[CSD-TR 60026] ParallAX: An Architecture for Real-Time Physics

Thomas Y. Yeh, Petros Faloutsos, Sanjay Patel, and Glenn Reinman

Abstract

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

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

1 Introduction and Motivation

Interactive entertainment (IE) applications demand high performance from all architectural components. In the future, this demand will increase even further. These applications will be composed of a diverse set of computationally demanding tasks. One critical task is modeling how objects and characters move and interact in a virtual environment. Most current IE applications make use of pre-recorded motion clips to synthesize the motion of virtual objects (kinematics), but as these objects and their interactions scale up in complexity, pre-recording has become impractical. *Physics-based simulation* has emerged as an attractive alternative to kinematics, providing high levels of physical realism through motion calculation. Future applications targeting a true immersive experience will demand this as a cornerstone of their design.

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

The difficulty in meeting the performance demands of physics-based simulation is somewhat mitigated by the high degree of parallelism available in these applications. Our results show that on average 91% of a physics workload can be broken down into parallel subtasks of varying granularities. Although only 9% of the workload is serialized, on a single desktop core this portion can take up to 125% of the available frame time. This argues for an architecture that has sufficient performance to tackle the serial tasks, but also has the flexibility to fully exploit parallel regions. There are existing designs that combine some number of large coarse-granularity (CG) cores with a larger number of fine-granularity (FG) cores, including GPUs connected to a host CPU through a system bus [22], the Cell's SPE's paired with its PPE [11], and a conventional core paired with multiple vector units [15]. However, these designs lack flexibility in how their cores can be utilized, making it difficult for them to *efficiently* meet the demands of real-time physics. This paper proposes a more flexible design that is optimized to meet these demands, based on a design space exploration that includes the number and type of cores required, the amount of cache state required, and the interconnect required between these components.

In this paper, we make the following contributions:

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

- We show that current multi-core architectures will not be able to sustain interactive frame rates even when the benchmark suite is aggressively parallelized. Operating system, cache contention, and control logic area overhead all contribute to this conclusion.

- We propose an architecture with both CG and FG cores that is able to sustain interactive frame rates for physics workloads through efficient area utilization. The key elements of this efficiency are:

  - *Intelligent, application-aware L2 management* – In section 6.1 we examine the L2 requirements for physics simulation and propose a partitioning strategy that reduces the required L2 space by more than half.

  - *Dynamic coupling/allocation of FG cores to CG cores* – In section 7.1 we propose an arbitration policy that balances the maximal utilization of available FG core resources and the exploitation of locality among FG cores working on the same CG task.

  - *Relaxed communication latency of FG and CG cores*– In sections 7.2 and 8.2.2 we explore design alternatives to interconnecting FG and CG cores. The tight coupling of FG and CG cores can restrict where these cores are placed (i.e. on/off chip) and how effectively we can dynamically leverage FG cores. To loosen this coupling, we consider the amount of buffering space and application parallelism required to overlap communication for a variety of interconnection strategies.

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

2 Related Work

The MDGRAPE-3 chip by RIKEN [28] and the PhysX chip by AGEIA [3] are currently the only dedicated physics simulation accelerator designs. While MD-GRAPE targets computational physics, PhysX targets real-time physics for games. Both designs are placed on accelerator boards which connect to the host CPU through a system bus. PhysX's architectural design is not public, and MD-GRAPE's design is specific to computing forces for molecular dynamics and astrophysical N-body simulations with limited programmability.

Two other closely related bodies of prior work are *vector processing* [10] and *stream computation* [17, 13].

Vector Processing. The massive parallelism available in real-time physics hints at the use of vector processors like VIRAM [16], Tarantula [8], and CODE [15]. VIRAM[14] has achieved an order of magnitude performance improvement on certain multimedia benchmarks. However, conventional vector architectures are constrained [15] by limitations like the complexity of a centralized register file, the difficulty to implement precise exceptions, and the requirement of an expensive on-chip memory system. Most importantly, the physics workload requires tremendous speedup that necessitates massive parallel execution. While CODE is scalable by increasing clusters and lanes, the data shows a plateau at eight clusters with eight lanes, and the cacheless CODE can not satisfy our measured physics workload.

Stream Computation. Stream architectures (SAs) aim to enable ASIC-like performance efficiency while being programmable with a high-level language. Stream programs express computation as a signal flow graph with streams of records flowing between computation kernels. While a broad range of designs populate this space, the high-level characteristics of SAs are described in [17].

