Accelerating Real-Time Physics Simulation by Leveraging High-Level Information

Thomas Y. Yeh* Petros Faloutsos†

Glenn Reinman‡

Department of Computer Science, University of California, Los Angeles

Abstract

Simulating the physical laws that govern the motion of complex groups of objects is becoming increasingly important for interactive entertainment applications. Looking forward in the future, the computational demands of such applications along with the requirement for real-time performance poses a unique computational load and challenge for microprocessor designers. In this paper, we introduce real-time physical simulation, as it applies to interactive entertainment applications, as a novel area of research in architecture. For the most demanding example we show, the worst case execution time for one frame's physics simulation is 133ms. This is 4 times the time allocated for an entire frame's computation of all tasks, not just physics simulation. We motivate the challenges and the idiosyncrasies of this computational load by analyzing one of its most complex elements, collision detection*, and suggest ways to accelerate it.*

*e-mail: tomyeh@cs.ucla.edu
†e-mail: pfal@cs.ucla.edu
‡e-mail: reinman@cs.ucla.edu

1 Introduction and Motivation

Interactive entertainment is one application that truly drives the demand for microprocessor performance. Future interactive entertainment applications will continue this trend, exploring more realistic, immersive virtual worlds populated with ever increasing numbers of objects and characters. Such applications will be composed of a diverse set of computationally demanding tasks. One critical component of this is modeling how such objects and characters move in a virtual environment and interact with one another.

Most interactive entertainment applications still make use of pre-recorded motion clips to synthesize the motion of virtual objects. But as these objects and their interactions grow increasingly more complex, it has become impractical to record the entire set of possible motions for a given object. *Physics-based simulation* has emerged as an attractive alternative to kinematic techniques, providing high levels of physical realism through automated motion calculation.

The benefits of physics-based simulation over current techniques comes with a considerably higher computational cost. Consider a multi-player on-line environment with 15 users racing cars around busy city streets littered with other cars, pedestrians, and other objects. The staggering amount of calculation required to model the complex interactions and collisions that could occur between participants and objects would be beyond any conventional processor. As demand for heightened realism and complexity scales, future hardware for interactive entertainment will need to scale performance to match this demand.

Furthermore, to maintain a fluid visual experience, interactive entertainment applications typically provide at least 30 frames of display each second. This minimum frame rate provides a bound on the amount of time a single frame can take - 1/30th of a second (i.e. 33 ms). This is the total time that will be allotted to physics-based simulation, graphics display, path-finding or other AI activities, and any other game engine code to glue everything together. For the most demanding example we show, the worst-case execution time on a 2.8GHz Pentium4 for one frame is 133ms with an average of 55ms.

This demand for performance is somewhat mitigated by the fact that physics-based simulation has

tremendous amounts of parallelism. Chip multiprocessors [19] and other multi-threaded processor designs have the potential to exploit this, but certain parts of physics simulation are more amenable to parallelization than others. Moreover, as the complexity of virtual worlds increases, even embarrassing amounts of parallelism may not be able to accelerate performance enough to satisfy the frame rate constraint.

In this paper, we make the following contributions:

- We introduce physical simulation, as it applies to interactive entertainment applications, as a novel area of research in microarchitecture.

- We motivate the challenges and the idiosyncrasies of this computational load by analyzing one of its most complex elements, collision detection, and we suggest ways to accelerate it. In particular:

  - We explore the locality and correlation of control values with higher-level application abstractions of physics simulation

  - We propose an architectural approach to leverage this higher-level application data for control speculation, along with alternatives for communicating higher-level application data to the architecture level.

  - We propose and evaluate a scheme to increase the parallelism in collision detection by leveraging object locality. We consider the hardware cost of such a scheme, and the impact of misspeculation.

The rest of the paper is organized as follows. In section 2 we discuss physics simulation and the specific engine we use in this paper, along with a set of benchmarks we have created to capture the key idiosyncrasies of interactive entertainment applications. Section 3 identifies important high-level features of the application data. Section 4 explores a branch prediction architecture that leverages this high-level application data. Section 5 proposes a scheme to parallelize collision detection, again leveraging this high-level application data. We conclude in section 6.

