Adaptive and Speculative Slack Simulations of CMPs on CMPs

Jianwei Chen, Lashkmi Kumar Dabbiru, Murali Annavaram, Michel Dubois

Adaptive and Speculative Slack Simulations of CMPs on CMPs

Jianwei Chen, Lashkmi Kumar Dabbiru, Murali Annavaram and Michel Dubois
Ming Hsieh department of Electrical Engineering
University of Southern California, Los Angeles CA90089-2560
jianwei.chen@oracle.com, {dabbiru, annavara}@usc.edu, dubois@paris.usc.edu

Abstract

Current trends signal an imminent crisis in the simulation of future CMPs (Chip MultiProcessors). Future micro-architectures will offer more and more thread contexts to execute parallel programs, but the execution speed of each thread will not improve at the same pace. CMPs with 10's or even 100's of cores are envisioned. Simulating these future CMPs efficiently without compromising accuracy is a challenge.

Slack simulation is a general parallel simulation paradigm which provides flexible trade-offs between simulation accuracy and speed. Simulation threads do not synchronize after every target core cycle as in cycle-by-cycle simulation. Rather a maximum slack (the slack bound) is enforced between the clocks of all simulated cores.

A slack simulation may become inaccurate because of simulation violations. Such violations occur when a resource is accessed by two cores in different orders in the simulation and in the target system. We introduce and demonstrate techniques to detect violations, to adapt the simulation slack to maintain a target violation rate, and to checkpoint and rollback a slack simulation when violations are detected. We show some simulation performance/accuracy data for a set of four Splash benchmarks in the context of an 8-core CMP with snooping protocol simulated on SlackSim, our versatile slack simulation platform.

1 Introduction

As computer architecture design rapidly moves into the chip multiprocessor (CMP) era, we cannot keep simulating CMPs in a sequential fashion as single processor systems are, because a single simulation thread must simulate the activities of 10s, 100s or even 1000s of cores.

Fortunately, CMPs have emerged not only as new research targets, but also as new computing platforms, thus offering new opportunities for parallel programs. In parallel programming paradigms, communication and synchronization between different computing processes are critical to performance. Because CMPs provide low-latency access to shared variables they are excellent parallel computing platforms for the implementation of parallel simulators. CMPs typically support the shared-memory programming model, in which a parallel CMP simulator can exploit fast Read/Write operations on shared variables in order to implement communication and synchronization between simulation threads.

In a single-threaded simulation of a CMP, the simulations of every core are interleaved on a clock by clock basis ("cycle-by-cycle" simulation). By incrementing the clock-cycle counter only when all cores have completed their execution of one cycle, any external effect (data, control, or structural hazard) which one core may have on another is faithfully simulated because events effectively take place at the end of each clock. We consider cycle-by-cycle simulation as the "gold standard" against which we measure success.

Cycle-by-cycle simulation can be easily parallelized. Let C be the number of cores in the target CMP and let N be the number of hardware thread contexts in the host CMP. A natural and scalable division of the simulation work is to allocate the simulation of one target core to a simulation thread and then map C/N simulation threads to each hardware thread context in the host. In cycle-by-cycle simulations, every simulation thread executes one cycle of its target core and then synchronizes with a barrier. This approach is scalable both from a programming and from a performance point of view. In some sense each host thread context emulates the behavior of a subset of the target cores.

With this approach the speed of parallel CMP simulation relies on effective implementations at four levels: application (benchmark) layer, target hardware layer, host hardware layer, and simulation layer. At the application (benchmark level) the simulation speedup is limited by the algorithmic speedup. If the target application has little or no parallelism, very little can be gained by running the simulation on a CMP, since each host thread context is merely an emulator of each core. Similarly if the target CMP architecture has limited parallel resources such as cores or if its memory architecture is very inefficient, parallel simulation will not help as simulation threads will be few and/or mostly idle. In the context we are in (the simulator layer), nothing can be done at the benchmark level or at the target CMP architecture level. At the host hardware level, the CMP host must have the resources to execute the parallel simulation efficiently, for example by supporting fast read/write sharing

and by having enough on-chip cache to maintain the working set of the simulation. In this paper we explore ways to make the simulation layer more efficient. The simulator could be designed to maximize the efficiency of the hardware resources offered by the host CMP, but this is an approach that is very implementation dependent and essentially non-portable. Rather we address a more general problem: the interactions among the threads simulating cores and excessive synchronizations among host threads. This synchronization may lead to the serialization of the simulation and low parallel speedups.

