Enabling Real-Time Physics Simulation in Future Interactive Entertainment

Thomas Y. Yeh Petros Faloutsos Glenn Reinman
Department of Computer Science, University of California, Los Angeles

Abstract

Interactive entertainment has long been one of the driving factors behind architectural innovation, pushing the boundaries of computing to achieve ever more realistic virtual experiences. Future entertainment applications will feature robust physics modeling to enable on-the-fly content creation. However, application designers must provide at least 30 graphical frames per second to provide the illusion of visual continuity. This constraint directly impacts the physics engine, which must deliver the results of physical interactions in the virtual world at a fraction of this frame rate. With more sophisticated applications combining massive numbers of complex entities, the cost of robust physics simulation will easily exceed the capability of today's most power machines.

This work explores the characteristics of real-time physics simulation, and proposes a suite of future-thinking benchmarks stressing different situations that represent the demands of future interactive entertainment. With this suite, we then explore techniques to help meet these demands, including parallel execution, a fast estimation approach that self-regulates error, and a value prediction technique that is allowed to get "close enough" to the real value. We demonstrate that parallel execution together with the proposed fast estimation approach can satisfy the demands of nearly all of the PhysicsBench suite.

CR Categories: I.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—; C.3 [Processor Architectures]: Multiple Data Stream Architectures—.

Keywords: interactive entertainment, real-time physics, benchmark, parallel execution, error tolerance

1 Introduction and Motivation

Interactive entertainment has grown to a substantial industry, with $7 billion in revenue for 2004. As the predominant form of interactive entertainment, gaming has driven mass demand for high-performance general purpose processors. Beyond entertainment, interactive gaming is being targeted for use in education, training, health, and public policy [Initiative]. An interesting example is the America's Army game, created by the United States Army for civilians to experience life as a soldier [U.S. Army]. Other creative uses include medical screening, fitness promotion, and hazmat training.

From a technical perspective, future games will be computationally intensive applications that involve various computation tasks: artificial intelligence, physics simulation, motion synthesis, scene database query, networking, graphics, audio, video, I/O, OS, tactile feedback, and GP game engine code. In order to provide the perception of smooth motion, gaming hardware typically produces 30 graphical *frames* every second.

Despite the social, economic, and technical importance of interactive entertainment applications, there has been very little academic effort to quantify their behavior and needs. Recent announcements of next generation game-consoles (Sony PlayStation 3 [CNET], Microsoft Xbox 360 [Microsoft], and Nintendo Revolution [Nintendo]) show 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. This spectrum of designs suggests that interactive entertainment software requirements are non-standardized and vary across genres.

In this work, we focus on the use of physics simulation in interactive entertainment. Such simulation has been augmenting recent applications, like HalfLife 2, but truly immersive virtual worlds with many interactive entities may prove far too computationally intensive for current microprocessors. Our contributions include the following:

- PhysicsBench, a suite of real-time physics benchmarks,
  - Analysis and comparison to other workloads.
  - Metrics for performance and error evaluation.
- Architectural exploration of PhysicsBench
  - How far are we from satisfying the 30 frame/sec requirement for interactive entertainment?
  - Exploration of parallel execution.
- Novel techniques to target high frame rates.
  - Fast Estimation with Error Control (FEEC)
  - Fuzzy Value Prediction (FVP)

The remainder of this paper is organized as follows: The rest of this section discusses related work. Section 2 introduces the challenges of physics simulation and the characteristics of its computational load. We propose our benchmark suite, *PhysicsBench*, in Section 3. Results from a real world processor running PhysicsBench are shown in Section 4, and we explore techniques to improve this performance further in Section 5. We conclude in Section 6.

1.1 Related Work

Throughout the paper we will explore work related to our techniques, however, there is little directly related work in interactive entertainment in the architecture community. [Matthews et al. 2004] compared the performance counter statistics of a single second's 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) [Olukoton et al. 1996] as a promising approach to providing performance.

Graphics Processing Units (GPUs) are specialized hardware cores designed to accelerate rendering and display. Because 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 accessed easily or efficiently by the CPU. This is problematic for physics simulation that works in a continuous feedback loop. In addition, graphics hardware is primarily designed to store 2D arrays (textures). This is suitable for computations involving grids (2D-fluids) but not 3D rigid bodies. Mapping constrained rigid body simulation to modern GPUs is not straightforward and is an active area of research.

2 Physics and Interactive Entertainment Applications

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 allowed interactive entertainment applications to approach cinematic quality. Unprecedented levels of visual quality and complexity in turn require high fidelity animation, and 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. Physics-based simulation provides physical realism and automated motion calculation, but also has greater computational cost, difficulty in object control, and potentially unstable results. We will focus on the computational cost of physical simulation, which can grow very high for complex scenes.

