Cache Behavior of Real-Time Physics Simulation

Thomas Y. Yeh* Petros Faloutsos†

Glenn Reinman‡

Department of Computer Science, University of California, Los Angeles

Abstract

Real-time physics simulation is becoming increasingly important for creating immersive interactive entertainment applications. Physics simulation's high computational demands along with the applications' real-time constraint pose a unique computational load and challenge for microprocessor designers.

Prior work has shown the tremendous performance required to satisfy physics simulation's real-time constraint. Parallel threads executing on chip-multiprocessors (CMPs) has been shown to be a promising approach for reaching the desired frame-rate. However, detailed cache behavior for CMPs has not been presented.

In this paper, we examine the cache behavior of real-time physics simulation for typical interactive entertainment (IE) scenarios. These scenarios include both constrained rigid body dynamics and cloth, and cover the main genres of games. Our study investigates single-thread and multi-thread executions utilizing private and shared caches. This detailed cache behavior study points toward dynamically adaptable L2 caches to exploit the varying cache demand between threads and computational phases.

*e-mail: tomyeh@cs.ucla.edu

†e-mail: pfal@cs.ucla.edu

‡e-mail: reinman@cs.ucla.edu

Figure 1: Snapshots from the FIFA World Cup 06 game by EA Sports.

1 Introduction and Motivation

Interactive entertainment has evolved into a major industry with a huge user-base. The recent advances in hardware and software have allowed the development of applications that are complex and visually stunning, Figure 1. However, interactive entertainment applications need to keep up with an ever-increasing demand for higher complexity and visual fidelity. To achieve this, future interactive entertainment applications will continue to drive the demand for microprocessor performance. One of the most important, and at the same time computationally demanding, components of such applications is *physics-based simulation*.

Most interactive entertainment applications synthesize the motion of virtual objects based on parameterized pre-recorded motion clips. Naturally, for any given virtual object or character the amount of motion that can be recorded is limited. As interactive entertainment applications depict increasingly complex objects and their interactions, it becomes impossible to pre-record all the necessary motion clips. For example, consider the entire range of motions that a human can do, from everyday motions to sports. Furthermore, consider the secondary elements that need to be animated as well, such as hair and clothes which are highly deformable. For many cases, physic-based simulation is a the right solution and one that offers high levels of realism and automation.

The automation and realism that physics-based simulation offers is accompanied with a considerable computational cost. This cost together with the real-time performance that interactive entertainment ap-

plications require, produces a formidable computational challenge for current and future microprocessors. In this paper, we take an important first step toward understanding the low level behavior of this load by evaluating the cache behavior of real-time physics simulation for single and multi-thread execution.

The contributions of this paper are twofold:

First, the cache behavior data presented will be valuable in balancing die-area allocation for computation vs on-chip storage. This trade-off will likely restrict the size of the last level cache to be much smaller than large sizes proposed for other parallel workloads.

Second, we show that real-time physics simulation's cache demand varies significantly across frames, phases, and islands. As more physics simulation threads execute in parallel, contention for data increases due to the lack of sharing across threads. This behavior is a prime target to be exploited by recently proposed dynamically adapting cache designs.

The remainder of the paper is organized as follows. Section 2 describes the physics-based simulation workload and related background. Section 3 presents an overview of related work. Section 4 discusses our methodology and our experiments. Sectionsr̃efsinglethreadand 6 presents our results for the single and the multiple thread cases respectively. Section 7 concludes the paper.

2 Physics Simulation Workload

In the context of interactive entertainment physics-based simulation synthesizes the motion of the virtual objects by numerically solving the laws of physics. In the real world, an object can affect the motion of another object only through contact forces. Therefore detecting and resolving contact is a key element of physics-based simulation and often it dictates the simulation's performance.

The next section presents the simulator we use in this paper and all the neccessary details.

2.1 Open Dynamics Engine

The publicly available Open Dynamics Engine (ODE) provides a comprehensive soluition for physics-based simulation. It is tailored for interactive entertainment applications where efficiency is often more important than accuracy. Although it focuses on contrained rigid bod dynamics following [5] it is straightforward to combine it with a cloth simulator based on [10]. The algorithmic structure of our physics-based simulator is as follows:

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

The computational load of ODE physics simulation is dominated by two main components: *Collision Detection (CD)* and the *Forward Dynamics Step*. The former uses geometrical approaches to identify bodies that are in contact and the location of contact points. The latter, given the applied forces and torques, first computes the *constraint forces* (both contact and joint), then computes the accelerations, and finally

integrates the accelerations to compute the new position and velocity of every body in the simulated world. With the addition of our cloth simulation code, *Cloth* is now the third major contributor of total execution time.

Collision detection not only is computationally expensive but it also dictates the level of parallelism that exists at an algorithic level: objects that are in direct or indirect contact need to be solved simultaneously while non-interacting objects can be solved independently of each other. Within the ODE framework interacting objects form what is called an *island*. Eah island can be solved independently from the others.

