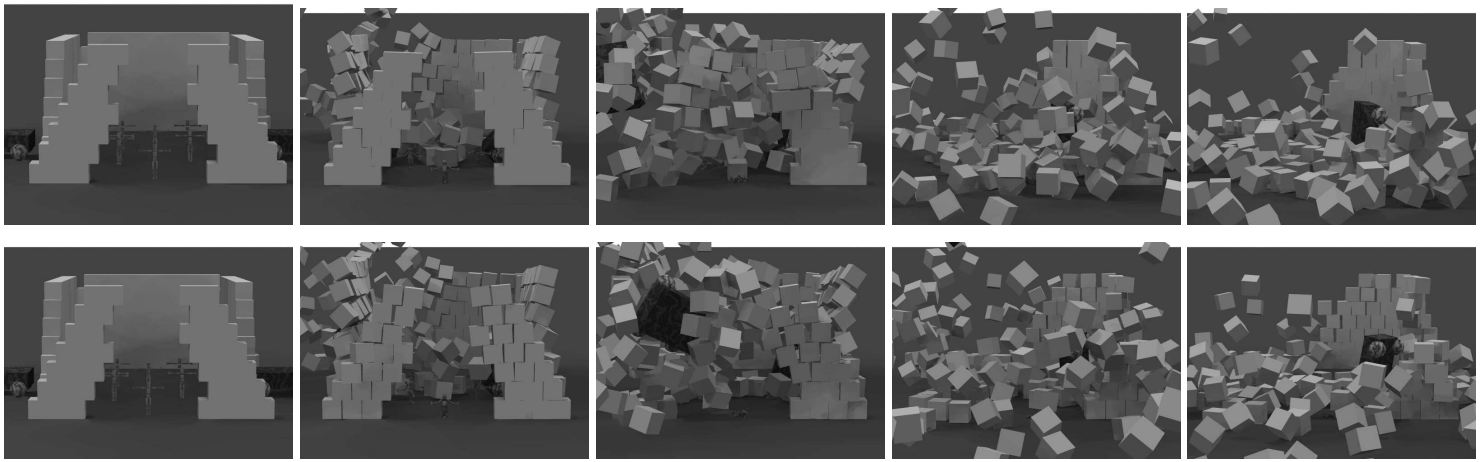


Figure 6.1: Snapshots of two simulation runs with the same initial conditions. The simulation results shown on top is the baseline, and the bottom row is simulation computed with 7-bit mantissa floating-point computation in *Narrowphase* and *LCP*. The results are different but both are visually correct.

mental 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[OHM04, OD01]. It has been demonstrated that there is a surprisingly large degree of error tolerance in our perceptual ability. This tolerance is independent of a viewer's understanding of physics [Pro].

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 [RP03a, CH97, MDC06] 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 [SR06]. At the compiler level, dependencies among parallel tasks could be broken to reduce 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 chapter, we address the challenging problem of leveraging perceptual error tolerances to improve the performance of real-time physics simulation. The main challenge is the need to establish a methodology using a set of error metrics that can measure the visual performance of a complex simulation. Prior perceptual thresholds do not scale to complex scenes. This chapter addresses this challenge and investigates specific hardware solutions.

The contributions are threefold:

- Methodology to evaluate physical simulation errors in complex dynamic scenes.
- Identification of the maximum error that can be injected into each phase of the low-level numerical PBA computation.

- Exploration of precision reduction and time-step tuning to exploit error tolerance to improve PBA performance.

6.1.2 Background

In this section, we present a brief overview of physics simulation, identify its main computational phases, and categorize possible errors. A review of the latest results in perceptual error metrics and their relation to physical simulation was described in Section 3.3.

6.1.2.1 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 [Bar97, MHH06, Havb]. 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 our work. As far as we are concerned, the time-step is a high-level parameter that should be controlled by the developer of an application. Our work aims to identify low-level trade-offs that are part of a hardware solution and as such are transparent to an application developer. As we demonstrate later, the errors that our approach introduces in the simulation results are no greater that those introduced by the discretization alone.

Physics simulation can be described by the data-flow of computational phases shown in Figure 6.2. 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

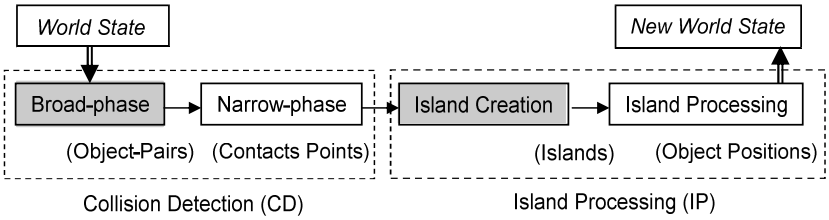


Figure 6.2: Physics Engine Flow. All phases are serialized with respect to each other, but **unshaded** stages can exploit parallelism within the stage.

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.

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 po-

sition 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)

6.1.2.2 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 [ODG03, HRP04a] for simple scenarios. All errors with magnitude smaller than these thresholds are considered imperceptible. All errors exceeding these thresholds without simulation blow-up are considered visible but bounded. All errors that lead to simulation blow-up are considered catastrophic.

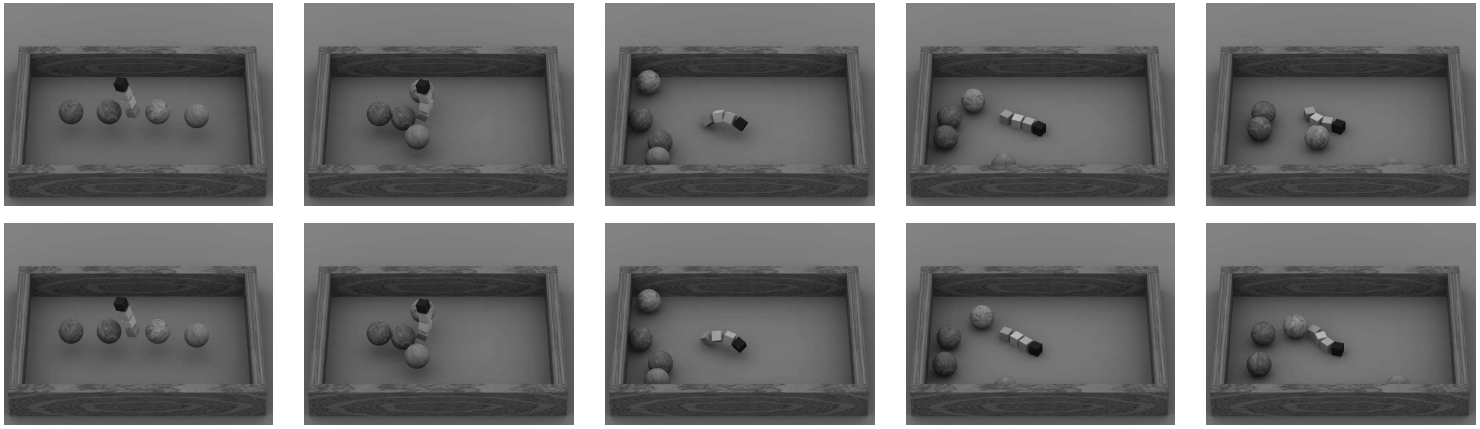


Figure 6.3: Snapshots of two simulation runs with the same initial conditions and different constraint ordering. The results are different but both are visually correct.

An interesting example that demonstrates imperceptible simulation errors is shown in Figure 6.3. 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. This behavior is due to the *constraint reordering* that the iterative constraint solver employs to reduce bias [Eng]. 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 [agea]. For our purposes though, it provides an objective way to establish what physical errors are acceptable when we develop the error metrics in Section 6.1.4.

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.

6.1.3 Methodology

One major challenge in exploring the trade-off 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.

6.1.3.1 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 6.1. The scene is composed of a building enclosed on all four sides by brick walls 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).

Benchmarks from the PhysicsBench 2.0 suite [YFP07] as described in 5.1 are also used for the reduced precision evaluation.

Our physics engine is a modified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [Eng]. ODE follows a constraint-based approach

for modeling articulated figures, similar to [Bar97, agea], 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.

6.1.3.2 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 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.

6.1.3.3 Error Metrics

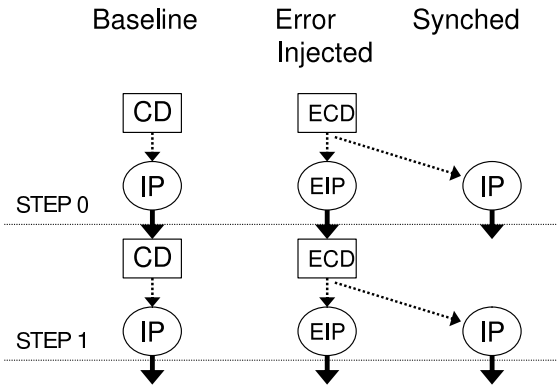


Figure 6.4: Simulation Worlds. CD = Collision Detection. IP = Island Processing. E = Error-injected. The Baseline world simulates without any errors. The Error-Injected world simulation has error injected. The Synched world copies the state of objects from Error-Injected after collision detection. Then continues the physics loop with no error injection.

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 6.3.

To address this, we make use of three simulation worlds as shown in Figure 6.4: *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.

6.1.4 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 game play. 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.

6.1.4.1 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 6.5 demonstrates the maximal error that results from error injection on our seven perceptual metrics. Following the error injection methodology of section 5.2.1, 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 6.5, 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.

While these maximal error values are interesting, as shown in figure 6.6, 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 results in significant percentage change.

6.1.4.2 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 6.5 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 6.6, 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 6.1: Max Error Tolerated for Each Computation Phase.

Table 6.1 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 samples 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 6.1, we consider four approaches: confirming our thresholds visually, comparing our errors to a very simple scenario

with clear error thresholds [ODG03], comparing the magnitude of our observed error to constraint reordering, and examining the effect of the time-step on this error.

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 [ODG03]. 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 6.1 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. [1]

Threshold Evaluation 2 We recreated the experiment used in [ODG03] (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 6.2, extrapolated from prior

[1] One such experiment is shown in the accompanying video.

work [ODG03, HRP04a], 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 [ODG03] 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 6.1 to inject error into this simple example. Using the thresholds from [ODG03], 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.

Threshold Evaluation 3 Interestingly, when we enable reordering of constraints in ODE, most of our perceptual metrics will exceed the thresholds from [ODG03], 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 6.1. 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 6.2 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 [ODG03]. 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, 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 6.2: 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 [ODG03] demonstrates that these thresholds are not useful as a means of guiding the trade-off 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 6.1.

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 6.5 are a function of the time-step and not the error injected.

