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

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

Abstract

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

1 Introduction and Motivation

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

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

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

Our paper makes the following contributions:

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

2. *Value Locality:* Precision reduction improves the locality that exists (a) among similar objects in similar scenarios and (b) across iterations during the relaxation of constraints.
3. *Reduced Precision FPU:* Smaller, faster, and less power hungry FPUs can be used to replace full precision FPUs. However, the varying precision requirement necessitates occasional access to full precision FPUs.

- We propose and investigate the hierarchical sharing of FPU resources among simple cores in PBA. The L1 FPU designs leverage the benefits of precision reduction to execute FP operations locally, and the full-precision L2 FPUs are shared among multiple cores for area reduction. Area savings from FPU sharing can translate directly to performance by adding more simple cores. Per phase performance is increased by up to 55% while per phase energy is simultaneously reduced by up to 50% when compared to a baseline without FPU sharing.

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

2 Background and Related Work

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

2.1 Real-Time Physics for Interactive Entertainment

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

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

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

2.2 Conjoined Cores

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

2.3 Floating-Point Optimization

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

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

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

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

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

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

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

3 Methodology

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

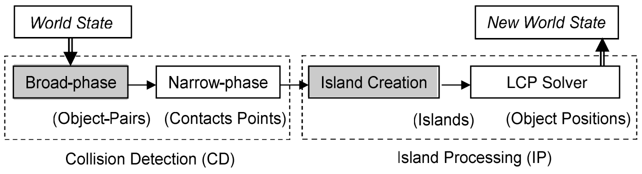


Figure 1. Physics engine flow. All phases are serialized with respect to each other. Unshaded stages can exploit parallelism.

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

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

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

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

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

4 Precision Reduction

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

4.1 Error Tolerance of FP Operations

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

4.1.1 Minimum Required Precision

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

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

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

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

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

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

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

4.2 Dynamic Adaptation of Floating Point Precision

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

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

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

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

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

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

4.3 Opportunities Enabled by Dynamic Precision Reduction

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

4.3.1 Trivial FP Operations

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

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

Table 2. Conventional trivial cases

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

Table 3. Factors increasing trivialization

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

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

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

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

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

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

4.3.2 Trivialization Details

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

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

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

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

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

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

4.3.3 Locality of FP Operations

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

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

4.3.4 Lookup Table

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

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

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

For FP add lookup, the smaller operand is first shifted based on the exponent difference. This operation only requires a small and fast 5-bit shifter. This means that the implicit 1 of the smaller operand will be seen. Then, the concatenation of the larger operand and the shifted operand is used as the index. In FP add lookup entries, an additional bit is stored to indicate the need to increment the exponent by 1 for normalization of a carry out. Since we allocate 8 bits per entry, there is room for such annotation. One corner case for FP add is when the two operands have the same exponent. In this case, indexing with the default scheme will result in aliasing with a smaller operand which has been shifted. One solution is to first detect this by an exponent difference of zero, then appropriately handle the most significant bit after the leading one. This solution does not require any changes to the default scheme.

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

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

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

Table 5. Lookup vs memoization table

5 FPU Sharing

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

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

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

Table 6. Fine-grain shader core design

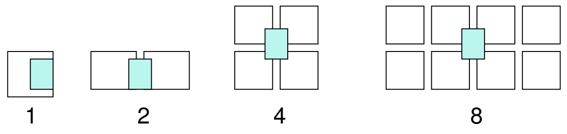


Figure 2. Layout for one, two, four, and eight cores sharing an FPU.

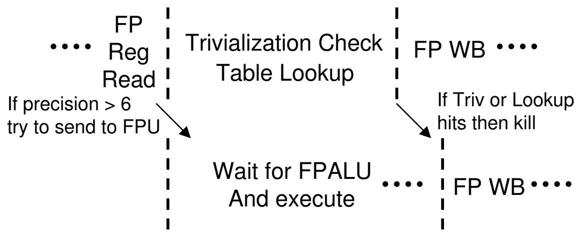

FP Reg Read
Trivialization Check Table Lookup
FP WB
If precision > 6 try to send to FPU
If Triv or Lookup hits then kill
Wait for FPALU And execute
FP WB

Figure 3. Pipeline for FPU sharing.