3 Prior Work

Interactive entertainment applications is emerging to become a new important application domain requiring tremendous amount of performance to satisfy its real-time constraint. Despite its social, economic, and technical importance, there has been little academic effor to quantify this workload. Yeh et al. [16] provide a set of benchmarks, evaluate the workload, and propose value and thread-level speculation techniques to improve performance. However, they did not address cache behavior which is the focus of this work. Furthermore, we consider also cloth simulation. Matthews et al. [12] compared the performance counter statistics of 1 second execution between two first person shooter games to music and video playback applications. This work shows the difference between gaming and multimedia applications due to game's content creation tasks. Both papers point to chip multiprocessor (CMP) [13] as a promising approach to providing performance. [1] presents Sony's findings from porting the AGEIA [4] physics engine to work on the Cell [9] processor.

Jaleel et al. [11] analyzed the last level cache performance of the emerging parallel bioinformatics workload. It concludes that large shared last level caches can tremendously reduce misses to memory. For our workload, the conclusion surprisingly is different.

Recently there have been various proposals for hybrid CMP cache designs to merge the benefits of

Figure 2: Snapshots from a sports or action game scenario in raster order.

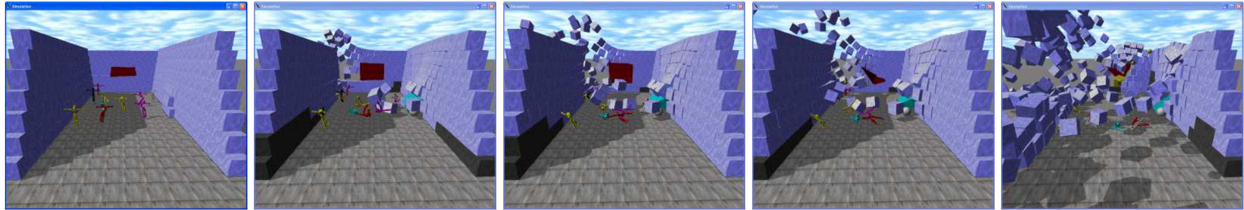

Figure 3: A complex battle scene: A cannon is shooting at the far wall and a car crashes on the right wall.

private and shared L2 caches, namely low access latency and low miss-rate to memory [15, 7, 6, 8, 17]. Our work suggests real-time physics simulation to be an ideal candidate for such designs.

Understanding the behavior of the physics workload is valuable for gaming computer and gaming console manufacturers. Some of them have already produced next generation gaming hardware such as [9, 1]. The physics board by AGEIA[4] is one of the most interesting efforts to accelerate physical simulation with off-the-chip dedicated hardware.

4 Methodology

4.1 Benchmarks

To evaluate the cache behavior of real-time physics simulation, we created three 3 benchmarks to represent the following game genres: Sports/Action, First-Person Shooter (FPS), and Massive Multiplayer Online Game (MMOG). These benchmarks build upon prior work in [16] by adding scalability and cloth simulation.

The benchmarks we have created represent scenes of realistic complexity (interactions) but not necessarily realistic motions. Our benchmarks involve virtual humans, cars, walls, projectiles, and cloth:

- *Humans*: The virtual humans are of anthropomorphic dimensions and mass properties. Each character

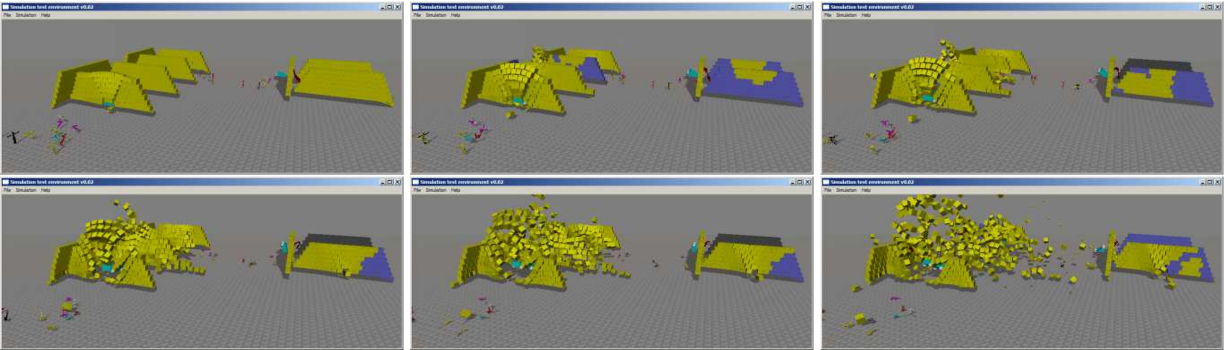

Figure 4: Snapshots of a massively multiple player online game scenario in raster order.

consists of 16 segments (bones) connected with idealized joints that allow movement similar to their real world counterparts.

