Adaptive and Speculative Slack Simulations of CMPs on CMPs

Jianwei Chen, Lashkmi Kumar Dabbiru, Daniel Wong, Murali Annavaram and Michel Dubois
Ming Hsieh department of Electrical Engineering
University of Southern California, Los Angeles CA90089-2560
jianwei.chen@oracle.com, {dabbiru, wongdani, 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 order 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 five Splash benchmarks in the context of an 8-core CMP with a snooping cache coherence protocol simulated on SlackSim, our universal 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, leading to unacceptable simulation slowdown. Fortunately, CMPs have emerged as new parallel computing platforms, thus offering new opportunities to overcome the sequential simulation bottleneck. It is imperative that future CMP designs exploit current CMPs to parallelize simulations. In its simplest form, each host CMP core can simulate a subset of target CMP cores. The parallel simulation paradigm of using host CMPs to simulate target CMPs is a scalable solution. As the host thread count increases target CMPs with potentially more cores can be simulated without encountering the dramatic slowdowns that plague sequential simulations. 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 (will be referred to as CC) simulation as the "gold standard" against which we measure the success of the proposed approach in this research.

CC 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 CC 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 performance point of view.

The speed of parallel CMP simulations relies on effective implementations at four layers: (1) application (benchmark) layer, (2) target hardware layer, (3) host hardware layer, and (4) simulation layer. At the application layer 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 target core. If the target CMP architecture has inherent design bottlenecks such as a small number of cores or if its memory architecture is very inefficient, parallel simulation will not help. For instance, if the target cores mostly wait on memory then the simulation threads that simulate the target cores will also be mostly idle. 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. The inherent inefficiencies at the benchmark, target CMP and host hardware layer are orthogonal to the focus of this research, which is to improve the performance of the simulation layer. For instance, 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 this research addresses a more general problem: the interactions among the host threads simulating cores and excessive synchronizations among them. The parallel CC simulation approach as described above is inefficient as the number of host instructions executed between two synchronizations is just several thousands. This frequent 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 [8][9]. 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 [18], rather they are granted a *slack*. The *simulation 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*. For example if the slack is set at 10 cycles, then the difference between any two target core clocks can be at most 10. 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. Our prior research [8][9] as well as more recent work [16] have shown that slack simulations may lead to simulation violations, which in turn affects simulation accuracy. In general, as the simulation slack between simulation threads increases, the number of simulation violations increase but, in general, the simulation speed increases because the simulations of the cores are less and less dependent on each other.

Slack simulations are different from quantum simulations [18], 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 target 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 [3]. If bus conflicts are accurately modeled then the critical latency of a quantum simulation would have to be one cycle. A quantum of one clock effectively degrades a quantum simulation to become a cycle-by-cycle simulation. In contrast to quantum, slack simulations do not rely on such rigid synchronization. Rather than strictly enforcing synchronization at the end of each quantum, bounded slack simulations enforce synchronization *only* when the simulation slack between any two cores reaches the bound. Furthermore, when the slower core advances one cycle the faster core simulation can be immediately released and can advance one cycle. Such flexibility enables slack simulations to provide consistent speedup over prior approaches that rely on rigid synchronization.

It is important to understand the distinction between simulation violation and simulation error. We first provide a brief description of simulation violations. A comprehensive description is provided in section 3.

Simulation Violations: In order to understand the sources of simulation violations in slack simulations, it is necessary to differentiate between simulated time and simulation time. Simulation time is defined as the amount of time the simulator has executed for, which can be measured using wall clock time. Simulated time is defined as the clock cycle that a target core is currently running at. Slack simulations cause distortion in simulated time. For instance, at simulation time (or wall clock time) T one target core may be running at cycle 1 while another target core may be running at cycle 5. This distortion in simulated time is a source of violations in slack simulation. For instance, a target core may access a shared resource such as an L2 cache bank at simulated cycle 5 while another target core may access the same resource at simulated cycle 1. Both accesses may happen at the same simulation time T resulting in an out-of-order access to the L2 cache bank in terms of simulated cycles.

Simulation Error: We define simulation error as the difference in any metric of interest, such as target CMP's execution time or CPI, between CC simulation and slack simulation. In this paper we use the target CMP's execution time difference to measure simulation error.

