Exploiting Simulation Slack to Improve Parallel Simulation Speed

Jianwei Chen Murali Annavaram Michel Dubois

Department of Electrical Engineering - Systems, University of Southern California.
E-mail: {jianwei, annavara, dubois}@usc.edu

ABSTRACT

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 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

As silicon technologies enable billions of transistors on a single die, CMPs with increasing number of cores will be the norm of the future. This development signals an imminent crisis in the simulation of future chip multiprocessors (CMPs) since simulations are crucial for exploring the vast state space of CMP designs.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. In particular, the aim of our research is to efficiently 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.

We revisit the problem of parallel architecture simulation in the new context of CMPs, namely the simulation of a CMP on multiple threads. Previously proposed approaches to parallel simulations rely on cycle-by-cycle synchronization, where 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, these simulations are accurate because they prevent timing violations, in which a simulated core is affected by an event happening in its past, a violation of temporal causality.

In this paper, we introduce the novel notion of slack simulations. In slack simulations 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.

The relaxation of simulation synchronization has been studied previously in the context of Parallel Discrete Event Simulation (PDES). There are two classes of PDES techniques to relax the synchronization imposed by cycle-by-cycle simulation: *conservative* and *optimistic* [10]. In order to avoid timing violations, a conservative approach processes an event only when no other event could possibly affect it. In other words, if event A could affect event B, a conservative approach must process event A at first, and then process event B. *Barrier synchronization* [10] or *lookahead* [3] are two well known conservative techniques. 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. 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 these simulations, 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].

In this paper, we present SlackSim, a flexible simulation environment to experiment with various schemes for simulating CMPs on CMPs. SlackSim is able to model the detailed microarchitecture of CMPs at cycle-accurate level and also provides the flexibility to configure and explore the design space of possible slack simulation schemes. Unlike the above two simulators [4] [14], SlackSim is based on the shared memory model. Therefore, it can take advantage of fast access to shared variable on existing CMP platforms. CMP's shared memory effectively implements fast communication among simulation threads.

We propose bounded slack (the slack is kept below a preset number of cycles, *without* using synchronization barriers) as the preferred simulation approach for future due to its large

performance potential but with negligible error rates. Computer architects are allowed to balance the need for simulation efficiency and accuracy according to different design requirements. We experiment with various slack simulation schemes on SlackSim. Our results show bounded slack schemes can achieve up to 278% speedup with less than 0.5% simulation error.

2 SLACKSIM

A typical target CMP evaluated in the paper consists of multiple cores on a die, where each core has a private L1 data/instruction caches and all cores on a die share a large L2 cache. 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 per core. In SlackSim, both target and host systems are CMPs.

2.1 Simulation Framework

In SlackSim, simulations are parallelized using the POSIX Threads programming model. Figure 1 shows the general framework of SlackSim. It is made of two types of Pthreads: several core threads and one simulation manager thread. A 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. Its second function is to orchestrate and pace the progress of the entire simulation.

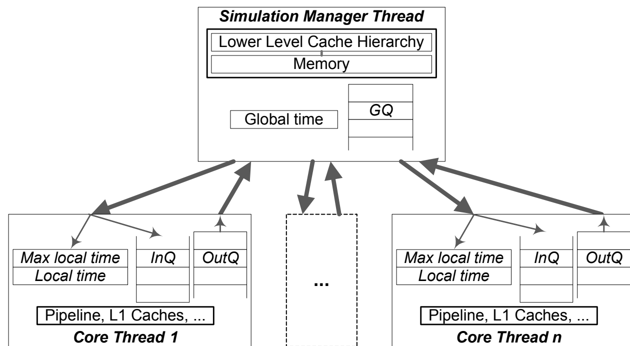

Figure 1. The architecture of the simulator.

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 for 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. Given the definitions of global, local, 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 1 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. If the simulation manager thread ever becomes a bottleneck it is possible to split the functionality of the manager thread also into several threads.

In the CMP simulator architecture illustrated in Figure 1, 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 such as a bus or an interconnection network. 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 manager thread has a global event queue (GQ). In each entry, a timestamp records the time (as defined by the local time of a given thread) an event initiates and should take effect. Events are labelled by their event type field. Event types in OutQ could be an L1 cache miss service request generated by a core thread. Similarly, event types in the InQ could be a cache line invalidation request from an other core thread or a L1 miss satisfied event notification. In addition, data and address fields are needed for any event involving data transfers.