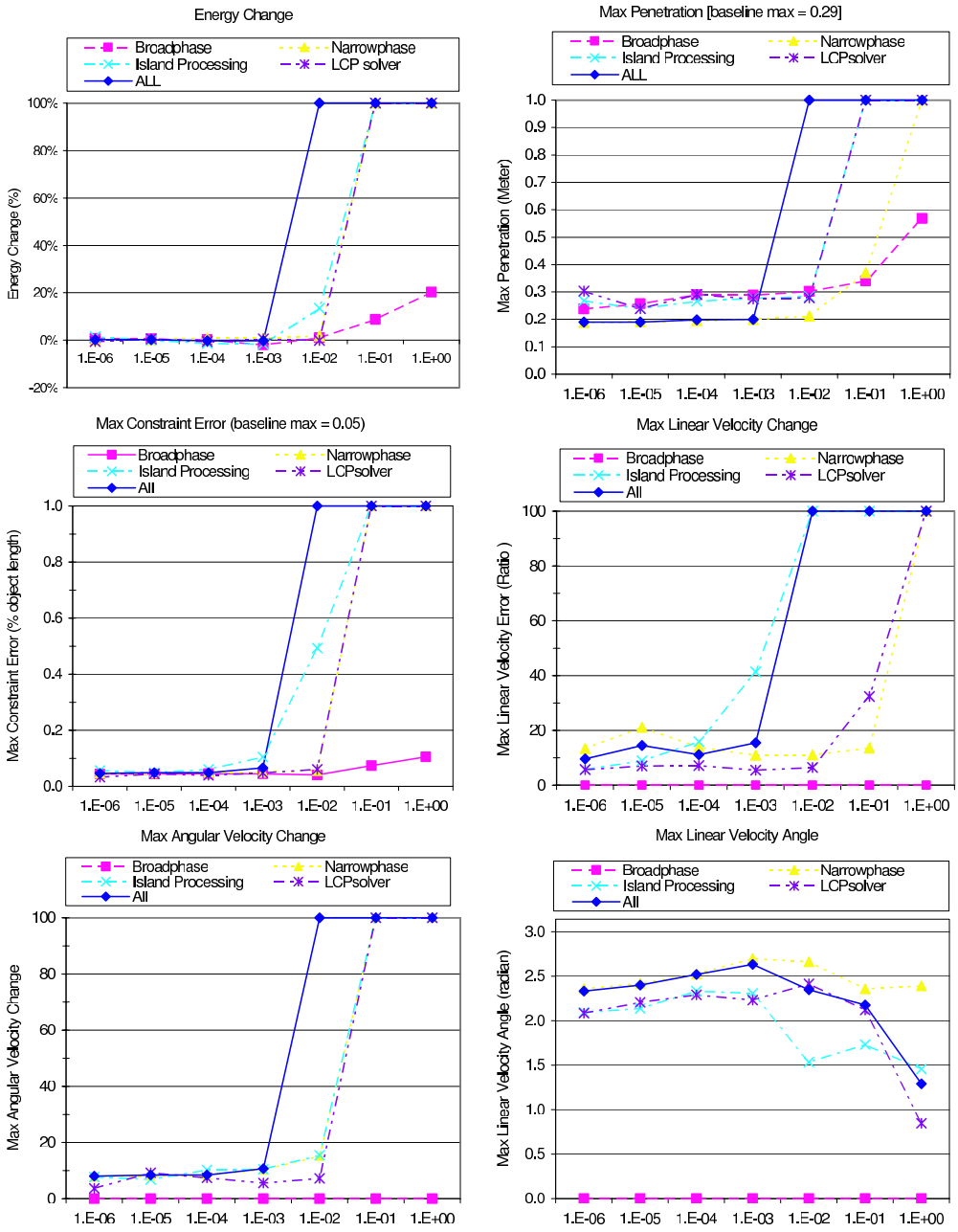


Figure 6.5: Perceptual Metrics Data for Error-Injection. X-axis shows the maximum possible injected error. Note: Extremely large numbers and infinity are converted to the max value of each Y-axis scale for better visualization.

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 6.6: Average and Standard Deviation of Error-Injection.

6.1.5 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.

6.1.5.1 Prior work

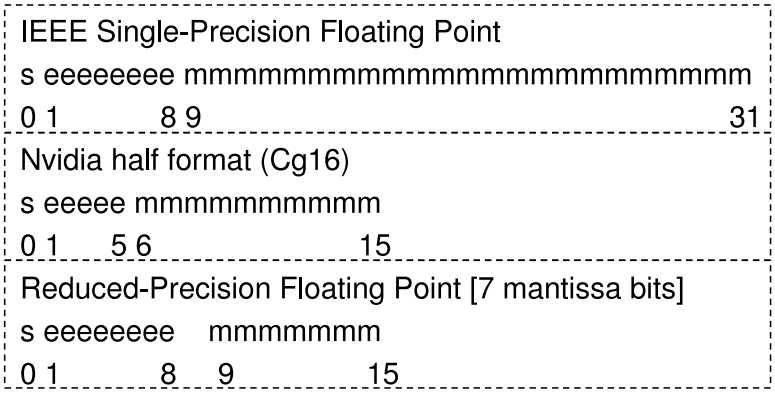


Figure 6.7: Floating-point Representation Formats (s = sign, e = exponent, and m = mantissa).

The IEEE 754 standard [Gol91] defines the single precision floating-point format as shown in figure 6.7. 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 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) [MGA03] as shown in Figure 6.7. There is one sign bit, and both the exponent and mantissa bits are reduced. The Cg16 format has been used for several embedded applications [EBL05, LEO05, EEL04] 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 [SEP93, FCR02]. Both prior work emulate variable-precision floating point arithmetic by using new C++ classes.

Prior work [GST06] 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.

6.1.5.2 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 [SEP93] and is comparable to [FCR02]. 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 granularity. 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).

6.1.5.3 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 6.1.2.2. 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 6.1, 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 6.1, we present a range of possible minimum mantissa bits in table 6.3 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.

Figure 6.8 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 6.1.4, we summarize the

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 6.3: Numerically-derived Min Mantissa Precision Tolerated for Each Computation Phase.

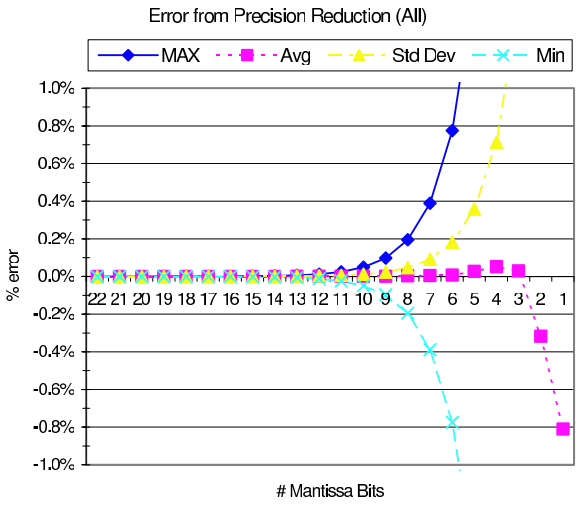


Figure 6.8: Percentage Error Injected from Precision Reduction. X-axis shows the number of mantissa bits used.

per-phase minimum precision required in Table 6.4 based on data in Figure 6.9. When comparing Table 6.4 and Table 6.3, 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 [ODG03, HRP04a].

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

Table 6.4: Simulation-based Min Mantissa Precision Tolerated for Each Computation Phase.

While the exact precision reduction tolerance may vary across different physics engines, this study shows the potential for leveraging precision reduction for hardware optimizations.

To obtain the exact tolerance for interactive entertainment applications using the ODE engine, we apply the same methodology to benchmarks in PhysicsBench 2.0. The results for fine-grain phases of *LCP* and *Cloth* with rounding are shown in Table 6.5.

Benchmark	LCP	Cloth
Breakable	[5]	
Continuous	[4]	
Deformable	[3]	[13]
Everything	[10]	[14]
Explosions	[11]	
Highspeed	[4]	
Periodic	[13]	
Ragdoll	[7]	

Table 6.5: Min Mantissa Precision Tolerated for PhysicsBench 2.0.

The precision requirement for *Cloth* is based on energy and stretch. *Cloth* energy is

computed from the particles' velocity and height (assuming mass of 1). The stretch is the difference between the initial distance vs the actual distance between two particles.

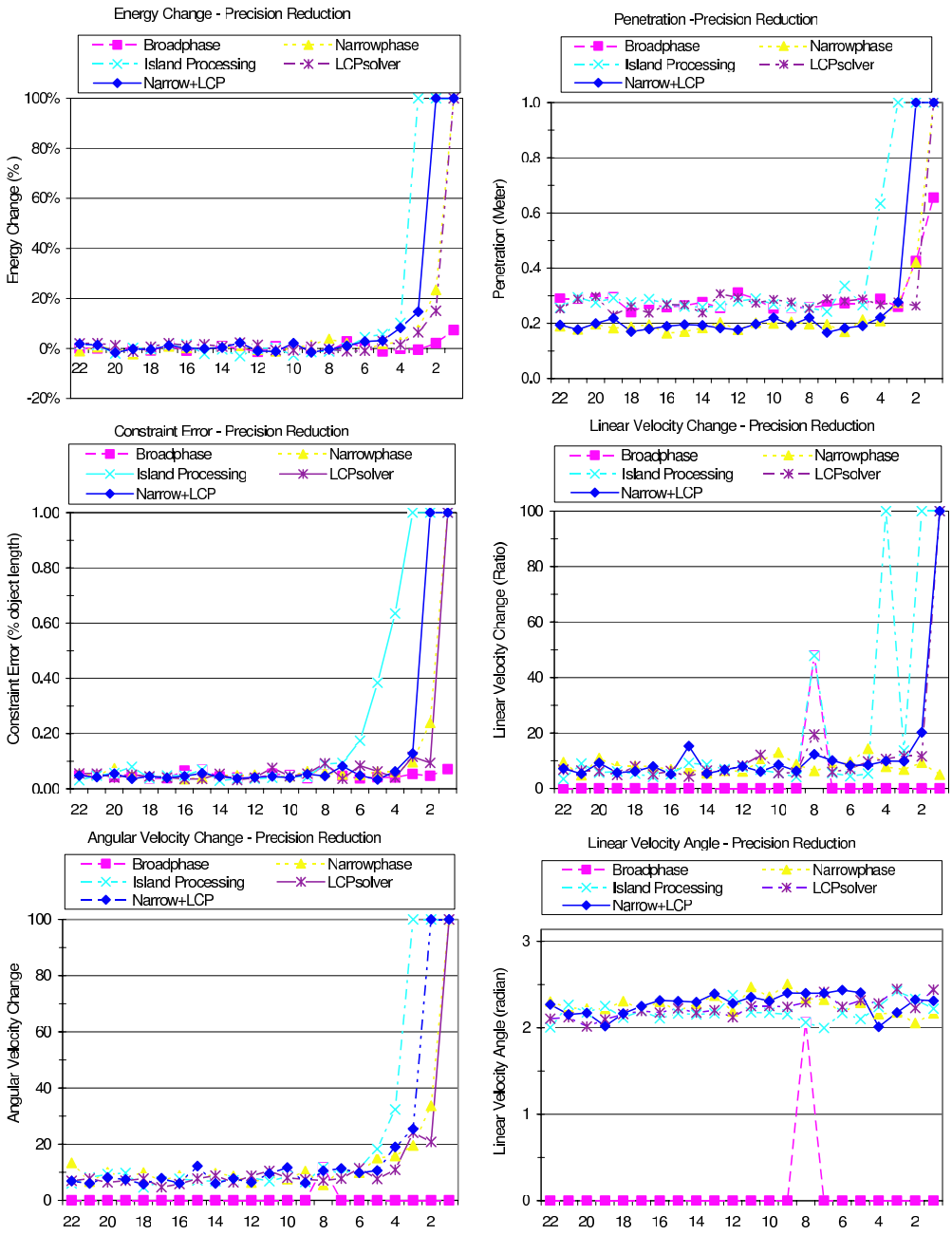


Figure 6.9: Perceptual Metrics Data for Precision Reduction. X-axis shows the number of mantissa bits used. Note: Extremely large numbers and infinity are converted to the max value of each Y-axis scale for better visualization.

6.1.6 Simulation Time-Step

As described in Chapter 2, 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 granularity 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.

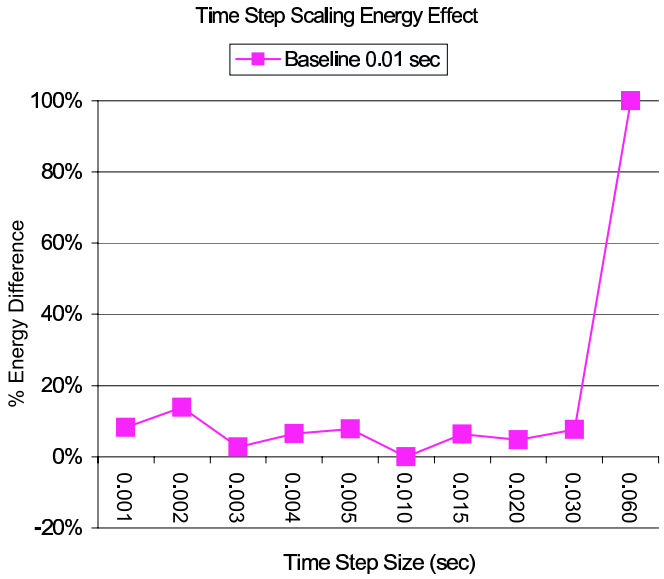


Figure 6.10: Effect on Energy with Time-Step Scaling.

As shown on Figure 6.10, the maximum time-step (out of the time-steps simulated)

for the scenario described in Section 6.1.3 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.

6.1.7 Summary

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, examined the minimum required precision and identified the maximum allowable time-step for physics simulation.

6.2 Fast Estimation with Error Control

QuickStep uses an iterative approach where during each iteration (a) each body in an Island is essentially considered a free body in space and solved independently of the others (b) a constraint relaxation step progressively enforces the constraints by some small amount. The constraint satisfaction increases with the number of iterations. According to the ODE manual, 20 iterations is considered the minimum for consistent robust simulation. As we will show later in this section, the error rate for low iteration counts can be quite severe.

The iterative nature of QuickStep implies that a solution can be obtained at the iteration granularity, by taking the result of the last completed iteration. However, simply reducing the QuickStep iteration count can drastically increase simulation errors which can accumulate and "blow up" calculations. Visually acceptable errors are dependent on the situation and are of much larger magnitude than what the physics engine can tolerate.