In essence, the flexible slack between target cores causes simulated time distortions, which eventually lead to simulation errors, because the simulated workload state, the simulated architecture state and the simulation state march at the pace of simulation time, not of simulated time.

The goal of our research is to improve simulation performance by reducing simulation time without compromising simulation accuracy as measured by simulation error. At this point it is important to emphasize that a slack simulation never deadlocks or becomes unstable, because both simulation and simulated times never decrease [6][7][16]. In the extreme case (unbounded slack), target cores can be out of sync by thousands of cycles, yet, surprisingly, the simulation error (on the target execution time metric) is often within single digit percent. The key observation made in prior work on slack simulations is that parallel architectural simulators of parallel machines survive violations naturally. Thus the major problem is that of measuring and controlling the error rate, while still taking advantage of slack for simulation performance. Industrial design teams explore the next generation of CMPs using the current generation of CMPs. In this context accuracy is paramount and hence cycle-by-cycle simulation may be the only acceptable approach. Even in exploratory studies evaluating large scale futuristic CMPs, the error rate must be tightly controlled to make sure that it does not compromise the intended purpose of simulation, namely the quantitative evaluation of the design.

Accuracy control in slack simulations is explored to enhance the broader use of slack simulations and it is the major contribution of this paper. In particular this paper makes the following three contributions:

1) A method to detect and track simulation violation rate and use it as a universal confidence factor in the simulation results.

2) A technique to adapt slack dynamically at run time to keep the violation rate at or below a desired threshold. The dynamic adaptation mechanism uses our violation tracking mechanism to achieve its goal.

3) Finally this paper explores the use of checkpoint and roll back the simulation when violations occur, thereby completely eliminating all simulation violations.

Concrete results on simulation accuracy and speed are shown for five Splash benchmarks [21] running on an eight core target CMP with a bus-based snooping protocol. In this paper we do not revisit the problem of parallel vs. sequential CC. Rather we look at the performance and accuracy of slack simulations. So our newly proposed schemes are compared against parallel CC both for accuracy and performance.

The rest of the paper is organized as follows. In section 2 the simulation environment used in this paper is briefly reviewed. In section 3 we overview simulation violations and their detection to assess the accuracy and quality of a slack simulation run. In sections 4 and 5 we introduce two new schemes called *adaptive slack simulation* and *speculative slack simulation* respectively. Section 6 presents related work and section 7 our conclusions and expose avenues for future work.

2 SlackSim

We have developed a simulation platform to experiment with slack simulations and called SlackSim[9]. In SlackSim, simulations are parallelized using the POSIX threads programming model. Figure 1 shows the general architecture 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.

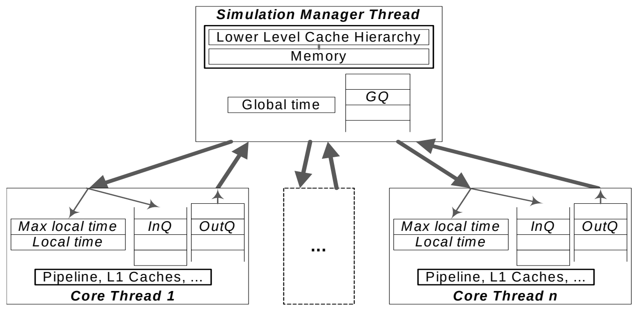


Figure 1. The architecture of SlackSim.

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. For instance, when simulating a bounded slack simulation with a slack of 5 cycles the max local time of each core is set to global time + 5. 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) at which 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 upon SimpleScalar [1]. The two most significant modifications we have made to SimpleScalar are: 1) modifications to enable the simulation of every core in separate threads and 2) modifications to support an Intel NetBurst-like OoO microarchitecture [12]. 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 seems to be a simulation bottleneck. However, prior work [7] has shown that the average amount of work in the manager thread is much less than that of 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 2MB shared L2 cache with an access latency of 8 clocks. 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. 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 five parallel benchmarks from the SPLASH-2 suite [21]: *Barnes*, *FFT*, *LU*, *Water-Nsquared* and Radix 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.

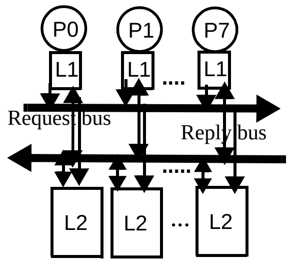


Figure 2. Target CMP

Benchmark	Input Set
Barnes	1024
FFT	64K points
LU	256 x 256 matrix
W-Nsq	216 molecules
Radix	262144 keys

Table 1. Benchmarks.