When a memory event, such as an L1 cache miss, 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. Depending on the microarchitecture of the core being simulated, the simulation continuation can be as simple as just incrementing the local clock of the core if the core is a simple in-order core that stalls on a cache miss, or a complex set of activities that are possible in case of a OoO processor with non-blocking L1 cache. 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. Note that GQ consolidates all the local thread OutQ requests in a single queue, which allows the thread manager to efficiently manage and schedule all the GQ events.

SlackSim is derived from SimpleScalar [1]. We have, however, made considerable modifications to SimpleScalar. The two most significant modifications are: 1) modifications to enable the simulation of every core in separate threads and 2) modifications to support an Intel NetBurst-like OoO microarchitecture [9]. For instance, in the target core, register values are fetched just before execution. Unlike SimpleScalar

which simulates instruction execution at the dispatch stage, SlackSim executes each instruction when it reaches an execution unit.

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. Note that while Slacksim can be configured to simulate any slack simulation scheme, in this section we describe only the schemes explored in this paper.

3.1 Slack Simulation Schemes

For the purpose of demonstrating the difference between various parallel simulation schemes, we use a 4-core CMP as a pedagogical configuration. Six possible slack simulation schemes are presented here. The first four are compared in Figure 2, 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*. At the end of this subsection, we explain another two schemes.

Figure 2(a) shows a simulation with 0 slack, also called cycle-by-cycle synchronization scheme. (This is our implementation of cycle-by-cycle simulation, our “gold standard” for accuracy) 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. In Slacksim cycle-by-cycle simulations are implemented as follows: At first, the global time is set to 0 at the beginning of the simulation. The simulation manager thread must at first calculate the new global time, which is equal to the smallest local time of all core threads. When the new global time becomes one cycle bigger than the old global time, all core threads have finished simulation of their current cycle. In order to move simulation forward, the simulation manager thread increments all core threads’ max local time by one cycle, and then wakes up all pending core threads so that they can resume simulation of their next cycle.

Figure 2(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 simulate 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.

A core thread works exactly the same fashion in the quantum-based simulation as in the cycle-by-cycle simulation. The difference lies in when and how requests from core threads become globally visible in the two algorithms. In the cycle-by-cycle scheme, core threads are synchronized at the end of each cycle. This tight synchronization guarantees that the effect of all requests becomes globally visible and the simulated system is consistent at the end of each cycle. On the other hand, in the quantum-based scheme, requests are not globally visible until the end of each quantum. After every core thread exhausts its quantum, the simulation manager thread updates the shared resources so that the target system presents consistent states to all core threads *only* in the next quantum.

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 $\{Y_i\}$:

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

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

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 parallel 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), quantum-based simulations are often deemed as accurate as cycle-by-cycle simulations. We call this minimum latency the *critical latency*, which is difficult to identify in a particular simulated system. For example, threads of a CMP often conflict for shared resources such as the interconnect between cores; such conflicts may occur in only one cycle of latency, which means that the critical latency and the quantum should be one clock. However we may neglect the impact of such low-level interactions to reach a reasonable quantum size. In this paper, as elsewhere [4], we adopt the unloaded latency of an L2 cache access as the critical latency.

Figure 2(c) illustrates one of our novel parallel simulation paradigms, which we propose and evaluate in this paper. We call it *bounded slack simulation*. In this scheme, the maximum slack among threads is bounded as in 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 2(c), the lower (left) boundary of 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., towards the end of the simulation. All threads are free to run as long as they stay within the window, and a simulation 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 times become 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.