To address the need for fast result turnaround and the importance of avoiding large errors that can drastically impact the quality of the solution, we consider decoupling these two components. While dependent software components require physics simulation to provide a solution within some fraction of a frame's time, there is no reason why physics simulation cannot continue to run *after* returning a solution on a separate thread/core. Consider running QuickStep for only a single iteration to provide results to the graphics engine – and then for the remainder of the 1/30th of a second, we continue the physics simulation loop for the same time-step. The eventual refined solution from this error correction thread is fed back as input to the next frame of the physics loop.

This technique shortens physics simulation's critical path by altering the control

flow to generate a fast solution for the graphics core. At the same time, all errors fed back into the physics engine are 100% eliminated, as compared to running the full 20 iterations, by the end of each frame. Thus, errors are never allowed to propagate beyond a frame's worth of time. We call this approach *fast estimation with error control* (FEEC) since the physics simulation is allowed to continue in parallel with the rest of the interactive entertainment application and effectively limit the error rate of doing faster, lower iteration simulation.

This is an effective approach in leveraging contexts in a CMP environment: some subset of cores can continue to refine a physics solution while other cores handle other game engine components. In cases where other components (such as AI) might require feedback from the physics engine, solutions can be provided on demand, depending on the time constraints of the different game engine components.

We numerically verified that the errors using FEEC are imperceptible. As will be shown in our results, over the entire course of the simulation they are numerically equivalent to using a 19 iteration QuickStep without FEEC.

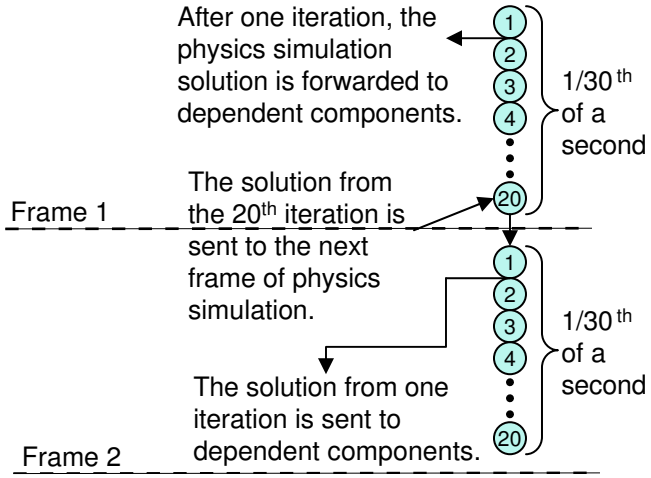


Figure 6.11: Fast Estimation with Error Control (FEEC).

The parallel results in section 4.3.1 demonstrated that while many benchmarks cannot achieve 300 fps, all but one were able to achieve 30 fps. Therefore, for our

FEEC study, we will assume that all threads can achieve 30 fps with a 20 iteration QuickStep, but only send the results of the first iteration to the graphics hardware. The full 20 iteration run will be passed to the start of the next physics simulation. For this technique, we will report errors based on the result of the first iteration (i.e. what would be seen by the graphics hardware). The FEEC approach is illustrated in Figure 6.11. Each circle represents one iteration of QuickStep. This figure shows FEEC when the physics engine's stepsize is 0.033 seconds. If the stepsize is smaller than one frame's time, then multiple threads of execution are required to enable FEEC.

The performance and error data of FEEC are presented in Section 7.2.1.

CHAPTER 7

Architectural Exploitation of Algorithmic Properties

This chapter discusses our architectural contributions in exploiting real-time physics simulation's algorithmic properties.

7.1 Leveraging Precision Reduction in FPU Design for CMPs

In section 6.1.5 we show how physics simulation can tolerate significant precision reduction. In this section we propose three approaches that exploit this result: *area reduction*, *trivialization* and and *memoization*.

7.1.1 Core Area Reduction

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 ($0.3mm^2$).

Based on the data presented in section 6.1.5, the minimum precision required to satisfy all phases of all benchmarks is 14 bits. One simple approach to leverage this reduced precision requirement is to design reduced precision FPUs.

Reducing the mantissa from 24 (23 + implicit 1) bits to 15 (14 + implicit 1) bits

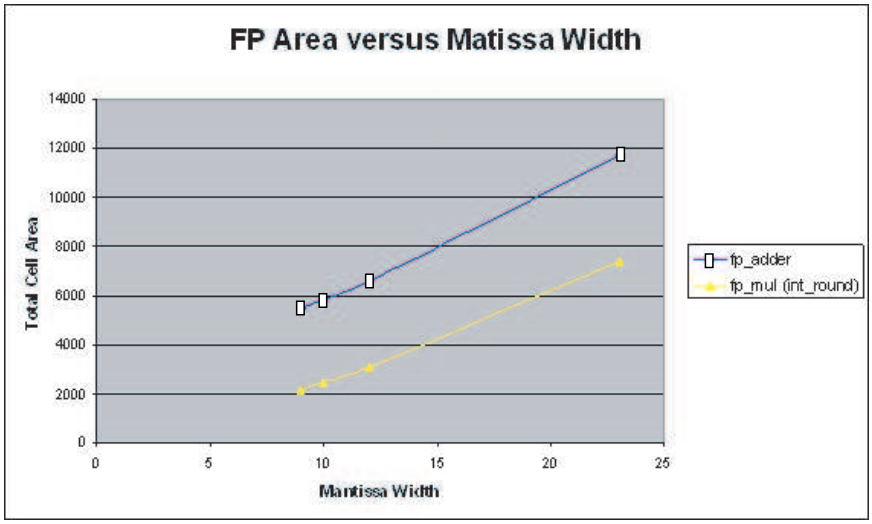


Figure 7.1: FP Adder/Multiplier Area with Varying Mantissa Width.

reduces the logic cost from $1.5mm^2$ to about $0.9mm^2$. The precision-reduced core is 20% 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 data-paths. Figure 7.1.1 shows a linear relationship between area cost and mantissa width for both the floating-point adder and multiplier.

This 20% 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.

Instead of using only reduced precision FPUs, a hybrid chip design can support both low-precision FP (at a higher throughput) and single-precision FP (at a lower throughput). This is another way to exploit perceptual tolerance while allowing the developer to choose between high-throughput and high-accuracy. Future explorations can address this approach.

7.1.2 Improve Floating-Point Trivialization and Memoization

Another method to leverage reduced precision is to improve the efficiency of trivialization and memoization.

7.1.2.1 Prior Work

Trivial Operations The first exploration of trivial computation is done by [Ric92, Ric93]. The SPEC 92 and Perfect Club and benchmarks were used and the studies focused on a set of 8 trivial computations of multiplication (by 0, 1, and -1), division (X/Y with X = 0,Y,-Y, and square roots of 0 and 1. More recent work by [YL02] expands the range of trivialized 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. In addition, this paper suggests that the amount of trivial computations are not highly dependent on input values by comparing simulation results using two input sets. The latest work by [AB05] studies the energy efficiency benefits of bypassing trivial operations.

Memoization A closely related technique to bypassing trivial operations is memoization or value reuse [SS97, CF98, HL99, MGT99, ACV05]. 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 chocked for a match in the table. If there is a match, the cached result is reused instead of executing the instruction. [ACV05] leverages media application's error tolerance to remove the lower significant bits of floating point mantissas for the input operand match. Full precision results are still stored.

7.1.2.2 Reduced Precision Optimizations

Prior work has examined the potential for improving performance and reducing energy by exploiting trivial operations and memoization. By coupling these two techniques with precision reduction, we can treat a significantly larger percentage of floating point operations as trivializable or memoizable. The minimum mantissa precisions used in this study are based on Table 6.5.

From the characterization in [YFP07], FP add and multiply make up between 35% to 40% of total dynamic instructions for fine-grain phases of *Island Processing* and *Cloth*. In this section, we will focus on FP add, subtract, and multiply operations.

FP Operation	Representation	Trivial
Add	X + Y	X=0, Y=0
Subtract	X - Y	Y=0
Multiply	X * Y	X=0, Y=0, X=1, Y=1

Table 7.1: Conventional Trivial Cases.

FP Operation	Representation	Trivial
Add	X + Y	X=0, Y=0 Exponent difference greater than number of mantissa bits
Subtract	X - Y	X=0, Y=0 Exponent difference greater than number of mantissa bits
Multiply	X * Y	X=0 or 1, Y=0 or 1 Mantissa of X or Y equals all zeroes

Table 7.2: Reduced Precision Trivial Cases.

Reduced Precision Trivial Operations Conventional trivialization logic for these operations involve zero, one, or negative one detection of operands as shown in Table 7.1. By examining the details of floating point computation, we have expanded the trivializable cases to the list shown in Table 7.2. The rightmost column refers to trivializable operations.

The two novel cases we propose are the following:

- For adds and subtracts, if the magnitude of exponent difference between the operands is greater than the number of valid mantissa bits then the operation becomes a bypass of the larger operand.
- For multiplies, if the mantissa bits of one operand are all zeroes then the mantissa logic can be trivialized to bypassing the other operand. The exponent and sign logic still needs to be executed. This is a general case of multiplication by 1 or -1.

Based on simulations of PhysicsBench 2.0 with object-disabling and rounding for 200 simulation steps, we have compiled the percentage of trivial operations with full precision versus reduced precision in Table 7.3 for fine-grain phases with large numbers of FP operations: *LCP*, *Cloth2*, and *Cloth3*.

Precision reduction increases 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.

Why are there so many trivial computations? To understand the reasons for such high percentages of trivial operations, we examine the implementation details of each phase.

In *LCP*, the core computation involves multiplications of 6-element matrices such as constraint force, jacobian matrix, and inverse jacobian. For each matrix, 3 elements refer to linear components of each axis with values from the normal vector at the contact point. The other 3 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.

In *Cloth2*, most of the computation revolves around satisfying the distance constraint between neighboring particles. This involves taking the difference between two particle's position, computing the distance, and finding the ratio between the distance due to collision vs the desired distance to maintain the cloth shape. Then, each particle is moved by 50% of the difference.

In *Cloth3*, most of the computation revolves around the intersection test between a cloth particle and a triangle mesh representing a part of a rigid body. This code finds the amount of penetration by the particle into the rigid body and moves the particle by 0.1 meter outside of the body. The computation involves dot product of point with plane normal, multiplication of new depth with plane normal, and subtraction of current position with delta required to move particle outside object.

Reduced Precision Memoization Fuzzy memoization proposed in [ACV05] is closely related to reduced precision memoization. The key difference is that our approach stores and uses reduced precision results rather than full precision results as in [ACV05]. This optimization further improves area efficiency and is shown by our simulations to produce tolerable errors.

The memoization method for evaluation is to match on both operands' mantissa values, and we focus this study on the efficiency of a shared memoization table for frequent, long latency FP multiply operations. The table we used contains 256 en-

tries with 16-way associativity, and it is indexed by the XOR of the most significant bits in the mantissas. Smaller tables were evaluated and found to be very ineffective. Trivializable operations are filtered from accessing the memoization table.

Our simulation results using full and reduced precisions for multiplies are shown in Table 7.4. Results for addition/subtraction is shown in Table 7.5, and Table 7.6 shows the results for sharing one table across all three instruction types.

For multiplies, precision reduction increases the effectiveness of memoization by 725% on average. This translates to an additional 16% of total FP multiplies being memoized on average. The effectiveness of memoization on addition and subtraction operations is less significant. When sharing one table among all types of operations, the overall hit rate is reduced due to contention between multiply and add/sub operations. Compared to the data on SPEC benchmarks from prior work [CF03], the physics workload has lower value locality. Significant hit-rate on the memoization table requires the use of precision reduction.

When precision reduction is applied to the combination of trivialization and memoization, only 54% of FP adds and 35% of FP multiplies on average requires execution on a FPU. Using data from prior work [YFP07], we see that for *LCP* and *Cloth* on average 20% of total instructions are FP adds and 17% are FP multiplies. Multiplying these percentages together shows that less than 17% of total instruction count requires the FPU. Ignoring routing and arbitration overhead, this data indicates the possibility of sharing one real FPU among 5 cores while keeping utilization under 100%.

Dynamic Precision Tuning From the data presented in Table 6.5, we see that the minimum required precision varies significantly between scenarios and across computation phases. In order to optimize the benefit of precision reduction, we propose a dynamic precision tuning mechanism whereby the application indicates to the hard-

ware the minimum required precision.

