SlackSim: A Platform for Parallel Simulations of CMPs on CMPs

Jianwei Chen   Murali Annavaram   Michel Dubois

Ming Hsieh Department of Electrical Engineering
Viterbi School of Engineering
University of Southern California

Technical Report CENG-2008-6

August 2008

SlackSim: A Platform for Parallel Simulations of CMPs on CMPs

Jianwei Chen Murali Annavaram Michel Dubois

Department of Electrical Engineering - Systems
University of Southern California,
Los Angeles, CA 90089-2562, U.S.
E-mail: {jianwei, annavara, dubois}@usc.edu

The fast simulation of chip multiprocessors (CMPs) presents a critical challenge to the architecture research community as both industry and academia shift their research focus to multi-core design. Parallel simulation is a technique to accelerate microarchitecture simulation of CMPs by exploiting the inherent parallelism of CMPs. In this paper, we explore the simulation paradigm of simulating each core of a target CMP in one thread and then spreading the threads across the hardware thread contexts of a host CMP. We implement several parallel simulation schemes using POSIX Threads (Pthreads). We start with cycle-by-cycle simulation and then relax the synchronization condition in various schemes, which we call ***slack*** *simulations.*

In slack simulations, the Pthreads simulating different simulated cores do not synchronize after each simulated cycle, but rather they are given some slack. The slack is the difference in cycle between the simulated times of any two target cores. Small slacks, such as a few cycles, greatly improve the efficiency of parallel CMP simulations, with no or negligible simulation error. We have developed a simulation framework called SlackSim to experiment with various slack simulation schemes. Unlike previous attempts to parallelize multiprocessor simulations on distributed memory machines, SlackSim takes advantage of the efficient sharing of data in the host CMP architecture.

We demonstrate the efficiency and accuracy of some well-known slack simulation schemes and of some new ones on SlackSim running on a state-of-the-art CMP platform.

1 INTRODUCTION

In the future, doubling the number of transistors in micro-processor chips every 18 months (as predicted by Moore's law) will translate into increased numbers of hardware thread contexts on a chip. Single-thread performance may not improve much and may even deteriorate. The current trend is even to simplify the architecture of cores in order to support more thread contexts in the same processor. Such trend is clear in the Sun Microsystems 8-core Ultra-SPARC T1 and T2 microchips (with 32 and 64 threads respectively).

This development signals an imminent crisis in the simulation of future chip multiprocessors (CMPs). Such simulation is a critical step of the entire design flow because computer architects rely on simulations to explore design space and evaluate the merit of any changes to a new architecture. Currently, CMPs are simulated in a single host thread, which must simulate many target cores and their interactions. To keep simulating future generation (target) CMPs on current (host) CMPs with acceptable efficiency, we need to be able to exploit the multiple thread contexts in the host CMP in a way that scales well with the number of threads in the target and in the host. We need to split the target simulation among the host thread contexts so that the simulation of future generation CMPs on current CMPs will continue to scale with each generation of CMPs. In this paper, we develop a general approach to carry out parallel simulation of CMP targets on CMP hosts.

In this paper, we revisit the problem of parallel architecture simulation in the new context of CMPs, namely the simulation of a CMP on multiple threads. In the past, single-thread performance was growing at such a high pace that the motivation for parallel simulations of multiprocessors was never as strong as it is now. Today, this research is particularly relevant because multiple cores in CMPs are arranged tightly on the same die, which allows the cores to share information efficiently during simulation. Moreover, CMPs are fast becoming ubiquitous, from desktop PCs to servers.

In cycle-by-cycle simulations, the simulation advances one cycle at a time. In parallel cycle-by-cycle simulation, the threads simulating the target cores must synchronize after every simulated cycle. This approach is inefficient as the number of host instructions executed between two synchronizations is just several thousands. However, cycle-by-cycle simulation is accurate. Throughout this paper, cycle-by-cycle simulation is considered the "gold standard" in terms of accuracy. We use cycle-by-cycle simulation as the baseline, against which we compare the accuracy and performance of all other simulation schemes.

The tight synchronization condition imposed in cycle-by-cycle simulations is relaxed in various schemes called *slack simulations*. In slack simulations, the simulated cores do not necessarily synchronize after every simulated cycle, but rather they are granted some slack. *Simulation Slack* is defined as the cycle count difference between any two target cores in the simulation. Small slacks --such as a few cycles-- greatly reduce the amount of synchronization among simulation threads and thus improve the simulation efficiency with no or negligible simulation errors. Besides evaluating well-known slack techniques employing barrier synchronizations [4] [14], we explore new slack simulation schemes including unbounded slack (slack can be as large as the whole simulated execution time) and bounded slack (the slack is kept below a preset number of cycles, without using synchronization barriers).

The reason cycle-by-cycle simulations are deemed accurate is because they prevent timing violations, in which a simulated core is affected by an event happening in its past, a violation of temporal causality. Relaxing the synchronization of Parallel Discrete Event Simulation (PDES) has been studied for decades. There are two classes of PDES techniques to relax the synchronization imposed by cycle-by-cycle simulation: *conservative* and *optimistic* [10]. The key idea behind conservative approaches is to avoid any timing violation by carefully pacing the simulation. Optimistic approaches take periodical checkpoints of the simulation state and detect timing violations. When a timing violation is detected, the simulation is rolled back to the previous checkpoint, and the simulation is re-run in a "safe" mode from the checkpoint. Optimistic approaches perform poorly when timing violations are frequent.

In order to avoid timing violations, a conservative approach processes an event only when no other events could possibly affect it. In other words, if event A could affect event B, a conservative approach must process A at first, and then process B. Well-known techniques to enforce such restriction, while relaxing the cycle-by-cycle constraint, are based on *Barrier synchronization* [10] or *lookahead* [3]. In the first case, the simulation is divided into time intervals made of several simulated cycles, which are separated by synchronization barriers. Within a time interval, all simulation threads can advance independently until they reach the barrier. Before the simulation moves into the next interval, all global events triggered during the current interval must become visible to all simulated cores so that their effect will be simulated in the following intervals. Provided the time intervals are smaller than the latency needed for an event to propagate from one core to another in the target architecture, temporal causality is preserved. A well-known parallel simulator using barrier synchronizations is the Wisconsin Wind Tunnel II [14]. In WWT II, the time interval between two barrier is referred to as "quantum", and the

simulation is “quantum-based”. In the second case, the *lookahead* is the maximum amount of time in the future of a simulated core so that the thread simulating it can run safely without timing violations. The oldest event is always processed first, and simulation threads are allowed to progress up to their lookahead from the time of that event. Chidester and George compared the barrier and the lookahead schemes in the context of CMP simulations on a distributed, message-passing system and concluded that the lookahead scheme performed poorly [4].