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

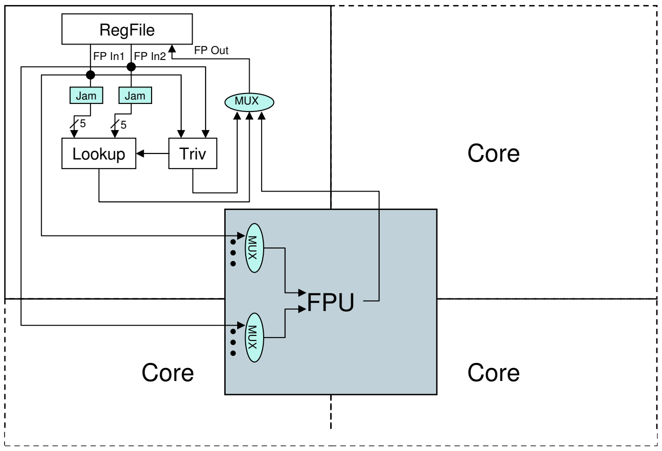


Figure 4. Four cores sharing a single FPU – each core has a trivialization and memoization unit.

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

5.1 Hierarchical FPUs

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

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

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

1. *Conventional Trivialization:* The L1 FPU only contains conventional trivialization logic – no precision reduction.

2. *Reduced Precision Trivialization:* The L1 FPU uses our reduced precision trivialization logic. Additional exponent logic is required.

3. *Lookup Table + Reduced Precision Trivialization:* The L1 FPU has our reduced precision trivialization logic, exponent logic, and a local 2K-entry lookup table. FP add and multiply instructions using mantissa precision of less than six bits make use of the lookup table. Figure 4 demonstrates a version of this design where four cores share a single L2 FPU. Each core has an L1 FPU with trivialization and lookup table hardware. The pipeline change required for this FPU sharing is illustrated in figure 3.

4. *mini-FPU + Reduced Precision Trivialization:* In addition to our reduced precision trivialization logic, this L1 FPU has a *mini-FPU*: an FPU design that uses a 14-bit mantissa and 8-bit exponents. FP add and multiply operations with mantissa precision of less than 15 bits can execute in this mini-FPU. The 14-bit precision is chosen based on the (benchmark coverage)/(area) ratio derived from Table 1. The intent of this design is to provide potentially higher coverage at the L1 FPU, but at a greater area cost.

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

Table 7. Variable FP Latency

5.1.1 Variable FP Latency

All components of HFPU's variable FP latency are listed in Table 7. If an operation can be satisfied by trivialization or look-up table, it takes one cycle to complete. If the mini-FPU is available and the operation can be satisfied by it, the operation takes three cycles. If an operation requires the full FPU, the exact latency is determined by accumulating the required latencies of arbitration, interconnect overhead, and full FPU operation.

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

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

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

1. Optimistically assumes 1 cycle latency. If the operation is not satisfied by the trivial or look-up table logic, the pipeline stalls until the operation is completed. Once an operation is detected to require the full FPU, the actual number of required cycles is known.

2. Pessimistically assumes the use of the full FPU. The required number of cycles is known at beginning of issue. This scheduling policy eliminates stalls/flushes/replays and retains the energy savings shown in the paper.

5.2 Results

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

For the 14-bit mantissa mini-FPU, we assume an area requirement of 60% that of the full precision 23-bit mantissa FPU. This ratio is based on the estimates of both prior work on reduced precision FPU [11] and results from a high-level synthesis tool tuned for generating area-efficient floating point data-paths. All area data is for $90nm$ technology.