Based on the results shown in Section 6.1, we see most of the perceptual tolerance metrics correlate with energy. By using the law of energy conservation, the application can compute the energy difference between successive simulation steps, while accounting for forces injected by user input, to determine whether the simulation is diverging towards instability.

Benchmark	23-bit %
	Add [LCP, CL2, CL3]
	Mult [LCP, CL2, CL3]
		Reduced %
		Add [LCP, CL2, CL3]
		Mult [LCP, CL2, CL3]

Bre	[36] [34]	[56] [51]
Con	[49] [43]	[71] [62]
Def	[32, 11, 18] [31, 35, 39]	[61, 32, 31] [64, 47, 51]
Eve	[35, 19, 27] [33, 39, 33]	[43, 32, 33] [38, 45, 39]
Exp	[28] [25]	[38] [29]
Hig	[27] [23]	[54] [49]
Per	[32] [32]	[34] [34]
Rag	[34] [33]	[47] [46]

Table 7.3: Percent Trivialized FP Operations for Full and Reduced Precision.

Benchmark	23-bit % [LCP, CL2, CL3]	Reduced % [LCP, CL2, CL3] test
Bre	[2]	[33]
Con	[1]	[38]
Def	[2, 2, 4]	[35, 8, 19]
Eve	[3, 2, 5]	[6, 7, 15]
Exp	[7]	[10]
Hig	[8]	[51]
Per	[0]	[0]
Rag	[0]	[4]

Table 7.4: Percent Memoized FP Multiply for Full and Reduced Precision.

Benchmark	23-bit % [LCP, CL2, CL3]	Reduced % [LCP, CL2, CL3] test
Bre	[0]	[2]
Con	[0]	[8]
Def	[0, 0, 6]	[7, 0, 15]
Eve	[0, 0, 6]	[1, 0, 15]
Exp	[0]	[1]
Hig	[0]	[11]
Per	[0]	[0]
Rag	[0]	[0]

Table 7.5: Percent Memoized FP Add/Sub for Full and Reduced Precision.

Benchmark	23-bit % [LCP, CL2, CL3] add/sub,mult	Reduced % [LCP, CL2, CL3] test add/sub,mult
Bre	[0/1]	[1/9]
Con	[0/0]	[6/36]
Def	[0/1, 0/2, 6/2]	[4/25, 0/8, 14/10]
Eve	[0/0, 0/3, 5/3]	[1/2, 0/8, 10/10]
Exp	[0/7]	[1/9]
Hig	[0/8]	[8/48]
Per	[0/0]	[0/0]
Rag	[0/0]	[0/1]

Table 7.6: Percent Memoized FP Mixed for Full and Reduced Precision.

7.2 Fuzzy Computation

7.2.1 Value Prediction

Prior work has demonstrated the effectiveness of using last value prediction at reducing instruction latency [LWS96]. Last value prediction is a speculative technique that breaks true data dependencies by using the last value produced by a given instruction as a prediction for the input to its dependent instructions. The value prediction is verified when the instruction actually completes, and the dependent instructions are re-executed if a misprediction has occurred. Confidence mechanisms [CRT99] can effectively limit the impact of mispredictions. Recent work on shows some benefit for floating point value predictions [N 05] using multi-threaded execution. Fuzzy instruction reuse [ACV05] has been further proposed to reduce power usage by floating point units. This latter technique is non-speculative, and cannot break data dependencies.

Exact value prediction can break true data dependencies, and it will not have any impact on the number of errors in the physics simulation. However, we can trade some accuracy to improve the instruction prediction rate. We propose *Fuzzy* value prediction (FVP), where floating point value prediction is allowed to err within a certain bound without resulting in a misprediction and subsequent recovery. The main motivation for fuzzy value prediction is that real-time physics simulation for gaming workloads already introduces small errors by using single-precision floating point, large step sizes, and approximation methods in solving the equations. We assume that an error in the 10^{-6} range is tolerable in our case. The quantities involved in the simulation are macroscopic and follow the metric system. An error, for example, in position of 10^{-6} meters is of microscopic scale and visually negligible. This optimization is unique to graphics related workloads, since other applications typically require precise state, especially for floating point calculations.

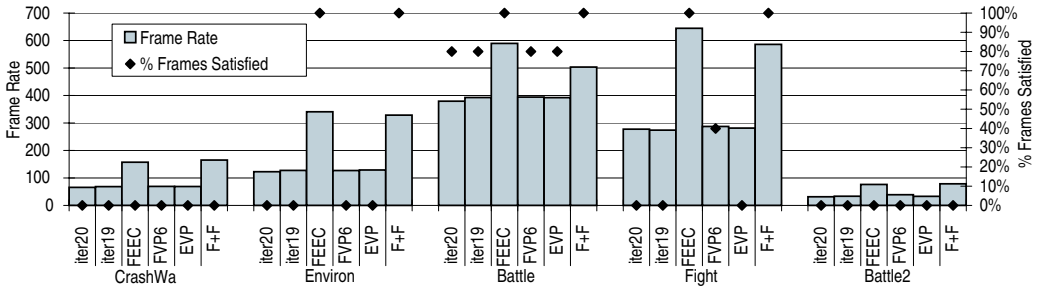


Figure 7.2: Performance of FEEC and fuzzy value prediction.

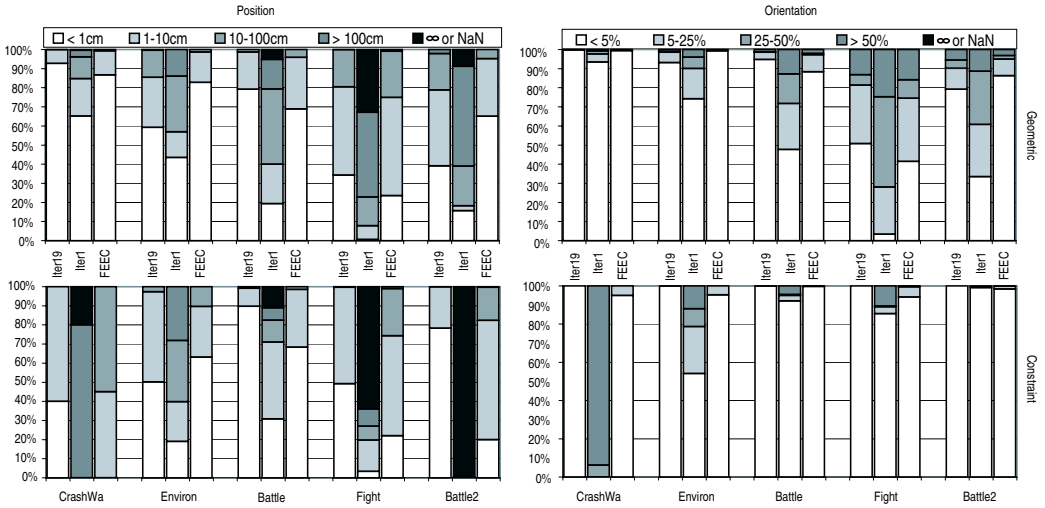


Figure 7.3: Error for FEEC relative to a 20-iteration QuickStep.

7.2.1.1 Methodology

Although we used the same binaries as section 4.3.1 in our simulations, we used PTLsim [You], a cycle accurate simulator supporting the full x86 instruction set, to allow architectural modification and to avoid the system-level nondeterminism that can occur in real system modeling (i.e. variability in system load, OS interaction, etc). PTLsim also models the translation of x86 instructions into micro-ops, similar to the translation in current x86 CPUs. We modified PTLsim to support value prediction. We use the x86 instruction PC concatenated with a micro-op ID to index the value prediction table. The micro-op ID is a simple 4-bit counter, since PTLsim generates up to 16 micro-ops per x86 macro-instruction. This ID uniquely identifies micro-ops that are

Frequency	3 GHZ
Issue	8-way out-of-order
Issue window size	4 16-entry Clusters
Branch Predictor	64K Gshare 4K 4-way BTB
Instruction L1 Cache Latency	32KB 4-way 2 cycles
Data L1 Cache Latency	16KB 4-way 2 cycles
Inst Window Load/Store Queue size	192 entries 144 entries
L2 Cache Latency	256KB 8-way 7 cycles
L3 Cache Latency	2MB 16-way 16 cycles
Functional Units (Int, Int, Mem port, FP)	2,2,2,2
Mem Latency	160 cycles

Table 7.7: Parameters for our architectural configuration.

generated from the same x86 instruction. The ID (starting at 0) is assigned in order to each micro-op.

From the data in figure 4.4, we see that the computational demand of complex PhysicsBench tests is above the performance of a contemporary design. To evaluate FEEC and FVP, we choose a more aggressive processor design, with both a high frequency and wider resources. Table 7.7 presents the simulation parameters used in this section.

To better characterize the error in geometric position or constraint violations, we classify errors into one of five categories according to the magnitude of the error. For position, the categories are: below 1 cm, between 1-10 cm, between 10-100cm, more than 100 cm, and infinity or NaN. For orientation, the categories are based on percentages of 2×PI: below 5%, between 5-25%, between 25-50%, above 50%, and infinity or NaN. The last category for both measures shows cases where the error has blown up. Note that in all cases, every constraint and every object position will map to one of these categories, even if there is an exact match (i.e. the position error is 0 cm). This ensures that we are always comparing the same number of possible errors for a given benchmark.

7.2.1.2 Results

Figure 7.2 shows the frame rate (primary axis, bars) and % frames satisfied (secondary axis, diamonds) for QuickStep with 20 iterations (20Iter), QuickStep with 19 iterations (19Iter), Fast Estimation with Error Control (FEEC), Fuzzy Value Prediction (fuzzy to 10^{-6} meters) (FVP6), Exact Value Prediction (EVP), and FEEC with FVP6 (F+F). FEEC has a dramatic effect on Environ, Battle, and Fight - all three applications are able to completely satisfy their frame constraints. CrashWa and Battle prove especially difficult to satisfy. On average, FEEC dramatically improves performance over 20Iter

by 220% with minimal error as described below. By trading some accuracy for performance, FEEC lowers the required speedup for parallel sections of Battle2 from the 84X to 30X.

Considering the hardware overhead of value prediction, this approach seems less attractive than FEEC when there are plenty of core contexts available to continue to run physics simulation. FVP6 is able to see more frames satisfied for Fight, but is unable to help other applications. Moreover, FVP6 degrades the performance of FEEC when these techniques are applied together. In the rest of this section, we focus on the errors generated by FEEC.

Despite FEEC's performance advantage, this same benefit could be obtained by using only a single iteration with Quickstep. Therefore, we need to compare the errors seen by FEEC to a single iteration with Quickstep.

Figure 7.3 shows the error breakdown for the FEEC approach. Results are shown in a grid of four figures: the first row of two figures represents geometric error, and the second row represents constraint error. The first column represents position and the second column represents orientation. In each figure, three architectures are shown: QuickStep with 19 iterations, QuickStep with 1 iteration, and FEEC. The figures are oriented such that the benchmarks and architectures line up vertically – so the geometric and constraint position errors for CrashWa for Iter19 are vertically aligned.

These results clearly demonstrate that FEEC is able to achieve errors comparable to Iter19, sometimes doing better and sometimes doing worse. In all cases, there are few cases where the computation blows up for either Iter19 or FEEC – unlike Iter1 which has a substantial number of the highest class of errors. Due to the severe errors introduced with Iter1, its performance data is meaningless in that image data created is not usable.

7.3 Object-Pair Information

We have shown that real-time physic's high demand for performance is somewhat mitigated by the fact that physics-based simulation has tremendous amounts of parallelism. Chip multiprocessors [ONH96a] and other multi-threaded processor designs have the potential to exploit this, but certain parts of physics simulation are more amenable to parallelization than others.

In this section, we analyze broad-phase and narrow-phase collision detection. We propose and evaluate two techniques to leverage object locality: object-pair based branch prediction and object-pair table. The object-pair table targets broad-phase collision algorithms which require a complete update between simulation steps such as spacial hashing. We consider the hardware cost of such a scheme, and the impact of misprediction.

This study makes use of 3 benchmarks from PhysicsBench 1.0: CrashWall, Battle, and Battle2. The behavior of different entity types affect collision detection computation in different ways. Bricks that make up walls produce stacking behavior. Humans represent highly articulated objects. Projectiles are small, fast moving objects, and cars are fast moving large objects.

7.3.1 Application-Level Correlation and Locality

Intuitively, the low-level behavior of physics simulation code should have a great deal of locality and correlation with higher-level application constructs. Motion at a simulation step by step basis should be smooth and collisions between particular objects can persist across many steps – unless they are from objects moving extremely fast. Behavioral locality in the form of temporal coherence has been shown by prior collision detection work [LC91, KHM98, LC98, EL01]. Temporal coherence describes

the fact that in dynamic environments, objects do not move large distances between consecutive steps of physics simulation, except for objects with high velocities.