Kinematic and physics-based techniques have strengths and weaknesses. Combining physics based simulation with kinematic techniques is clearly the right approach for future applications. It is also an active area of research that has started to produce interesting results [Zordan et al. 2005; Shapiro et al. 2003]. Such techniques are already being incorporated into the new generation of interactive entertainment applications albeit at a small scale. In addition, applications often resort to heuristics in order to reduce the computational load and achieve interactive rates for complex scenarios. Such heuristics may involve simplified models, quasi-static dynamics et al. Our work aims to understand the computational load of the simulation as a first step toward designing a new generation of processors that can support the computational load of physical simulation at a large scale.

In this study we focus exclusively on constrained rigid body simulation [Smith ; AGEIA ; Havok] and leave the soft-body simulation domain for future work. In the majority of games, the central elements are humanoid characters. Humanoid motion is dominated by the rigid body motion of the character's body parts. Soft-body simulation such as flesh, cloth and hair animation are typically secondary effects. Among the various types of physics simulation, rigid body physics dominates in terms of computational load. Most recently, this was demonstrated by Sony in [V. Kokkevis 2006].

2.2 High Level Characteristics of the Simulation Load

Efficiency is crucial in interactive entertainment: each frame of animation must be computed at 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. For interactive applications such as games and urban simulations, the components include: artificial intelligence operations, path planning, user input, motion synthesis, networking, audio and video processing, and graphics. In this work, we assume that 10% of this frame rate can be used for physics-based simulation. *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 constraints placed on the objects (i.e. bones bending, walking through walls), IE applications generally have looser requirements on accuracy than most scientific applications. Recent research in animation [Harrison et al. 2004; Reitsma and Pollard 2003] has actually studied and quantified errors that are visually imperceptible. For instance, length changes below 2.7% cannot be perceived by an average observer[Harrison et al. 2004] while changes of over 20% are allways visible. This error tolerance increase with scene clutter and high-speed motions[Harrison et al. 2004].

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, there is usually a human viewer/user involved, so the application can focus on the area of the world that falls within the field of view of the viewer or in general, the area around the user.

In summary, the physics load, specifically as it applies to interactive entertainment applications, seems to be *distributed*, *sparse*, *restricted* by the interactive nature of the applications and subject to low level vector-based acceleration. 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* [Smith] as a representative physics-based simulator for interactive entertainment applications.

2.3 Open Dynamics Engine Algorithmic Load

The Open Dynamics Engine follows a constraint-based approach for modeling articulated figures, similar to [Baraff 1997]. 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).

2.3.1 Collision Detection

Collision detection (CD) uses geometrical approaches to identify bodies that are in contact and appropriate contact points. A *space* in CD contains geometric objects that represent the outline of rigid bodies [Smith]. Spaces are used to accelerate collision detection by allowing the removal of certain object pairs that would result in useless tests. This concept of space allows for hierarchical CD and isolated CD. Hierarchical CD provides fine granularity CD without frequently incurring the high load of doing tests on a large number of objects (i.e. collisions between 2 skeletons composed of 16 bones each). Isolated CD allows CD without frequent communication (i.e. 2 pairs of armies interacting in spaces that are significantly far apart). Both of these are exploited in the ODE engine to improve CD performance.

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.

The number and type of spaces have significant impact on CD requirements. At a finer granularity, the geometric shape used to model individual bodies in the system also contributes to differences in requirement.

2.3.2 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 the collision detection module. This is 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. Thus, in this paper we exclusively use the quick-step approach.

ODE's integrator trades 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 0.001 to 0.01.

ODE is designed to exploit parallelization: the user can create multiple worlds which are handled independently of each other. Within each world ODE automatically separates objects into independent groups, called *islands*. Each world and each island within a world can be solved independently and potentially on a different thread/processor.

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 [Smith], 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.

ODE supports a number of parameters that model the material properties of simulated objects, such as the coefficients of friction and the elasticity of collisions. In our experiments, we cover a wide range of materials ranging from elastic balls to rigid bricks.

3 PhysicsBench

In order to suggest architectural improvements to enable real-time physics simulation, we need to first characterize the computational load of the real-time physics engine kernel. Due to the lack of prior work, this requires the creation of a representative suite of benchmarks that covers a wide range of situations in future interactive entertainment applications. It is important to note that current games do not employ significant amount of physics simulation due to existing hardware platforms' limitations. With the introduction of next generation consoles and physics accelerators, applications are starting to integrate physics into the game-play. Our work examines *future* IE applications employing physics simulation. This section covers the details and reasoning in PhysicsBench's creation.