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

IBM's Cell [11] consists of one general purpose PowerPC core (PPE) and eight application specific streaming engines (SPE) – all connected by a ring of on-chip interconnect (EIB). Although the Cell's programming model is described as cellular computing, the design can be included in the broad space of streaming computation. The PPE is a 64-bit, 2-way SMT, in-order execution PowerPC design with 32KB L1 caches and a 512KB L2 cache. The SPEs are RISC cores each with 128 128-bit SIMD registers, customized SIMD instructions, and 256KB local private memory. SPEs are not ISA compatible with conventional PowerPC cores. Heterogenous CMP designs such as the Cell are able to target the best *task* specific performance using different cores. The PPE targets control intensive tasks such as the OS and the SPEs target compute intensive tasks. However, according to our exploration, both the PPE and SPE designs are not optimal for physics computation. Serial components' performance on the PPE wil take a significant amount of each frame's time, and the SPE's complexity prevents the placement of the required number of cores to achieve 30 FPS. This may be a result of the fact that the Cell is designed to execute *all* components of a game, not just physics simulation.

The Graphics Processing Unit (GPU) [22] is another design point within the streaming architecture space. GPUs are specialized hardware cores designed to accelerate rendering and display. While Havok's FX allows effect physics simulation on GPUs, GPUs are designed to maximize throughput from the graphics card to the display – data that enters the pipeline and the results of intermediate computations cannot be easily accessed by the CPU. Furthermore, the host CPU is connected to the GPU via a system bus. This communication latency is problematic for physics simulation working in a continuous feedback loop. This may be one reason why Havok's FX only enables effect physics and not game-play physics. This limitation also exists for the Xbox360 system [4], which combines a 3-core CMP and GPU shaders for physics computation. In addition, the 48 GPU shaders will be shared between physics and display processing. However, this system allows the GPU to read from the FSB rather than main memory and L2 data compression reduces the required bandwidth.

In terms of physics engine software, both Ageia [3] and Havok [9] provide their own proprietary SDKs. Open Dynamics Engine (ODE) [7] is the most popular open-source alternative. It has been used in commercial settings, and provides APIs and numerical techniques similar in nature to proprietary engines. ODE is the basis of our physics engine.

3 Physics Simulation and Workload

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

3.1 Our Physics Engine

Our physics engine is a heavily modified implementation of the publicly available Open Dynamics Engine (ODE) version 0.7 [7]. ODE follows a constraint-based approach for modeling articulated figures, similar to [5, 20], and it is designed for efficiency rather than accuracy. Our implementation supports more complex physical functions, including cloth simulation, pre-fractured objects, and explosions. We have parallelized it using pthreads and a work-queue model with persistent worker threads. Pthreads minimize thread overhead, while persistent threads eliminate thread creation and destruction costs.

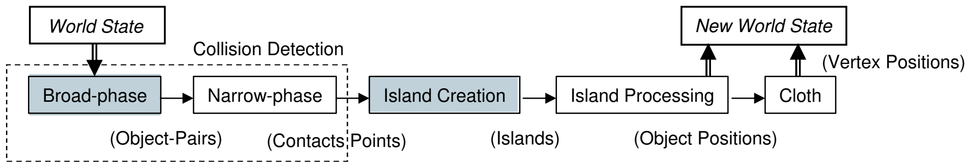


Figure 1: Physics Engine Flow. All phases are serialized with respect to each other, but **unshaded** *stages can exploit parallelism within the stage.*

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

1. Create and setup a dynamics world.
2. While $(time < time_{end})$
   (a) Apply forces to the objects as necessary (e.g. gravity).
   (b) Calculate all pairs of objects that are in contact.
   (c) For each pair of objects in contact do the following:
      i. Compute the contact points and create the associated contact constraints (joints).
      ii. *If an explosive object makes contact with another object, create a sphere representing blast radius.*
      iii. *If a body makes contact with a cloth's bounding volume, insert body on cloth's contact list.*
      iv. *If a pre-fractured object is in contact with a blast volume (sphere) break object into debris.*
   (d) Form groups (islands) of objects interconnected with joints, i.e. find the connected components.
   (e) Forward simulation step: For each island compute the applied loads and the new positions/velocities of each object.
   (f) Check all breakable joints: if joint's applied load has exceeded a threshold, break the joint.
   (g) *Process all cloth objects by taking a forward simulation step.*
   (h) Advance the time: $time = time + \Delta t$
3. End.

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

For our benchmarks, the time-step is 0.01 seconds and 3 steps are executed per frame to ensure stability and prevent fast objects from passing through other objects. We also use 20 solver iterations as recommended by [7].