- *Cars*: The car consists of a single rigid body and four wheels that can rotate around their main axis. Four slider joints model the suspension at the wheels.
- *Walls*: The walls are modeled with blocks of light concrete that are stacked on top of one another.
- *Projectiles*: The projectiles are single bodies with spherical, cylindrical or box geometry.
- *Cloth*: Each cloth is a single soft body modelled by 20x20 number of particles.

The behavior of different entity types affect real-time physics computation in different ways. Bricks that make up walls produce stacking behavior. Humans represent highly articulated objects. Projectiles are small, fast moving objects. Cars are fast moving large objects, and cloth represents particle simulation.

In all benchmarks, the simulator is configured to resolve collisions and resting contact with friction. The three benchmarks are:

- Sports/Action: 2 teams of 15 humans fighting in clusters of 10. Total of 480 rigid bodies.
- First-Person Shooter (FPS): a relatively complex battle scene (Figure 3). A tank is behind the far wall shooting projectiles in different directions. A car crashes on the right wall while two groups of five people are fighting inside the compound. The walls eventually get destroyed and fall on the people.

One large tapestry on the center wall is deformed by moving objects. Total of 616 rigid bodies and 1 cloth made up of 400 particles.

- Massive Multiplayer Online Game (MMOG): a scaled up version of FPS. There are 3 separate battle areas with 60 humans and 25 cars sparsely distributed. There are 3 tapestries, one for each building, Figure 4. Total of 2255 rigid bodies and 3 cloth, each made of 400 particles.

4.2 Cache Simulation

In this section, we describe our method for data collection. All benchmarks utilizes ODE 0.5 compiled for the x86 ISA using gcc 4.1.0 at optimization level -O2 (recommended by ODE), using single precision floating point and the following flags: -ffast-math, -mmmx, -msse2, -msse, -mfpmath=sse and -march=pentium4. The timing data and memory traces are collected with a Intel Pentium 4 Xenon with 512KB L2 cache.

We utilize Pin [2], a dynamic instrumentation tool for application binaries, to generate the memory traces and collect instruction profiles. Pin is similar in functionality to the ATOM [14] toolkit for Compaq's Tru64 Unix on Alpha processors. By inserting instrumentation code, we are able to trace all memory accesses and categorize them into the 4 main components of physics simulation.

The 3 parallel components of physics simulation are all parallelized at different granularities: per island for island processing, per 50 object-pairs for *collision detection*, and per cloth for *cloth* simulation. The thread assignment for all components is done with a greedy algorithm that finds the thread with the least number of memory accesses. In this respect, the threads created are ideally balanced.

We evaluated parallelizing collision at 50-pair granularity instead of per object-pair to simulate parallelizing at the spatial level where nearby object pairs are computed on 1 thread. This spatially-aware assignment removes the unnecessary sharing across parallel *collision threads*, and can be easily realized by leveraging broad-phase collision detection's spatial partitioning.

The memory traces gathered is then fed to our version of DineroIV [3] modified for parallel thread cache

simulation. Dinero is a trace-driven uniprocessor cache simulator.

4.3 Cache Parameters

For the single thread cache simulation, cache sizes equal and greater than 256KB utilize 64B blocks. Smaller caches use 32B blocks. Cache sizes equal and less than 8KB are direct mapped. 16KB is 2-way. 32KB and 64KB are 4way. Larger caches are 8-way associative. For multi-thread simulations, the 4MB cache is 16-way associative. All caches implement LRU replacement policy, demand fetch, writeback, and write-allocate.

5 Single-Thread Characterization

5.1 Single-Thread Performance

Interactive entertainment applications require each frame of animation to be computed at a mininum of approximately 30 frames per second. All necessary components of the application must complete within a fraction of the 33 ms allocated for each frame. For games, the components can include: physics simulation, aritificial intelligence, user input, motion synthesis, networking, audio and video processing, graphics, and game-engine code. Because the complexity of each component varies across different game genres, there is no set standard for the time allocated to physics simulation.

To illustrate the high performance required by real-time physics simulation, we executed the benchmarks on a 2.8 GHz Intel P4 Zeon CPU with a 512KB L2. Each benchmark is executed for 8 frames, and per frame execution time breakdown is shown on Figure 5. Each frame is divided into the 4 main components of *island* processing, *collision* detection, *cloth* simulation, and *misc* which includes all other physics engine computation.