SlackSim is derived from SimpleScalar [1]. To enable correct simulation of individual cores of a modern CMP, we have made considerable modifications to SimpleScalar. The two most significant kinds of modifications are: 1) modifications to enable the simulation of every core in separate threads; 2) modifications to support an Intel NetBurst-like OoO micro-architecture [9]. For instance, in the target core, register values are fetched just before execution. Moreover, unlike SimpleScalar which simulates instruction execution at the dispatch stage, our simulator models the execution of each instruction when it reaches an execution unit.

In the following sections of this paper, we first describe the SlackSim infrastructure, and we describe and analyze several slack simulation schemes that are implementable on SlackSim in Section 2 and 3. Then in Section 4, we evaluate the performance and accuracy of slack simulation on an actual multithreaded machine. Finally, after reviewing prior work, we conclude and propose future research in Sections 5 and 6.

2 SLACKSIM

A chip-multiprocessor (CMP) consists of multiple cores or CPUs integrated on a single die. A typical 4-core CMP organization is depicted in Figure 1. Chip multiprocessors use the shared memory communication model.

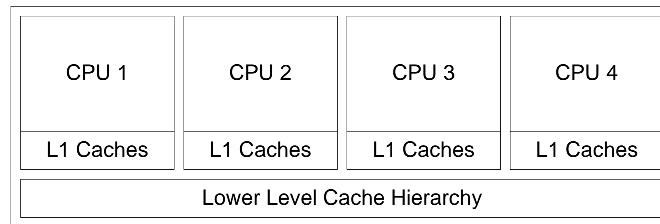

Figure 1: A typical 4-core CMP.

Typically, each core of a CMP has a first-level (L1) instruction cache and a first-level data cache, which may be write-through or write-back. Coherence must be maintained among the first-level data caches, with either a snooping or a directory protocol. The lower-level cache hierarchy is made of L2 cache banks. The L2 cache is typically organized as a set of banks with non-uniform access times (NUCA [7] [11]). Banks can be shared or private to each core.

2.1 Simulation Framework

In SlackSim, simulations are parallelized using the POSIX Threads programming model. POSIX threads, or Pthreads are defined as a set of C language programming types and function calls. The overhead of creating and managing Pthreads is much lower than the overhead of creating and managing processes. Compliant with the shared memory model, all Pthreads have access to the same global, shared memory. Pthreads can also have their own private data. Simulations written with

Pthreads can run on any number of cores and can be ported on any CMP environment that supports Pthreads. We have run our simulations so far on a PC server with two Intel Xeon chips (4 cores and 4 hardware thread contexts per chip).

Figure 2 shows the general framework of SlackSim. It comprises two types of Pthreads: several core threads and one simulation manager thread. One core thread simulates a single target core of a CMP with its L1 caches. The simulation manager thread has two functions. Its first function is to simulate the on-chip lower-level cache hierarchy including L2 cache banks and their interconnection to cores. The second function is to orchestrate and pace the progress of the entire simulation.

The simulation pace is controlled by two variables shared by each core thread and the simulation manager thread: *local time* and *max local time*. A core thread increments its local time after every simulated clock cycle of its target core. The max local time of each core is set by the simulation manager thread in accordance with the slack simulation scheme. The way the max local time is updated by the simulation manager will be described in Section 3 dedicated to various slack simulation schemes. A core thread can advance its own simulation and local time as long as its local time is less than or equal to its max local time. It suspends itself when the local time reaches the max local time. The simulation manager thread maintains the global time, which is equal to the smallest local time of all core threads. As the global time increases, the simulation moves forward. The simulation manager thread synchronizes the progress of the simulation by setting the max local time of each core thread.

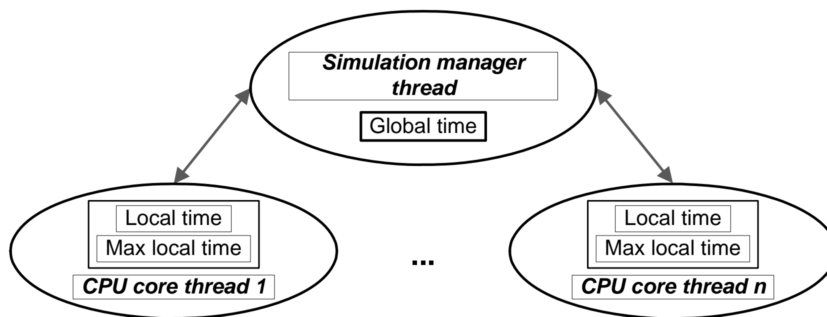

Figure 2: Simulation framework.

Given the definitions of global time, local time, and max local time, the following relation is always enforced among them:

$$Global\ Time \leq Local\ Time \leq Max\ Local\ Time$$

2.2 Detailed Architecture of SlackSim

Figure 3 shows the overall structure of the parallel simulation platform in more details including the two types of threads: the core threads and the simulation manager thread. There may be a concern that the simulation manager thread could become a bottleneck in the simulation as the number of core threads grows. However, the work involved to simulate the lower-level cache hierarchy in each cycle is much less than the work in each core thread. Nevertheless, if the manager thread becomes a bottleneck, the functions of the simulation manager should be hierarchically partitioned and assigned to several threads.

In the CMP simulator architecture illustrated in Figure 3, target cores affect one another through the simulation manager. For example, target cores can affect each other through the cache coherence protocol or by conflicting for shared resources. The communication between the core threads and the simulation manager thread is primarily realized through event queues. Each core thread has two queues: an outgoing event queue (OutQ) and an incoming event queue (InQ). The simulation man-

ager thread has a global event queue (GQ). In each entry, a timestamp records the time an event initiates and should take effect. Events are labelled by their event type field. In addition, data and address fields are needed for any event involving data transfers.

When a memory event takes place in a core, the core thread allocates and fills an OutQ entry for the request, and then it continues its simulation until its local time reaches the max local time. Meanwhile, the simulation manager thread continually fetches entries from the head of every core thread's OutQ. Once the simulation manager thread reads out an entry, it allocates a GQ entry for the request, and then fills it. For example, the most common request is an L2 cache access from a core. In this case, the simulation manager thread figures out both latency and data associated with the access on behalf of the core thread. Next, it generates an InQ entry filled with the returned value from the L2 cache and a timestamp indicating when the InQ event should be simulated by the core thread. Meanwhile, the core thread enquires its InQ in every cycle in order to see if its request has been processed by the manager thread. If so, the core thread reads out the data field of the entry when its local time becomes equal to the timestamp of the entry.