Slack simulation is a new paradigm for the parallel simulation of CMPs on CMPs first introduced in [5]. In slack simulations the simulated cores do not synchronize after each simulated cycle as in cycle-by-cycle simulations or after a fixed number of cycles as in quantum simulations [16][9]. The s*imulation slack* of any two target cores in a slack simulation is the difference between their clocks. In *bounded slack* simulations, the simulated clocks of all target cores are kept within a range called the *slack bound*: The simulation of a target core is stalled whenever its clock falls outside the slack bound because its simulation progresses too fast. When the slack is *unbounded*, target cores are simulated independently of each other with no synchronization between simulation threads. As the simulation slack between simulation threads increases, the accuracy may deteriorate but generally simulation speed increases because the simulation of all cores are less and less dependent of each other. In this context, we define accuracy as the difference in any metric of interest, such as CPI, between cycle-by-cycle simulation and slack simulation.

Slack simulations are different from quantum simulations [10][16], in which simulation threads execute a barrier synchronization after a number of simulated cycles. The accuracy of quantum simulations depends on the size of the quantum. When the quantum size is not more than 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 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. For example, threads of a CMP often conflict for shared resources such as the interconnect between cores and L2 banks and such conflicts may occur in only one cycle of latency, which would set the quantum to one clock [2]. A quantum of one clock effectively degrades a quantum simulation to become a cycle-by-cycle simulation. In this paper we do model bus conflicts and thus the critical latency of a quantum simulation would have to be one cycle. In contrast to quantum, slack simulations do not rely on such tight synchronization. The synchronization between simulation threads in slack simulations is much looser and is due to the enforcement of the slack bound, akin to a sliding window in time [7][8].

The sources of simulation errors in slack simulations have been documented at length in [6] and [8]. The flexible slack between target cores in effect causes simulated time distortions, a source of errors for the simulation metrics, because the simulated workload state, the simulated architecture state and the simulation state march at the pace of simulation time, not of simulated time. However, it has been observed [7][15] and was shown in [8] that a slack simulation never deadlocks or becomes unstable, because simulation and simulated times never decrease. In the extreme case (unbounded slack), target cores can be out of sync by thousands of cycles, yet, surprisingly, the error on the execution time is often within single digit (in percent). The key observation made in prior work on slack simulations is that parallel architectural simulators of parallel machines survive violations naturally and thus the major problem is that of measuring and controlling the error rate.

A slack simulation may become inaccurate because of *simulation violations*. A simulation violation occurs when a resource is accessed by two cores in different orders in the simulation and in the targeted system. Accuracy control in slack simulations is the major contribution of this paper. In this paper, we show 1) how to detect and count simulation violations to obtain a confidence level in the simulation results, 2) how to exploit these dynamic measurements to control the rate of errors in adaptive schemes and 3) how to design simulation checkpoints and rollback to recover from violations in speculative schemes.

At first we briefly review the versatile simulation environment we have developed (called SlackSim) to prototype slack simulation schemes and explore the full potential of slack simulations rapidly. More details on SlackSim can be found in [6]. Then we explain how to detect and count simulation violations to estimate the accuracy of a slack simulation run. A new scheme called *adaptive slack simulation* is introduced, in which the simulation violation rate is measured dynamically and kept to a preset minimum by changing the slack bound dynamically. Finally we explore a scheme to checkpoint and rollback a slack simulation in order to preserve its accuracy. Concrete results on simulation accuracy and speed are shown for four Splash benchmarks [18] running on the simulation of an eight core machine with a bus-based snooping protocol in our SlackSim framework.

2 SlackSim

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 threads: 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.

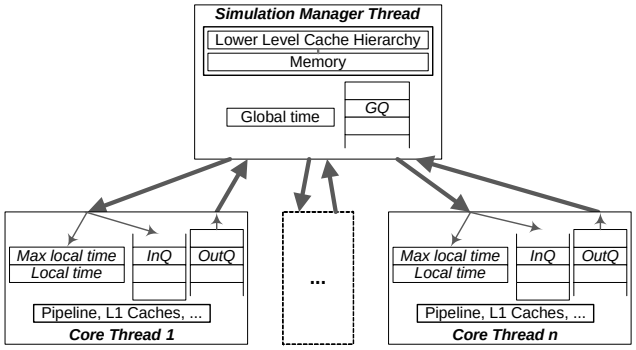


Figure 1. The architecture of SlackSim.

The simulation pace of each core thread is controlled by two variables per core thread shared by the 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. A core thread can advance its own simulation for as long as its local time is less than or equal to its max local time. It suspends itself when its local time reaches its max local time.

The simulation manager thread also 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 according to the slack scheme. When the slowest simulation thread completes a target cycle, the max local times of all simulation threads are incremented.

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 should take effect.

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 (assuming a 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.

Each core thread checks its InQ in every simulated cycle in order to see if one of its requests 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 for various slack simulation schemes.

SlackSim is built around 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 [11]. 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.

The simulation manager thread may look like a serious simulation bottleneck. However, the average amount of work in the manager thread is much less that in each core thread. If the manager thread becomes a bottleneck, then it should be organized hierarchically, reflecting the structure of future large-scale CMPs.