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

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

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

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

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

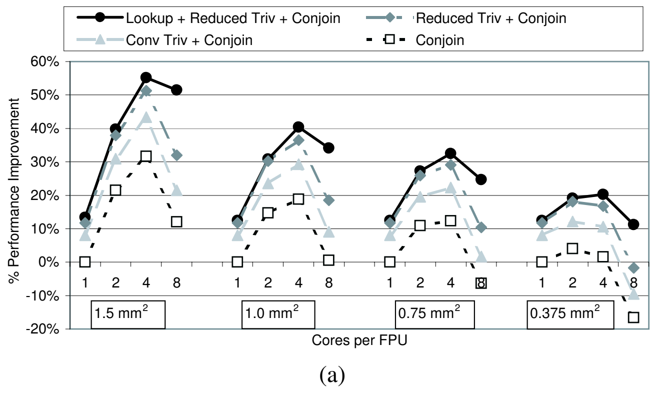
To estimate the total dynamic energy consumed by FP operations, we leverage the energy data for different subunits of the FPU presented in [10]. For configurations with trivialization, all FP operations are charged the trivialization logic energy. Non-trivial operations are then charged for the FPU energy.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

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

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

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

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



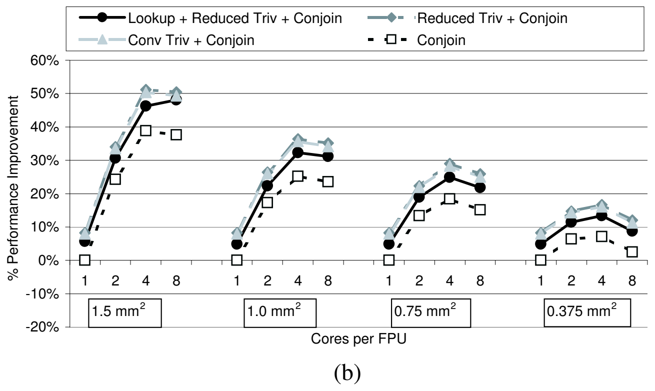


Figure 5. (a) HFPU LCP performance — (b) HFPU narrow-phase performance

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

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

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

6 Summary and Future Work

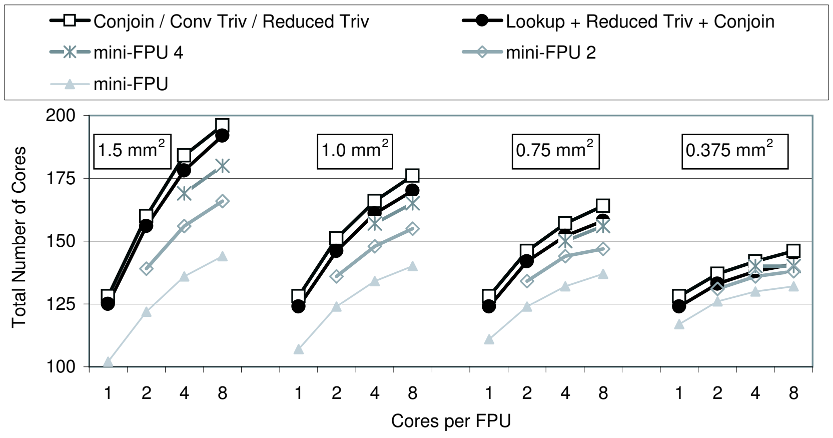
In this study, we exploit the inherent perceptual error tolerance of physics-based animation to dynamically adapt the required precision for floating-point computation. Then, we identify and leverage the benefits made possible by dynamic precision tuning to design a hierarchical floating-point unit (HFPU). This consists of an area-efficient L1 FPU and a full-precision L2 FPU that is shared among a number of cores in the ParallAX architecture. Our best performing L1 FPU design consists of our enhanced trivialization logic and a mantissa lookup table – an area efficient design that allows more cores to be packed into a particular silicon area. For the range of FPU designs we consider, this L1 FPU coupled with an L2 FPU that is shared among four cores provides an improvement in overall throughput up to 55% and reduces energy expended in the FPU by up to 50%.