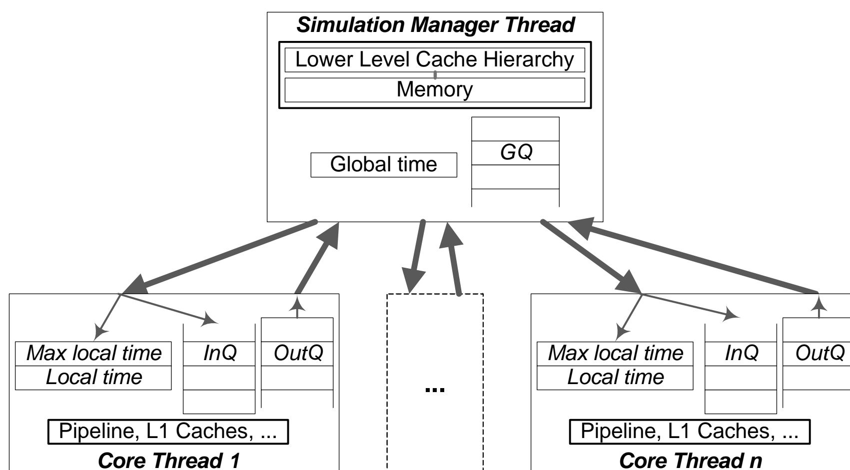

Figure 3: The architecture of the simulator.

3 SLACK SIMULATION

Various slack simulation schemes can be implemented in the framework described above. Updates to max local times for each core thread are dictated by the particular simulation slack scheme deployed in a particular simulation. We do not intend to describe all the possible slack simulation scheme in this section, just the ones we have implemented so far.

3.1 Slack Simulation Schemes

Simulations of a 4-core CMP are used to demonstrate each scheme in this section. Four possible slack simulation schemes are compared in Figure 4, where the X axis stands for simulation time, and the numbers on the top identify simulated cycles. The simulation starts at cycle 1 and ends when the local time of every core thread has reached *End*.

Figure 4(a) shows a simulation with 0 slack (this is our first scheme, our "gold standard" for accuracy, our implementation of cycle-by-cycle simulation). All threads must synchronize after every simulated cycle. For instance, at simulated clock 1, P4

finishes its first clock of target core simulation in the shortest time. On the other hand, P1 takes the longest time to finish its first simulated clock, and hence all threads must wait for P1 to be done before advancing to the next clock.

Figure 4(b) shows quantum-based simulation, in which all core threads must synchronize every three cycles, i.e. a 3-cycle quantum and a 2-cycle slack. In this scenario, when P4 finishes its simulation of clock 1, it does not wait for P1 to be done. Instead, P4 continues on to execute clocks 2 and 3. Because P1 and P2 take the longest time to complete the simulation of three clock cycles of their target cores, other threads have to wait for P1 and P2 after they have completed three cycles of their simulation.

As the figure shows, it should be clear that the quantum-based scheme is more efficient than cycle-by-cycle scheme because the number of synchronization barriers are cut by two-thirds. In general, the efficiency is better with less synchronizations. This derives from a basic mathematical relation among two sets of numbers $\{X_i\}$ and $\{Y_i\}$:

$$Max_i(X_i + Y_i) \le Max_i X_i + Max_i Y_i$$

However, because it takes different amounts of time for the threads to reach a synchronization point, the speed of simulation is set by the slowest thread within each quantum.

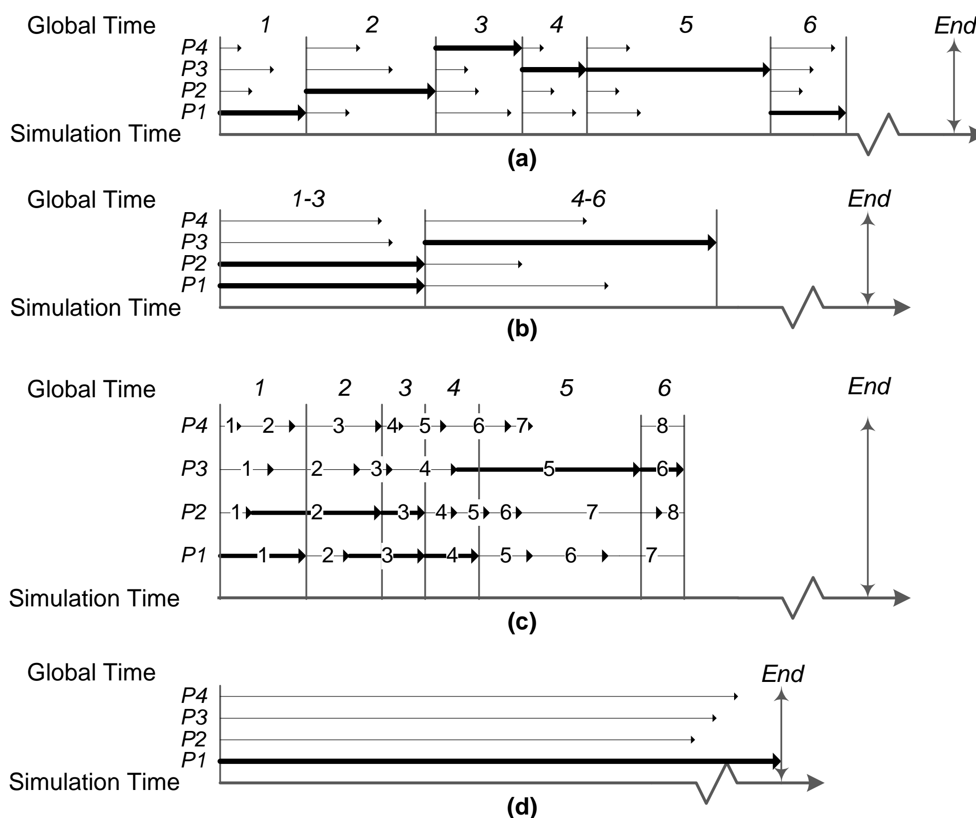

Figure 4: Cycle-by-cycle (a), quantum-based (b), bounded slack (c), and unbounded slack (d) simulations.

Quantum-based simulation is a well-known parallel simulation technique [10], and it has been the simulation paradigm for several simulators [4][8][14]. The accuracy of this type of slack simulation depends on the size of the quantum. When the quantum size is not more that the minimum latency needed to propagate an event generated by a target core to a point where it could affect another core's simulation (i.e., by communication, synchronization, or resource conflicts), the quantum-based

simulations are deemed as accurate as cycle-by-cycle simulations. We call this minimum latency the *critical latency.* Identifying the critical latency in a particular simulated system may be difficult and should be done safely. In this paper, as elsewhere [4], we adopt the unloaded latency of an L2 cache access as the critical latency. Thus quantum-based simulations are safe or accurate when the quantum is no more than the critical latency.

Figure 4(c) illustrates a third (new) parallel simulation paradigm, which we are proposing in this research. We call it *bounded slack simulation*. In this scheme, the maximum slack among threads is bounded as in the quantum-based simulation, but the simulation threads do not synchronize periodically at barriers. The maximum slack restriction forces all simulation threads to stay within a cycle window whose size is the maximum slack. Referring to Figure 4(c), the lower (left) boundary the window is always equal to the global time, T_g, shown at the top of the figure. With a maximum slack of S cycles, the upper (right) boundary of the window is $T_g + S$, which must be the max local time of all threads. The window slides every time the global time is updated. Because the global time always increases, the window always moves towards the right, i.e., the end of simulation. All threads are free to run as long as they stay within the window, and a thread is blocked only when its local time becomes equal to $T_g + S$, i.e., it reaches the right boundary of the window.