3.2 Computational Phases of The Physics Workload

The steps in the above algorithm form a dataflow of computational phases shown in Figure 1. Simulations such as cloth and fluid are specializations of this. However, because cloth simulation presents a conceptually and numerically different load than rigid-body simulation we considered it as a separate phase in our study. Below we describe the 5 computational phases in more detail.

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

Narrow-phase. This is the second step of CD that determines the contact points between each pair of colliding objects. Each pair's computational load depends on the geometric properties of the objects involved. The overall performance is affected by broad-phase's ability to minimize the number of pairs considered in this phase. This phase exhibits massive fine-grain (FG) parallelism since object-pairs are independent of each other. ODE originally uses a single joint-group where all contact joints are stored, enforcing an artificial dependency across all object-pairs. We added a data structure for each thread to store created contacts, allowing all object-pairs to be processed in parallel. Based on the number of worker threads, we partition the object-pairs into equal sets.

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

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

Cloth Simulation. We have implemented cloth largely based on Jakobsen's position-based approach[12]. An extension of this approach that handles more general constraints has been proposed by AGEIA[20]. A cloth object is represented using a triangular mesh where each edge represents a length constraint. The constraints are solved using an iterative constraint relaxation solver and the mesh is simulated forward in time using a Verlet integrator. Collision detection is based on a combination of ray casting and axis-aligned bounding volume hierarchies. Collision resolution is based on a vertex projection scheme. This phase also exhibits both CG and FG parallelism. Each cloth object is independent, and the integrator contains independent tasks for each vertex in the cloth object. We parallelized the engine at both the object and vertex levels.

4 Future Physics Workload and Benchmarks

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

We plan to make the suite publicly available to stimulate research in this area.

Our benchmark design was guided by the approach shown in Table 1, and based on the set of required features demonstrated in Table 2. Table 3 explains the benchmarks while Table 4 provides some statistical data.

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

Table 1: Our benchmarks cover a wide range of parameterized situtaions within different game genres.

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

Past benchmarks [29] are limited to small scale rigid-body interactions. Our suite has dramatically increased the types of simulations and number of interacting objects. Based on personal communications with Ageia, future games in development (like Cell Factor Revolution, Auto Assault, and Stoked Rider) targeting the PhysX card use 1000-10,000s of rigid body objects, 10,000s of particles, and 1000s of vertices for deformable meshes (cloth). In contrast, physics targeting a dual-core desktop processor tops out at 500 rigid bodies, 1000s of particles, and no deformable meshes.

5 Experimental Setup

We used a Simics-based [18] full-system execution-driven simulator. Both the cache and processor timing simulators are from the GEMS toolset [19]. All L2 caches are based on 1MB 4-way banks, and Table 5 shows the configuration parameters used for the coarse-grain cores in our simulations. Fine-grain cores are described in section 8.

All benchmarks are executed for 3 frames of physics simulation due to very long simulation times – the average number of instructions in a frame is shown in table 3. Some benchmarks require more than 4 days to complete a single frame when simulating a 4-core design. The simulation proceeds with 0.01 second time steps. The benchmarks are setup so that most of the activity happen in the first 10 frames. The frames 5-7 are executed, and the worst-case frame in terms of execution time is chosen. All benchmarks are warmed up for one physics simulation step prior to execution of the selected frames. The

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

Table 2: Features Found in Our Benchmarks

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

Table 3: Our Physics Benchmarking Suite

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

Table 4: Benchmark Specs

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

Table 5: Coarse-grain Core Design

performance target is 30 frames-per-second (FPS).

The cores simulated within Simics are SPARC ISA processors running Solaris 10 in single user mode. All benchmark binaries executed for performance simulation are compiled with gcc 4.1.1 for the SPARC ISA using the following optimization flags to enable O2 optimizations, SPARC's SIMD instructions, 32-bit pointers, and the pthreads library: [-mcpu=v9 -mtune=ultrasparc3 -mvis -m32 -pthreads -O2]. The SPARC binaries used for performance simulation do not include any graphical display code. For visual verification, the benchmarks with visual display code are compiled for the x86 ISA and executed on real x86 machines. All instrumentation for separating computation phases uses Simics' MAGIC instruction, which is not counted when calculating execution time.

5.1 Interconnect Models

For the on-chip 2D mesh interconnect, we used the data in Table I from [26] for 90 nm technology. The per hop delay is 1 cycle, and the router pipeline is 5 cycles (2GHz clock). This network uses 64-bit flits, and four virtual channels can simultaneously send data. We assume the packet header to be 8-bit long, so each packet's payload is 56-bit.