However, such collision locality is difficult to extract by conventional program counter based methods because of the diversity of objects using the same set of engine code. If the notion of higher-level application constructs can be communicated down to the architectural level, new forms of locality and correlation can be exploited.

To better demonstrate this notion of locality and correlation with high-level application constructs, we focus on the collision detection part of the physics simulation loop, and show how the notion of object-pairs can be leveraged

7.3.1.1 Collision Detection

Collision Detection is a well studied problem and an excellent collection of theory and available software can be found at [GAM]. The brute force approach to CD would be to compare each object with all of the other objects which results in $O(NxN)$ complexity, where N is the total number of objects. The most common approach to speed up CD is to use a two-step algorithm [Hub95] composed of *broad-phase* and *narrow-phase*.

7.3.1.2 Broad-Phase CD

Broad-phase refers to the first step of CD which efficiently culls away pairs of objects that cannot possibly collide. This step uses a fast approximation test to quickly prune pairs from the total NxN pairs. Most broad-phase methods use hierarchical bounding volumes. The object is enclosed inside a sufficiently large volume of simpler shape. This volume is used instead of the object's true shape by broad-phase for fast, but approximate collision detection. The pairs which pass broad-phase are passed to narrow-phase.

7.3.1.3 Narrow-Phase CD

Narrow-phase CD determines the exact intersection points between two objects. Each pair's computational load depends significantly on the geometric properties of the objects involved, and the overall performance is affected by the ability of broad-phase to minimize the required number of comparisons.

A recent study [LCF05] examines the trade-off of broad-phase accuracy vs total computation time for different broad-phase methods. The results show that broad-phase needs to be very fast, even at the expense of generating a larger number of collision pairs because of the dependencies between not only broad-phase CD and narrow-phase CD, but also between CD as a whole and the forward dynamics step.

7.3.1.4 Correlation and Locality Exploration

In a sense, the branches in both broad-phase and narrow-phase CD are conditioned on whether or not the two objects under consideration collide. Broad-phase CD attempts to filter out pairs of objects that are easily identified as not being able to collide together. Narrow-phase CD takes this filtered set of object pairs and determines whether or not they *actually* collide and where the collision occurs.

At the application level, there are some factors that can be correlated to help branch prediction in CD. For example, objects that collide tend to collide for several physics simulation steps, unless they are moving extremely fast. High level objects may also have repeated collision patterns. One simple example would be a human walking where one foot stays in contact with the ground for some time until the other foot lands, and the whole process repeats. Due to the dynamic nature of objects and the flexibility inherent in a physics engine, it is difficult to capture this locality entirely in software.

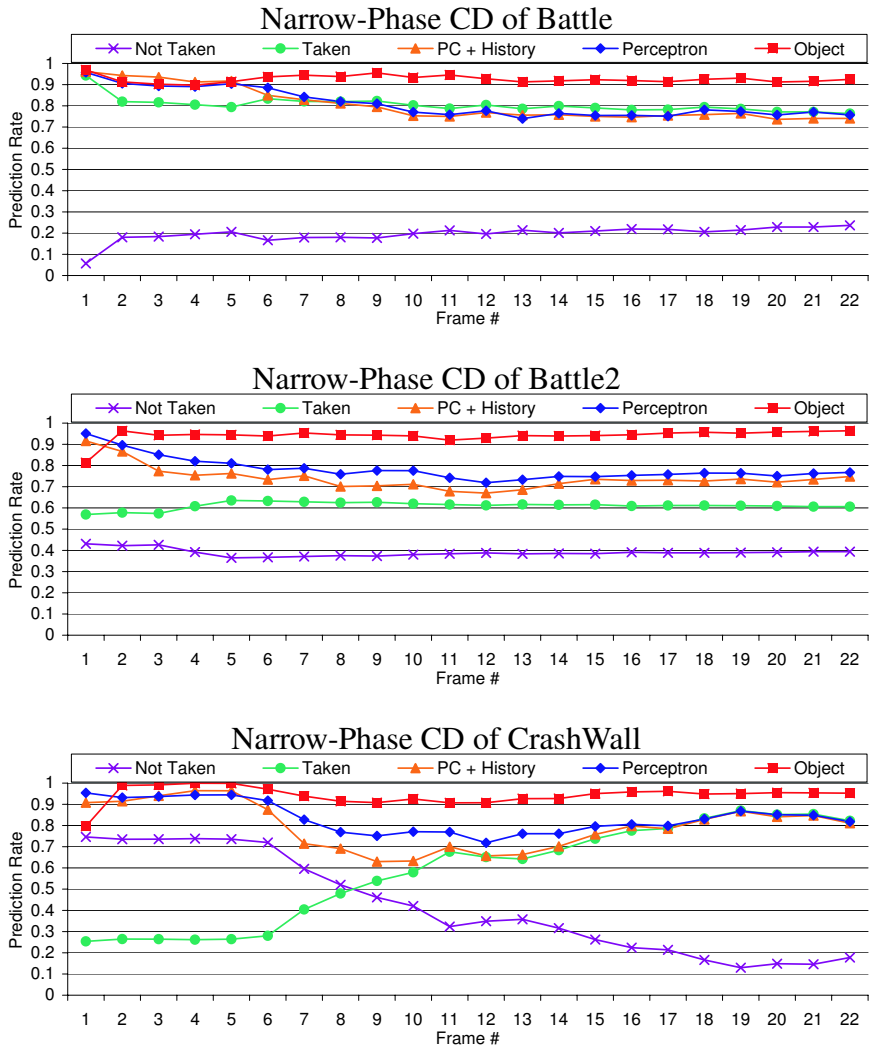


Figure 7.4: Narrowphase branch prediction rate correlation with the program counter (PC), branch history, and high-level objects.

To explore this further, we demonstrate how predictable the main conditional branches of narrow-phase CD are when correlated with the PC of the branch, branch history, and the two objects being compared. Figure 7.4 presents data for the three benchmarks from PhysicsBench 1.0 (CrashWall, Battle, and Battle2). The y-axis in each graph represents the branch prediction accuracy during the execution of the frame (only counting the predictions of correct path instructions). Each frame in the execution of the benchmark is tracked along the x-axis. We evaluate five architectures: a simple Not Taken predictor (always guess branch not taken), a simple Taken predictor (always guess branch taken), a gshare [McF93] predictor (PC + History) (16KB), a Perceptron branch predictor [JL02] (16KB), and an object-pair predictor (16KB). This latter predictor correlates with the base address of the two objects under consideration. The object-based predictor clearly outperforms any other option by a margin.

7.3.2 Branch Prediction for CD

Based on the branch correlation with object-pair addresses we saw in section 7.3.1, we propose an approach to leverage this information at the architecture level and improve control speculation for collision detection.

7.3.2.1 Object-Pair Based Branch Prediction

There has been an enormous amount of prior work on branch direction prediction. And while schemes have addressed correlating with global behavior [YP93], reducing aliasing [McF93, EM98], improving correlation [JL02], or even focusing on specific branch types [CHP97], all of these techniques rely on information present in the architecture to help make an accurate prediction. The majority of approaches are based on some combination of the program counter (PC) and either local or global branch history. All of these approaches are orthogonal to our goal, which is to improve corre-

lation and locality using application-level information in the architecture.

We consider indexing a pattern history table (PHT) - a table of two-bit counters indicating a prediction of taken/not taken - using the base address of the current object-pair under consideration. In collision detection, objects are compared in pairs, and therefore we will augment the architecture with two registers to hold the base addresses in memory of the two objects under consideration at any point in time. These registers will not be visible to the compiler, and techniques to set them will be discussed in section 7.3.2.

While object-pair information correlates well with certain control decisions, it does not work for others. We must also consider how to avoid using object-pair information in cases where it is not helpful. One approach is to leverage existing confidence techniques to selectively use the object-pair register only in cases where it is useful. Consider a simple pattern history table (PHT) indexed via PC and the object-pair register. This can be used in concert with a conventional PC-based PHT. An additional PHT can be used to select which PHT to use for a given prediction, just as in the bi-mode predictor [LCP97].

7.3.2.2 Loading the Object Registers

While high-level application information can dramatically improve low-level architectural correlation and locality for physics simulation, the question remains how to communicate this information from the application to the architecture.

The most straightforward technique conceptually, would be to augment the ISA with new instructions that propagate high-level information. The use of specialized move instructions, for example, would allow the compiler or application writer to place values into the object registers or clear the object registers.

Another approach might be to leverage the flexibility of ISA-specific address generation instructions for this purpose. For example, the x86 ISA features the `load effective address (LEA)` instruction. `LEA` is used to set a register with the value of an address computation. This address computation can take the form of any of the addressing modes supported by x86. For example `LEA edx, [esi+4*ebx]` would place 4 bytes of data at address `ESI+4*EBX` into `EDX`. We can use this instruction to set the object registers with the effective address calculated by LEA. We can change the architectural implementation of LEA so that in addition to writing to the register specified by the instruction itself, it will also implicitly write to one of the two object registers. The syntax of the effective address computation will determine which of the two will actually be written. Note that we are *not* making the object registers visible to the compiler with this approach – the LEA instruction from the perspective of the ISA need not change, we are simply enhancing it in the microarchitecture.

By using an existing instruction, we avoid adding opcodes to the ISA, but we may need to restrict the use of LEA. This can either be a complete restriction where LEA is only used for the purpose of setting the object registers, or a partial restriction where certain addressing modes are dedicated for the purpose of setting the object registers.

7.3.3 Increasing Parallelism for CD with the Object Table

High-level application information can also be leveraged by adding application-specific structures that accelerate certain components of execution. For example, a texture cache [HG97] is an application-specific structure that helps GPUs achieve higher performance. Another example is network processors, which use content-addressable memories for fast searches [Ageb]. In this section, we propose an application-specific structure to add parallelism to collision detection.

To show the performance potential of decoupling collision detection by the use of

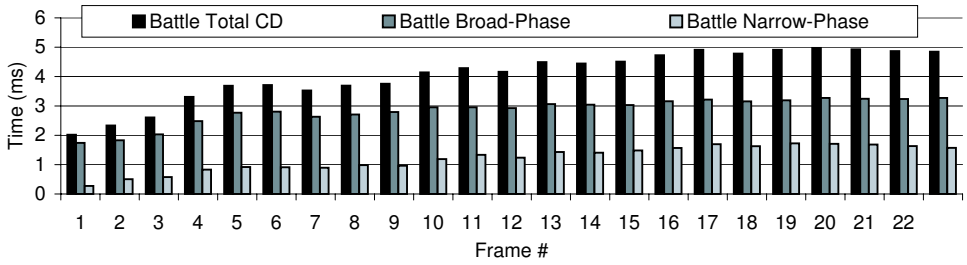


Figure 7.5: Battle Execution Time Breakdown for Collision Detection

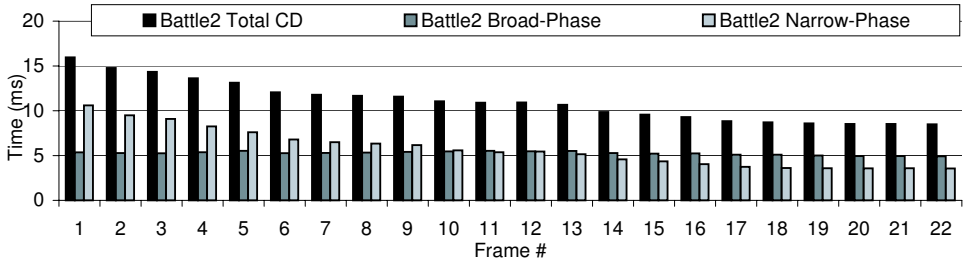


Figure 7.6: Battle2 Execution Time Breakdown for Collision Detection

object-pair filter, we measured the contribution of both broad-phase and narrow-phase on collision detection's execution time. Figures 7.5, 7.6, and 7.7 show the execution time breakdown for collision detection on a 2.8 GHz Pentium 4. It is interesting to note the variation in the proportional amount of time spent in broad-phase versus narrow-phase for these different benchmarks. Collision detection for CrashWall is dominated by the runtime of narrow-phase CD, Battle is dominated by broad-phase CD, and Battle2 is somewhat evenly distributed. One way to reduce the overall runtime of CD is

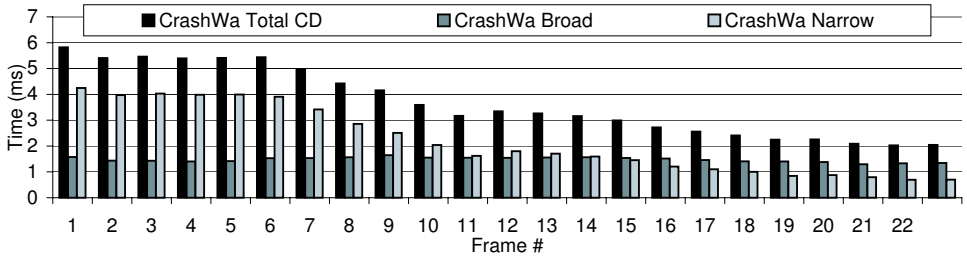


Figure 7.7: CrashWa Execution Time Breakdown for Collision Detection

to overlap broad-phase and narrow-phase as much as possible. This would allow us to optimally get a total CD runtime that is the maximum latency of the two components. Since broad-phase is effectively a technique to filter the work done by narrow-phase, it should also be possible to dynamically balance the amount of work done by each of these. In the rest of this section, we will explore a technique to provide this parallelism.