A significant difference exists between the bounded slack scheme and the quantum-based scheme. In the quantum-based scheme, the global time is updated only after all threads' local time becomes equal to their max local time. By contrast, in the bounded slack simulation, the global time is updated whenever the smallest local time advances. Although there is still some synchronization among threads in the bounded slack simulation, it is much looser than in the quantum-based simulation. In the ideal situation, no thread is ever blocked to wait for other threads during the entire simulation provided all threads remain within the sliding window, without ever reaching its upper boundary. Another difference is that the quantum-based simulation is deemed accurate as long as the quantum is no more than the critical latency, whereas bounded slack simulation may theoretically have timing violations even if the maximum slack is no greater than the critical latency.

Figure 4(c) shows a case where the slack is bounded to 2 (also equivalent to a quantum of 3). As can be seen, the threads simply run within a maximum sliding window of 3 cycles, instead of being forced to synchronize every 3 cycles. For instance, when the global time is 3, the left boundary of the window is 3, and the right boundary is 5. Within this window, P4 and P3 have already started simulation of their cycles 5 and 4, respectively. Meanwhile, P2 and P1 are working on their simulation of cycle 3. As a result, the bounded slack scheme not only finishes the simulation of cycle 6 earlier than the quantum-based scheme, but also has already started the simulation of cycles 7 and 8 of some cores. The only wasted simulation cycles are due to P4 at global time 5.

Figure 4(d) shows a fourth scheme, in which the slack is unbounded. This is an extreme case of bounded slack simulation, as no synchronization is enforced at all during the entire simulation, i.e., the bound on the slack is the entire simulated time. It is not hard to imagine why the minimum simulation time will be achieved if the slack is unbounded. At the same time, it is also obvious that such wild, uncontrolled simulation run could lead to inaccurate simulation statistics and unpredictable simulation behavior. We will dispel such concerns, and then we will show actual data to validate the efficiency and accuracy of slack simulation. Before that, we now analyze various possible simulation violations.

3.2 Simulation Violations due to Slack

There is always a danger in parallel computing that operations can be out of synchronization. This is certainly a major concern in parallel simulation as well. Cycle-by-cycle simulation depicted in Figure 4(a) is always accurate. The quantum-based simulation depicted in Figure 4(b) is also deemed accurate provided that the quantum size is no more than the critical latency. The other slack schemes are designed to accelerate simulation by granting more slack to core thread simulations. The slack helps balancing the load by relaxing the synchronization among simulation threads, as illustrated in Figure 4. When the slack remains bounded and small, simulation errors or violations are few or even non-existent. However, when the slack becomes very large, such as in unbounded slack simulations, the number of errors and violations might be very large.

We now illustrate how simulation violations can occur in bounded or unbounded slack simulations (Figures 4(c) and (d)).

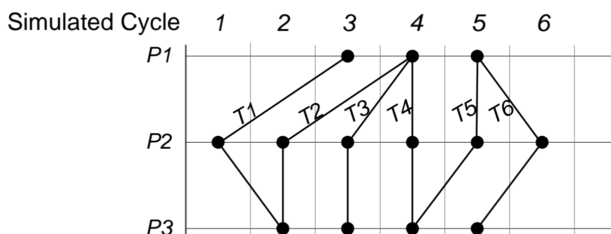

Figure 5: Isochrones connecting clocks at various simulation times.

In Figure 5, the simulated cycles in each of three cores are connected by isochrones corresponding to a given simulation time. At simulation time T1, the local times of P1, P2, and P3 are clocks 3, 1, and 2 respectively. Note that the isochrones never cross each other because both simulation time and simulated time always increase. In this example, the slack is small (maximum two), and the effect of slack is simply a slight, temporary, and variable distortion of the state of the simulated system. Although these distortions are small, they might cause violations of the simulation state, of the state of the simulated system, and of the state of the workload. These violations might cause the simulation to stray, to become meaningless, or even to deadlock. In the following subsections, we show how these distortions may cause violations.

3.2.1 Simulation State Violations

Every simulation maintains state variables internally to order to record and track the state of each resource in the target system. We need to make sure that the simulation state remains consistent in the face of simulated time distortions due to slack. We use a simple case, the simulation of a bus access, as an example to illustrate how violations to simulation state may happen, and how we can compensate for such violations.

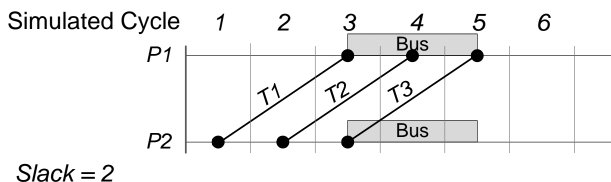

Figure 6: Violation of simulation state on a bus access.

Figure 6 shows a situation where the slack is 2 cycles with isochrones T1, T2, and T3. At simulation time T1, the local times of P1 and P2 are cycle 3 and 1 respectively. P1 requests and gains control of the bus because the bus is "free" at simulation time T1. As simulation moves to simulation time T2, which is later than T1, the local time of P1 and P2 becomes cycle 4 and 2. P2 requests control of the bus and finds it "busy". At simulation time T3 (clock 5 for P1, clock 3 for P2), the bus is

released by P1, and at that "time" the bus is used by P2. This is an inconsistency for the simulation because the bus appears to satisfy two bus requests at the same time in the simulated system. However, this can also be seen as a temporary time distortion, which statistically does not affect final simulation statistics much. Because the isochrones never cross, the simulation state (i.e., the values of the variables keeping track of the occupancy of resources in the target system) remains consistent in simulation time (which is the time that counts for the simulator state), although simulated time is distorted by the effect of slack. Resource conflicts are simulated although accesses to shared resources may be in a different order than in cycle-by-cycle simulation.

3.2.2 Simulated System State Violations

Next, we show how the integrity of the simulated system can be affected by simulation slack. This refers to storage structures in the target system, which keep the information needed to enforce correct hardware operations. These structures are not visible to the software.

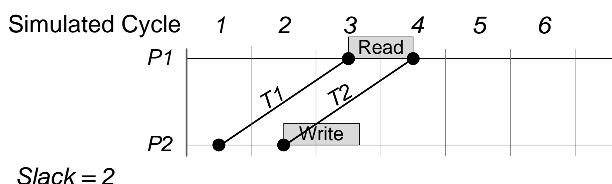

Figure 7: Read/Write violation on the same block.