2 Physics Simulation

The laws of physics offer the most general constraint over the motion of virtual objects. Not only do they guarantee realistic motion, but they also avoid repetition. Any variation in the initial conditions (i.e. contact points) will produce a different motion. In a sense, the set of possible actions is as large as the domain of the initial conditions, and not restricted to a small set of recorded motions. Once the equations of motion are provided for each object in a virtual world, motion can be computed automatically based on the applied forces and torques.

In this study, we focus exclusively on constrained rigid body simulation [7, 1, 10] and leave the soft-body simulation domain for future work. In the majority of games, the central elements are humanoid characters. Humanoid motion is dominated by the rigid body motion of the character's body parts. Soft-body simulation such as flesh, cloth and hair animation are typically secondary effects.

The physics board by AGEIA [1] is one of the most interesting efforts to accelerate physical simulation with dedicated hardware. However, in addition to being off the chip, its current version seems to target mostly the main parallel elements of the physical simulation load.

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

To evaluate the use of application-level information, we employed a freely available physics engine and constructed three diverse benchmarks.

2.1 Open Dynamics Engine

We make use of the Open Dynamics Engine (ODE) to evaluate the ideas and designs proposed in this paper. ODE follows a constraint-based approach for modeling articulated figures, similar to [3]. ODE is specifically designed for efficient rather than accurate computation, and is specifically tuned for constrained rigid body dynamics simulation. Applications employing ODE would utilize the following high-level algorithmic structure:

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

As objects move in space they come into contact - the resolution of this contact is the most complex part of physics simulation. 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

Figure 1: Battle 2

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.

2.2 Physics Benchmarks

Name	Number of Islands (Max, Min, Avg, Dev)	Island Complexity	Temporal Behavior	Number of Spaces	Inter-space Comm
CrashWall	105, 99, 101, 1.4	wall, car, sphere projectiles	stable, abrupt change	1,3	range from none to high
Battle	120, 2, 93, 18	groups of humans projectiles	stable, fast change	1, 3	none
Battle2	156, 113, 134, 18	multiple walls, humans, car, simple projectiles	fast changing	1,15	high

Table 1: Parameters Affecting Computation Load

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

- *Humans*: The virtual humans are of anthropomorphic dimensions and mass properties. Each character consists of 16 segments (bones) connected with idealized joints that allow movement similar to their real world 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 - these are shot from a stationary tank.

The behavior of different entity types affect collision detection computation in different ways. Bricks that make up walls produce stacking behavior. Humans represent highly articulated objects. Projectiles are small, fast moving objects, and cars are fast moving large objects.

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

- CrashWall: Extreme-speed Car crashing on wall, tank shooting projectiles - a high speed car (velocity 200Mph) crashing into a wall, while a tank shoots varying shape projectiles towards the wall. The wall consists of a large number of blocks.
- Battle: Battle scene 1 - One group of 10 humanoids being attacked by a tank. 2 groups of 4 and 6 humanoids crashing into each other.
- Battle2: Battle scene 2 - a relatively complex battle scene (Figure 1). 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.

Table 1 summarizes the quantitative differences between benchmarks.

3 Application-Level Correlation and Locality

Intuitively, the low-level behavior of physics simulation code should have a great deal of locality and correlation with higher-level application constructs. Motion at a simulation step by step basis should be smooth and collisions between particular objects can persist across many steps – unless they are from objects moving extremely fast. Behavioral locality in the form of temporal coherence has been shown by prior collision detection work [16, 13, 15, 6]. Temporal coherence describes the fact that in dynamic environments, objects

do not move large distances between consecutive steps of physics simulation, except for objects with high velocities.

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

To better demonstrate this notion of locality and correlation with high-level application constructs, we focus on the collision detection part of the physics simulation loop, and show how the notion of object-pairs effectively correlates with branch behavior in collision detection.