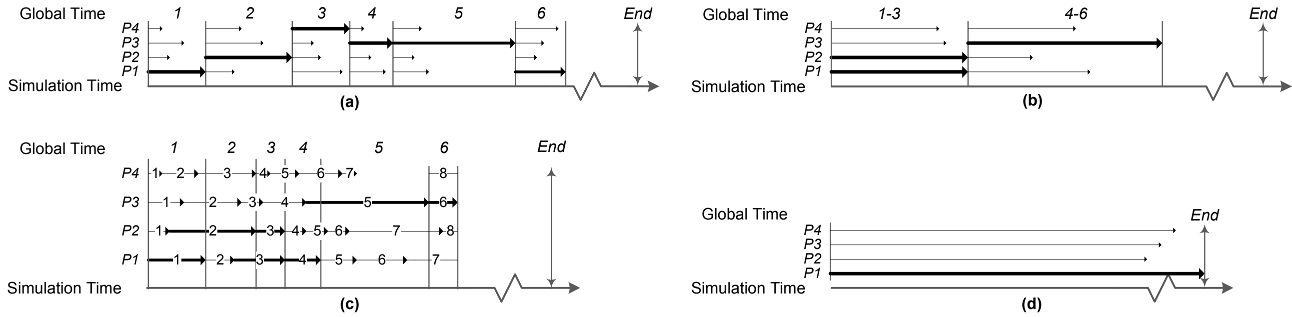

Figure 2. Cycle-by-cycle (a), quantum-based (b), bounded slack (c), and unbounded slack (d) simulations.

Figure 2(c) shows a case where the slack is bounded to 2 (similar 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, when P4 already completed simulation of its local clock cycle 7 while P3 is still simulating its local clock cycle 5. Hence P4 has reached the maximum allowed slack and waits for P3 to complete its cycle 5 before moving forward.

Figure 2(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. This scheme places few limits on the threads. The core thread *never* suspends itself based on the relative speed of progress of other core threads. Similarly, simulation manager thread processes any requests whenever they appear in GQ.

In addition to the above four schemes, we also implemented one optimization over our bounded slack simulation scheme called *oldest-first* bounded slack simulation. In this scheme, the manager thread globally orders all the requests from core threads based on the time stamps of each request. It then processes the oldest request first *only* if the timestamp of the oldest request is equal to the global timestamp. The manager thread does not process any request whose time stamp is greater than the global time. On the other hand, the original bounded slack scheme has no such constraint. This is the most significant difference between the two bounded slack schemes. Note that by ordering the requests based on timestamps and by delaying the processing of requests till the request timestamp equals the global timestamp the oldest-first approach eliminates all simulation violations when the slack is less than the critical latency. Thus, our oldest-first approach provides the same accuracy as quantum simulations but can potentially achieve higher speedup (shown later in our results section). It is also important to note that if the slack is more than critical latency even the oldest-first simulation can potentially cause simulation violations.

Finally, we discuss one additional simulation scheme called lookahead simulation scheme. Lookahead simulation is a conservative approach. In this scheme, a core thread still works in the same way as in the cycle-by-cycle simulation. However, in the simulation manager thread, actions are taken only when either a request is received or a core thread has exhausted its lookahead, i.e., it is blocked. After global time is calculated, a request is processed and becomes globally visible only when its time stamp is equal to global time.

Some slack simulation schemes could result in 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 2(a) is always accurate. The quantum-based simulation illustrated in Figure 2(b) is also deemed accurate provided 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 2. 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.

In Figure 3, 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 never decrease. 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.

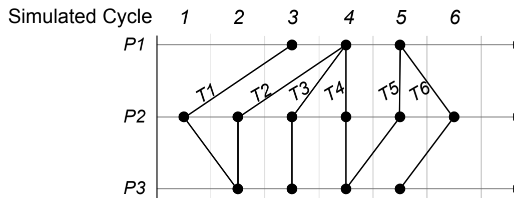

Figure 3. Isochrones connecting clocks at various simulation times.

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 4 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 cycles 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 times of P1 and P2 become cycles 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 simulation 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.

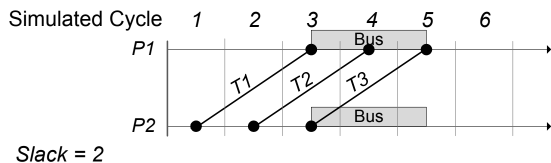

Figure 4. Violation of simulation state on a bus access.

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

Figures 5 and 6 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 6(a) and (a'): The block is present in the cache of core P2 and is clean. Figure 5 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 6(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 6(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 6(b'). Figure 6(c') displays the state of the directory entry after the read of P1. Figure 6(c) and (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.

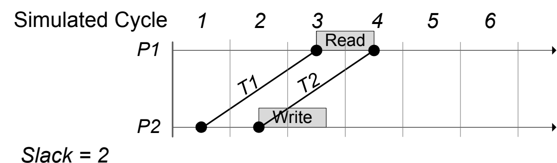

Figure 5. Read/Write violation on the same block.