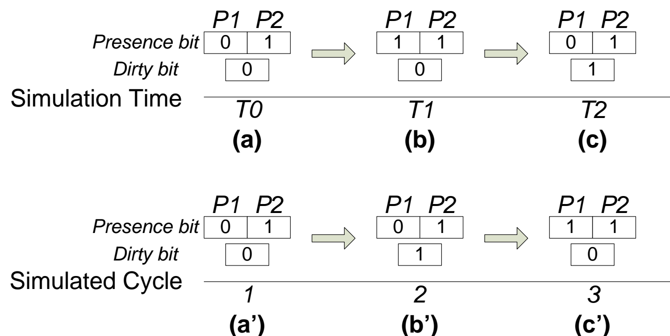

Figure 8: Directory entry state violation.

Figures 7 and 8 illustrate a simple example of such violations based on the state bits of a directory entry for a cache block in a directory protocol. The original state of the directory entry is shown in Figures 8(a) and (a'): The block is present in the cache of core P2 and is clean. Figure 7 shows a possible timing of the simulation. As before, cycles simulated at the same simulation time are connected by isochrones. At simulation time T1, the local times of P1 and P2 are 3 and 1 cycles respectively, and P1 reads the block. Consequently, the block now is still clean but shared by both P1 and P2, and the presence bits are both 1 (Figure 8(b)). As simulation moves to simulation time T2, P2 writes into the block (not necessarily the same word as P1 reads), P1 is invalidated, and P2 owns a dirty copy. The final state of the entry is shown in Figure 8(c). This is the outcome of the directory state, based on simulation time. However, the states of the simulated directory would be different in a cycle-by-cycle simulation. The write by P2 would occur first, and the block would first be dirty in P2's cache, as shown in Figure 8(b'). Figure 8(c') displays the state of the directory entry after the read of P1. Figure 8(c) and Figure 8(c') show the different outcomes in the two simulations. As a result, the timing of memory accesses and their effects on the system state

are different from the cycle-by-cycle simulation, which may impact the accuracy of performance metrics collected by the simulation.

Again, because isochrones never cross, the maintenance of hardware structures in the simulator remains consistent, which therefore enforces correct hardware behavior, albeit possibly different from a cycle-by-cycle simulation.

3.2.3 Simulated Workload State Violations

Finally, we discuss workload behavior violations. This refers to the execution flow of the workload and the data values written and read by the workload. These violations are potentially much more damaging than the previous ones because they could result in an incorrect execution of the benchmark on the simulator, or even situations where the benchmark cannot execute completely because of errors in its execution.

This type of error results from the timing of memory accesses to the same memory word, as governed by the memory consistency model. The only way that a workload thread of the target CMP can affect another workload thread is by a Store followed by a Load to the same word. This is often called a conflicting pair of accesses. If the program is properly synchronized, and synchronization is handled carefully, this type of violations should not have an impact on the correct execution of the benchmark. Thus we could ignore them as well in this framework. Nevertheless, our intent is to run workload intended for various memory consistency models. Currently, we support Sun TSO (Total Store Order). So at the very least all programs written for TSO should execute correctly and finish.

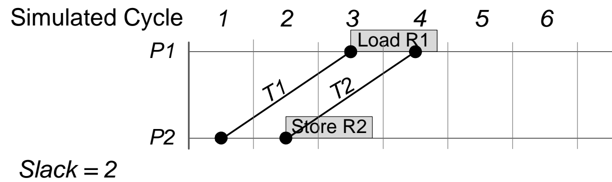

Figure 9: Read/Write violation on the same word.

Figure 9 illustrates the situation. P1 loads a value from a memory address M into register R1, and then P2 stores the value of register R2 into M in simulation time order. However, the memory access order is reversed in a cycle-by-cycle simulation. Because the order of the Load and the Store in the slack simulation is different from that in the cycle-by-cycle simulation, the value returned by the Load is different in the two cases. Such a violation can be detected by the simulator in the global queue of the simulation manager thread, which buffers all in-flight accesses.

To enforce correct execution under a memory consistency model that does not rely on synchronization, we need to detect and compensate for such violations. One possible method is *fast-forwarding*. Referring again to Figure 9, when the simulation manager thread detects the violation, it informs P2 to increase its local time by 1 through P2's InQ. Meanwhile, it changes the timestamp associated with the memory request from 2 to 3. By fast-forwarding, the Store seems to happen contemporaneously with the Load. This is true because NO programs designed for a particular memory model may rely on the relative speed of processors. Fast-forwarding a core emulates a situation where the core idles or slows down for some cycles, and this idle time must be undetectable by the program. A form of fast-forwarding is employed in [8] to correct the effect of violation in a quantum-based simulation.

Currently, we do not compensate for the violations in SlackSim. Nevertheless, the benchmarks we have tested still execute correctly. The simulation state and simulated system states remain coherent because simulation actions are taken at isochrones, which never cross, as if the simulated time was distorted.

3.3 Impact of the Programming Environment

The performance of our parallel simulation tool heavily depends on how it is programmed. In particular, the scheduling of the threads on the host greatly affects overall simulation performance.

Our simulation environment is built on top of Linux. We have adopted the POSIX Threads programming model because of its portability and flexibility. Under the POSIX Threads model, when a core thread's max local time is reached, the thread suspends itself. The OS kernel switches out the thread, and runs any other ready threads. Later on, after the sleeping core thread is awakened, it is enabled to run again.

When the number of ready threads is less than the number of available host thread contexts, some thread contexts or host cores become idle, which will lead to longer simulation time and lower host core utilization. In order to avoid this, we must prevent core threads from reaching their max local time, thus favoring schemes such as bounded slack with larger slack or even unbounded slack. Additionally, the global time must increase as fast as possible. Slower core threads must be explicitly scheduled with higher priorities so that the global time can advance. On the basis of this analysis, we choose to assign different scheduling priorities to core threads according to their local time: the lower a core thread's local time, the higher its priority.

In general, a parallel program incurs more system overhead than its sequential counterpart does. We measured the system overhead of parallel programming by forcing our simulator to run on top of one single host core. The results show the simulator on the average spends 10.9% of its time in system mode across all benchmarks. More importantly, the system overhead increases very modestly as the number of available host cores grows. As a result, the parallel multi-threaded simulator is able to deliver better throughput on more host cores.

4 COMPARISON OF SLACK SIMULATION SCHEMES ON SLACKSIM

The development of the simulation platform started with SimpleScalar/PISA. It was modified extensively in order to support the simulation of CMPs. SlackSim is still a user-level simulator. Memory management, file system handling, and other system functions are emulated outside the simulator. A POSIX Threads style application programming interface (API) is provided to parallel workloads. Table 1 shows a sample subset of APIs that are used for workload thread synchronization. The parallel programming APIs of SlackSim are based on previously implemented APIs from MP_Simplesim [13]. Note that no new instructions were added to the PISA instruction set to support our APIs. Instead, we implement them outside the simulator as we do for other system calls. In order to run on top of SlackSim, parallel programs must be modified to use the APIs. According to our experience, most common changes to benchmarks involve replacing the native thread creation and synchronization primitives with our APIs. This was achieved by editing the benchmarks' source code with simple search and replace commands.