3.1 Collision Detection

Collision Detection is a well studied problem and an excellent collection of theory and available software can be found at [8]. The brute force approach to CD would be to compare each object with all of the other objects which results in $O(NxN)$ complexity, where N is the total number of objects. The most common approach to speed up CD is to use a two-step algorithm [11] composed of *broad-phase* and *narrow-phase*.

3.1.1 Broad-Phase CD

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

To more accurately characterize broad-phase, we can further break down this task into two steps: (1)

update of spatial partitioning structure using new object position and orientation, and (2) filter object-pairs by bounding box comparisons.

3.1.2 Narrow-Phase CD

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

A recent study [17] examines the tradeoff of broad-phase accuracy vs total computation time for different broad-phase methods. The results show that broad-phase needs to be very fast, even at the expense of generating a larger number of collision pairs because of the dependencies between not only broad-phase CD and narrow-phase CD, but also between CD as a whole and the forward dynamics step.

3.2 Correlation and Locality Exploration

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

It is particularly difficult for branches in broad-phase CD to leverage traditional means of correlation. First, broad-phase CD does not use a consistent ordering when iterating through the set of available objects – something which can be particularly harmful for predictors that use branch history to correlate. The ordering of pair-wise comparisons is based on the spatial partitioning commonly done by broad-phase. Second, a diverse set of objects which may have no correlation or interaction with one another make use of the same code to detect collisions – something which can impair predictors that correlate on the PC of the branch.

However, at the application level, there are some factors that can be correlated to help branch prediction

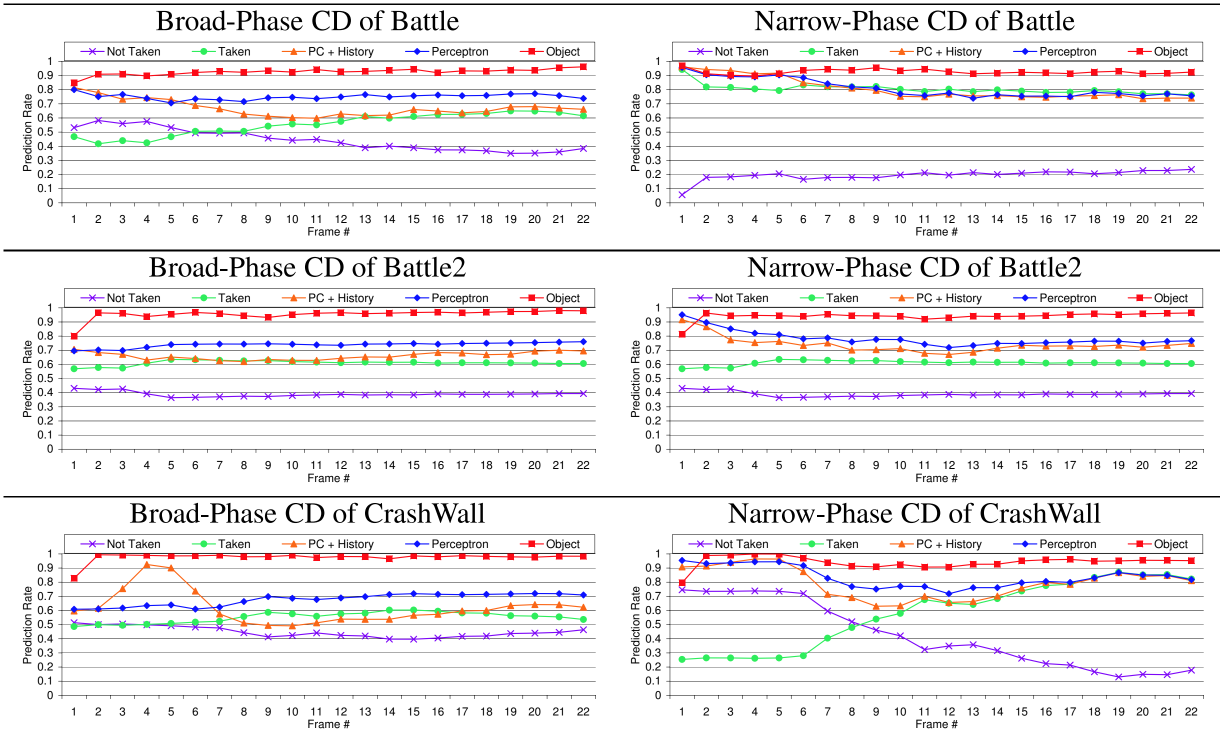

Figure 2: Correlation of the branch prediction rate with the program counter (PC), branch history, and high-level objects.