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

6 Performance Demands of Real-Time Physics Workload

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

Figure 2(a) shows the total execution time of one frame for single-threaded benchmarks running on a 2GHz single-core desktop-class processor with a 1MB L2 cache. The distribution of execution times shows good complexity scaling ranging from *Periodic* to *Mix*. Only two benchmarks, *Periodic* and *Ragdoll* can be completed within a frame's worth of time and the most complicated benchmark *Mix* requires over an order of magnitude performance improvement to reach 30 FPS. Execution time is broken down into contributions by the phases described in section 3.

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

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

6.1 Working Set Analysis: Serial and Parallel Phases

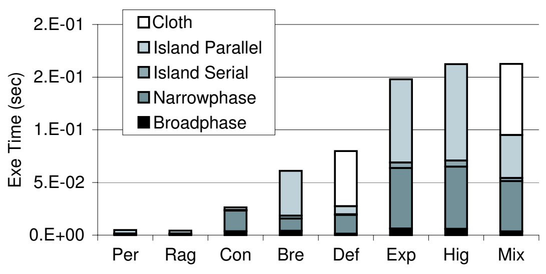
After evaluating different core designs and L2 configurations, we found the L2 cache design to be the dominant factor affecting the serial phase performance. Figure 2 (b) shows execution time for the serial phases as the L2 cache is scaled from 1MB to 32MB by incrementing the number of 4-way 1MB banks. The banks are configured with a point-to-point network and use a 2-level directory based MOESI coherency protocol [19].

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

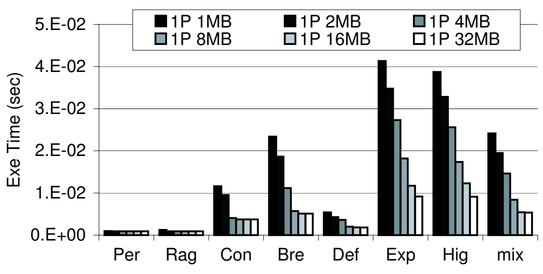
Figures 3(b), 4(b), and 5(a) show the isolated L2 scaling performance data for *Narrowphase*, *Island Processing*, and *Cloth* respectively. Both *Island Processing* and *Cloth* are relatively insensitive to L2 cache scaling. For *Narrowphase*, the benchmarks *Explosions* and *Highspeed* show roughly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

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

When executing the parallel phases in single-thread mode, 1MB of additional L2 space is sufficient to obtain the bulk of the performance. This 1MB can be shared between all three phases since there will be sharing between *Broadphase* and

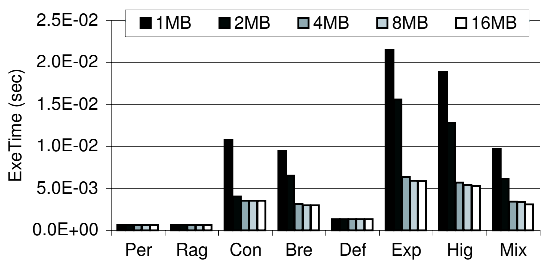


(a)

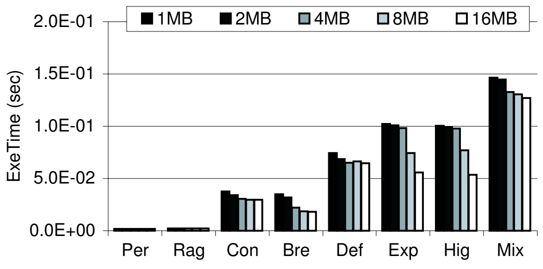


(b)

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

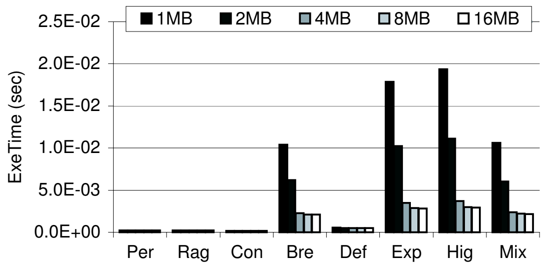


(a)

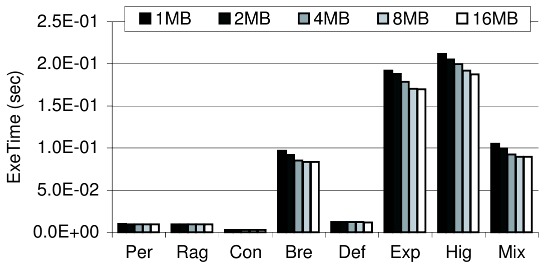


(b)

Figure 3: (a) Performance of Broadphase with dedicated L2 — (b) Performance of Narrowphase with dedicated L2