	Lock	Barrier	Semaphore
Operations	init_lock() lock() trylock() unlock()	init_barrier() barrier()	init_sema() sema_wait() sema_signal()

Table 1: Synchronization primitives and operations.

4.1 Experimental Setup

In all the results presented in this section, the target system is an 8-core CMP with the SimpleScalar PISA instruction set. Table 2 presents the relevant microarchitectural parameters of the target system. Each core thread in SlackSim is capable of simulating a Pentium IV-like OoO architecture with the PISA instruction set. In SlackSim, the core pipeline executes instructions in the execute stage of the pipeline rather than in the dispatch stage. The micro-architecture supports a physical register file separated from the ROB and speculative scheduling of instructions. Each target core has private L1 instruction and data caches, and all 8 cores share a single L2 cache. Caches are kept coherence using MESI protocol. The currently supported memory model is Total Store Ordering (TSO).

Parameter	Value
Number of Cores	8
Instruction Window	64 entries
Issue Mechanism	4-way OoO
Branch Predictor	Combined
L1 I-cache	Private 16K, direct-mapped, 32B blocks, 1 cycle latency
L1 D-cache	Private 16K, 4-way (LRU), 32B blocks, 1 cycle latency
L2 cache	Shared 256K, 4-way (LRU), 64B blocks, 10 cycle latency
Memory	300/100 cycles for first/inter chunk
Coherence Model	Directory-based MESI coherence protocol
Memory Consistency Model	Total Store Order (TSO)

Table 2: Microarchitectural parameters of the target.

Table 3 presents the host environment for our simulations. Our experiment platform is a Dell PC server powered by two Intel Quad-core Xeon processors. The operating system is Ubuntu Linux Version 6.06. The simulator is compiled using GCC 4.1.2 with “-O3” as flag.

Parameter	PC Server
Processor Model	Intel Xeon E5310
Processor number	2
Cores/processor	1
Threads/core	1
Processor Frequency	1.6 GHz
Memory	4 GB
OS	Ubuntu Linux 6.06
Compiler	GCC 4.1.2 -O3

Table 3: Host parameters.

In order to evaluate parallel CMP simulations, parallel benchmarks that have thread-level interactions should be chosen. Hence, instead of selecting separate single-threaded applications, such as in SPEC CPU2K and running them on our target

CMP, we choose six parallel benchmarks: *Barnes*, *FFT*, *LU*, *Water-Nsquared*, *Jacobi*, and *SimpleSum*, shown in Table 4. *Barnes*, *FFT*, *LU*, *Water-Nsquared* are programs from the SPLASH-2 suite [2]. *Jacobi* and *SimpleSum* are internally developed programs. *Jacobi* is quite simple and based on an iterative algorithm to calculate the Jacobian matrix of a differentiable function. Because of its small data set, the communication between cores in *Jacobi* overwhelms the amount of computation. By contrast, in *SimpleSum*, each workload thread computes the sum of integers from 1 to n, and the threads are totally independent. Although *Jacobi* and *SimpleSum* are simple kernels, they helped us quickly debug and understand thread-level interactions. Every benchmark starts as one single workload thread. Then it spawns other workload threads by calling a system call, SS_MP_CREATE_THREAD(). In our experiments, every benchmark is composed of a total of 8 workload threads. In order to skip the initialization phase of the benchmarks, we start collecting simulation data right after all workload threads are created. Then, 100M committed instructions are simulated in all configurations.

Benchmark	Input Set
Barnes	1024
FFT	64K points
LU	256 x 256 matrix
Water-Nsquared	216 molecules
Jacobi	32 x 32 matrix
SimpleSum	n = 1,000,000

Table 4: Benchmarks.

As described earlier, each CMP core is simulated by one POSIX thread. L2 and other shared resources are simulated by a separated POSIX thread, which also controls the simulation. As a result, the simulation is composed of 9 POSIX threads that simulate an 8-core target CMP. We use a maximum of 8 hardware thread contexts in the host CMP.

4.2 Simulation Experiments

4.2.1 Simulation Speed Improvements with Slack

Table 5 shows the instruction throughput of the cycle-by-cycle simulations of the six benchmarks when all threads are executed by one single host core. This single-core cycle-by-cycle simulation of our 8-core target is used as the baseline for comparisons throughout the following discussions.

	Barnes	FFT	LU	Water-Nsquared	Jacobi	SimpleSum
Baseline Instruction Throughput (KIPS)	104.9	85.5	110.1	109.6	133.3	171.2

Table 5: Instruction throughput of cycle-by-cycle simulations on one host core.

The simulation speedup of a given simulation scheme is calculated with the following formula:

$$\text{Simulation Speedup} = \frac{\text{Baseline Simulation Time}}{\text{Simulation Time}} = \frac{\text{Instruction Throughput}}{\text{Baseline Instruction Throughput}}$$

Figures 10 and 11 show the simulation speedup of each benchmark with different simulation schemes: cycle-by-cycle simulation (CC); quantum-based simulation with 10-cycle quantum (Q10); bounded slack simulation with 9-cycle slack (S9); bounded slack simulation with 100-cycle slack (S100); and unbounded slack simulation (SU). We choose a 10-cycle quantum because the *critical latency* for the quantum-based scheme is 10, the latency of an L2 cache access in the target system.

The X-axis of the figures stands for the number of host cores, from 2 to 8. We vary the number of host cores but keep the number of target cores fixed at 8. Note that the simulation speedup is a function of both the number of host cores and the slack scheme.

Several observations are in order. First, the simulation speedup always improves with more host cores. Second, the cycle-by-cycle simulation is relatively poor and does not scale well with the number of cores because of its high synchronization overhead. Its speedup slightly increases when number of host cores grows from 2 to 8. The best speedup of 2.5 is achieved in *LU* and *Water-Nsquared*. Third, all slack simulation schemes (including quantum-based) significantly enhance simulation efficiency. Even when simulation threads are limited to run on 2 host cores, their speedups are at least 3.5. Fourth, SU delivers the best speedups in all configurations. S100, which is safer than SU, outperforms S9 and Q10 significantly across all benchmarks. S9 also yields better speedup over Q10. Interestingly, S9 exhibits approximately 20% higher speedup than Q10 when there are 8 host cores, but very modest differences for 2 or 4 host cores.