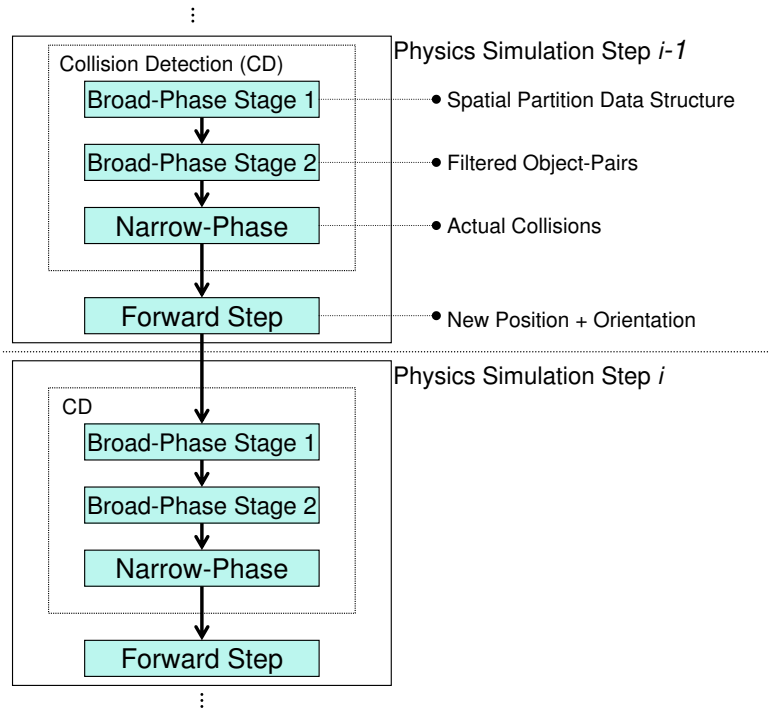


Figure 7.8: Collision Detection's Role in the Physics Simulation Flow

Figure 7.8 shows the inter-task dependencies for existing collision detection methods. Both narrow-phase CD and broad-phase CD must wait on the forward dynamics from the previous step. The forward dynamics of one step relies on the narrow-phase CD of that same step – and it cannot form islands (clusters of colliding objects) and begin the actual physics simulation until the narrow-phase CD has completed all object pairs. Narrow-phase CD depends on the output of broad-phase CD. Note that this is shown for an arbitrary step in the calculation of a single frame – there would also be

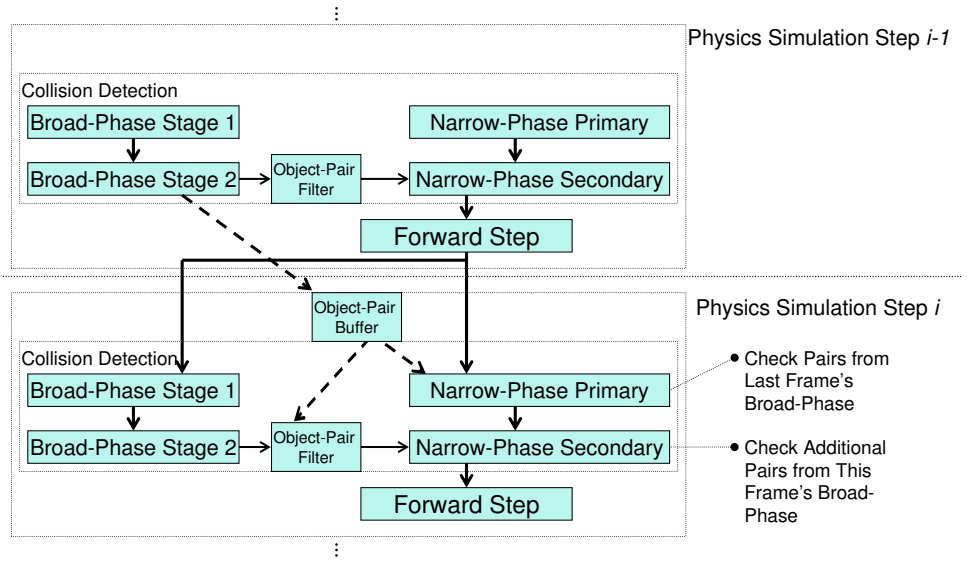


Figure 7.9: Collision Detection Flow with Decoupled Broad-Phase and Narrow-Phase

information streaming from step $i-2$ to step $i-1$ in the same manner that step $i-1$ streams to step i.

While different object-pairs can perform bounding box filtering and narrow-phase CD in parallel, broad-phase stage 1 (the update of spatial partitioning data structures) is serial with respect to all other components. This serial task will represent an increasingly larger amount of the total execution time as the number of objects in the virtual world and the number of processor cores on-die increases for future CMPs. This observation is confirmed by Luque et al [LCF05] – they conclude that broad-phase CD must be done as efficiently as possible for good overall physics performance, even if it means filtering fewer object pairs.

In section 7.3.1, we observed that there is considerable locality in CD. One approach to creating more parallelism in CD would be to use the result of broad-phase CD from *one* step to feed the narrow-phase CD of the *next* step. This would allow the broad-phase and narrow-phase components of CD to be done in parallel. However, there are two problems with this approach. First, it is certainly possible that the result

of broad-phase CD from a *previous* step before does not include all pairs from the *current* result of broad-phase CD. Narrow-phase CD only uses pairs from broad-phase, so this is clearly a correctness issue where we may miss a collision. Therefore, we would like to add a correction mechanism that puts any extra pairs detected in the *current step's* broad-phase CD through narrow-phase CD. We will refer to these extra pairs that must be done serially as *serial narrow-phase comparisons*.

Second, there may be pairs that were in the *previous* step's broad-phase result that are not in the current step's broad-phase result. Because narrow-phase will verify any pairs from broad-phase, this is not a correctness problem – but if too many extra pairs are added, we may lose the benefit from parallelization. We will refer to pairs that are in the former step's broad-phase CD result but not in the current step's result as *unnecessary narrow-phase comparisons*.

The overall flow of this new approach to CD is shown in figure 7.9. We are effectively decoupling broad-phase and narrow-phase CD as much as possible to improve parallelism. We split narrow-phase CD into two components: a primary stage that handles the speculative set of object pairs from the prior step's broad-phase CD and a secondary stage that handles any serial narrow-phase comparisons. Note that step i's primary stage of narrow-phase CD is using step $i-1$'s broad-phase result. We use a simple object-pair buffer to queue pairs from one step to the other – we will discuss the size of this structure a little later in this section. Step i's secondary stage of narrow-phase CD is using step i's broad-phase CD as input. This broad-phase CD result can potentially have pairs that were already given to the primary stage of narrow-phase CD – we will call these *redundant narrow-phase comparisons*. Redundant comparisons are not functionally incorrect, but can potentially negate any performance gain. The critical component of this new approach to CD is how to efficiently filter these redundant narrow-phase comparisons at the output of broad-phase CD. In the diagram, we

refer to this generally as an *object-pair filter*, but in the next section we will investigate a particular design of this filter.

Note again that this is shown for an arbitrary step in the calculation of a single frame – there would also be information streaming from step $i-2$ to step $i-1$ in the same manner that step $i-1$ streams to step i. The object-pair filter would have been filled from the broad-phase calculation of step $i-2$. The initial step in a frame would leverage information from the last step of the previous frame.

Figure 7.10 shows the increase in both unnecessary narrow-phase comparisons and serial narrow-phase comparisons for our three benchmarks. The y-axis shows the percent increase in object pairs relative to the total number of pairs that would have been passed directly from broad-phase in the normal CD flow (figure 7.8). For all frames simulated, this never grows above 4% on average.

To model the increase in redundant narrow-phase comparisons, we must consider the actual implementation of the object-pair filter. We could use a software-based filter here, but the cost of storing all of the object pairs to memory could interfere with the locality of other data blocks. We instead propose a hardware structure that can efficiently filter redundant comparisons. These structures are mapped into a special section of memory so that application software can access them directly using conventional data transfer instructions.

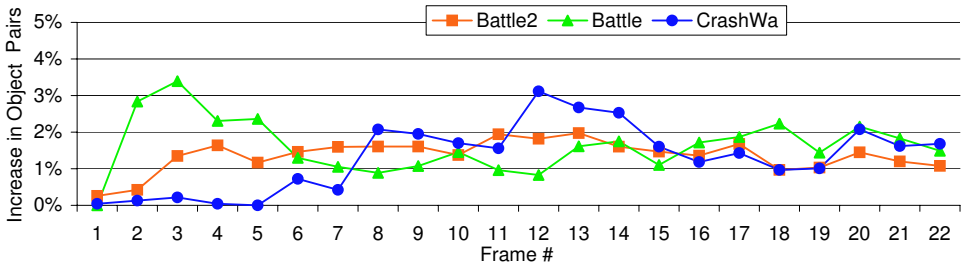
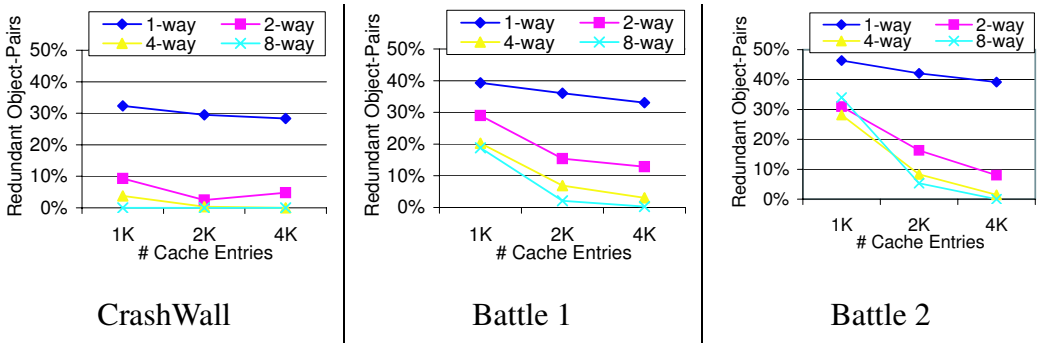


Figure 7.10: Unnecessary Narrow-Phase Comparisons and New Object-Pairs

7.3.3.1 Object-Pair Filter

The object-pair filter needs to determine, for a given pair of objects, whether or not these objects have already been communicated to the narrow-phase CD. And so given two object addresses, the filter gives a yes or no.

Due to the serial nature of broad-phase CD, it will be unlikely that more than one core on a future CMP will be handling this component of CD (unless the application contains relatively disjoint spaces). Therefore, we will likely only need a single object-pair filter for the results of broad-phase. However, this structure is challenging since it needs to contain a potentially large number of objects – this may make the use of a CAM structure expensive.



The object filter we evaluate is a cache-like structure (it has a number of sets and an associativity), but has a few differences from a conventional cache. First, on a miss, we do not index a second level structure – when an object pair misses in the filter, it is sent to narrow-phase CD. Second, we never evict anything from the filter until the end of the current physics step. If we run out of room in the filter, we simply disregard the object pairs that do not fit in the filter. In the worst case, these object pairs will be processed twice by narrow-phase CD. In order to ensure correctness, the software must prevent duplicated object pairs from creating multiple contacts at the same positions. Each filter entry only stores the two 32-bit addresses - if it is in the filter, it does not

need to be sent to narrow-phase CD. If it is not in the filter, it was either not sent to narrow-phase CD or could not fit in the filter, and will be sent to narrow-phase CD.