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

Branch prediction in narrow-phase CD may do slightly better than broad-phase because it is already looking at a filtered set of object pairs. If broad-phase had done a good job, the branches in narrow-phase will be strongly biased. But as [17] points out, broad-phase accuracy needs to be balanced with broad-phase's execution time.

To explore this further, we demonstrate how predictable the main conditional branches of broad-phase and narrow-phase CD are when correlated with the PC of the branch, branch history, and the two objects being compared. Figure 2 presents data for the three benchmarks detailed in Section 2.2 – in each case we

show locality for broad-phase CD on the left and narrow-phase CD on the right. The y-axis in each graph represents the branch prediction accuracy during the execution of the frame (only counting the predictions of correct path instructions). Each frame in the execution of the benchmark is tracked along the x-axis. We evaluate five architectures: a simple Not Taken predictor (always guess branch not taken), a simple Taken predictor (always guess branch taken), a gshare [18] predictor (PC + History) (16KB), a Perceptron branch predictor [12] (16KB), and an object-pair predictor (16KB). This latter predictor correlates with the base address of the two objects under consideration.

For broad-phase CD, the object-based predictor clearly outperforms any other option by a wide margin. The gshare predictor ranges in performance – particularly for the CrashWall benchmark. The object-based predictor improves over the perceptron predictor by an average of 22% for broadphase and an average of 13% for narrowphase.

The perceptron predictor does a little better than other PC-based methods, but still cannot leverage the rich correlation that exists with the higher-level application notion of the object-pair.

Narrow-phase CD performs similarly, but due to the filtered nature of the objects going through narrow-phase, the conventional approaches perform slightly better.

4 Branch Prediction for CD

Based on the branch correlation with object-pair addresses we saw in section 3, we propose an approach to leverage this information at the architecture level and improve control speculation for collision detection.

4.1 Object-Pair Based Branch Prediction

There has been an enormous amount of prior work on branch direction prediction. And while schemes have addressed correlating with global behavior [20], reducing aliasing [18, 5], improving correlation [12], or even focusing on specific branch types [4], all of these techniques rely on information present in the archi-

tecture to help make an accurate prediction. The majority of approaches are based on some combination of the program counter (PC) and either local or global branch history. All of these approaches are orthogonal to our goal, which is to improve correlation and locality using application-level information in the architecture.

We consider indexing a pattern history table (PHT) - a table of two-bit counters indicating a prediction of taken/not taken - using the base address of the current object-pair under consideration. In collision detection, objects are compared in pairs, and therefore we will augment the architecture with two registers to hold the base addresses in memory of the two objects under consideration at any point in time. These registers will not be visible to the compiler, and techniques to set them will be discussed in section 4.

While object-pair information correlates well with certain control decisions, it does not work for others. We must also consider how to avoid using object-pair information in cases where it is not helpful. One approach is to leverage existing confidence techniques to selectively use the object-pair register only in cases where it is useful. Consider a simple pattern history table (PHT) indexed via PC and the object-pair register. This can be used in concert with a conventional PC-based PHT. An additional PHT can be used to select which PHT to use for a given prediction, just as in the bi-mode predictor [14].

4.2 Loading the Object Registers

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

The most straightforward technique conceptually, would be to augment the ISA with new instructions that propagate high-level information. The use of specialized move instructions, for example, would allow the compiler or application writer to place values into the object registers or clear the object registers. Specialized instructions would increase code size in addition to the cost of consuming potentially scarce opcodes in the ISA.