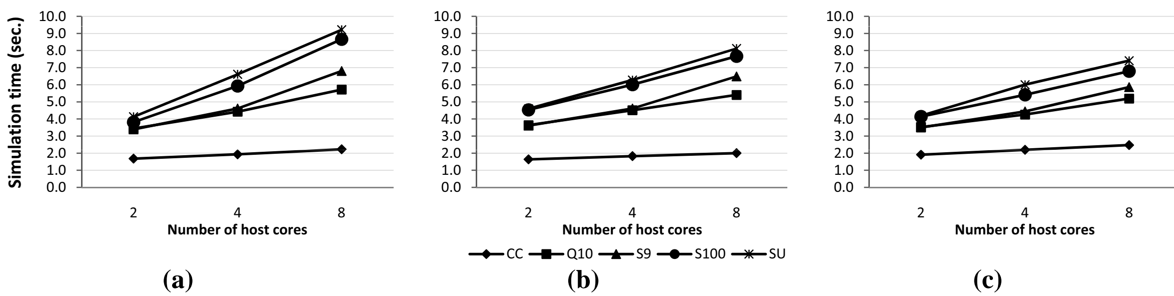

Figure 10: Simulation speedups of Barnes (a), FFT (b), and LU (c).

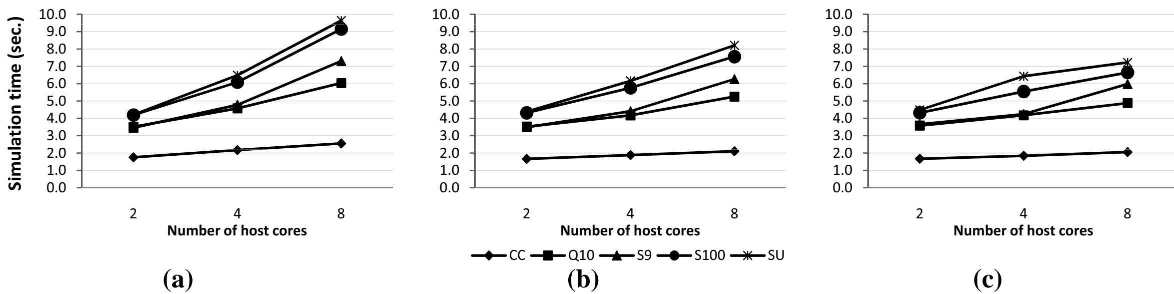

Figure 11: Simulation speedups of Water-Nsquared (a), Jacobi (b), and SimpleSum (c).

4.2.2 Impact of Slack on Simulation Errors

As explained earlier in Section 3.2, different slack schemes can potentially generate different sequence of microarchitectural events. Cycle-by-cycle simulations are the most accurate. The quantum-based simulation results reported here are also considered accurate because the quantum is equal to the critical latency. Other slack schemes may cause timing violations. The relative errors in the execution times for S9, S100, and SU with 8 host cores are shown in Table 6. These errors are very small, especially for SPLASH-2 benchmarks that have a large amount of inter-core communications. What is even more surprising is the low relative errors observed in the case of unbounded slack.

	S9	S100	SU
Barnes	0.08%	0.70%	3.94%
FFT	0.00%	0.06%	0.24%
LU	0.02%	0.03%	0.98%
Water-Nsquared	0.01%	0.06%	2.60%
Jacobi	0.14%	0.28%	2.75%
SimpleSum	0.00%	0.00%	0.05%

Table 6: Relative simulation errors in the execution times due to slack.

In order to understand why even unbounded slack causes such a small error in our simulations, we have collected the dynamic slack length statistics in *Barnes* and *FFT* when the slack is unbounded. For these results, we measured the slack (maximum difference between the local simulation clocks of any core threads) once every 100,000 simulated cycles. In Figures 12 and 14, the Y-axis shows the maximum slack between any two target cores for *Barnes* and *FFT,* respectively as a function of time. Figures 13 and 15 show the histogram of the slack of the two benchmarks for the entire execution. Figures 14 and 15 show that even with unbounded slack in *FFT,* the actual slack is relatively small most of the time. There are only a few peaks where the slack is more than one thousand cycles. Table 6 shows that unbounded slack causes a relative error of 3.94% in *Barnes*. Figure 12 shows that the slack for this benchmark is more uniformly distributed over time compared to *FFT*. Even though the peak slack of *Barnes* is smaller than *FFT,* the average slack is much higher. As a result, *Barnes* suffers higher error rates than *FFT*.

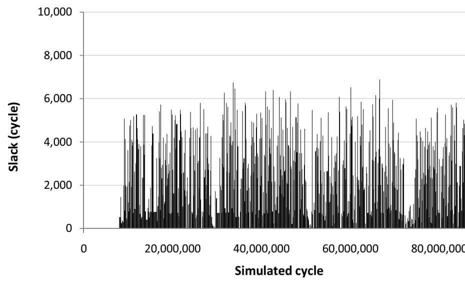

Figure 12: Slack of Barnes

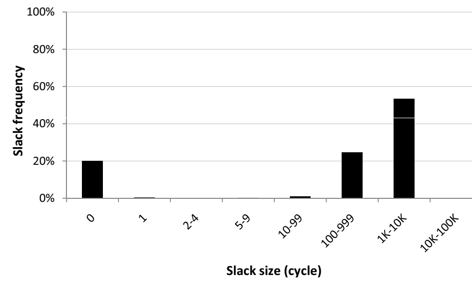

Figure 13: Slack frequency of Barnes

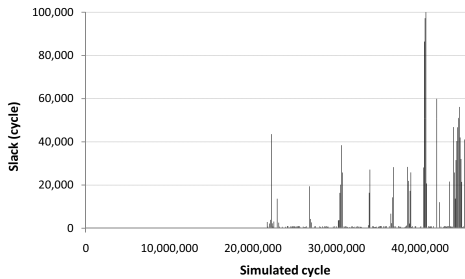

Figure 14: Slack of FFT

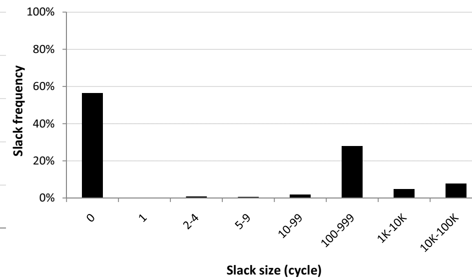

Figure 15: Slack frequency of FFT

5 PRIOR WORK

Parallel simulation has been an active research topic for several decades. Long before its application to computer architecture simulation, parallelization was a popular way to accelerate Discrete Event Simulation (DES). Parallel Discrete Event Simulation (PDES) employs two categories of methods: *conservative* and *optimistic*. The latter is sometimes referred to as *speculative*. Conservative approach guarantees correctness by blocking the simulation whenever there could be a possible

timing violation. *Barrier synchronization* [10] and *lookahead* [3] are two methods to implement such restriction. In the optimistic approach, simulation does not avoid violations. Rather, it recovers by rolling back to a previously saved checkpoint when timing violations are detected. The best known framework for optimistic simulation is Jefferson's Time Warp [12]. Unfortunately, the application of the Time Warp algorithm was not very successful due to several practical issues, such as large memory usage, excessive rollbacks, and wasted lookahead simulation.