If we assume a generous time allocation of 50% for physics simulation (16.5 ms), none of the frames simulated can be satisfied. Even if 100% of a frame's time is dedicated to physics, the FPS and MMOG

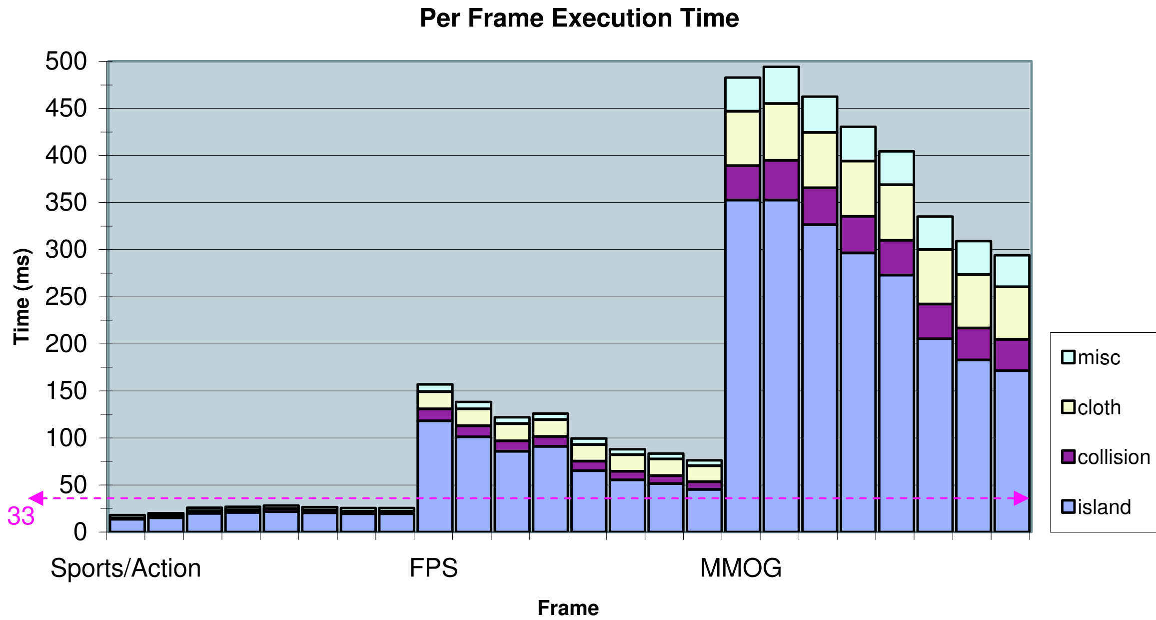

Figure 5: Per Frame P4 Execution Time Breakdown

benchmarks require orders of magnitude performance improvements to enable real-time interactivity. The geometric mean of frame-rates across the 8 frames is 41 for Sports/Action, 7 for FPS, and 2 for MMOG.

As mentioned earlier, different phases of the physics engine are parallelizable. These include the per *island* processing, bounding box and narrow-phase *collision* detection, and *cloth* simulation. We assume the rest of the physics computation is serially executed on the main thread.

Referring back to the data of Figure 5, we see that *island*, *collision*, and *cloth* contribute to the bulk of the computation. It is interesting to note the significant time-changing behavior of execution time across frames. Most of this variation can be traced to *island's* fluctuation, and the rest of the computation requires roughly the same amount of time across frames per benchmark.

5.2 Instruction Profile

Figure 6 presents the dynamic instruction profile for single-thread execution. For each benchmark, we show the total number of instruction executed per frame. To give an indication on memory latencies effect

Instructions per Frame (Millions)	Sports/Action	FPS	MMOG
Average	52	184	602
Frame 1	46	248	782
Frame 2	42	225	724
Frame 3	54	205	677
Frame 4	56	186	636
Frame 5	59	172	555
Frame 6	56	153	508
Frame 7	53	147	482
Frame 8	53	136	455
% of Total Inst	Sports/Action	FPS	MMOG
Island	84%	71%	71%
Collision	6%	8%	8%
Cloth	0%	17%	18%
Misc	10%	3%	3%
% of Total Exe Time	Sports/Action	FPS	MMOG
Island	76%	67%	64%
Collision	10%	9%	10%
Cloth	0%	17%	16%
Misc	14%	6%	10%
Mem Access Per Instruction	Sports/Action	FPS	MMOG
Total	0.56	0.56	0.66
Island	0.56	0.57	0.57
Collision	0.50	0.45	0.46
Cloth	0.00	0.55	0.55
Misc	0.51	0.52	0.60
Mem Read Per Instruction	Sports/Action	FPS	MMOG
Total	0.45	0.44	0.50
Island	0.47	0.47	0.47
Collision	0.33	0.30	0.30
Cloth	0.00	0.38	0.38
Misc	0.39	0.41	0.48