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

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

5 Increasing Parallelism for CD with the Object Table

High-level application information can also be leveraged by adding application-specific structures that accelerate certain components of execution. For example, a texture cache [9] is an application-specific structure that helps GPUs achieve higher performance. Another example is network processors, which use content-addressable memories for fast searches [2]. In this section, we propose an application-specific structure to add parallelism to collision detection.

To show the performance potential of decoupling collision detection by the use of object-pair filter, we measured the contribution of both broad-phase and narrow-phase on collision detection's execution time.

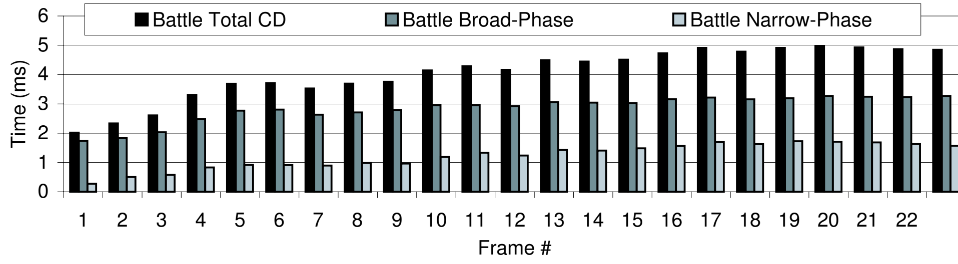

Figure 3: Battle Execution Time Breakdown for Collision Detection

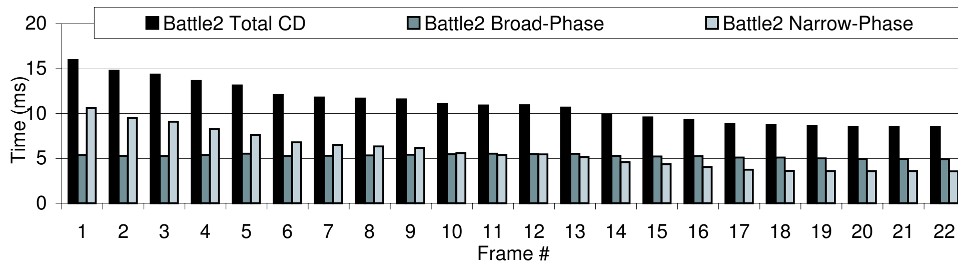

Figure 4: Battle2 Execution Time Breakdown for Collision Detection

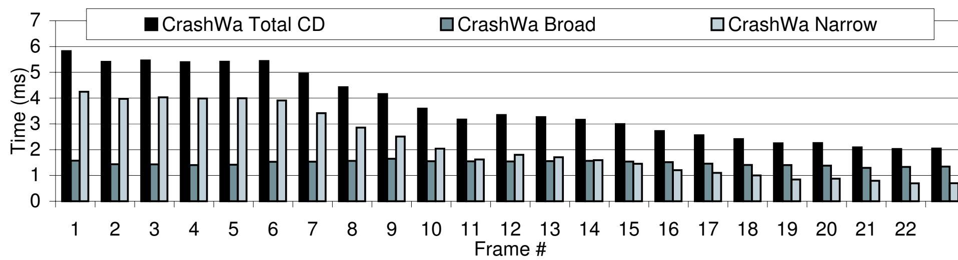

Figure 5: CrashWa Execution Time Breakdown for Collision Detection

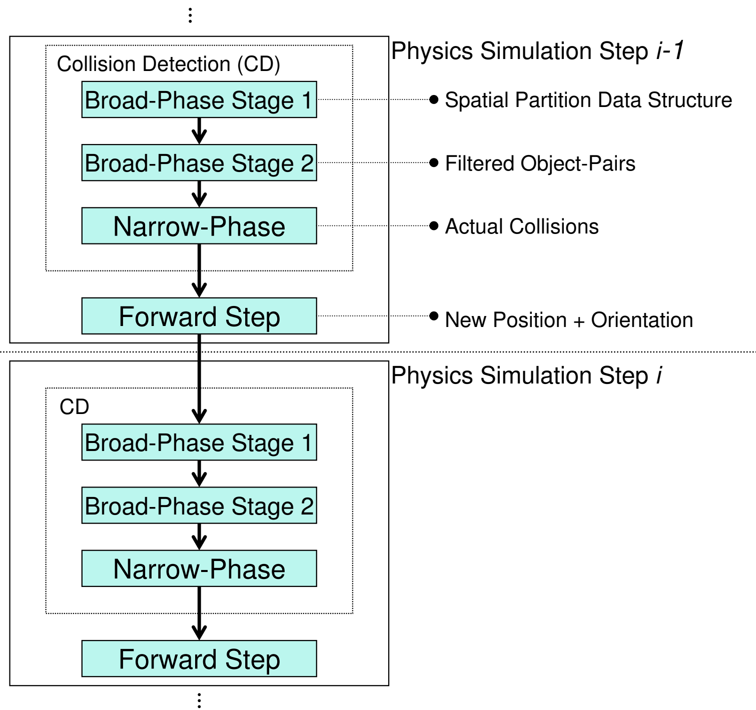

Figure 6: Collision Detection's Role in the Physics Simulation Flow

Figures 3, 4, and 5 show the execution time breakdown for collision detection on a 2.8 GHz Pentium 4. It is interesting to note the variation in the proportional amount of time spent in broad-phase versus narrow-phase for these different benchmarks. Collision detection for