The Wisconsin Wind Tunnel II is a direct-execution (i.e. the simulated code is executed directly on the host machine), discrete-event simulator that can be executed on shared-memory multiprocessors or networks of workstations [14]. WWT II uses a conservative approach with barrier synchronizations. The simulation accuracy of this so-called quantum-based simulation is guaranteed if the quantum size is no greater than the target system's critical latency. If the quantum is larger than the critical latency, then simulation accuracy is compromised. Our bounded/unbounded slack schemes can provide better performance with very high accuracy. The trade-off between simulation performance and accuracy is dictated by the size of the slack.

More recently, parallel simulation has been applied to CMP simulation [4]. The architecture of this CMP simulator is similar to the one we have adopted in SlackSim, but it was conceived to run on a distributed system, not on a CMP. Instead of shared-memory, inter-thread communication is implemented by message-passing. Instead of Pthreads, the Message Passing Interface (MPI) is employed for inter-process communication on the host platform. Two conservative schemes are compared: barrier and lookahead. The paper concludes that barrier is far superior to lookahead in terms of simulation performance. There are two significant differences between bounded/unbounded slack simulations and [4]. First, we choose to tolerate simulation errors: bounded/unbounded slack simulations can run with any slack size. Larger slacks accelerate the simulation with no or negligible errors. Second, because our simulator uses R/W accesses to shared variables to synchronize simulation threads, it is able to take full advantage of low-latency access to shared variables in the host CMP, instead of exchanging messages through MPI.

In order to speed up the parallel simulation of a full-system cluster, an adaptive quantum-based synchronization scheme is proposed in [8]. The size of the quantum is adaptively adjusted according to the amount of network traffic. The quantum is increased when packets are not exchanged, and it is shortened as the packet traffic increases. With this simple mechanism, the simulation speedup is improved dramatically with less than 5% errors in most experiments.

When simulating a multiprocessor system with distributed memory or a cluster system, the simulation quantum can be very large. In WWT II simulations, the critical latency is 100 cycles, and in [8] the critical latency is one microsecond (equivalent to about 1,000 cycles). However, for the parallel simulation of CMPs, the critical latency is much smaller. For example, a critical latency of 12 cycles is adopted in [4]. Because of the short critical latency of CMPs, the possible amount of simulation slack is small, and the speedup of quantum-based simulation is limited as shown in [4].

Finally, a rapid parallelization approach for parallel CMP simulation is presented in [6], where SimpleScalar and IBM Turandot are augmented to run in parallel by employing Pthreads programming model and thread-local storage technique. However, its simulation ability is severely limited due to lack of cache coherence and non-realistic microarchitectural model.

6 CONCLUSION AND FUTURE WORK

Slack simulation offers new trade-offs between simulation speed and accuracy. Slack simulation accelerates the parallel simulation of CMPs by relaxing the tight synchronization enforced between simulation threads in cycle-by-cycle (cycle accurate) simulation. Compared to cycle-by-cycle simulations, slack simulations can significantly improve the efficiency of parallel CMP simulation. Conservative approach to slack simulations uses barriers or lookahead to secure simulation accuracy. We have introduced other types of slack simulations, bounded and unbounded slack simulations. The added flexibility of these schemes improves simulation speed at the cost of simulation accuracy. Errors due to slack are attributable to simulated time distortions. After careful study of how simulated time distortions may cause simulation violations, we have shown that the workloads always execute correctly, and that the simulation state and simulated system states always remain consistent, although the distortions may result in different transitions of the system state and ultimately lead to errors in the reported simulation statistics.

We have developed *SlackSim*, a user-level simulation infrastructure to explore different slack simulation schemes. SlackSim takes advantage of efficient data sharing on CMP platforms. To evaluate the performance and accuracy of various slack schemes on SlackSim, we have simulated the execution of six parallel benchmarks on a target CMP under several slack simulation schemes. In the experiments we have run, errors on metrics such as execution time are practically negligible for slacks bounded to 100 cycles and small for unbounded slack.

Implementing simulation violations detection and compensation mechanisms can help us better understand the characteristics of slack simulation, and also can provide future improvement opportunities. These opportunities will be one focus of our future work. Another possible extension is to make SlackSim more dependable through dynamically modifying the size of slack to throttle simulation error rate within an acceptable range. Although we are still using SimpleScalar's PISA instruction set at this point, we plan to move to SPARC V9. Meanwhile SlackSim's efficiency could be further improved by careful tuning and by removing low efficient code.

References

[1] T. Austin, E. Larson and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling", *IEEE Computer*, Feb. 2002, pp. 59-67.

[2] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The Splash-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the International Symposium on Computer Architecture*, pp. 24–36, June 1995.

[3] K. Chandy and J. Misra , "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, Vol. 5 No. 5, pp. 440-452, 1979.

[4] M. Chidester and A. George, "Parallel Simulation of Chip-multiprocessor Architectures," *ACM Transactions on Modeling and Computer Simulation*, Vol. 12 , Issue 3, pp 176 - 200, July 2002.

[5] R. Devries, "Reducing Null Messages in Misra' s Distributed Discrete Event Simulation Method," *IEEE Transactions on Software Engineering*, Vol. 16 No. 1, pp. 82-91, 1990.

[6] J. Donald, M. Martonosi, "An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation," *IEEE Computer Architecture Letters*, Vol.5 No. 2, 2006.

[7] H. Dybdahl and P. Stenstrom,"An Adaptive Shared/Private NUCA Cache Partioning Scheme for Chip Multiprocessors, " in *Proc. of the Int. Symposium on High Performance Architecture (HPCA)*, pp. 2-12, 2007

[8] A. Falcon, P. Faraboschi, D. Ortega, "An Adaptive Synchronization Technique for Parallel Simulation of Networked Clusters," in *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 22-31, April 2008.

[9] Glenn Hinton et al.,"The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, Q1, 2001.

[10] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communication of the ACM*, Vol 33 No. 10, pp. 30-53, Oct. 1990.

[11] J. Huh et al.,"A NUCA Substrate for flexible CMP Cache Sharing," *IEEE Transactions on Parallel and Distributed Systems*, Vol.18, No.8, August 2007, pp.1028-1040.

[12] D. R. Jefferson, B. Beckman, F. Wieland, and L. Blume, "Distributed Simulation and the Time Warp Operating System," *Operating Systems Review*, vol. 21, pp. 77-93, 1987.

[13] N. Manjikian, "Parallel Simulation of Multiprocessor Execution: Implementation and Results for SimpleScalar," in *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 147-151, Nov 2001.

[14] S. S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood, "Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator," *IEEE Concurrency*, Vol.8, No. 4, pp 12-20, 2000.

[15] A. Over, B. Clarke, P. E. Strazdins, "A Comparison of Two Approaches to Parallel Simulation of Multiprocessors," in *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, pp.12-22, April 2007.