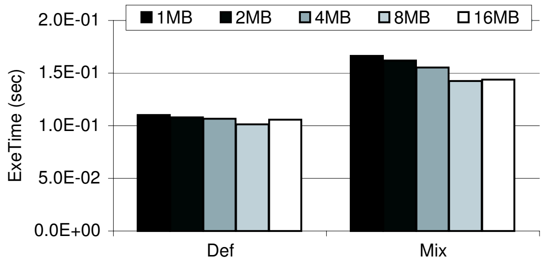


(a)

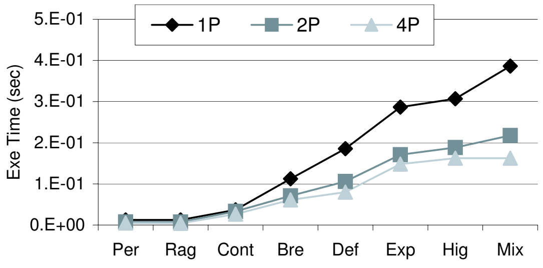


(b)

Figure 4: (a) Performance of Island Creation with dedicated L2 — (b) Performance of Island Processing with dedicated L2

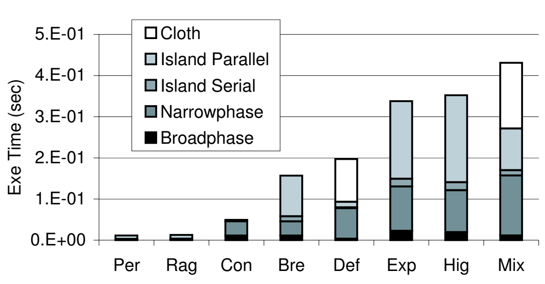


(a)

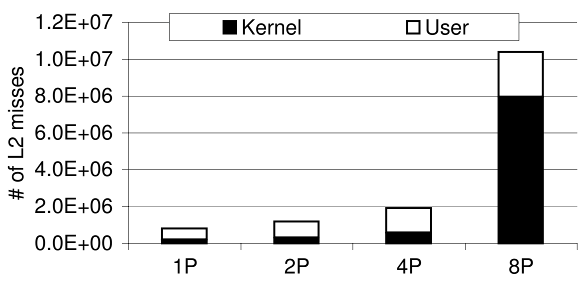


(b)

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

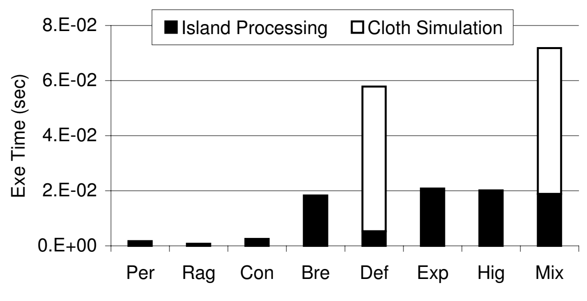


(a)



(b)

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



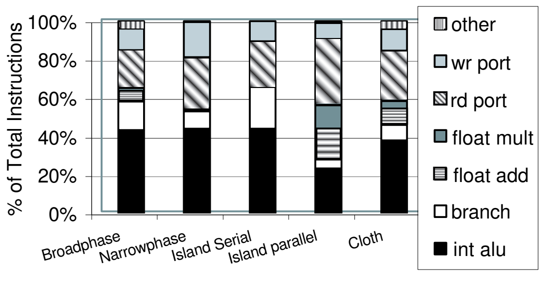


Figure 7: (a) Limit of Coarse-grain Parallelism — (b) Instruction Mix for all 5 Phases

Narrowphase and between *Island Creation* and *Island Processing*. The cache space dedicated to the serial phases should be readable but not modifiable during parallel phases.

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

6.2 Exploiting Coarse-Grain Thread-Level Parallelism

Figure 5 (b) shows the performance as we scale up the number of cores per processor. The L2 cache is sized according to the information presented so far (12MB total – organized into three 4MB partitions [27, 23]:one for *Broadphase*, one for *Island Creation*, and one for the parallel sections). With every additional core, we add an additional worker thread. Future work will examine L2 cache size reduction by prefetching, per-thread cache management, and DMA transfers.

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

Figure 5 (b) clearly shows that the performance improvement starts to plateau at 4 cores. On average, scaling from one to two cores improves performance by 53% and scaling from two to four cores improves performance by 29%. Figure 6 (a) shows the execution time breakdown for a four core processor with 12MB total L2 cache space. The performance has improved by roughly 3X, but we still need an additional 5X improvement to satisfy all benchmarks. Performance starts to degrade at eight cores (not shown). This degradation is surprising, but can be attributed to two main causes: an increased working set in the L2 cache and operating system overhead. For all parallel phases, additional threads consume more cache space by simultaneously accessing data that had not needed to co-exist in the cache. The large increase in L2 misses is shown in Figure 6 (b).