To reduce thrashing in the filter, we use two filters together. Each filter employs a different hash, but both filters have the same number of entries. For a pair of object filters with n sets, the primary filter takes $logn$ bits from each object address and xor's these to form the index into the filter. The secondary filter takes $\frac{logn}{2}$ bits from each object address and concatenates them to form a $logn$ bit index. When the broad-phase CD output is written to the object filter, we first use the primary filter until the set we are writing to has filled. We do *not* evict pairs from a set, but instead use the secondary filter to find an alternative location to place the pair. If the corresponding set in the secondary filter is not full, we write the object pair to the secondary filter. This helps to better distribute the sets that heavily thrash. On an access to the filter (when broad-phase CD is determining what to send to the narrow-phase CD within its own step), both filters are checked in parallel – each using its own hash function.

At the end of the step, all filter entries are invalidated, and the filters are refilled using the object buffer.

Figure 7.3.3.1 shows the performance of our cache when varying the total number of object-pairs it can hold and the associativity of the cache.

7.3.3.2 Further Reduction in Size

A naive implementation of this filter would store pairs of 32-bit addresses for each entry. To reduce the storage requirement, we can utilize a dictionary table to map 32-bit addresses to a much smaller object number. Now, the dictionary table stores all unique objects' 32-bit addresses, with the index of the entry as the implicit object number. The object-pair filter then just stores a pair of object numbers, but requires translation from the 32-bit addresses to object numbers in order to access the filter.

7.3.3.3 Object-Pair Buffer

The object-pair buffer needs to be able to hold incoming object-pairs for the primary narrow-phase CD from the broad-phase CD of the previous step (see figure 7.9). There are two ways to do this in a CMP environment: 1) the buffer is distributed among the cores responsible for narrow-phase CD or 2) the buffer is centralized at the core responsible for broad-phase CD. As mentioned for the object-pair filter, it is unlikely that more than one core will be doing broad-phase CD. In either approach, the worst-case number of object-pairs in the examples we looked at was 1700. This would require a total buffer capacity of 14KB. However, this is a simple FIFO buffer since it does not require CAM logic.

CHAPTER 8

Conclusion and Future Directions

Interactive entertainment applications are rapidly gaining significance from both technical and economical point of views. In this dissertation, we present a holistic exploration to accelerate one core component of this emerging workload, namely real-time physics simulation or physics based animation. Our contributions can be categorized into benchmark creation, workload characterization, architectural acceleration, algorithmic acceleration, and architectural exploitation of algorithmic properties.

To represent this emerging workload, we developed and refined PhysicsBench, a set of benchmarks to capture the complexity and scale of physics simulation in interactive entertainment applications. The 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 utilize a large set of features often used found in these applications. PhysicsBench can be leveraged by both computer architects and application designers to explore hardware or software optimizations. To support the studies describe earlier, we have generated binaries for Alpha, MIPS, SPARC, and x86 ISAs.

Using the PhysicsBench suite, we characterized real-time physics simulation (RTPS) with simulators (SimpleScalar [BA97], Ptlsim [You], Simics/Gems [MCE, MSB], and SESC [RFT05]) as well as real multi-core x86 machines. The suite has been parallelized using both POSIX threads and the SESC API. After comparing to conventional workloads such as SPEC and MediaBench, we have identified RTPS's 5 key differenti-

ating factors. First, its real-time constraint requires consistent, high performance. This deems certain predictive techniques to be ineffective. Second, error tolerance allows incorrect computation to still produce valid result. This tolerance applies to both data-flow and control-flow operations. Third, significant execution time is spent on serial, thread parallel (coarse-grain), and data parallel (fine-grain) sections. This behavior suggests the requirement of optimizing all components. Fourth, there is tight feedback between data parallel (fine-grain) and thread parallel (coarse-grain) components which exposes communication latency. Finally, RTPS has easily identifiable, explicit computation phases. While some of these features overlap with either SPEC or MediaBench, RTPS combines unique traits with different aspects of established workloads.

Based on the workload characterization, we propose ParallAX, an architecture to sustain interactive frame rates for real-time physics. The ParallAX architecture is a heterogeneous CMP design 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.

As exemplified by the simulations for ParallAX, more active cores per chip increases the load on the lowest-level cache. To alleviate cache thrashing from parallel threads in workloads such as physics simulation, SPEC, or MediaBench, we propose the Performance Driven Adaptive Sharing (PDAS) cache design. PDAS is a scalable,

multi-ported NUCA that dynamically allocates its distributed cache resources through an intelligent, realizable on-line partitioning strategy. We achieve improved partitions by taking actual performance into account rather than just the miss rate. PDAS guarantees a minimum performance bound for each core while pursuing high throughput.

While perceptual error tolerance of physics simulation has been studied, no prior work has addressed the evaluation of complex scenes often used in IE applications. Using previous perceptual error tolerance studies of simpler scenes as a starting point, we extrapolate from these results a methodology for evaluation of complex scenes. This work is the first to bring together the speculative perceptual studies of previous years into a practical framework and evaluate their potential usefulness. It closes the loop between perceptual studies and practical applications. We conclude that total energy is the main metric to consider when evaluating the output quality of physics simulation. This study evaluates the maximum tolerable error from three sources: random numerical errors, precision reduction errors, and time-step tuning errors.

Furthermore, we propose the algorithmic optimization of Fast Estimation with Error Control (FEEC). FEEC executes two simulation worlds. While the fast, estimate simulation generates intermediate results for dependent software components, the slow, precise simulation feeds accurate results to both simulation worlds at the end of each step. This mechanism allows for the use of intermediate results while correcting all errors at the end of each step.

Leveraging our findings from the workload characterization and the perceptual error tolerance study, we propose architectural techniques to exploit algorithmic properties of the physics workload, namely perceptual error tolerance and the notion of object-pairs. For perceptual error tolerance, we evaluate the effects of reduced precision floating-point computation on core area reduction as well as improving the effectiveness of trivialization and memoization. With regard to object-pairs, we propose a

object-pair branch predictor and the object-pair table, an application specific structure to increase the amount of parallelism for collision detection.

To summarize, this dissertation is an in-depth study on the acceleration of real-time physics simulation. We created representative benchmarks, characterized its behavior, proposed an architecture, proposed a methodology to analyze perceptual error tolerance, and developed a set of architectural and algorithmic techniques for acceleration. Given physics' overlap of constraints and behaviors with other software components, the proposed methodologies and techniques can be applied to other components of IE applications. For example, physics' spatial partitioning for collision detection pruning are also required for real-time artificial intelligence (AI) [Ope] and real-time ray-tracing (RT) [Wal04]. In addition, the data flow model representing the physics pipeline as shown in Figure 5.1 resembles those of AI and RT.

There are many directions to extend the work presented in this dissertation. Some immediate extensions include the exploration of additional software components such as AI or RT, evaluation of precision reduction on FPU sharing among cores, evaluation of object-pair correlating branch predictor, and fuzzy branch recovery, where branch misprediction recovery is avoided for selected control flow decisions.

REFERENCES

[360] Microsoft Xbox 360. http://www.xbox360.com/.

[3DM] Futuremark Homepage. http://www.futuremark.com.

[AB05] E. Atoofian and A. Baniasadi. "Improving Energy-Efficiency by Bypassing Trivial Computations." In *The 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[AB06] J. Andrews and N. Baker. "Xbox 360 System Architecture." In *IEEE Computer Society*, 2006.

[ACV05] C. Alvarez, J. Corbal, and M. Valero. "Fuzzy memoization for floating-point multimedia applications." In *IEEE Transactions on Computers*, 2005.

[agea] http://www.ageia.com/.

[Ageb] Agere. "Building Next Generation Network Processors." www.agere.com/telecom/docs/.

[AHK00] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. "Clock rate vs. IPC: The end of the road for conventional microprocessors." In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[Alb99] D. Albonesi. "Selective Cache Ways: On-Demand Cache Resource Allocation." In *32st International Symposium on Microarchitecture*, November 1999.

[Arm] U.S. Army. "The Official U.S. Army Game: America's Army. http://www.americasarmy.com/.".

[BA97] D. C. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[Bar97] David Baraff. *Physically Based Modeling: Principals and Practice*. SIGGRAPH Online Course Notes, 1997.

[BGM00] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing." In *27th Annual International Symposium on Computer Architecture*, 2000.

[BHW96] R. Barzel, J.F. Hughes, and D.N. Wood. “Plausible motion simulation for computer graphics animation.” In *Computer Animation and Simulation*, 1996.

[Boh95] M. Bohr. “Interconnect Scaling - The Real Limiter to High-Performance ULSI.” In *Tech. Dig. of the International Electron Devices Meeting*, pp. 241–244, December 1995.

[Bro] Broadcom. “BCM1480 product brief.” www.broadcom.com/collateral/pb/1480-PB02-R.pdf.

[BW98] David Baraff and Andrew Witkin. “Large steps in cloth simulation.” In *SIGGRAPH ’98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 43–54, New York, NY, USA, 1998. ACM Press.

[BW04] B. Beckmann and D. Wood. “Managing Wire Delay in Large Chip-Multiprocessor Caches.” In *The 37th Annual IEEE/ACM International Symposium on Microarchitectur*, 2004.

[CCC02] C. Cascaval, J. Castanos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. Moreira, K. Strauss, and Jr. H. Warren. “Evaluation of a Multithreaded Architecture for Cellular Computing.” In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, 2002.

[CF98] D. Citron and D. Feitelson. “Accelerating Multi-Media processing by Implementing Memoing in Multiplication and Division Units.” In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[CF00] S. Chenney and D. Forsyth. “Sampling plausible solutions to multi-body constraint problems.” In *ACM SIGGRAPH*, 2000.

[CF03] 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 *Power-Aware Computer Systems*, 2003.

[CH97] D.A. Carlson and J.L. Hodgins. “Simulation levels of detail for real-time animation.” In *Proceedings of Graphics interface*, 1997.

[CHP97] P. Chang, E. Hao, and Y. Patt. “Target Prediction for Indirect Jumps.” In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 274–283, June 1997.

[CJD00] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. "Dynamic Cache Partitioning via Columnization." In *Proceedings of Design Automation Conference, Los Angeles*, June 2000.

[CJY02] Johnny T. Chang, Jingyi Jin, and Yizhou Yu. "A practical model for hair mutual interactions." In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 73–80, New York, NY, USA, 2002. ACM Press.

[CK02] Kwang-Jin Choi and Hyeong-Seok Ko. "Stable but responsive cloth." In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 604–611, New York, NY, USA, 2002. ACM Press.

[CLR92] T. Cormen, C. Leiserson, and R. Rivest. "Introduction to Algorithms." 1992.

[CNE] CNET. "Playstation 3: the next generation. http://news.com.com/2100-1040-866288.html." http://news.com.com/2100-1040-866288.html.

[CPV03] Z. Chishti, M. Powell, and T. Vijaykumar. "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures." In *36th International Symposium on Microarchitecture*, 2003.

[CRT99] B. Calder, G. Reinman, and D.M. Tullsen. "Selective Value Prediction." In *26th Annual International Symposium on Computer Architecture*, pp. 64–74, June 1999.

[CS] R. W. Crandall and J. G. Sidak. "Video Games Serious Business for America's Economy.".

[DS02] A. Dhodapkar and J. E. Smith. "Managing Multi-Configuration Hardware via Dynamic Working Set Analy sis." In *29th Annual International Symposium on Computer Architecture*, May 2002.

[EAE02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. "Tarantula: a vector extension to the alpha architecture." In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pp. 281–292, Washington, DC, USA, 2002. IEEE Computer Society.

[EBL05] D. Etiemble, S. Bouaziz, and L. Lacassagne. "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*, 2005.

[EEL04] J. Eilert, A. Ehliar, and D. Liu. "Using Low Precision Floating Point Numbers to Reduce Memory Cost for MP3 Decoding." In *IEEE 6th Workshop on Multimedia Signal Processing*, 2004.

[EL01] S. Ehmann and M. Lin. "Accurate and fast proximity queries between polyhedra using convex surface decomposition." In *Computer Graphics Forum*, 2001.

[EM98] A. Eden and T. Mudge. "The YAGS branch predictor." In *31st International Symposium on Microarchitecture*, December 1998.

[Eng] Open Dynamics Engine. http://ode.org/.

[ESA] ESA. "Entertainment Software Association.".

[FCR02] F. Fang, T. Chen, and R.A. Rutenbar. "Floating-point Bit-width Optimization for Low-power Signal Processing Applications." In *International Conference on Acoustic, Speech, and Signal Processing*, 2002.

[FL04] Raanan Fattal and Dani Lischinski. "Target-driven smoke animation." *ACM Trans. Graph.*, **23**(3):441–448, 2004.