3.1 High Level Considerations

PhysicsBench covers a wide range of typical IE situations that involve object interaction. Our scenarios represent typical game-play situations for the most popular genres based on sales of current generation platforms [Everything and Nothing ; MagicBox b; MagicBox a]: simulation (The Sims), sandbox (Grand Theft Auto), racing (Grand Turismo), fps (Half-life), rts (Starcraft), mmog (World of Warcraft), rpg (Diablo), and sports (FIFA) . The benchmarks include high-velocity vehicles, fighting humans, object to human collisions, object to object collisions, exploding structures, and fairly complex battle scenes. They are designed to test the scalability of the simulation with different distribution of interactions: stacking vs a large battle scene.

Because of time constraints some of the benchmarks represent scenes of realistic complexity (interactions) but not necessarily realistic motions. The visual representation of the scenes shows the geometries used for collisions, not the ones used for visual display. We are only interested in the simulation load, not the graphics load.

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 IE applications, the combined computational load can be roughly extrapolated from the results of the individual scenarios. Since the use of real-time physics in applications is an active area of research and development, PhysicsBench will continue to evolve and be augmented with new scenarios and new physical interactions.

3.2 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.

In addition to representing realistic application scenarios, we aim for broad coverage on the low-level parameters that affect computation load as shown in Figure 1. Table 1 summarizes the quantitative differences between benchmarks.

The benchmarks are as follows:

- 2-Cars: Two cars driving - two cars that are steered to run in parallel then collide. One car goes over a wooden ramp.
- 10-Cars: Ten cars driving - to evaluate how the load changes with scale, we extend the 2-Cars scenario to ten cars.
- CrashSk: Car crashing on two people - a car with four wheels crashing into two 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 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.
- Battle: Battle scene I - One group of 10 humans attacked by tank. 2 groups of 4 and 6 humans crashing into each other.
- Fight: Fighting Scene, 2 groups of 5 humans - two groups of five humans that come in contact in pairs and eventually form a number of piles.
- Battle2: Battle scene II - a relative complex battle scene. A tank behind the far wall shoots 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.

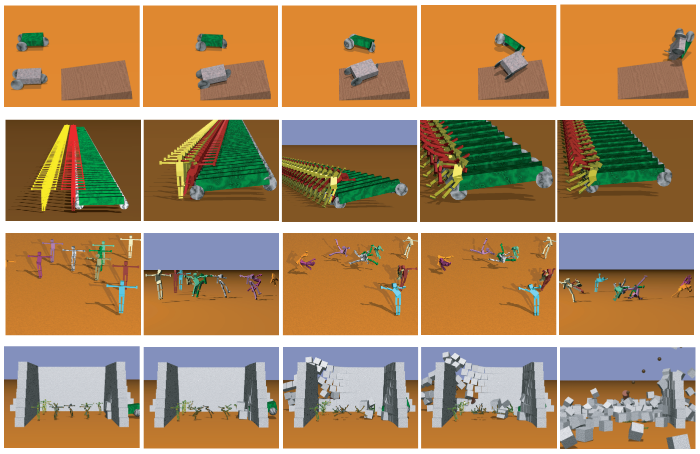
Figure 2: Benchmarks 2-Cars, 100CrSk, Fight, and Battle2 from top to bottom. Images in raster order.

These scenarios can capture some extremely complex interactions. The computational load of *2-Cars*, for example, relates to a wide range of two objects interactions that arise in games. These include car racing, 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.

3.3 Comparison Against Other Workloads

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. Collision detection (section 2) makes up an average 7% of all executed instructions, ranging between 2% and 20% for the various benchmarks.

The *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 IE 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