Figure 6: Dynamic Instruction Profile

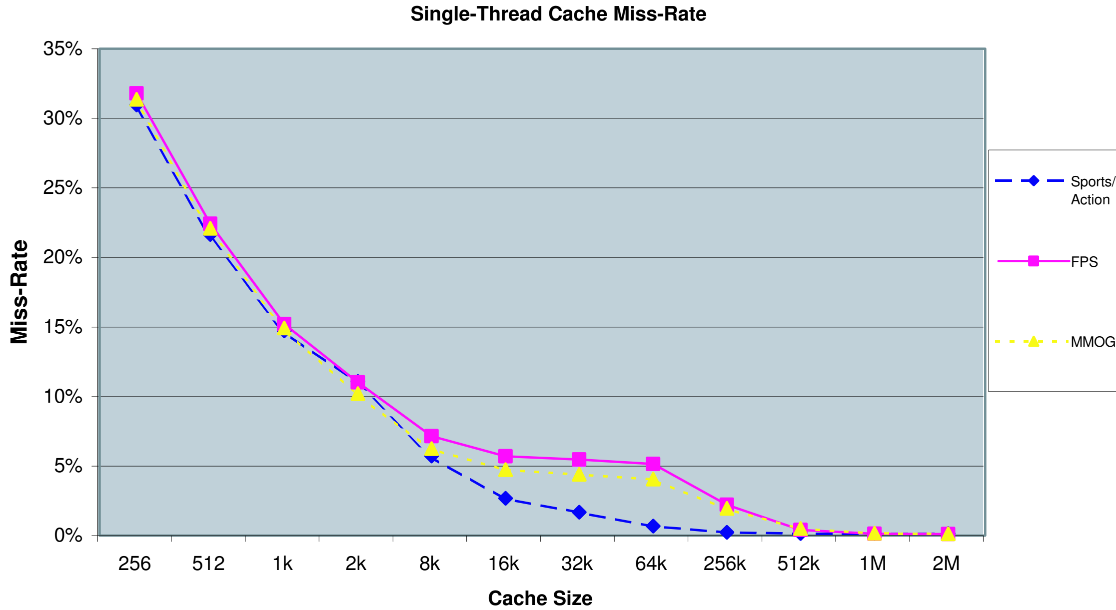

Figure 7: Single Thread Cache Behavior

on performance, we compute memory accesses per instruction and memory read per instruction. All data except instructions per frame are averaged across all frames and categorized into the 4 components described above.

On average, rougly 600 memory references are generated per 1000 instructions executed, and 80% of the accesses are reads. *Island* computation shows the highest ratios of accesses and reads per instruction while *collision* computation shows the lowest.

5.3 Cache behavior

Before exploring the massive multi-thread cache behavior of physics simulation, we evaluate single-thread cache miss-rate on one level of cache spanning from 256B to 2M. The detailed cache configuration for each size is described in section 4.

This data, as shown on Figure 7, gives valuable information for designing different cache levels (filter

L0, L1, and L2) for real-time physics simulation.

Assuming inclusion between cache levels, this graph shows the percentage of all memory accesses that will be sent to the next level in the memory hierarchy (next level cache or memory) if a given sized cache is used for this level.

Figure 7 shows 2 distinct knees in the miss-rate curves. The first occurs between 8k and 16k, and the second happens between 256k and 512k. These 2 sets of sizes represent cost effective sizes for L1 and L2 caches for executing this workload with a single thread.

At first glance, the general trend of increased missrate with increasted physical complexity does not hold when comparing the data of *FPS* and *MMOG*. *MMOG* is similar to *FPS* but with scaled up number of entities and a different distribution of objects. However, the scaling factor used for different type of entities are different. The number of walls, humans, and cars is scaled by 3, 6, and 25 respectively. Walls make up the most complicated islands, and *MMOG's* wall island to simpler island ratio is much lower than that of *FPS*. This ratio difference explains this unexpected cache behavior, and this reason will be corroborated by per island miss-rate data in the next section.

To gain further understanding of physics simulation's cache behavior we breakdown the single-thread cache miss-rate into the four components of *island*, *collision*, *cloth*, and *misc*. In the interest of clarity, we only show the parallelizable components in Figure 8. There is no cloth simulation in *Sports/Action*, so its contribution is always 0%.

Island processing demands the most amount of storage for a low miss-rate while *cloth* simulation's working set is easily satisfied by a small 16KB cache. Collision detection shows the most percentage of compulsive misses. This data shows that physics simulation's demand for on-chip storage varies across different phases of computation which repeats in the same order for every frame. In addition, this storage demand varies across different frames.