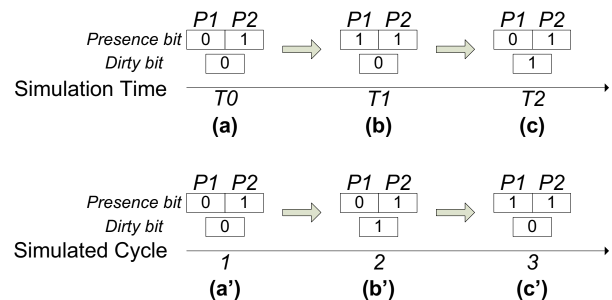

Figure 6. Directory entry state violation.

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.

Figure 7 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.

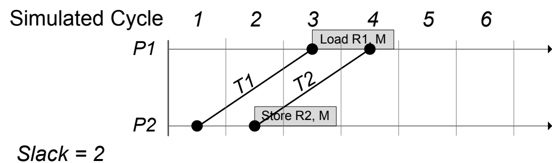

Figure 7. Read/Write violation on the same word.

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 7, when the violation is detected, P2 must increase its local time by 1. Meanwhile, the timestamp associated with the memory request is changed from 2 to 3. By fast-forwarding, the Store seems to happen contemporaneously with the Load. This conforms to the memory consistency model 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.

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 and construction of parallel simulation workloads. SlackSim is still a user-level simulator. When memory management, file system handling, and other system functions are called by the simulation workloads, they are emulated outside the simulator. A POSIX Threads style application programming interface (API) is provided to build the 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]. In order to simplify the simulation infrastructure, we currently support parallel benchmarks that use a Pthread style programming API similar to the functions shown in Table 1. 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 that do not use these Pthread style API 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 in our current implementation, which is a rather simple task.

Table 1. Synchronization primitives and operations.

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

4.1 Experimental Setup

SlackSim can simulate a variety of target CMP configurations. However, in all the results presented in this section, the target system is an 8-core CMP with the SimpleScalar PISA instruction set. Each core in the CMP is modeled as a 4-way issue Out-of-Order processor with 64 in-flight instructions, 16KB I/D caches and 256K shared L2 cache. L1 caches are kept coherent using a directory-based MESI protocol.

Our host experimental platform is a Dell PC server powered by two Intel Quad-core Xeon processors running at 1.6 GHz and with 4GB memory. The operating system is Ubuntu Linux Version 6.06. The simulator is compiled using GCC 4.1.2 with "-O3" as flag.

In order to evaluate parallel CMP simulations, we need to select parallel benchmarks with thread-level interactions. 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*, shown in Table 2. *Barnes*, *FFT*, *LU*, *Water-Nsquared* are programs from the SPLASH-2 suite [2]. Every benchmark starts as one single workload thread. Then it spawns other workload threads. 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.

Each CMP core is simulated by one POSIX thread. L2 and other shared resources are simulated by a separate thread, which also controls the simulation. Hence, simulation is composed of 9 POSIX threads that simulate an 8-core target CMP. We use a maximum of 8 hardware threads in the host CMP.

4.2 Simulation Experiments

4.2.1 Simulation Speed Improvements with Slack

In Table 2, the KIPS column 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 this section.

The simulation speedup of a given simulation scheme is calculated as ratio of baseline simulation time over simulation