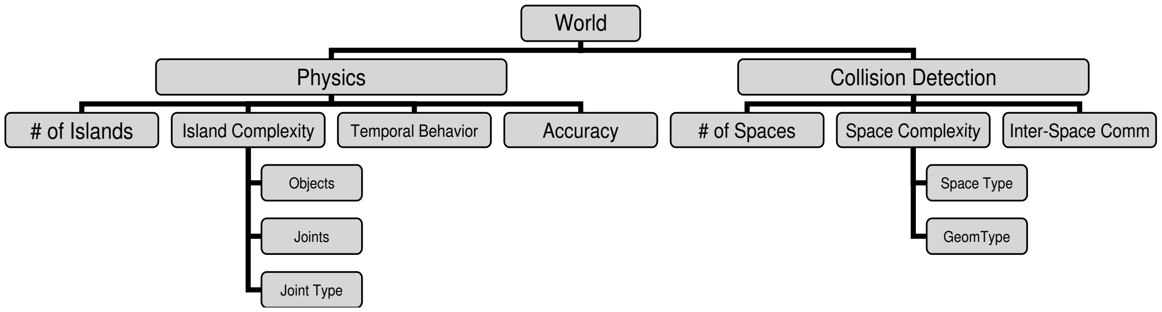

Figure 1: Low-level Parameters Affecting Computation Load

Benchmark	Number of Islands (Max, Min, Avg, Dev)	Island Complexity	Temporal Behavior	Number of Spaces	Inter-space Comm
2 Cars	2, 2, 2, 0	simple, 5 obj + 4joints	constant	1	NA
10 Cars	10, 10, 10, 0	simple, 5 obj + 4 joints, few contacts	stable, few collisions	1	NA
Car Crash Sk	3, 1, 2, 0.65	from 2 complex 1 simple to one large complex	fast changing	1	NA
Car Crash Wall	105, 99, 101, 1.4	complex stack, simple car, simple cannon, sphere projectiles	stable, abrupt change	1,3	range from none to high
Environment	337, 196, 245, 46	complex stack, monster, simple car, cannon, projectiles	fast changing	1,10	high
Car Crash Sf x100	300, 100, 220, 64	same as Car Crash Sk	stable, fast change	100	none
Battle I	120, 2, 93, 18	groups of multiple complex skeletons, standalone skeletons, projectiles	stable, fast change	1, 3	none
Fight	10, 7, 8, 1.1	complex skeletons interacting	4 stable, 6 change	1,10	moderate
Battle II	156, 113, 134, 18	multiple complex stack, complex skeletons, simple car, simple cannon, projectiles	fast changing	1,15	high

Table 1: Parameters Affecting Computation Load

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.

Certain applications in the SPEC CPU 2000 FP suite make use of similar numerical methods, but the constraints imposed by games and the particular instruction mix of PhysicsBench distinguish these applications from the SPEC suite. The real-time constraint of games requires high performance to allow the use of real-time physics simulation. Also, the relaxed accuracy requirement allows multiple levels of approximations and optimizations, such as higher error thresholds, restricted size matrices, constraint violations, higher time-steps, interpenetrations, approximate iterative techniques etc. Furthermore, the high cache hit-rates of PhysicsBench contrasts with scientific applications' large memory working-set.

Embedded application suites like MiBench [Guthaus et al. 2001] are also quite different from PhysicsBench. Embedded real-time schedules are typically easily satisfied by low-power, lower-performance single-core processors. Another major difference is the relatively small amount of floating point instructions seen in these applications.

4 Physics Performance

In this section, we explore the performance of PhysicsBench on a real processor.

4.1 Methodology

The real-time physics engine used in this study is the Open Dynamics Engine (ODE), described in Section 2. We selected ODE because it is inherently designed to support the requirements and exploit the characteristics of constrained rigid body dynamics as they apply to interactive entertainment (IE) applications. ODE has been widely used in both research and commercial applications such as computer games and 3D authoring tools.

PhysicsBench includes tests using both the *big-matrix* and the iterative solvers. In this paper, 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.

While the workloads' level of physical interaction varies between different points in time, we will initially focus on the peak computation demand during frames of high activity. 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 after warm-up (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. Physics data frequently communicates with the core engines of IE applications [Havok ; Wu 2005], requiring short latencies for data transfers to and from core engine threads that execute on the general-purpose CPU.

As described in the introduction and [Wu 2005], 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 there is no standard for this time allocation, we feel that future complex applications using a real-time physics engine will undoubtedly be using advanced algorithms for other core components. Therefore, to provide low variance on the frame rate we need to aim for a more aggressive time allocation.

For this real processor study, we used a 2.4GHz Intel P4 Xeon CPU with 512KB L2 cache, with support for SSE/2 instructions.

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 games – particularly since the performance goal is measured in frames per second. Each benchmark requires different amount of instructions per frame as shown in Figure 3. The data points with a Q suffix refers to Quickstep, and data points without the suffix refers to Normalstep. QuickStep requires significantly less instructions than Normalstep.

4.1.1 Parallel Physics Simulation

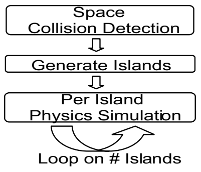

Figure 4: Core Physics Simulation Loop

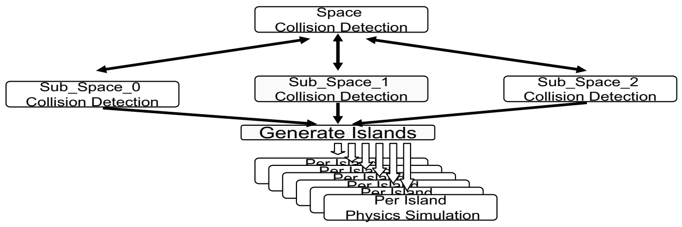

Figure 5: Parallel Implementation of Physics Simulation Loop

As shown in Figure 4, the high level core physics simulation loop has three 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 – and, on average, this component consumes approximately 92% of the total physics simulation time.