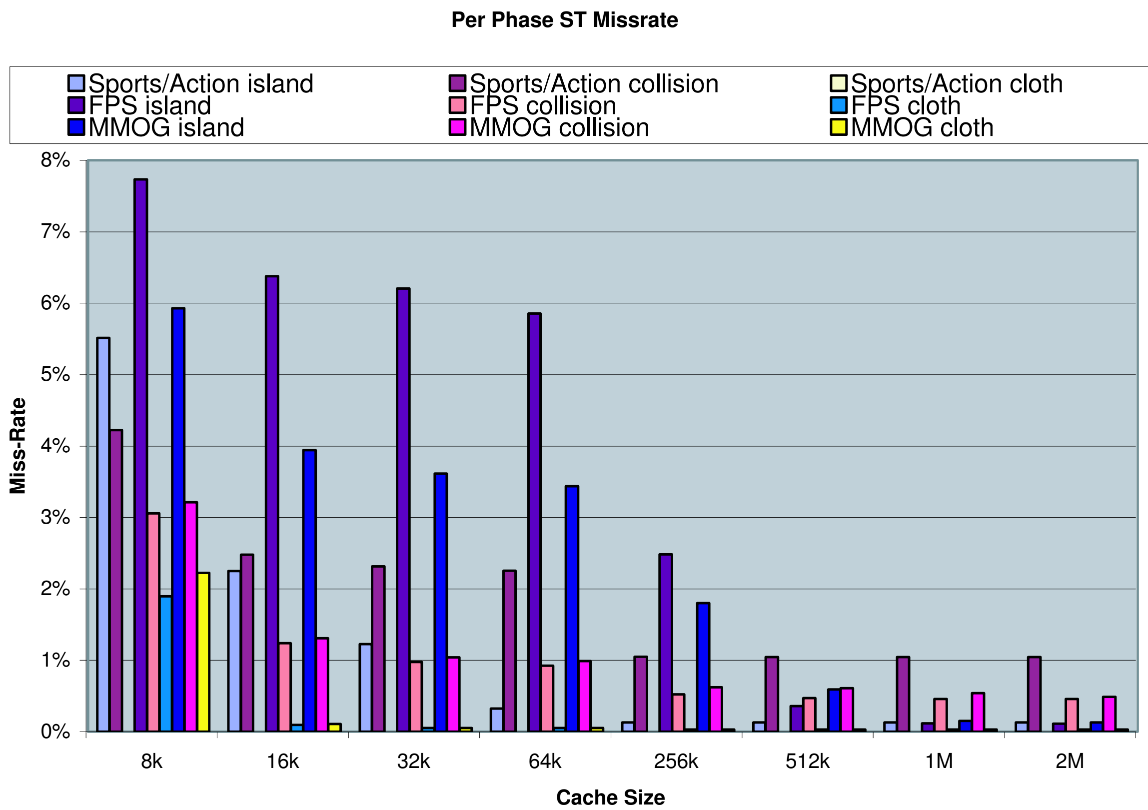

Figure 8: Per Phase Cache Behavior

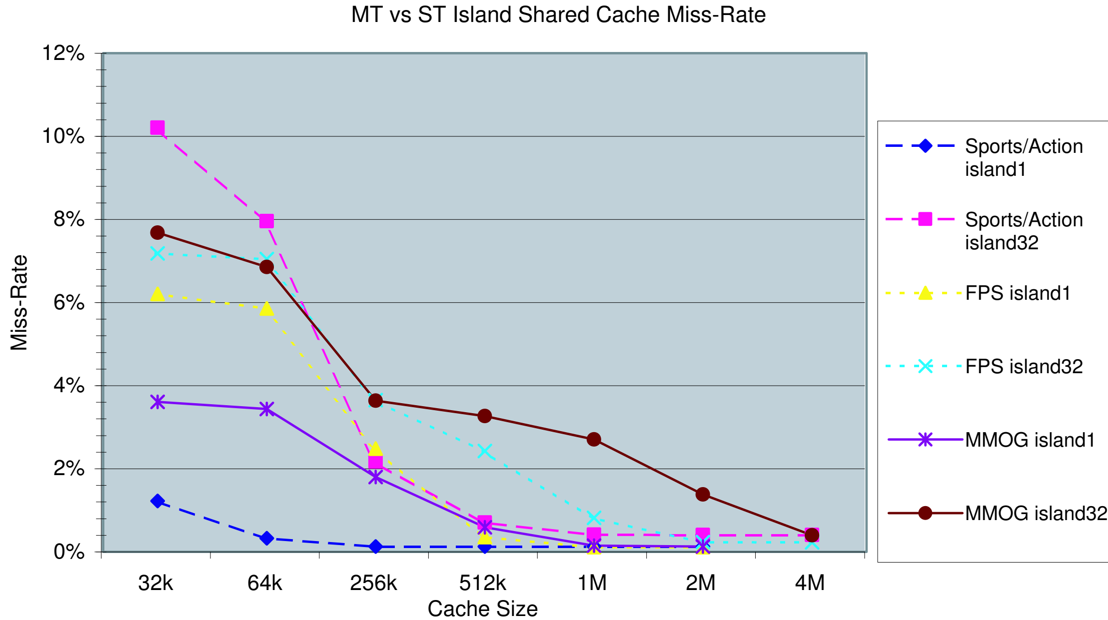

Figure 9: Multi-Thread vs. Single-Thread Cache Behavior

6 Multi-Thread Characterization

Based on the single-thread data, we concentrate on *island* processing for the rest of the paper. In this section, we examine the cache behavior of executing parallel *island* processing with 32 cores.