2.1 Experimental Setup

SlackSim can simulate a variety of target CMP configurations. However, in all the results presented in this paper, 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 up to 64 in-flight instructions, 16KB I/D caches and 256K shared L2 cache with an access latency of 8 clocks. The reason behind the small cache sizes is that the benchmarks are considerably scaled down as compared to real workloads and thus the caches must be scaled down as well. L1 caches are lock-up free and kept coherent using a MESI protocol on a request/response bus. Snoop requests are put on the request bus and all cores plus the L2 cache bank snoop the request. Responses (data) propagate on the response bus. The L2 miss latency is 100 clocks. The target CMP is illustrated in Figure 2. Complete details can be found in [6].

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

We have selected four parallel benchmarks from the SPLASH-2 suite [18]: *Barnes*, *FFT*, *LU*, *Water-Nsquared*, shown in Table 1. Every benchmark starts as one single

thread. Then this thread spawns other workload threads. In our experiments, every benchmark is composed of a total of eight 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.

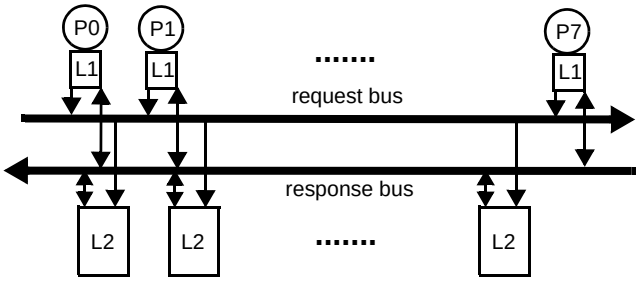


Figure 2. Target CMP

Table 1. Benchmarks.

Benchmark	Input Set
Barnes	1024
FFT	64K points
LU	256 x 256 matrix
Water-Nsquared	216 molecules

Each CMP core is simulated by one POSIX thread. L2 and the bus interconnect are simulated by a separate Pthread, which also controls the simulation. Hence, a simulation is composed of nine POSIX threads simulating an 8-core target CMP.

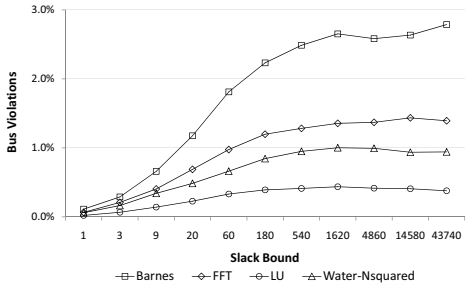
3 Detection of Simulation Violations

Three types of simulation violations can be detected, as introduced in [5]: simulation state violations, simulated system state violations, and simulated workload state violations. These violations can potentially cause simulation errors. Hence, detecting simulation violations allows us to reduce the loss of accuracy, which is the primary goal of this paper.

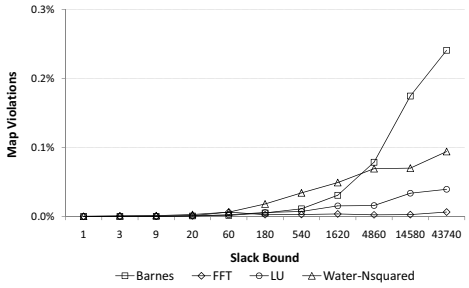
Simulation state violations occur because variables internal to the simulator and keeping track of the occupancy of resources are updated by some earlier operations. One way to detect such violations is to attach a monitoring variable to a resource tracking variable in SlackSim. The monitoring variable records the largest timestamp of any incoming operations on the tracking variable. When an operation is conducted, its timestamp is compared against the value of the monitoring variable. If the incoming timestamp is larger, the monitoring variable is updated with the new value. Otherwise, a violation is detected.

Simulated system state violations happen when storage structures in the target system are accessed by some earlier operations. Such violations are more expensive to monitor since the target system contains large amount of storage structures such as for example protocol directory entries. Nevertheless, the basic detection mechanism is quite similar as for simulation state violations. This time, we need to attach a monitoring variable to every directory entry in order to record the timestamp of incoming operations. A violation is reported when an operation's timestamp is smaller than the current value of the monitoring variable.

Simulated workload state violations occur if values of the same address which is part of the target memory cross each other differently in the simulation than in the target system due to race conditions. In SlackSim, this type of violations does not happen when synchronization in simulation workloads are properly programmed and synchronized because synchronizations are reliably executed inside the simulator using the parallel programming APIs from MP_Simplesim [14].



(a) Bus



(b) Map

Figure 3. Violation rates of bus and cache map with bounded slack.