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.

With parallel physics simulation, the % of cycles taken up by collision detection becomes much more significant (20% on average, maximum of 55%). Therefore, we also parallelize collision detection through parallel spaces on top of ODE's spatial hash broadphase collision detection [Smith].

Figure 5 shows our implementation of the core physics loop with both collision detection and physics simulation in parallel. 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.

To evaluate the potential of this optimization, we capture the upper bound on performance by assuming an unlimited supply of homogeneous cores and ignoring the overhead from sources such as thread creation, thread migration, data migration, and setup codes.

4.2 PhysicsBench Results

Figure 6 presents results for actual runs of PhysicsBench on the architecture in section 4.1. It is clear why current IE 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. In situations where physical interactions can be partitioned cleanly as in 100CrSk, tremendous speedup can be obtained from the use of parallel threads. Battle and Fight see benefit from parallel threads, but are still unable to achieve frame rates high enough to satisfy the time constraint for any of their frames. The remaining benchmarks have little parallelism to exploit, and therefore see little or no improvement in frame rate. For example, CrashWa contains a large complex island simulating a brick wall. The 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.

For parallel physics simulation, the maximum island count for each benchmark indicates the maximum number of cores we would need to achieve the improvements shown earlier. As shown in Table 1, the maximum island count in the suite is 337. However, we may need far fewer cores to achieve parallelization benefits since there exist simple islands which can be serialized on one core, overlapping the execution of more complex islands. We present the resource requirements using optimal load balancing in Table 2. The data shows that most benchmarks with high island counts can actually be satisfied 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.

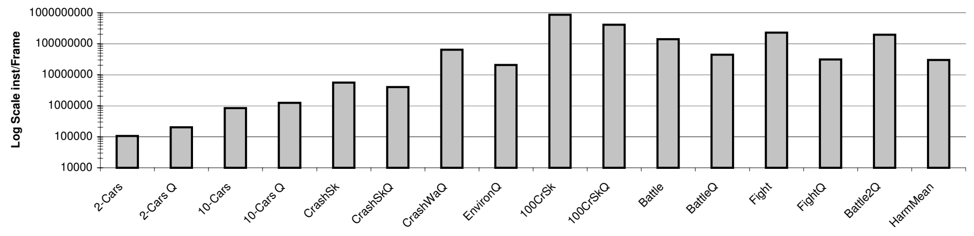

Figure 3: Instructions Per Frame for PhysicsBench

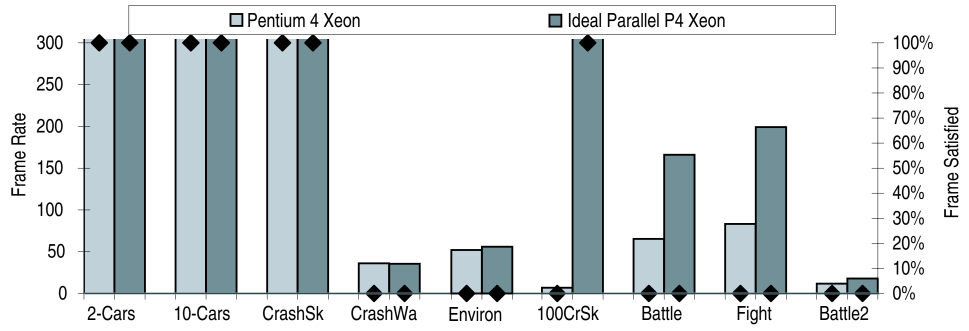

Figure 6: PhysicsBench Performance.

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 2: Resource Requirement for Full Parallelization

5 Physics Acceleration

Even though our idealized parallel physics simulation is very effective, we are still some distance away from satisfying the demands of PhysicsBench applications CrashWa, Environ, Battle, Fight, and Battle2. In this section, we consider techniques to meet the 30 fps constraint.

Because error tolerance is a main differentiating feature of the IE workload, we will explore the trade-off of accuracy for performance to develop novel optimizations. This will be done at both the architectural and algorithmic levels. Errors in physics simulation can result in different geometric positions and orientations, as well as constraint violations. This latter error can be more severe, and could include objects passing through one another, elongation of the object, or joints separating or bending in incorrect ways.

One main limiting factor in trading accuracy for performance is the need for deterministic behavior. This is important for both application verification and coherent behavior across different clients.