7 Acknowledgements

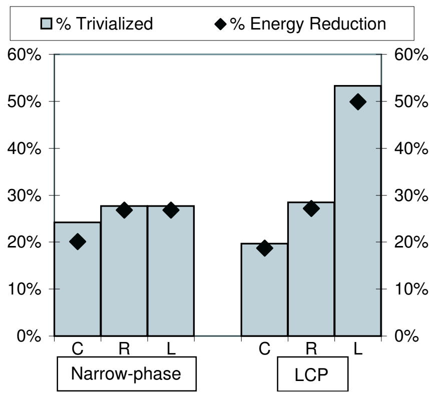
We would like to thank the anonymous reviewers for their helpful comments. Intel Corp., Microsoft Corp., AMD/ATI Corp., and AGEIA Technologies Corp. helped us with their generous support through equipment and software grants. This work was partially supported by NSF grants CCF-0429983 and CCF-0133997,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

References

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



(a)



(b)

Figure 6. (a) Total number of cores in the same die area as 128-core baseline. The total area used: 472 mm^2 for 1.5 mm^2 FPU, 408 mm^2 for 1.0 mm^2 FPU, 376 mm^2 for 0.75 mm^2 FPU, and 328 mm^2 for 0.375 mm^2 FPU. — (b) FP computation % trivialized and energy reduction. C = Conv Triv, R = Reduced Triv, and L = Lookup Table + Reduced Triv.

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

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

[8] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronics computing instrument. *Computer Structures: Reading and Examples, McGraw-Hill Inc.*, 1971.

[9] G. Chen, H. Chen, M. Haurylau, N. A. Nelson, D. H. Albonesi, P. M. Fauchet, and E. G. Friedman. On-chip copper-based vs. optical interconnects: Delay uncertainty, latency, power, and bandwidth density comparative predictions. In *IEEE International Interconnect Technology Conference*, 2006.

[10] D. Citron and D. G. Feitelson. Look it up or do the math: An energy, area, and timing analysis of instruction reuse and memoization. In *Workshop on Power-Aware Computer Systems*, 2003.

[11] F. Fang, T. Chen, and R. Rutenbar. Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform. *EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication*, 2002.

[12] D. Goddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in fem simulations. In *International Journal of Parallel, Emergent and Distributed Systems*, 2006.

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

[14] P. Hofstee. Power efficient architecture and the cell processor. In *HPCA11*, 2005.

[15] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y. Chen. Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors. In *34th Annual International Symposium on Computer Architecture*, 2007.

[16] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara, a 32-way multithreaded sparc processor. In *IEEE Micro*, 2005.

[17] K. Krewell. Ultrasparc iv mirrors predecessor. In *Microprocessor Report*, 2003.

[18] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *The 37th International Symposium on Microarchitecture (MICRO)*, 2004.

[19] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

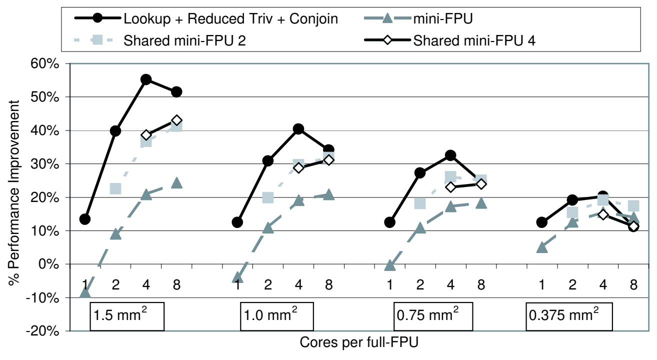
[20] M. Matthias, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. In *Proceedings of the 3rd Workshop in Virtual Reality Interactions and Physical Simulation*, 2006.