Figure 3 shows how simulation violation rates trend as the slack bound increases. The X-axis is labelled with the value of the slack bound. The Y-axis stands for simulation violation rate, which is defined as violation count divided by the number of simulated cycles. Figures 3(a) and (b) display the

simulation violation rates of the bus and the cache status map respectively. “Bus violations” counts the number of times when the bus is accessed in a different order in the simulation and in the target system. In the same way “Map violations” counts the number of times the global cache state transitions are not consistent with messages received by the simulation manager thread. We observe the following. First, bus violations are much more frequent than map violations, by at least one order of magnitude. Second, as the size of the slack bound increases, the number of bus violations gradually grows until it reaches a plateau. On the other hand, the number of cache map violations is negligible for small slack bounds (up to 60), and then gradually grows. These observations are consistent with the fact that occurrences of map violations take much longer latencies than bus violations and also that map violations are spread across a much larger state than bus violations.

The data show that simulation violations for an interconnect as simple as a bus are much more frequent than violations of the simulated system state embodied in the cache status map maintained in the simulation manager thread, even when the slack is much larger than the critical latency (here the critical latency is the access latency to the L2 cache, i.e., eight clocks).

In a practical situation, the simulator detects and counts the number of violations of each type and reports these numbers at the end of a simulation run. These numbers help assess the validity of the results. Note that the detection of violations takes place during simulation and unavoidably disturbs the execution of SlackSim itself. Therefore the progress of target threads might be slightly different from what really happens when the violation detection mechanism is turned off. Different violation types may have different impacts on simulation results and users may want to overlook some types of violations based on their perception, experience, or design goals. For example, map violations could have a much greater impact in an evaluation of the cache protocol than bus violations, in which case bus violations might be ignored.

Because of simulation violations, the error rate could become so high as to ruin the credibility of the simulation results. Although we have not experienced such problems, it is prudent to have safeguards in place so that the error rate can be bounded within a predefined range. One way to deal with this problem is to track some error measure such as the simulation violation rate (as suggested above) and to obtain some theoretical bound on the error for the metric of choice. Unfortunately, such bound remains elusive. Rather, in the balance of this paper, we explore two dynamic schemes to reduce or even eliminate simulation errors: adaptive slack simulations and speculative slack simulations.

Adaptive slack simulation aims to control the error rate by changing the slack bound dynamically based on a measure of simulation error. The goal of speculative slack simulation is to eliminate certain (or all) types of simulation violations by checkpointing the simulation periodically, tracking selected violation types and rolling back the simulation whenever selected violations are detected. Both approaches rely on violations detection mechanisms instrumented in a bounded or unbounded slack simulation.

4 Adaptive Slack Simulations

There is no good way to prevent simulation violations from happening once the slack becomes too large. One feasible approach is to pace the simulation speed and violation rate by adaptively adjusting the slack bound size.

This proposed scheme, simply called *adaptive slack*, creates a feedback control loop into a bounded slack simulation to adjust the slack bound based on a running estimate of simulation error. Ideally, the dynamic error rate on the chosen output metric(s) should guide this feedback mechanism. However, it seems impossible to calculate this dynamic error rate in practice.

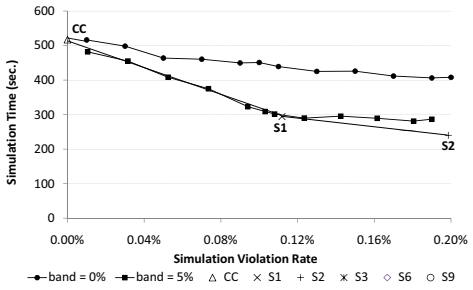
Because of the difficulties associated with computing simulation errors on a specific metric on the fly, we need to choose a convenient proxy for the simulation error to control an adaptive slack simulation. We have chosen the simulation violation rate as the proxy to steer the adaptive slack simulation because it can be easily tracked dynamically, correlates well with errors on the execution time, and is a measure of simulation errors that is likely to impact any metric besides the execution time.

The basic idea is to increase the slack bound when violations happen infrequently, and to decrease it when violations happen frequently (a technique called *slack throttling*). The violation rate is the total number of violations divided by the number of cycles. The violation rates for different types of violations are maintained by the manager thread. Before the simulation starts, the slack bound is set to a default value. If the violation rate is less than a preset target, the slack bound is increased. On the other hand, if the violation rate becomes larger than the target, the slack bound is decreased until it reaches the lowest possible value for the slack bound.

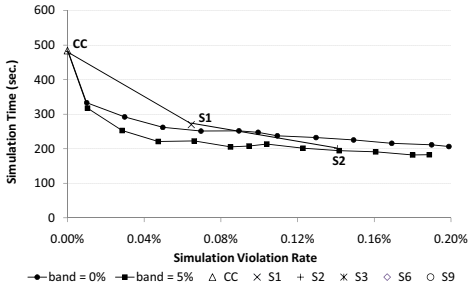
We do not adjust the slack bound for as long as the simulation violation rate remains within a preset range above or below the target violation rate. This range is called the *violation band*.