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

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

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

7 ParallAX Architecture

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

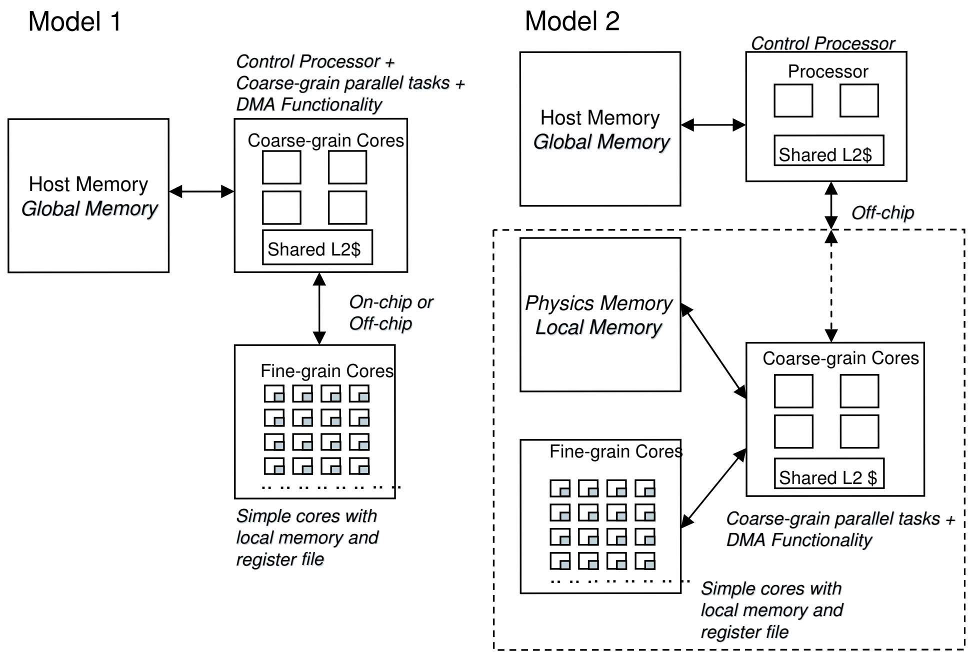


Figure 8: ParallAX - Parallel Physics Accelerator

section.

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

FG parallelism refers to parallelizing object-pairs in *Narrowphase*, degrees of freedom removed in the LCP solver of *Island Processing*, and vertices of each piece of cloth in the *Cloth*. Each FG task is an independent inner iteration of a multiply-nested for loop. Even when dividing up at this granularity, each task is independent, allowing us to get around the fundamental performance limitations from the last section.

FG cores use local instruction and data memories instead of caches, since the instructions and data required for each task are easily determined. Each FG core will iterate through all the tasks assigned to it, and no communication is required between these FG cores as there are no data dependencies between iterations. All FG cores will execute the same kernel at one time due to the dependency between phases of physics computation. This argues for one pool of shared FG resources for all kernels instead of unique cores dedicated to each kernel. Although this observation is similar to unified shader design [21], physics computation is inherently different from graphics where little pipelining of the phases described in section 3 can occur. There is additional overhead involved in pushing the data to the FG cores and retrieving the data back after computation completes.

7.1 Mapping Fine-Grain to Coarse-Grain Cores

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

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

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

Another design issue we explore is buffering tasks at the FG cores to hide communication latency between CG and FG cores. The more tasks that are sent to each FG core at once, the more potential communication latency we can hide, and the looser we can make the coupling between CG and FG cores. However, this requires that we have sufficient buffering space and sufficient application parallelism to overlap communication and computation. Looser coupling of cores will allow the use of less expensive interconnect and the more flexible placement of cores (i.e. off-chip). And looser coupling facilitates the flexible mapping of FG cores to CG cores.

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

Our CG cores and L2 cache banks connect via a 2D mesh, and we assume the same for FG cores. The 2D mesh combines simplicity, area effective design, and power efficiency. Its latency and power consumption are slightly worse than a 2D torus, but its design simplicity should translate to less design effort.