[21] H. Oh, S. M. Mueller, C. Jacobi, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong. A fully pipelined single-precision floating-point unit in the synergistic processor element of a cell processor. In *IEEE Journal of Solid-State Circuits*, 2006.

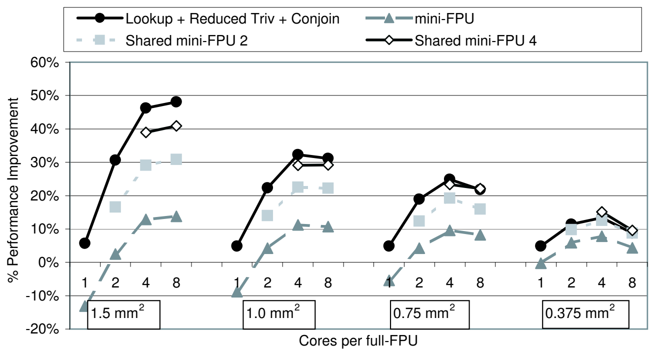
[22] K. Olukoton, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII*, 1996.

[23] C. O'Sullivan, S. Howlett, R. McDonnell, Y. Morvan, and K. O'Conor. Perceptually adaptive graphics. In *Eurographics 2004, State of the Art Report*, 2004.

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

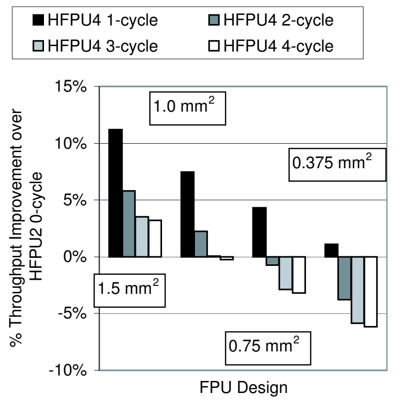


(a)

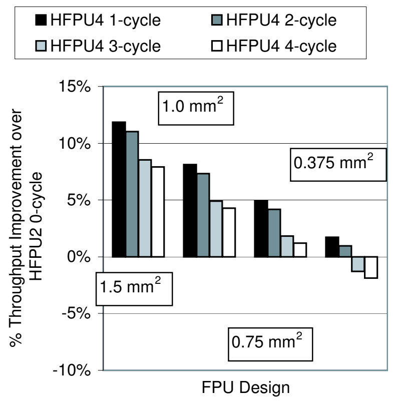


(b)

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



(a)



(b)

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

[25] S. E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. In *IEEE Symposium on Computer Arithmetic*, 1993.

[26] S. E. Richardson. Exploiting trivial and redundant computation. 1993.

[27] A. Seugling and M. Rolin. Evaluation of physics engines and implementation of a physics module ina 3d-authoring tool. In *Master's Thesis*, 2006.

[28] A. Shayesteh, G. Reinman, N. Jouppi, S. Sair, and T. Sherwood. Improving the performance and power efficiency of shared helpers in cmps. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2006.

[29] D. Sheahan. Developing and tuning applications on ultrasparc t1 chip multithreading systems. In *Sun Blue Prints Online*, 2007.

[30] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Compaq WRL 2001/2, 2001.

[31] V. Soteriou, N. Eisley, H. Wang, B. Li, and L. Peh. Polaris: A system-level roadmap for on-chip interconnection networks. In *Proceedings of the 24th International Conference on Computer Design*, 2006.

[32] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman. Parallax: An architecture for real-time physics. In *The 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[33] T. Y. Yeh, P. Faloutsos, and G. Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Video Game Symposium*, 2006.

[34] T. Y. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. In *2007 ACM Transactions on Graphics (TOG) – accepted with major revisions. http://www.cs.ucla.edu/ tomyeh/tog07.pdf.*

[35] J. Yi and D. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *IEEE International Conference on Computer Design: VLSI in computers and Processors (ICCD)*, 2002.