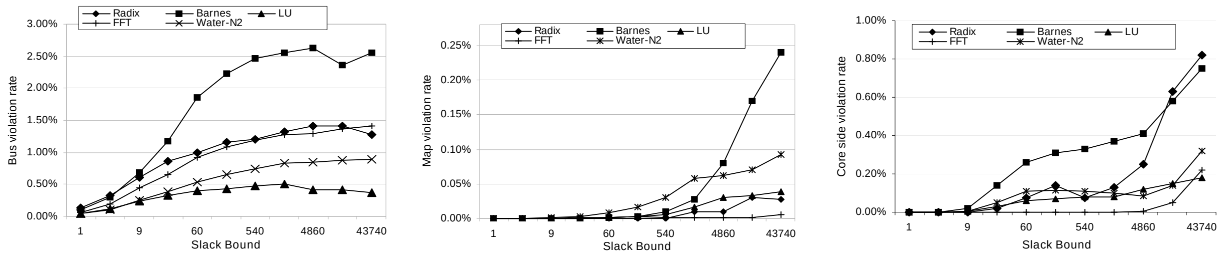
Each CMP core is simulated by one POSIX thread. L2 and the bus interconnect are simulated by a separate Pthread, which also controls the simulation. Hence, a simulation is composed of 9 POSIX threads simulating an 8-core target CMP.

3 Categorization and Detection of Simulation Violations

Simulation violations are categorized into three types. In this section we overview them and provide methods to detect them. This detection of simulation violations is at the root of the adaptive and checkpoint-rollback schemes proposed in this paper to control the accuracy of slack simulations.

3.1 Simulation State Violations

Simulation state violations occur when variables internal to the simulator that track and record the state of each resource are updated out of order. For instance two target cores may both access the bus at the same simulated cycle 5 since the two target cores may reach their respective simulated cycle 5 at different simulation times. This is an inconsistency in the simulation because the bus appears to satisfy two bus requests at the same time in the simulated system. In the above example the simulation state of the resource that is shared between multiple target cores is violated. It is also possible that the simulation state of a target core itself can



(a) Bus violations (b) Map violations (c) Core side violations

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

be violated. For instance, if two L1 miss requests are satisfied in a different order in slack simulation as compared to CC then the core may execute instructions in different order compared to CC resulting in a simulation state violation on the core side.

Simulation state violations can be detected by attaching a monitoring variable to a resource tracking variable. The monitoring variable records the largest timestamp of any incoming operation on a 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.

In SlackSim, the manager thread maintains a monitoring variable for the shared bus to detect bus violations. To detect violations at the core thread side a gross approximation is used. Instead of maintaining a monitoring variable for each internal resource in the core, the entire core node is treated as one resource and if any incoming event's timestamp is less than the local time of the core a violation is reported. This simplification may lead to over-counting simulation state violations since not all late incoming events might create a simulation state violation.

3.2 Simulated System State Violations

Simulated system state violations happen when storage structures in the target system which keep the information needed to enforce correct hardware operation are accessed out of order. For instance, consider a two core CMP with directory based coherence. Target core 1 may write to a cache block at its simulated cycle 4 at simulation time T1 thereby setting the directory structure's presence bit and dirty bit to point to target core 1. However, target core 2 may reach its own simulated cycle 3 at a later simulation time T2 (T2>T1) and access (read/write) a different word in the same cache block. Here at simulated cycle 3 of target core 2 the directory structure content is inconsistent with what target core 2 would have seen in the CC simulation. Even the final directory structure content after both target cores accessed the cache block is inconsistent with what would be in the CC simulation. Just like simulation state violations, simulated system state violations can occur either in target cores or on the shared resources of the target CMP. These violations are more expensive to monitor since the target system contains large amounts of storage structures, such as cache protocol directory entries, but the basic detection mechanism is similar to that of simulation state violations.