Figure 9 presents the miss-rates of executing only *island* processing serially on 1 core and in parallel on 32 cores. The side-by-side data for serial vs parallel execution clearly shows the degradation in cache miss-rate. This data represents the cache miss-rate of 1 level of cache storage, similar to Figure 7.

For *Sports/Action*, multi-thread execution increases the cache miss-rates of small sizes by as much as 10X. With *FPS*, the miss-rate is consistently higher by roughly 1% until a 1MB cache.For *MMOG*, we see a 2X increase until the size of 4MB.

Because each island's computation is completely independent of other islands (independent of the method for solving), there is no sharing at all between the 32 threads. This behavior is very different from other emerging parallel workloads such as bioinformatics [11].

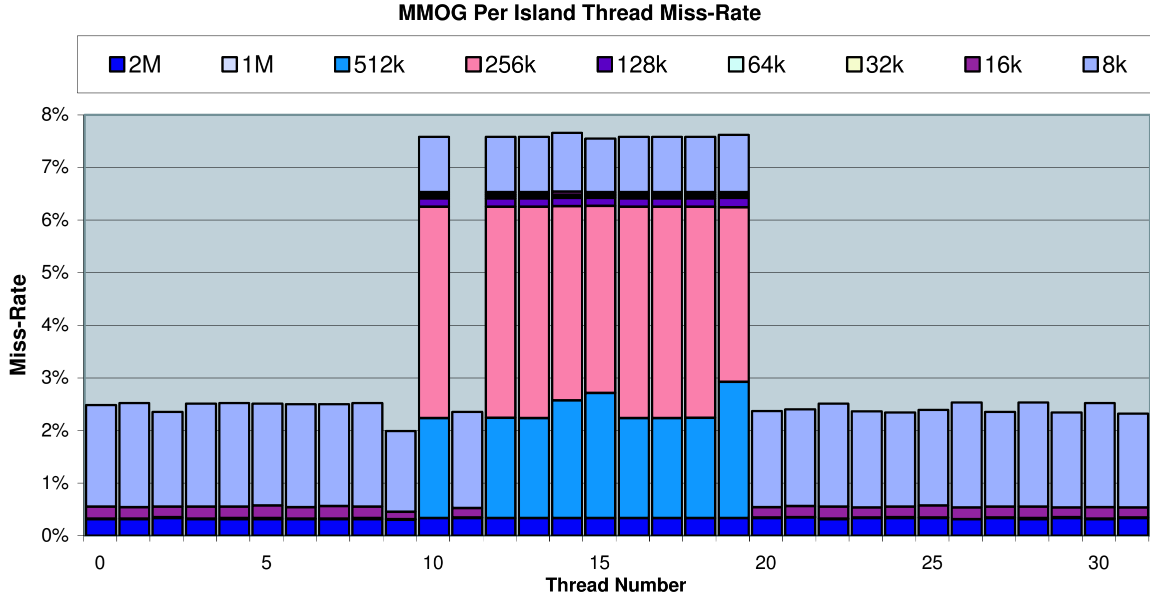

Figure 10: 32 Core Per Thread Cache Behavior

To gain a better understanding of multi-thread cache behavior, we capture the miss-rates of every thread during *island processing* with private caches. Figure 10 shows this data for the *MMOG* benchmark. This figure is a stacked bar graph where each bar shows each thread's miss-rate for different cache sizes. Each cache size's miss-rate is determined adding its missrate with contributions from larger sizes. For example, the 256K cache miss-rate for thread 10 is 6.2% and the 2MB cache missrate for the same thread is only 0.3%.

This data shows 2 distinct behaviors among all 32 threads. The first set includes threads 10, 12, 13, 14, 15, 16, 17, 18, 19, and 20. These threads require 1MB of storage to obtain $< 1\%$ miss-rate. The rest forms the second group, and these only require 16KB of storage for $< 1\%$ miss-rate.

The first set of threads is clearly dominated by the stacking behavior of the 9 walls. Each wall is assigned to one thread based on the parallelization scheme described in section 4.

Based on the data showing differing cache demands across threads, phases, and frames, a dynamically adaptable last level cache as described in prior work [15, 7] seems promising. Next, we evaluate the perfor-