5.1 Fuzzy Value Prediction [Architectural]

Prior work has demonstrated the effectiveness of using last value prediction at reducing instruction latency [Lipasti et al. 1996]. 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. Values are predicted as each instruction is decoded by indexing a table by the program counter. The value prediction is verified when the instruction actually completes, and the dependent instructions are re-executed if a misprediction has occurred. Traditional value prediction verifies by exact binary comparison. Confidence mechanisms [Calder et al. 1999] can effectively limit the degradation of mispredictions. Recent work shows some benefit for floating point value predictions [Tuck and Tullsen 2005] using multi-threaded execution. Fuzzy instruction reuse [Alvarez et al. 2005] 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 does not impact the accuracy of the physics simulation. However, we can trade some accuracy to improve the value 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. During verification, the predicted and executed values are subtracted and the magnitude of the error is checked. 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.

We evaluated FVP using thresholds ranging from 10^{-6} to 10^{-12}. Similar behaviors were observed between applying FVP to collision detection only, forward step only, and across all physics simulation computation. While the error FVP tolerates for individual operations is relatively small, the physics engine can produce “blown up” results in the form of infinity or NaN. This occurs even when using a threshold of 10^{-12}.

Motivated by FVP’s behavior and prior work on perceptual metrics, we present an algorithmic approach to trade accuracy for performance, while maintaining the deterministic behavior preferred by the industry.

5.2 Fast Estimation with Error Control [Algorithmic]

Fast Estimation with Error Control (FEEC) describes a high-level frame-work consisting of 2 logical threads and 2 evaluation methods. This technique leverages 2 observations about IE applications. First, different consumers (in terms of software components) require different accuracy for the same input data. Second, different consumers consume the same input data at different real-time schedules.

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 into 2 logical threads. These logical threads can be described as the (1) slow *precise* thread and the (2) *fast estimation* thread. Each thread utilizes its own evaluation method for the same computation. The data produced by the *fast estimation* thread is fed to critical dependent components while the data produced by the *precise* thread is fed back to the physics engine for both threads at the frame-boundary. This feedback of *precise* data is critical in containing errors since each frame's errors are 100% corrected for next frame's computation.

We call this approach *fast estimation with error control* (FEEC) since the *precise* physics simulation thread is allowed to continue in parallel with the rest of the interactive entertainment application and effectively limit the error rate of the estimation thread.

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.

Various estimation methods can be utilized with FEEC. In this paper, we evaluate one estimation method, namely LCP solver iteration reduction.

5.2.1 Reducing Iteration Count

The LCP solver ODE employs, 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.

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.

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 accumulate and "blow up" calculations. As described in section 3, visually acceptable errors are dependent on the situation and are of much larger magnitude than what the physics engine can tolerate.

Consider running QuickStep for only a single iteration to provide estimated results to the critical dependent components – 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 *precise* thread is fed back as input to the next frame of the physics loop to correct all errors.

This estimation method shortens physics simulation's critical path by altering the control flow of the LCP solver to generate a fast solution for critical dependent components. 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 verified both numerically and visually that the errors using FEEC with iteration reduction 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. The position and orientation data produced by FEEC were used as input to reconstruct images for visual inspection.

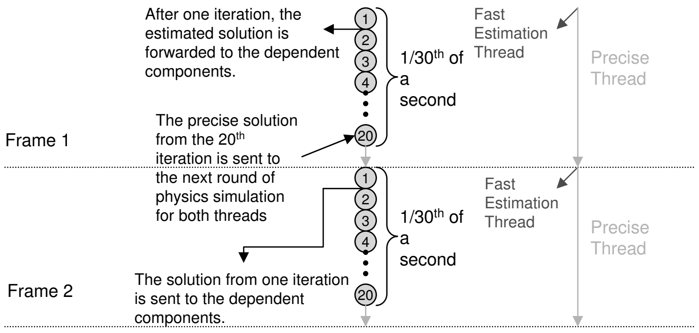
After one iteration, the estimated solution is forwarded to the dependent components.
1
2
3
4
20
1/30th of a second
Fast Estimation Thread
Precise Thread
Frame 1
The precise solution from the 20th iteration is sent to the next round of physics simulation for both threads
Frame 2
The solution from one iteration is sent to the dependent components.

Figure 7: Fast Estimation with Error Control (FEEC)

The parallel results in section 4 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 after the first iteration to critical dependent components. The results of 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 critical dependent components).

The FEEC approach is illustrated in Figure 7. The right side represents the high-level FEEC framework, and the left side represents the use of iteration reduction with FEEC. The Y-axis represents time, and two frames' worth is represented. The green arrows represent the *fast estimation* thread, and the red arrows represent the *precise* thread. Each circle on the left represents one iteration of the LCP solver for QuickStep.

5.3 Methodology

Although we used the same binaries as section 4.1 in our simulations, we used PTLsim [Yourst], 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.