Figure 4 illustrates the relationship between simulation violation rate and simulation time. There are three series of data. The first two series are results from adaptive slack simulations with different violation bands: 0% and 5%. When the violation band is 5%, the slack bound does not

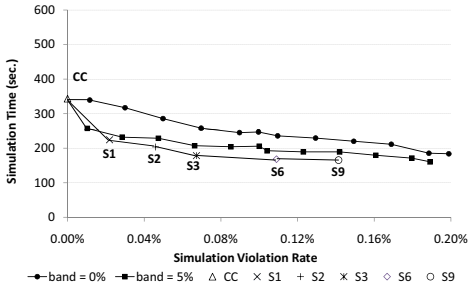
adjust for as long as the current violation rate falls in between 95% to 105% of the target violation rate.



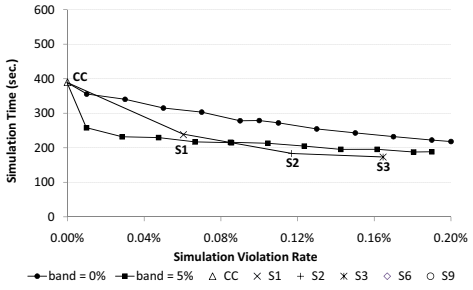
(a) Barnes



(b) FFT



(c) LU



(d) Water-Nsquared

Figure 4. Simulation time vs. violation rates of bounded slack and adaptive slack.

Each of these two series is made of 12 data points, with target violation rates of 0.01%, 0.03%, 0.05%, 0.07%, 0.09%, 0.10%, 0.11%, 0.13%, 0.15%, 0.17%, 0.19%, and 0.20% respectively. Wider violation bands lead to shorter simulation times. We believe that this phenomenon is caused by the overhead of slack bound adjustment.

For the purpose of comparison, the figure also connects a third series of data points including the results of cycle-by-cycle simulation (CC) and bounded slack simulations with slack bounds from 1 to 9 (S1-9). Adaptive slack simulations always run faster than cycle-by-cycle simulation but bounded slack simulations with similar violation rates run faster than their adaptive slack counterparts. This is expected because adaptive slack simulations include overhead to achieve a guaranteed target violation rate, providing a "safety net". This lower performance is the price to pay for this "safety net".

5 Speculative Slack Simulation

In speculative slack stimulation, simulation violations are allowed to occur, but a recovery mechanism is provided to restore the simulation to a consistent state whenever they happen. Speculation and rollback on violations can potentially eliminate all violations from any slack simulation.

As simulation progresses, checkpoints of the entire simulator are made periodically. When a violation is detected, the simulation is rolled back to a previously saved checkpoint where the violation has not occurred. The wasted simulation time between the violation and the closest checkpoint is called the *rollback distance*. After the rollback, simulation resumes from a correct and consistent state. In general, speculative simulation may incur excessive rollbacks due to over-optimistic execution. Whenever a rollback happens, both simulation time and simulation work are wasted. Eventually, rollbacks may cause substantial simulation performance degradation, offsetting the gains of slack simulation to a point that it is worse than cycle-by-cycle simulations. Speculative slack simulation is only useful if it can execute faster than cycle-by-cycle simulation, i.e., when violations are rare. Violations can be made rare and controllable when speculative slack simulation is deployed in conjunction with adaptive slack simulation. This is the approach we have taken in this paper, although speculation can be applied to any slack simulation scheme.

There are four critical mechanisms to make speculative slack simulation work: 1) checkpointing, 2) violation detection, 3) rollback, and 4) forward progress. Although we have not implemented speculative slack simulation completely on SlackSim, we have run simulation experiments and developed a simple analytical model to assess its performance. Checkpointing affects the overall simulation time

the most. Therefore, we have implemented it in details in order to assess the overhead of checkpointing.

5.1 Checkpointing

In order to make a global checkpoint of the simulation, all threads must synchronize, establish a consistent checkpoint, and then proceed [12]. We first overview the thread checkpointing mechanism we have implemented in each simulator thread.

The basic idea of memory-based checkpoint is built on the *fork()* system call [17]. *Fork()* is used to create processes in Unix or Unix-like operating systems, including Linux. After the *fork()* system call, the parent process suspends itself by executing *waitpid()* immediately. The child process proceeds. In the context of speculative slack simulation, the entire context of the parent process serves as a checkpoint of the simulation state. When a rollback becomes necessary, the child process simply exits and the parent process preserving the checkpoint of the simulation is awakened. On a violation, rollback is initiated by the child process. The operating system implicitly handles all the details. After the child process calls *_exit()* to terminate itself, its parent process is awakened to resume from the point where the checkpoint was made. As the simulation moves forward, new checkpoints must be made to preserve recently finished work in case future rollback. At the same time, old checkpoints become useless and should be discarded in order to release system resources. Removal of an old checkpoint begins in the child process. When *kill()* is called from the child, a signal is sent to the suspended parent process that preserves the checkpoint. The process terminates itself and releases all resources associated with the checkpoint.