In the current simulations coherence in the target system is maintained by snooping on a bus. However the manager thread must keep track of the state of all blocks in L1 caches in order to simulate snooping. The data structure that does that is very much like a directory. We call it the *cache map* to avoid confusion. The cache map records the content of all L1 caches in the target system. It is equivalent to a copy of the L1 cache directories. The manager thread maintains a monitoring variable for every cache map entry. At the core thread side, we use the same approximation as for simulation state violations, leading to the over-counting of violations. For instance, consider the case where a core's local time is 10 and the core receives an incoming request with a timestamp of 5. We conservatively count this incoming request as a simulated system state violation. However, this incoming request should have generated a simulated system violation *only* if the core had accessed the affected cache line sometime after its local time 5. In order to remove this over-counting of violations one would have to associate a monitoring variable to timestamp the accesses of each L1 cache line in the core.

3.3 Simulated Workload State Violations

Simulated workload state violations occur if values of the same address which is part of the target memory system cross each other differently in the simulation than in the target system due to race conditions. In the above directory example used to illustrate simulated system state violations, if both cores were to access the same data address, then target core 2 would read stale data since its simulated cycle 3 was executed later than the simulated cycle 4 of target core 1. Clearly these violations result in an incorrect execution of the benchmark. The workloads we use are properly synchronized to avoid data races on shared variables. Furthermore, in SlackSim, synchronizations are reliably executed inside the simulator using the parallel programming APIs from MP_Simplesim [15] so that simulated workload state violations do not occur.

3.4 Basic Results for Simulation Violations

Figure 3 shows how simulation violation rates trend as the slack bound increases. The X-axis is labelled with the slack bound. The Y-axis is labelled with the simulation violation rate, which is defined as the violation count divided by the number of simulated cycles. Figures 3(a) and (b) display the simulation violation rates of the bus and cache map, which correspond to the simulation state and simulated system state violations, respectively. Both violations are detected by the slack manager thread. Figure 3(c) shows the violation rate at the core side.

The results shown in Figure 3 lead to the following observations. First, bus violations are much more frequent than core and cache map violations. 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. As explained in section 3.1 and 3.2, our simplified approach to measuring core violations overcounts the number of violations and yet the violation rate is negligibly small for slack bounds up to 9 cycles and then gradually increases. These observations are consistent with the fact that cache map modifications have much longer latencies than bus accesses and are distributed across a large state.

Figure 4 shows how the relative simulation error rate varies as the slack increases. The relative simulation error is the difference between the execution times obtained with a slack simulation and with CC divided by the execution time obtained with CC. The figure shows that the error rate increases with slack, and the trend is very similar to that of simulation violations. The strong correlation between simulation violation rate and simulation error rate is an important observation that we will exploit in our adaptive and speculative slack simulation frameworks.

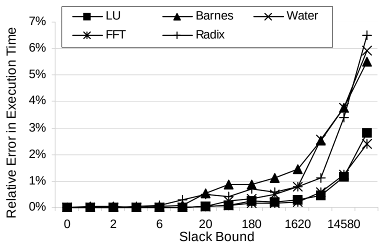


Figure 4. Simulation error with bounded slack.