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

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

7.3 Programming Model

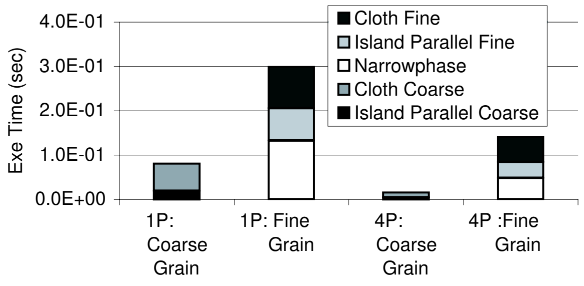
The orchestration of the FG cores need not require ISA modifications. The memory locations of instructions and data structure will need to be remapped into the local memory space of the FG cores, and FG kernel functions will need to be inlined. All of this can leverage existing compilers by adding a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

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

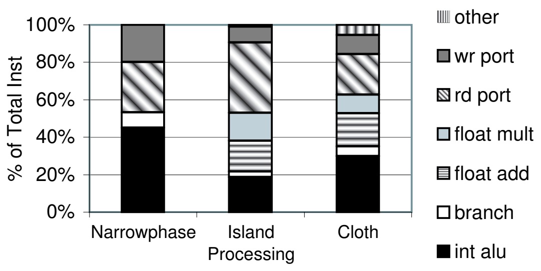
The control packet, in conjunction with the previously described arbiters, sets up the flow of data packets to FG cores. Once the a full set of data is received on a FG core, the kernel id chooses the kernel to execute (kernel code already resides in FG cores). The iteration count indicates the number of iterations to execute (the code will assume all FG tasks start from iteration 0 and the data has been packed accordingly). The task id uniquely identifies the CG thread which submitted the FG request, and the data set id uniquely identifies each FG core. This information will be tracked on the CG core to identify the results returned back from FG cores. Each CG core will have a network interface module to send, receive, and buffer these packets. Each arbiter tracks FG core activity by examining data packet headers.

8 Architectural Design Exploration

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



(a)



(b)

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

8.1 Characterizing Fine-Grain Computation

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

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

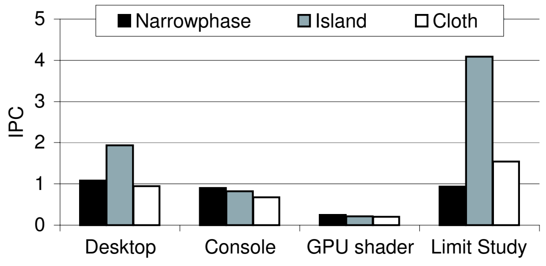
8.1.1 Kernel Characterization

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

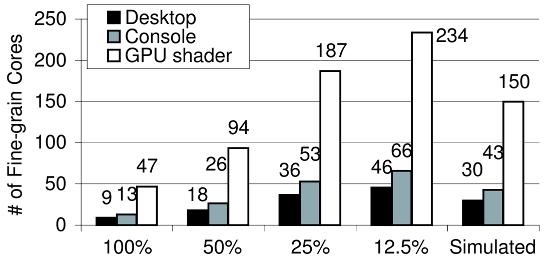
Narrowphase contains 8% branch instructions and few FP adds and multiplies. This contrasts greatly with both *Island Processing* and *cloth*, where these instructions make up 32% and 28% of the total respectively. *Cloth* differs from *Island Processing* with more branches and its use of integer multiplies,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

8.1.2 Memory Required for Instruction and Data Storage

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



(a)



(b)

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

Narrowphase, 0.7KB for *Island Processing*, and 0.9KB for *Cloth*.)

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

8.2 Fine-grain Core Design and Requirements

In order to complete the FG computations within the time left, we would like to use as many cores as required to exploit the level of parallelism needed. In addition to evaluating the performance of a desktop core on these kernels, we examine more area-efficient, simpler cores modeled after next generation console and GPU shader designs. An unrealistically large core design is also used as a limit study on the available instruction level parallelism. Table 6 summarizes these designs. As discussed previously, the memory behavior of the FG cores is extremely regular. CG cores will send all data required to the FG cores – i.e. memory requests at the FG cores will always hit in single-cycle local memory. All cores run at 2GHz.

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

Table 6: Our Fine-Grain Core Designs

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

8.2.1 Number of Finegrain Cores Required

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

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

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

Area Estimation: By using published die areas and photos of *single* cores from Intel Core Duo 2 [24], IBM Cell [11], and Nvidia G80 [25], we derive area estimates for each core type using 90nm technology. The required interconnect area is derived from Table III of [26]. The area estimates for 30 desktop, 43 console, and 150 shader cores are 1388 mm^2, 926 mm^2, and 591 mm^2 respectively. This argues for the simplest cores as the most area-efficient alternative and points to a severe need of area optimization, which will be one main thrust of our future work.