CrashWall is dominated by the runtime of narrow-phase CD, Battle is dominated by broad-phase CD, and Battle2 is somewhat evenly distributed. One way to reduce the overall runtime of CD is to overlap broad-phase and narrow-phase as much as possible. This would allow us to optimally get a total CD runtime that is the maximum latency of the two components. Since broad-phase is effectively a technique to filter the work done by narrow-phase, it should also be possible to dynamically balance the amount of work done by each of these. In this way, if we can achieve parallelism between the two components, we should be able to cut the runtime of CD at most in half. In the rest of this section, we will explore a technique to provide this parallelism.

Figure 6 shows the inter-task dependencies for existing collision detection methods. Both narrow-phase CD and broad-phase CD must wait on the forward dynamics from the previous step. The forward dynamics

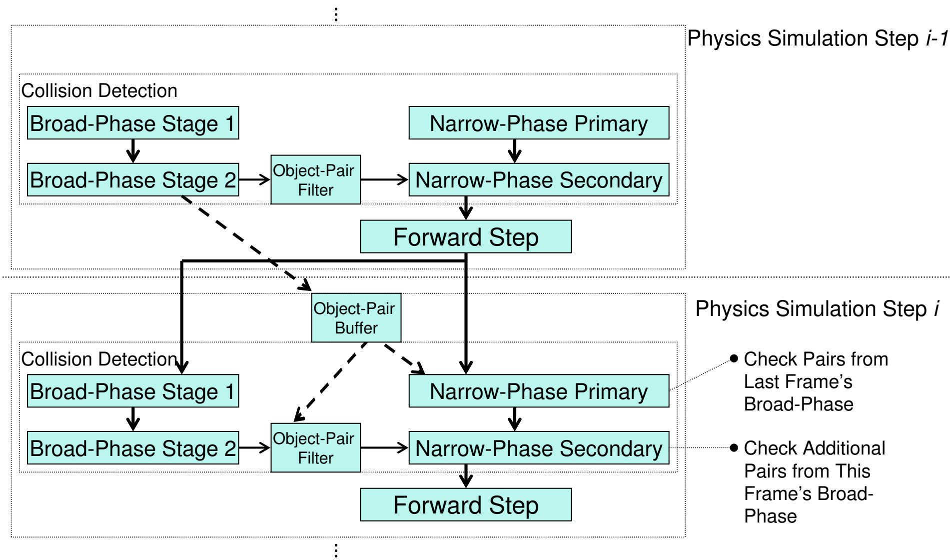

Figure 7: Collision Detection Flow with Decoupled Broad-Phase and Narrow-Phase

of one step relies on the narrow-phase CD of that same step – and it cannot form islands (clusters of colliding objects) and begin the actual physics simulation until the narrow-phase CD has completed all object pairs. Narrow-phase CD depends on the output of broad-phase CD. Note that this is shown for an arbitrary step in the calculation of a single frame – there would also be information streaming from step $i-2$ to step $i-1$ in the same manner that step $i-1$ streams to step i.