[FOA03] Bryan E. Feldman, James F. O'Brien, and Okan Arikan. "Animating suspended particle explosions." *ACM Trans. Graph.*, **22**(3):708–715, 2003.

[GAM] GAMMA Team, University of North Carolina. "Collision Detection." www.cs.unc.edu/geom/collide.

[Gol91] D. Goldberg. "What every computer scientist should know about floating-point arithmetic." In *ACM Computing Surveys (CSUR)*, 1991.

[GRE01] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. "MiBench: A free, commercially representative embedded benchmark suite." In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[GST06] 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.

[Hava] Havok. http://www.havok.com/.

[Havb] Havok. http://www.havok.com/content/view/187/77.

[HBK01] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. "Exploring the Design Space of Future CMPs." In *Proceedings of the 2001 International Conference on Parallel A rchitectures and Compilation Techniques*, pp. 199–210. IEEE Computer Society, 2001.

[HG97] Ziyad S. Hakura and Anoop Gupta. "The design and analysis of a cache architecture for texture mapping." *SIGARCH Comput. Archit. News*, **25**(2):108–120, 1997.

[HL99] J. Huang and D. Lilja. "Exploiting Basic Block Locality with Block Reuse." In *International Symposium on High Performance Computer Architecture (HPCA)*, 1999.

[Hof05] P. Hofstee. "Power Efficient Architecture and the Cell Processor." In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, 2005.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[HRP04a] J. Harrison, R. A. Rensink, and M. van de Panne. "Obscuring Length Changes During Animated Motion." In *ACM Transactions on Graphics, SIGGRAPH 2004*, 2004.

[HRP04b] Jason Harrison, Ronald A. Rensink, and Michiel van de Panne. "Obscuring length changes during animated motion." *ACM Trans. Graph.*, **23**(3):569–573, 2004.

[HS98] S. Hily and A. Seznec. "Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading." In *Proceedings of MTEAC'98 Workshop*, 1998.

[HTX] HyperTransport http://www.hypertransport.org/docs/tech/HT3pres.pdf.

[Hub95] P. Hubbard. "Collision detection for interactive graphics applications." In *IEEE Transactions on Visualization and Computer Graphics*, 1995.

[Ini] Serious Games Initiative. http://www.seriousgames.org/.

[Jak01] T Jakobsen. "Advanced character physics using the fysix engine." www.gamasutra.com, 2001.

[JL02] D. Jimenez and C. Lin. "Neural Methods for Dynamic Branch Prediction." In *ACM Transactions on Computer Systems, Vol. 20, No. 4*, November 2002.

[KBK02] C. Kim, D. Burger, and S. Keckler. "An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches." In *ASPLOS10*, 2002.

[KCS04] S. Kim, D. Chandra, and Y. Solihin. "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture." In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[Kel] Brian Keller. "XNA Studio: Introduction to XNA." Game Developer Conference 2006.

[KHM98] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K.Zikan. "Efficient collision detection using bounding volume hierarchies of k-DOPS." In *IEEE Transactions on Visualization and Computer Graphics*, 1998.

[KP02] Christoforos Kozyrakis and David Patterson. "Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks." In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 283–293, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[KP03] C. Kozyrakis and D. Patterson. "Overcoming the limitations of conventional vector processors." 2003.

[KPP97] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanović, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. "Scalable Processors in the Billion-Transistor Era: IRAM." *Computer*, **30**(9):75–78, 1997.

[KRD03] 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.

[LC91] M. Lin and J. Canny. "A Fast Algorithm for Incremental Distance Calculation." In *IEEE Int. Conf. on Robotics and Automation*, 1991.

[LC98] T.-Y. Li and J.-S. Chen. "Incremental 3D collision detection with hierarchical data structures." In *VRST*, 1998.

[LCF05] R. Luque, J. Comba, and C. Freitas. "Broad-phase collision detection using semi-adjusting BSP-trees." In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pp. 179–186, New York, NY, USA, 2005. ACM Press.

[LCP97] C. Lee, I. Chen, and Y. Patt. "The Bi-Mode Branch Predictor." In *30th International Symposium on Microarchitecture*, 1997.

[LEO05] L. Lacassagne, D. Etiemble, and S.A. Ould Kablia. "16-bit floating point instructions for embedded multimedia applications." In *Seventh International Workshop on Computer Architecture for Machine Perception*, 2005.

[LL00] John E. Laird and Michael van Lent. "Human-Level AI's Killer Application: Interactive Computer Games." In *AAAI/IAAI*, pp. 1171–1178, 2000.

[LMT04] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. "The Stream Virtual Machine." *pact*, **00**:267–277, 2004.

[LNM04] X. Li, H. Negi, T. Mitra, and A. Roychoudhury. "Design space exploration of caches using compressed traces." In *Proceedings of the 18th annual international conference on Supercomputing*, pp. 116–125. ACM Press, 2004.

[LSK04] C. Liu, A. Sivasubramaniam, and M. Kandemir. "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs." In *International Symposium on High-Performance Computer Architecture (HPCA-10)*, February 2004.

[LWS96] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. "Value Locality and Load Value Prediction." In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[MCE] 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*.

[McF93] S. McFarling. "Combining Branch Predictors." Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[MDC06] R. McDonnell, S. Dobbyn, S. Collins, and C. O'Sullivan. "Perceptual Evaluation of LOD Clothing for Virtual Humans." In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 117–126, 2006.

[MGA03] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. "Cg: a system for programming graphics hardware in a C-like language." In *ACM SIGGRAPH*, 2003.

[MGR05] V. Moya, C. Gonzalex, J. Roca, A. Fernandez, and R. Espasa. "Shader Performance Analysis on a Modern GPU Architecture." In *MICRO 38: Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, 2005.

[MGT99] C. Molina, A. Gonzalez, and J. Tubella. "Dynamic Removal of Redundant Computations." In *International Conference on Supercomputing (ICS)*, 1999.

[MHH06] 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.

[Mic04] P. Michaud. "Exploiting the Cache Capacity of a Single-Chip Multi-Core Processor With Execution Migration." In *10th International Symposium on High Performance Computer Architecture*, 2004.

[MSB] 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*.

[MTP04] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. "Fluid control using the adjoint method." *ACM Trans. Graph.*, **23**(3):449–456, 2004.

[MWG04] B. Matthews, J. Wellman, and M. Gschwind. "Exploring Real Time Multimedia Content Creation in Video Games." In *6th Workshop on Media and Streaming Processors*, 2004.

[N 05] D. Tullsen N. Tuck. "Multithreaded Value Prediction." In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, 2005.

[New] Newton. http://www.newtondynamics.com/.

[OD01] C. O'Sullivan and J. Dingliana. "Collisions and Perception." In *ACM Transactions on Graphics*, 2001.

[ODG03] C. O'Sullivan, J. Dingliana, T. Giang, and M. Kaiser. "Evaluating the visual fidelity of physically based animations." In *ACM Transactions on Graphics, SIGGRAPH*, 2003.

[OHM04] C. O'Sullivan, S. Howlett, R. McDonnell, Y. Morvan, and K. O'Conor. "Perceptually Adaptive Graphics." In *Eurographics 2004, State of the Art Report*, 2004.

[ONH96a] K. Olukoton, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. "The Case for a Single-Chip Multiprocessor." In *ASPLOS-VII*, 1996.

[ONH96b] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Cha ng. "The case for a single-chip multiprocessor." In *Proceedings of the seventh international conference on Archite ctural support for programming languages and operating systems*, pp. 2–11. ACM Press, 1996.

[Ope] OpenSteer. http://opensteer.sourceforge.net/.

[P] K. Olukotun P. Kongetira, K. Aingaran. "Niagara: A 32-Way Multithreaded Sparc Processor.".

[PCI] PCIE Express http://www.pcisig.com/specifications/pciexpress/.

[PF05] M. Pharr and R. Fernando. *GPU Gems 2*. Pearson Education, 2005.

[Pro] Dennis Proffit. "Viewing Animations: what people see and understand and what they don't." Keynote address, ACM SIGGRAPH/Eurographics Symosium on Computer Animation 2006.

[RAJ00a] P. Ranganathan, S. Adve, and N. Jouppi. "Reconfigurable Caches and their Application to Media Processing." In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[RAJ00b] 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.

[Rev] Nintendo Revolution. http://www.nintendo.com/.

[RFT05] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. "SESC simulator.", January 2005. http://sesc.sourceforge.net.

[Ric92] S. E. Richardson. "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation." In *International Symposium on Computer Arithmetic*, 1992.

[Ric93] S. E. Richardson. "Exploiting Trivial and Redundant Computation." In *Computer Arithmetic*, 1993.

[RP03a] Paul S. A. Reitsma and Nancy S. Pollard. "Perceptual metrics for character animation: sensitivity to errors in ballistic motion." *ACM Trans. Graph.*, **22**(3):537–542, 2003.

[RP03b] P.S.A. Reitsma and N.S. Pollard. "Perceptual Metrics for Character Animation: Sensitivity to Errors in Ballistic Motion." In *ACM Transactions on Graphics*, 2003.

[SA95] R. Sugumar and S. Abraham. "Set-associative cache simulation using generalized binomial trees." *ACM Trans. Comput. Syst.*, **13**(1):32–56, 1995.

[SDR02] G. Suh, S. Devadas, and L. Rudolph. "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning." In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[SEP93] D. M. Samani, J. Ellinger, E. J. Powers, and E. E. Jr. Swartzlander. "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*, 1993.

[SEW06] V. Soteriou, N. Eisley, H. Wang, B. Li, and L. Peh. "Polaris: A System-Level Roadmap for On-chip Interconnection Networks." In *Proceedings of the 24th International Conference on Computer Design*, 2006.

[SJ01] P. Shivakumar and N. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model." Compaq WRL 2001/2, 2001.

[Som] R. Sommefeldt. "Nvidia G80: Architecture and GPU Analysis." http://www.beyond3d.com/reviews/nvidia/g80-arch/.

[SPH02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior." In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[SR06] A. Seugling and M. Rolin. "Evaluation of Physics Engines and Implementation of a Physics Module ina 3d-Authoring Tool." In *Master's Thesis*, 2006.

[SRD04] G. E. Suh, L. Rudolph, and S. Devadas. "Dynamic Partitioning of Shared Cache Memory." *J. Supercomput.*, **28**(1):7–26, 2004.

[SS97] A. Sodani and G. Sohi. "Dynamic Instruction Reuse." In *International Symposium on computer Architecture (ISCA)*, 1997.

[SSC03] T. Sherwood, S. Sair, and B. Calder. "Phase Tracking and Prediction." In *30th Annual International Symposium on Computer Architecture*, June 2003.

[ST] P. Schmid and B. Topelt. "Game Over? Core 2 Duo Knocks Out Athlon 64." http://www.tomshardware.com/2006/07/14/.

[ST00] A. Snavely and D. M. Tullsen. "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor." In *In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[Sto] Jon Stokes. "Inside the Xbox 360.".

[STW92] H. S. Stone, J. Turek, and J. L. Wolf. "Optimal partitioning of cache memory." In *IEEE transactions on Computers*, 1992.

[Tai04] M. Taiji. "MDGRAPE-3 chip: a 165 Gflops Application Specific LSI for Molecular Dynamics Simulations." In *Hot Chips 16*, 2004.

[Tok] Tokamak. http://www.tokamakphysics.com/.

[Wal04] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[Wu05] D. Wu. "Physics in Parallel: Simulation on 7th Generation Hardware." In *Game Developers Conference*, 2005.

[YFP07] T. Y. Yeh, P. Faloutsos, S. Patel, and G. Reinman. "ParallAX: An Architecture for Real-Time Physics." In *The 34th International Symposium on Computer Architecture (ISCA-34)*, 2007.

[YFR06] T. Y. Yeh, P. Faloutsos, and G. Reinman. "Enabling Real-Time Physics Simulation in Future Interactive Entertainment." In *ACM SIGGRAPH Video Game Symposium*, 2006.

[YL02] 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.

[You] M. T. Yourst. "PTLsim User's Guide and Reference: The Anatomy of an x86-64 Out of Order Microprocessor." In *http://www.ptlsim.org*.

[YP93] Tse-Yu Yeh and Yale N. Patt. “A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History.” In *20th Annual International Symposium on Computer Architecture*, pp. 257–266, San Diego, CA, May 1993. ACM.

[YR] Thomas Y. Yeh and Glenn Reinman. “Fast and Fair: Data-stream Quality of Service.” In *CASES ’05: Proceedings of the 2005 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*.