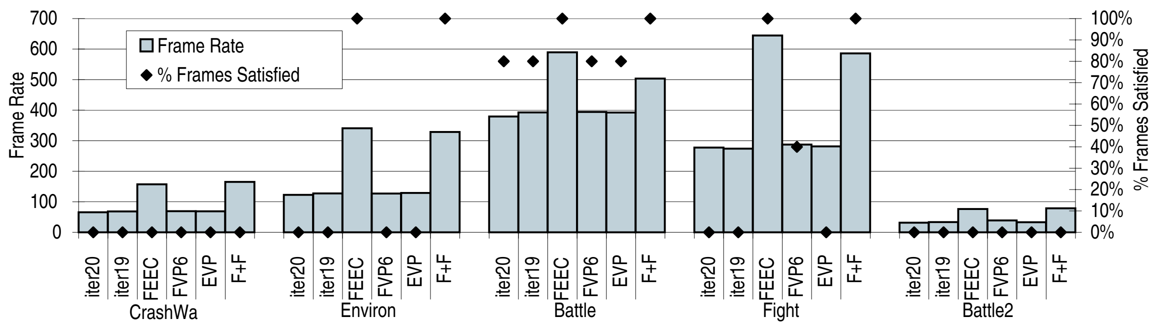

Figure 8: Performance of FEEC and fuzzy value prediction.

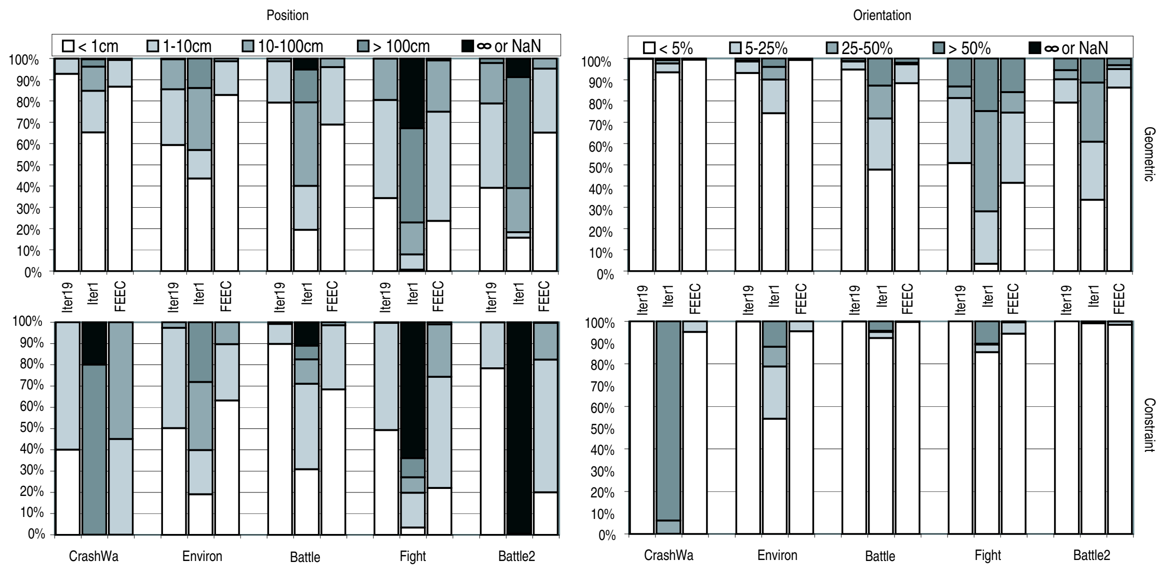

Figure 9: Error for FEEC relative to a 20-iteration QuickStep.

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 per Cluster (Int, Int, Mem port, FP)	2,2,2,2
Mem Latency	160 cycles

Table 3: Parameters for our architectural configuration.

This ID uniquely identifies micro-ops that are generated from the same x86 instruction. The ID (starting at 0) is assigned in order to each micro-op.

From the data in figure 6, 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 3 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.

5.4 Results

Figure 8 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 improves performance over 20Iter by 220% with minimal error as described below.

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 alone is able to see more frames satisfied for Fight and improves Battle2's frame-rate by 24%. When combined with FEEC, FVP6 degrades the frame-rate of three tests and marginally improves the rest. The degradation mainly comes from the formation of larger islands due to the errors introduced. The in-depth study of this behavior will be future work. 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 reducing the iteration count directly. Therefore, we need to compare the errors seen by FEEC to a single iteration with Quickstep.

Figure 9 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. This is verified visually with image reconstructions.