While different object-pairs can perform bounding box filtering and narrow-phase CD in parallel, broad-phase stage 1 (the update of spatial partitioning data structures) is serial with respect to all other components. This serial task will represent an increasingly larger amount of the total execution time as the number of objects in the virtual world and the number of processor cores on-die increases for future CMPs. This observation is confirmed by Luque et al [17] – they conclude that broad-phase CD must be done as efficiently as possible for good overall physics performance, even if it means filtering fewer object pairs.

In section 3, we observed that there is considerable locality in CD. One approach to creating more parallelism in CD would be to use the result of broad-phase CD from *one* step to feed the narrow-phase CD of the *next* step. This would allow the broad-phase and narrow-phase components of CD to be done in

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

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

The overall flow of this new approach to CD is shown in figure 7. We are effectively decoupling broad-phase and narrow-phase CD as much as possible to improve parallelism. We split narrow-phase CD into two components: a primary stage that handles the speculative set of object pairs from the prior step's broad-phase CD and a secondary stage that handles any serial narrow-phase comparisons. Note that step i's primary stage of narrow-phase CD is using step $i-1$'s broad-phase result. We use a simple object-pair buffer to queue pairs from one step to the other – we will discuss the size of this structure a little later in this section. Step i's secondary stage of narrow-phase CD is using step i's broad-phase CD as input. This broad-phase CD result can potentially have pairs that were already given to the primary stage of narrow-phase CD – we will call these *redundant narrow-phase comparisons*. Redundant comparisons are not functionally incorrect, but can potentially negate any performance gain. The critical component of this new approach to CD is how to efficiently filter these redundant narrow-phase comparisons at the output of broad-phase CD. In the diagram, we refer to this generally as an *object-pair filter*, but in the next section we will investigate a particular design of this filter.

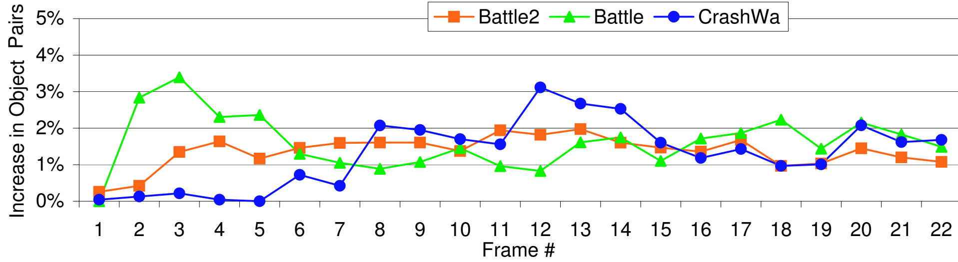

Figure 8: Unnecessary Narrow-Phase Comparisons and New Object-Pairs

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

Figure 8 shows the increase in both unnecessary narrow-phase comparisons and serial narrow-phase comparisons for our three benchmarks. The y-axis shows the percent increase in object pairs relative to the total number of pairs that would have been passed directly from broad-phase in the normal CD flow (figure 6). For all frames simulated, this never grows above 4% on average.