The above algorithm describes the checkpoint and rollback of a single thread. To take a global checkpoint in SlackSim, the simulation manager thread must force all core threads to stop at the same local time and inform all core threads to take their own checkpoint. At the same time, the simulation manager thread also makes a checkpoint of itself. This set of checkpoints composes the global checkpoint. Whenever a selected violation is detected by the simulation manager thread, the thread instructs all core threads to rollback to their previous checkpoint and rolls itself back as well. Thus, the entire simulation is rolled back to the previous global checkpoint. To ensure forward progress, the simulation is replayed in cycle-by-cycle mode until it safely reaches the next checkpoint.

5.2 Performance of Speculative Slack Simulation in SlackSim

During speculative simulation, four types of overhead are incurred. The first one is the overhead of taking periodic checkpoints. Second, after every rollback, the simulation must return to a previously saved checkpoint, and then start over. The simulation work between the point where the rollback is called and the checkpoint is wasted. Third, after a rollback, we must run simulation in cycle-by-cycle mode until the next checkpoint in order to avoid simulation livelocks. The fourth overhead is the overhead of rolling simulation back to a previous checkpoint. This overhead is omitted in this discussion because it is a secondary factor, and is hard to estimate. Thus, our model slightly underestimates the speculative slack simulation time. Our simple analytical model for the time taken by speculative slack simulation is given by the following formula.

$$T_s = (1-F)T_{cpt} + \frac{FD_rT_{cpt}}{I} + FT_{cc}$$

where

- T_s denotes the simulation time of speculative slack simulation;
- T_{cc} and T_{cpt} denote the simulation times of cycle-by-cycle simulation and of slack simulation with checkpointing respectively;
- F denotes the fraction of checkpoint intervals that have at least one violation;
- D_r denotes the average rollback distance in simulated cycles; and
- I denotes the length of each checkpoint interval in simulated cycles.

The first term in the formula corresponds to the time spent in normal simulation, where no violation happens. The second term is the wasted simulation time due to rollback. The last term is the time spent in cycle-by-cycle mode after a rollback.

In order to reduce the frequency of checkpoints, violations must be few and far between. Adaptive slack simulation is an effective way to reduce the number of simulation violations to any level and in a predictable way. Hence, an obvious strategy is to combine adaptive and speculative slack simulations. In the following, we have adopted a base adaptive slack simulation scheme with a target violation rate of 0.01% (one violation in 10,000 cycles) as the baseline to evaluate the impact of speculation. In order to estimate the overhead of taking periodic checkpoints in SlackSim, we force every core thread to take a checkpoint periodically. The number of cycles between two checkpoints is called the *checkpoint interval*.

Table 2 compares the simulation times of cycle-by-cycle simulation (CC) and adaptive slack simulation (Adaptive) with a target violation rate of 0.01% and violation band of 5%, with the simulation times of Adaptive in which checkpoints are taken periodically every 5k, 10k, 50k, and 100k simulated cycles. Note that we have implemented the

checkpointing mechanism in Linux and the overhead of checkpointing includes the time taken by the fork and join, plus all the effects of making a copy of the virtual space such as page faults, copy on write, etc, as the child thread executes. In the implementations, checkpoints always succeed.

Table 2. Simulation time of schemes with 0.01% target violation rate (sec.).

	CC	SU	Adapt	5K	10K	50K	100K
Barnes	517	157	482	1063	795	537	506
FFT	484	158	318	811	625	382	346
LU	343	142	258	856	594	324	320
W-Nsq	390	135	258	673	512	313	295

The different simulation times are due to different types of overheads. In cycle-by-cycle simulation, the simulation overhead is essentially due to barrier synchronization after each target core cycle. Unbounded slack (third column) runs much faster than cycle-by-cycle by a factor 2 to 3 (but with errors). In Adaptive, the synchronization is relaxed because core threads may have different local times, but collecting information about violations is time consuming. In the adaptive simulations with checkpoints, the overhead of maintaining checkpoints at regular intervals, from 5K to 100K cycles, is added to the overhead of controlling the simulation violation rate in the baseline adaptive slack scheme.

Results in Table 2 show a mixed picture for speculative slack simulation. All the simulations with either 5k or 10k checkpoint intervals run slower than cycle-by-cycle simulations, just because of the overhead of checkpointing. Long simulation times are the effect of frequent checkpointing. As the checkpoint interval grows to 50k, the simulation time drops dramatically. This happens in all benchmarks. The simulation times change very little for 100k checkpoints.