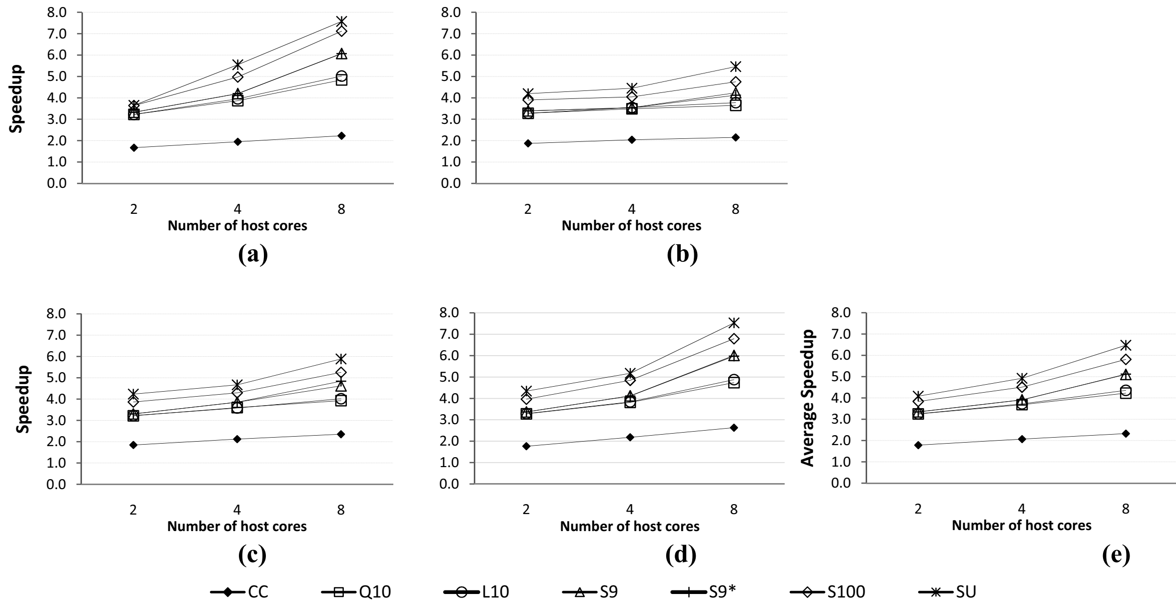

Figure 8. Simulation speedups of Barnes (a), FFT (b), LU (c), Water-Nsquared (d), and their harmonic means (e).

Table 2. Benchmarks.

Benchmark	Input Set	KIPS
Barnes	1024	111.3
FFT	64K points	120.5
LU	256 x 256 matrix	114.4
Water-Nsquared	216 molecules	127.1

time. Figures 8 shows the simulation speedup of each benchmark with different simulation schemes: cycle-by-cycle simulation (CC); quantum-based simulation with 10-cycle quantum (Q10); lookahead simulation with 10-cycle lookahead (L10); bounded slack simulation with 9-cycle slack (S9); oldest-first bounded slack simulation (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. Figures 8(e) shows harmonic means of the benchmarks' speedup.

Several observations are in order. First, the simulation speedup always improves with more host cores. Second, the speedup of 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 the number of host cores grows from 2 to 8. The best speedup of 2.6 is achieved in *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 3.3. 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. The speedup of S9 is approximately 20% higher than the speedup of Q10 in the case of 8 host cores. The speedup of S9* is almost the same as the speedup of S9. The speedup of L10 is a little higher than the speedup of Q10 because global time advances faster in L10 thanks to less synchronization. For 2 or 4 cores, the speedup of S9, S9*, Q10, and L10 are very close.

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 micro-architectural 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. Lookahead simulation is accurate because it is a conservative scheme. S9* is a conservative simulation scheme as well because it processes requests in the order of their timestamps. 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 3. 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.

Table 3. Relative errors in the execution times due to slack.

	S9	S100	SU
Barnes	0.08%	1.82%	5.94%
FFT	0.01%	0.07%	1.83%
LU	0.03%	0.09%	1.98%
Water-Nsquared	0.01%	0.12%	5.11%

5 PRIOR WORK

Parallel simulation has been an active research topic for several decades, first used in parallelizing discrete event simulation[12]. 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. Our (un)bounded slack schemes provide significant speedup at the expense of small errors.

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. Because our simulator uses R/W accesses to shared variables to synchronize 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.

An adaptive quantum-based synchronization scheme is proposed in [8]. The quantum size is adaptively adjusted according to the amount of network traffic resulting in dramatic speedup with less than 5% error. 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 was 100 cycles, and in [8] the critical latency is one microsecond (~ 1,000 cycles). However, for the parallel simulation of CMPs, the critical latency is much smaller. A critical latency of 12 cycles is adopted in [4]. Because of the short critical latency of CMPs, the quantum size is small, and the speedup of quantum-based simulation is limited[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 CONCLUSIONS

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, simulation errors on the execution time are practically negligible for slacks bounded to 100 cycles and small for unbounded slack.

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, "Multiprocessor enhancements of the SimpleScalar tool set," *ACM SIGARCH Computer Architecture News*, Vol. 29, Issue 1, pp. 8-15, Mar. 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.