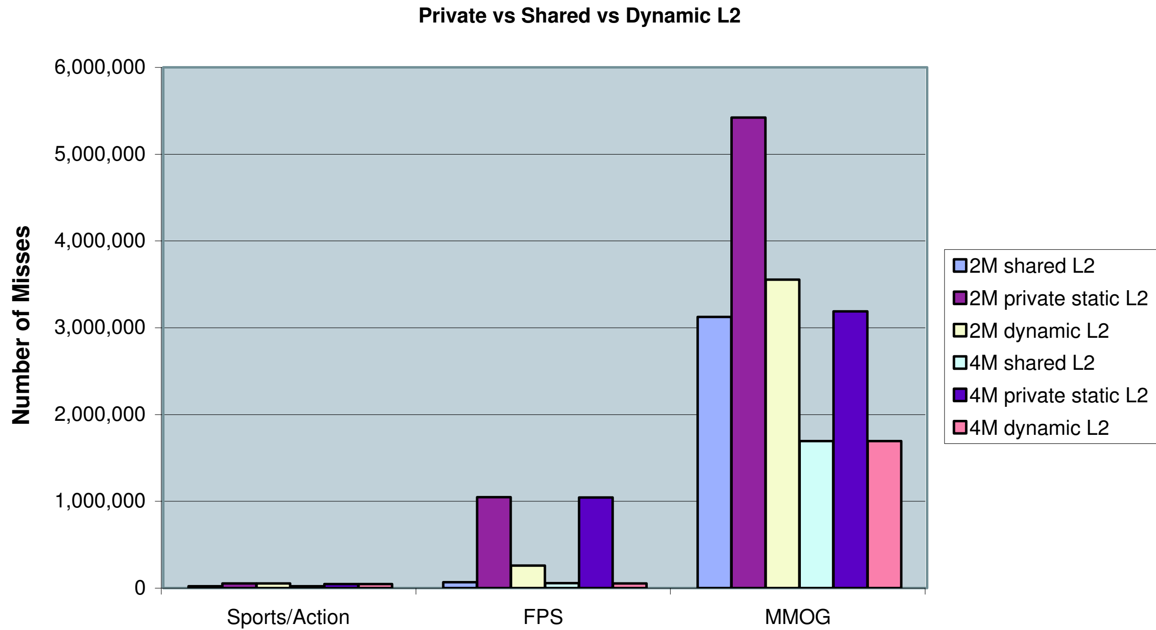

Figure 11: L2 Cache Behavior

mance of private, shared, and dynamically partitioned L2 caches.

6.1 L2 Cache Performance

For this study, we allocate 32KB private L1 caches to each of the 32 cores. For each type of L2 cache, we evaluate versions with 2MB and 4MB of total storage. The total cache resource is evenly distributed for private L2 caches, 64KB and 128KB for each core respectively. For the dynamically partitioned L2 cache, we follow the partitioning scheme described in [15] using granularity of 64KB and 128KB for 2MB and 4MB versions respectively.

Figure 11 shows the total number of misses out to memory for 1 step of simulation. The varying demand across threads for *FPS* and *MMOG* leads to ineffective utilization of statically partitioned cache resources. The dynamically paritioned L2 shows promise with miss-rates comparable to that of the monolithic shared cache. Future work will include the evaluation of actual execution time performce of different L2 organizations.

7 Conclusion

In this paper we have studied the cache behavior of an important and emerging workload, plysics-based simulation, as it applies to interactive entertainment applications. This embarrassingly parallel workload will be a key driver for future CMP designs with massive number of cores.

We have shown detailed cache behavior data useful for balancing die-area allocation for computation vs on-chip storage. Given the massive performance required for real-time physics simulation, this trade-off will restrict the last level cache size to be much smaller than what is proposed for other parallel workloads.

Real-time physics simulation's cache demand varies significantly across frames, phases, and islands. This behavior naturally suits recently proposed dynamically adapting cache designs. As more physics threads execute in parallel, increased contention will require management to prevent thrashing.

Looking forward, we plan to evaluate the performance implications of our findings as well as the area trade-off between cores and on-chip storage.

References

[1] High Performance Physics Solver Design for Next Generation Consoles. http://www.research.scea.com/research/pdfs/.

[2] Pin Homepage. http://rogue.colorado.edu/pin/.

[3] Dinero IV Homepage. http://www.cs.wisc.edu/ markhill/DineroIV.

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

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

[6] B. Beckmann and D. Wood. Managing wire delay in large chip-multiprocessor caches. In *The 37th Annual IEEE/ACM International Symposium on Microarchitectur*, 2004.

[7] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA '06: The 33rd Annual International Symposium on Computer Architecture*, 2006.

[8] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing. Optimizing replication, communication and capacity allocation in cmps. In *The 32th ISCA*, 2005.

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

[10] T. Jacobsen. Game Developers Conference, 2001.

[11] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. Last-level cache (llc) performance of data-mining workloads on a cmp–a cache study of parallel bioinformatics workloads. In *HPCA '06: Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[12] B. Matthews, J. Wellman, and M. Gschwind. Exploring real time multimedia content creation in video games. In *6th Workshop on Media and Streaming Processors*, 2004.

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

[14] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[15] Thomas Y. Yeh and Glenn Reinman. Fast and fair: data-stream quality of service. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, 2005.

[16] Tom Yeh, Petros Faloutsos, and Glenn Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Video Game Symposium*, 2006, to appear.

[17] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled cmps. In *The 32th ISCA*, 2005.