Table 3. Fraction of checkpoint intervals that have at least one violation.

	10K	50K	100K
Barnes	83%	93%	94%
FFT	37%	61%	88%
LU	13%	30%	31%
Water-Nsq	55%	97%	100%

During the execution of a benchmark program, violations may not happen in some checkpoint intervals. We need to estimate the fraction of checkpoint intervals that violate. Table 3 displays the fraction of checkpoint intervals that have at least one violation. In other words, rollback is necessary in these situations. We observe that violations do not happen evenly across benchmarks. For example, with 10k checkpointing intervals, violations happen in 83 percent of checkpoint intervals for *Barnes* while only 13 percent of checkpoint intervals contain at least one violation in *LU*. As the interval becomes larger, the fraction of intervals that violate also increases. For instance, with 100k interval, there is always at least one violation happening in every interval for *Water-Nsquared*.

To complete the model for the performance of speculative slack simulation on SlackSim, we need to estimate the rollback distance when a violation is detected. Table 4 shows the average distance (in simulated cycles) between the beginning of a checkpointing interval that violates and the first violation. This distance gives an estimate of the wasted simulation time because of a rollback triggered by a simulation violation. The data in the table suggest that 12% to 32% of simulation is wasted in *FFT*, *LU*, and *Water-Nsquared* due to violations when intervals are set to 50k or 100k. This wasted simulation time will further reduce the speed of speculative slack simulation.

Table 4. Average distance of first violation within one interval.

	10K	50K	100K
Barnes	4.6k	6.0k	8.0k
FFT	4.0k	16k	27k
LU	4.3k	16k	25k
Water-Nsq	4.9k	12k	12k

By plugging values from Tables 2, 3, and 4 into our analytical model above, we are able to calculate good estimates of the simulation times of a fully functional speculative slack simulation. Table 5 gives these estimates for speculative slack simulations with 50k and 100k checkpoints. To be acceptable, speculative slack simulation must run at least faster than cycle-by-cycle simulation (CC). These results show that the estimated execution time of speculative simulation is always longer than cycle-by-cycle simulation.

Table 5. Estimated overall simulation time of speculative simulation (sec.).

	CC	50K	100K
Barnes	517	578	554
FFT	484	519	550
LU	343	361	352
Water-Nsq	390	461	425

Based on these results, it is hard to reach a definite conclusion on the superiority of speculative slack simulation over cycle-by-cycle simulation. More benchmarks should be run. Also, simulation experiments with violation rates lower than 0.01% in the base adaptive slack scheme might yield better performance. Nevertheless, based on the data we have gathered and given its complexity, speculative slack simulation does not look promising unless violation rates can be brought down significantly.

One way to lower the rollback rate is to focus on violations that have the most impact on simulation accuracy. For example, it may be futile to eliminate bus violations because their impact on simulation errors may be very small. If one would focus on cache map violations alone, which are very rare and have the potential to cause more simulation errors especially on some metrics such as coherence overhead or miss rates, then the overhead of rollbacks may be greatly diminished and checkpoint intervals may be much longer thus reducing checkpointing overhead further. In our results we have tracked all violations, including bus violations. To be accurate the critical latency must be one cycle and quantum would be the same as cycle-by-cycle.

6 Related 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* [3][10]. The best known framework for optimistic simulation is Jefferson's Time Warp [13]. 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 target of Time Warp was the simulation of queueing networks. We believe that our work on speculative slack simulation is the first attempt to apply the optimistic approach of Time Warp to CMP simulations.

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 [16]. WWT II uses a conservative approach with barrier synchronizations. The simulation accuracy of this so-called quantum 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 accuracy is compromised. Due to short latencies in CMPs the quantum size must be kept short. When conflicts in the interconnect are simulated the critical latency drops to 1 clock [2].

Slack simulations are different from quantum simulations. In quantum simulation a barrier synchronization is executed periodically to re-synchronize the simulation threads. In slack simulations [5][6][7][8], the synchronization is more relaxed as simulation threads must just remain within an a time window. In effect, with slack simulation the barrier advances every time the slowest simulated core advances by one cycle. Simulation errors may results and this paper explores two schemes to eliminate errors in Slack simulations.

Parallel simulation has been applied to CMP simulation in [4]. In this work, the architecture of the 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. Two conservative slack 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 accept small simulation errors as trade-off with simulation speed. Bounded/unbounded slack simulations can run with any slack size. Second, 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 [9] in a message-passing parallel server envoronment. The size of the quantum is adaptively adjusted according to the amount of network traffic in the target system. The quantum is increased when packets are not exchanged, and it is shortened as the packet traffic increases. It is surmised that the error rate increases when message traffic increases because message exchanges are not modeled accurately as the quantum size is larger than the critical latency. With this simple mechanism, the simulation speedup is improved with less than a 5% error. In our adaptive scheme we have proposed ways to measure dynamically the rate of simulation violations, which is a more direct measure of errors.