6 Summary

Interactive entertainment applications are rapidly gaining significance from both technical and economical point of views. In this paper, we explore a core content-creation mechanism – real-time physics simulation – that will be central to future immersive workloads. First, we present a suite of benchmarks, PhysicsBench, to represent this new workload. Then, we explore PhysicsBench on a real world system. After showing the significant computational power required to satisfy PhysicsBench's frame demands, we detail three paths of architectural exploration to help achieve these requirements with existing hardware.

Parallel execution of both collision detection and physics simulation components enables significant speedup to satisfy the requirements for several benchmarks. With the addition of Fast Estimation with Error Control (FEEC), current processors can begin to satisfy the constraints of even complicated scenarios. This technique provides a unique opportunity to leverage multiple cores by allowing physics simulation to continue in parallel with other game engine tasks. Despite the promise of breaking true data dependencies, fuzzy value prediction is not able to significantly accelerate the majority of workloads we explored.

Given the rapid change and wide range of future real-time physics simulation, we will continue to refine and augment PhysicsBench. Future work will examine techniques to exploit the available massive fine-grain parallelism, i.e. parallelizing the computation of one massive island. This exploration will look at both single chip and discrete chip solutions such as Cell [Hofstee 2005], Xenon [Microsoft], GPU [Havok], and PPU [AGEIA]. More complex estimation methods for FEEC will also be evaluated.

From the findings of this paper, we expect highly parallelized real-time physics engines in the near future to enable more realistic applications. Furthermore, the recent push toward chip-multiprocessors will enable game programmers to exploit the available parallelism in creating large immersive experiences.

Acknowledgments

The work in this paper was supported by NSF CAREER grant No. CCR-0133997 and NSF grant No. CCF-0429983. We thank Intel Corp., Microsoft Corp., ATI Corp. and Alias Corp. for their generous support through equipment and software grants. We also thank Eric Wood for simulation infrastructure support.

References

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

ALVAREZ, C., CORBAL, J., AND VALERO, M. 2005. Fuzzy memoization for floating-point multimedia applications. In *IEEE Transactions on Computers*.

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

CALDER, B., REINMAN, G., AND TULLSEN, D. 1999. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, 64–74.

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

EVERYTHING, AND NOTHING. World wide selling software. In *www.everythingandnothing.org.uk*.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*.

HARRISON, J., RENSINK, R. A., AND VAN DE PANNE, M. 2004. Obscuring length changes during animated motion. *ACM Trans. Graph. 23*, 3, 569–573.

HAVOK. http://www.havok.com/content/view/187/77.

HOFSTEE, P. 2005. Power efficient architecture and the cell processor. In *HPCA11*.

INITIATIVE, S. G. http://www.seriousgames.org/.

LIPASTI, M., WILKERSON, C., AND SHEN, J. 1996. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 138–147.

MAGICBOX. Japan platinum videogame chart. In *www.the-magicbox.com*.

MAGICBOX. Us platinum videogame chart. In *www.the-magicbox.com*.

MATTHEWS, B., WELLMAN, J., AND GSCHWIND, M. 2004. Exploring real time multimedia content creation in video games. In *6th Workshop on Media and Streaming Processors*.

MICROSOFT. Xbox 360. http://www.xbox360.com/.

NINTENDO. Revolution. http://www.nintendo.com/newsarticle?articleid=5aa8631e-d4a0-45d9-a88c-e5931b807091.

OLUKOTON, K., NAYFEH, B., HAMMOND, L., WILSON, K., AND CHANG, K. 1996. The case for a single-chip multiprocessor. In *ASPLOS-VII*.

REITSMA, P. S. A., AND POLLARD, N. S. 2003. Perceptual metrics for character animation: sensitivity to errors in ballistic motion. *ACM Trans. Graph. 22*, 3.

SHAPIRO, A., PIGHIN, F., AND FALOUTSOS, P. 2003. Hybrid control for interactive character animation. In *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*.

SMITH, R. Open dynamics engine. http://www.ode.org.

TUCK, N., AND TULLSEN, D. 2005. Multithreaded value prediction. In *HPCA11*.

U.S. ARMY. The official u.s. army game: America's army. http://www.americasarmy.com/.

V. KOKKEVIS, S. OSMAN, E. L. 2006. High-performance physics solver design for next generation consoles. In *Game Developers Conference*.

WU, D. 2005. Physics in parallel: Simulation on 7th generation hardware. In *Game Developers Conference*.

YOURST, M. T. Ptlsim user's guide and reference: The anatomy of an x86-64 out of order microprocessor. In *http://www.ptlsim.org*.

ZORDAN, V. B., MAJKOWSKA, A., CHIU, B., AND FAST, M. 2005. Dynamic response for motion capture animation. *ACM Trans. Graph. 24*, 3, 697–701.