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

5.1 Object-Pair Filter

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

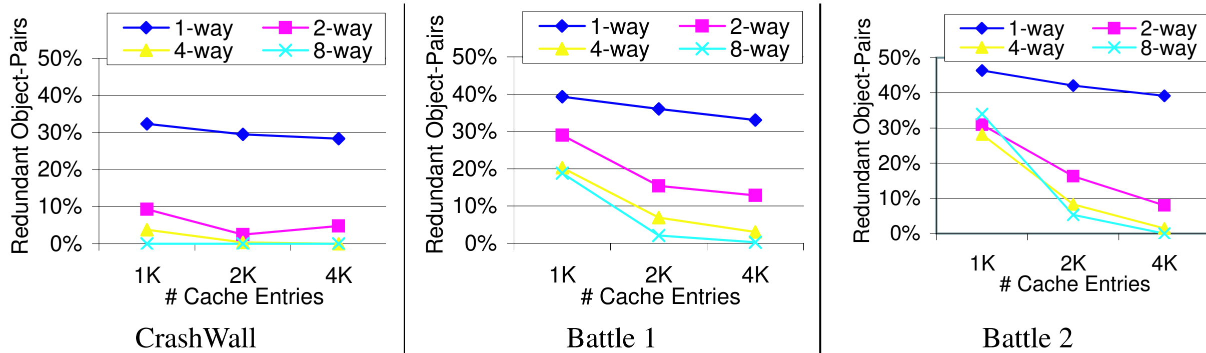

Figure 9: Object-Pair Filter Performance: the total redundant narrow-phase comparisons as a function of the number of stored object-pairs and associativity.

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

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

To reduce thrashing in the filter, we use two filters together. Each filter employs a different hash, but both filters have the same number of entries. For a pair of object filters with n sets, the primary filter takes $log n$ bits from each object address and xor's these to form the index into the filter. The secondary filter takes

$\frac{logn}{2}$ bits from each object address and concatenates them to form a $logn$ bit index. When the broad-phase CD output is written to the object filter, we first use the primary filter until the set we are writing to has filled. We do *not* evict pairs from a set, but instead use the secondary filter to find an alternative location to place the pair. If the corresponding set in the secondary filter is not full, we write the object pair to the secondary filter. This helps to better distribute the sets that heavily thrash. On an access to the filter (when broad-phase CD is determining what to send to the narrow-phase CD within its own step), both filters are checked in parallel – each using its own hash function.

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

Figure 9 shows the performance of our cache when varying the total number of object-pairs it can hold and the associativity of the cache.

5.1.1 Further Reduction in Size

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

5.2 Object-Pair Buffer

The object-pair buffer needs to be able to hold incoming object-pairs for the primary narrow-phase CD from the broad-phase CD of the previous step (see figure 7). There are two ways to do this in a CMP environment: 1) the buffer is distributed among the cores responsible for narrow-phase CD or 2) the buffer is centralized at the core responsible for broad-phase CD. As mentioned for the object-pair filter, it is unlikely that more than one core will be doing broad-phase CD. In either approach, the worst-case number of object-pairs in

the examples we looked at was 1700. This would require a total buffer capacity of 14KB. However, this is a simple FIFO buffer since it does not require CAM logic.

6 Summary

Physical simulation is becoming a significant component of current and future interactive entertainment applications. We have demonstrated the locality and correlation that exists with high-level application data in collision detection, a critical component of future physics simulation. And we have further leveraged this locality to perform control prediction and to provide parallelism between broad-phase and narrow-phase collision detection.

By proposing architectural techniques to accelerate collision detection, we have shown that the idiosyncrasies and the real-time performance requirement of interactive applications create many opportunities and challenges for microprocessor architecture design. We hope that our paper will make the community aware of these opportunities and spark further research.

References

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

[2] Agere. Building next generation network processors. www.agere.com/telecom/docs/.

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

[4] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 274–283, June 1997.

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

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

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

[8] GAMMA Team, University of North Carolina. Collision detection. www.cs.unc.edu/geom/collide.

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

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

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

[12] D. Jimenez and C. Lin. Neural methods for dynamic branch prediction. In *ACM Transactions on Computer Systems, Vol. 20, No. 4*, November 2002.

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

[14] C. Lee, I. Chen, and Y. Patt. The bi-mode branch predictor. In *30th International Symposium on Microarchitecture*, 1997.

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

[16] M. Lin and J. Canny. A fast algorithm for incremental distance calculation. In *IEEE Int. Conf. on Robotics and Automation*, 1991.

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

[18] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

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

[20] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, May 1993.