A recent CMP simulator called Graphite [15] targets multicore systems with 1000's of cores running on large scale distributed systems with a mix of simulations and analytical models. Three different simulation synchronization schemes called *Lax*, *Lax-Barrier* and *Lax-P2P* are contemplated. All these schemes allow some slack between the simulations of cores. *Lax* uses the same approach as unbounded slack and the authors confirm our experience that simulation errors are tolerable. *Lax-Barrier* is similar to quantum simulations. In *Lax-P2P*, each core periodically chooses another core at

random and synchronizes its simulation with it; this is an interesting approach, which we plan to explore further.

7 Conclusions

Slack simulation offers new trade-offs between simulation speed and accuracy. It accelerates the parallel simulation of CMPs by relaxing the tight synchronization enforced between simulation threads in cycle-by-cycle (cycle accurate) simulation but it suffers from simulation errors due to violations. To control the error rate caused by violations, we have introduced an adaptive slack simulation algorithm. Our experiments have shown that adaptive slack simulation is effective at controlling the rate of simulation violations.

We have proposed an optimistic simulation scheme called speculative slack simulation. We have described in details an implementation involving periodic simulation checkpointing, and rolling back whenever violations are detected. We have proposed a simple analytical model, which in conjunction with simple simulation measurement can evaluate the efficiency of speculative slack simulation.

The model suggests that speculative simulation may be worth considering provided the violation rate of the base slack simulation scheme can be kept to a minimum while keeping simulation efficiency high. At the end the performance of checkpoint/rollback is a compromise between the overhead of avoiding violations in the base scheme and the overhead of recovering from a violation. We are currently working in that direction. Another avenue for improvement is to minimize the checkpointing overhead.

The experiments we have run so far are of modest scale (simulation of eight cores on an 8-core CMP host). Larger-scale simulations must be run to reach a definite conclusion about the viability of speculative slack simulations. The problem is that most current commercial CMPs still support a small number of thread contexts. In the future we plan to run SlackSim on larger scale systems and to expand the pool of our benchmark programs. Finally we plan to fully deploy the speculative slack simulation scheme on top of SlackSim.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CSR-0615428 and CCF-0834798.

References

[1] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: an infrastructure for computer system modeling,” *IEEE Computer*, vol. 35, 2002, pp. 59-67.

[2] D. Burger and D. Wood, “Accuracy vs. Performance in Parallel Simulation of Interconnection Networks,” in *Processings of 9th International International Symposium on Parallel Processing*, 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, No. 3, pp. 176-200, July 2002.

[5] J. Chen, M. Annavaram, and M. Dubois, “SlackSim: A Platform for Parallel Simulations of CMPs on CMPs”, CENG-2008-6, Department of Electrical Engineering, University of Southern California, 2008.

[6] J. Chen. Parallel Simulations of Chip Multiprocessors. Ph.D. Thesis. Ming-Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, August 2009.

[7] J. Chen, M. Annavaram and M. Dubois,”SlackSim: A Platform for the Parallel Simulation of CMPs on CMPs,” *Sigmetrics/ Performance 2009, also Performance Evaluation Review*, Vol. 37, Issue 2, pp. 77-78, September 2009 .

[8] J. Chen, M.Annavaram, and M. Dubois,"Exploiting Simulation Slack to Improve Parallel Simulation Speed," *38th Int. Conference on Parallel Processing (ICPP)*, September 2009.

[9] 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.

[10] R. M. Fujimoto, “Parallel discrete event simulation,” *Communications of the ACM*, Vol. 33, No. 10, pp. 30 - 53, Oct, 1990.

[11] G. Hinton et al.,”The Microarchitecture of the Pentium 4 Processor,” *Intel Technology Journal*, Q1, 2001.

[12] G. Janakiraman and Y. Tamir, “Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory MultiMulticomputers,” in *Proc. of the 13th Symposium on Reliable Distributed Systems*, pp. 42-51, Oct. 1994.

[13] 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.

[14] N. Manjikian, “Multiprocessor Enhancements of the SimpleScalar Tool Set,” *ACM SIGARCH Computer Architecture News*, Mar. 2001, pp. 8-15.

[15] J. Miller et al., “Graphite: A Distributed Parallel Simulator for Multicores,” Proc. of the 16th Int. Symposium on High-Performance Computer Architecture (HPCA), January 2010.

[16] 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.

[17] J. Plank, K. Li, M. Puening, “Diskless Checkpointing,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, Issue 10, Oct. 1998, pp. 972-986.

[18] 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.