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

8.2.2 Available Parallelism to Hide Interconnect Latency

So far, we have optimistically assumed that the parallel phases of our workload could achieve 100% utilization of the fine grain cores. To verify this, we gather the amount of available FG tasks in each phase, assuming four CG threads. As described earlier, the limit of CG parallelism is determined by the size of large islands. Table 7 shows the amount of parallel FG tasks required to hide communication latency for different core types and interconnect technologies.

With four threads, Figure 11 shows all benchmarks contain enough parallel FG tasks to hide on-chip interconnect latency

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

Table 7: Number of Fine-Grain Tasks Required to Hide Communication

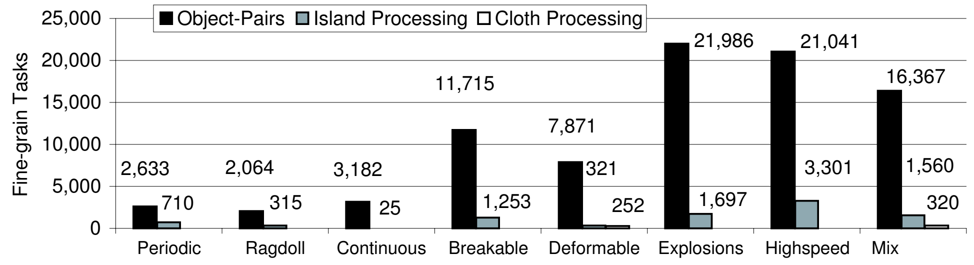


Figure 11: Average Number of Available Fine-grain Parallel Tasks

to the number of FG cores indicated on Figure 10 (b) except *Island Processing* for *Continuous* and *Deformable*, and *Cloth* for *Deformable*. Both benchmarks contain no large islands with more than 25 parallel FG tasks. Figure 6 (a) had demonstrated that *Continuous* already executes at 30 FPS without the use of any FG cores. For *Deformable*, *Island Processing* does not take up a significant component of one frame's time. Both *Continuous* and *Deformable* can achieve 30 FPS without any FG parallelization of *Island Processing*.

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

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

Although cloth is shown to have the least amount of FG parallelism for the next generation of games, the number of vertices used to model each cloth will likely scale up when creating realistic cloth movement further into the future.

8.3 Implementation Alternatives

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

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

When using a GPU for physics acceleration, the GPU is only effective for FG tasks. The model, as shown on Figure 5 of [17], ties a general purpose CPU to the GPU using an off-chip interconnect. The serial and CG computation are done on a CPU which may not be optimized for this workload, and the off-chip latency from the CPU to the GPU will be exposed many times within a frame (similar to our off-chip evaluation) except when handling massive islands or cloths (with 7700 to 9400 parallel FG tasks).

While first-generation physics accelerators may only need to send object position and orientation back to the main processor core(s), future demands will be more significant. Truly immersive reality may only be possible when the results of physics-guided motion are communicated to other components of IE applications. For example, an object that breaks may have a sound attached to it or may alert an AI character.

9 Summary

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

References

[1] HyperTransport 3.0 Spec. http://www.hypertransport.org/docs/tech/HT3pres.pdf.

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

[3] AGEIA. Physx product overview. www.ageia.com.

[4] J. Andrews and N. Baker. Xbox 360 system architecture. In *IEEE Computer Society*, 2006.

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

[6] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference, Los Angeles*, 2000.

[7] Open Dynamics Engine. http://www.ode.org/ode-latest-userguide.html.

[8] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. In *ISCA*, 2002.

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

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

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

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

[13] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. In *IEEE Computer*, 2003.

[14] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002.

[15] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *30th Annual International Symposium on Computer Architecture*, 2003.

[16] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, 1997.

[17] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. *PACT*, 2004.

[18] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer, Feb 2002*.

[19] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. In *Computer Architecture New, Sep 2005*.

[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] V. Moya, C. Gonzalex, J. Roca, A. Fernandez, and R. Espasa. Shader performance analysis on a modern gpu architecture. In *MICRO 38: Proceedings of the 38th annual ACM/IEEE international symposium on Microarchitecture*, 2005.

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

[23] P. Ranganathan, S. V. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, 2000.

[24] P. Schmid and B. Topelt. Game over? core 2 duo knocks out athlon 64. http://www.tomshardware.com/2006/07/14/.

[25] R. Sommefeldt. Nvidia g80: Architecture and gpu analysis. http://www.beyond3d.com/reviews/nvidia/g80-arch/.

[26] 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.

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

[28] M. Taiji. Mdgrape-3 chip: a 165 gflops application specific lsi for molecular dynamics simulations. In *Hot Chips 16*, 2004.

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