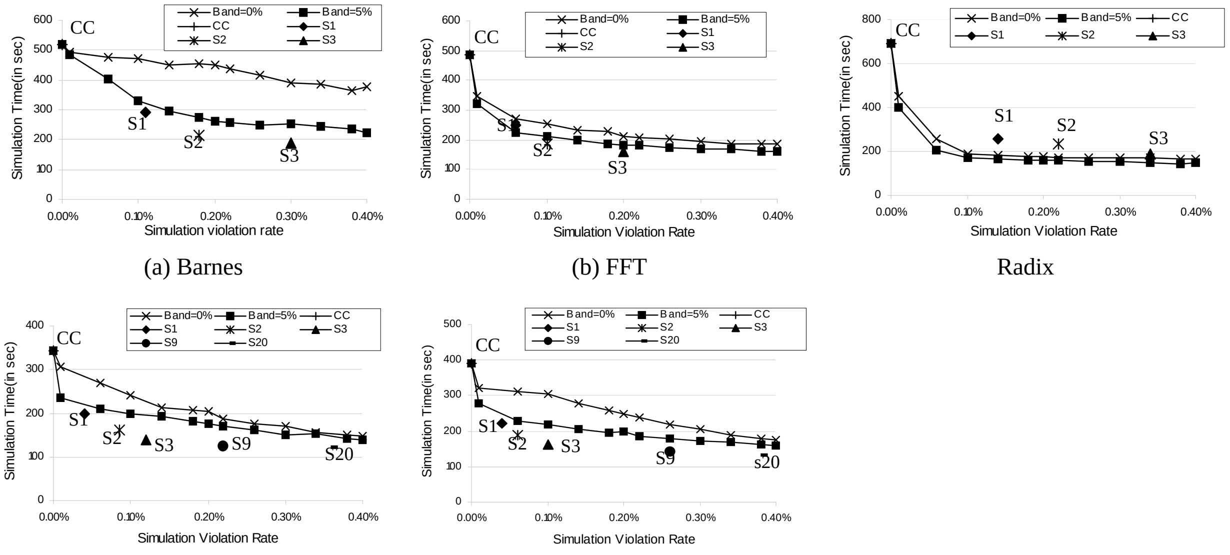
3.5 Controlling Simulation Errors

We have demonstrated in section 3.4 (as well as in our previous work [8][9]) that with small slack bounds simulation violations and the resulting simulation error rate can be kept small while achieving good simulation speedup. Even when some violation rates are high it is worth noting that not all violations are equally important. 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, cache 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.

Ideally users would like to know the simulation error rate at the end of a simulation. Unfortunately measuring the error rate on a metric dynamically is not feasible. However, our results show that there is a strong correlation between the simulation error rate and the simulation violation rate. Hence one can detect and count the number of violations of each type and report these numbers at the end of a simulation run. Given that the simulation error is correlated with the violation rate a user can assess the validity of the results based on the measured violation rate. 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. The overhead of tracking violations in SlackSim slows the simulation down by a factor of 5 to 10% for all benchmarks.

While simulation violation rates are quite low the basic SlackSim mechanism does not necessarily guarantee that violation rates stay low in all cases. Depending on the interactions between benchmarks and target CMP architecture violation rates can be high, and 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 would be to derive some theoretical bound on the error rate for the metric of choice based on the simulation violation rate. Unfortunately, such bound remains elusive. Rather, in the balance of this paper, we explore two dynamic schemes to reduce or even eliminate simulation violations: 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 violation rate. Ideally one would prefer to control the simulation error rate rather than the violation rate. But as mentioned earlier due to the infeasibility of



(a) Barnes (b) FFT Radix

(c) LU (d) Water-Nsquared

Figure 5. Simulation time vs. simulation violation rate of bounded slack and adaptive slack.

measuring simulation error rate we use simulation violation rate as a proxy. 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 bounded slack simulations.

4 Adaptive Slack Simulations

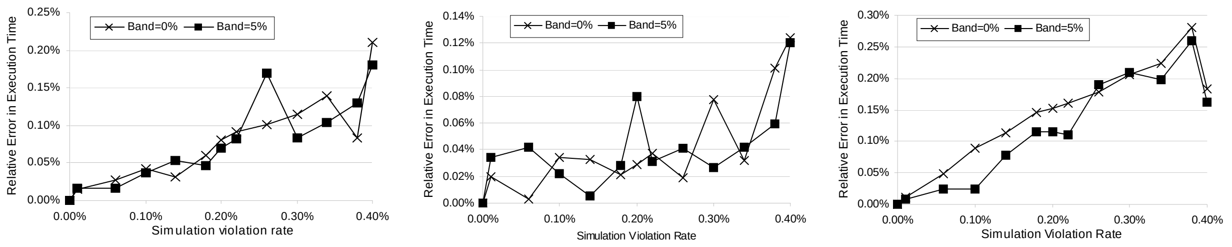
In this section we present an approach to control violation rate by adaptively adjusting the slack bound. This proposed scheme, called *adaptive slack*, creates a feedback control loop into a bounded slack simulation to adjust the slack bound based on a running estimate of simulation error. Ideally, the dynamic error rate on the chosen output metric(s) should guide this feedback mechanism.

However tracking the error rate on the metric dynamically is unfeasible. Hence 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. We compute the total simulation violations using the summation of simulation state violations and simulated system state violations. These violations can be easily tracked dynamically, and their number correlates well with errors on the execution time.

The basic idea is to increase the slack bound when violations happen infrequently, and to reduce it when violations happen frequently. The violation rates for the bus and cache map are maintained by the manager thread. Core violations are tracked by individual target core threads, which communicate their violation count to the manager through a shared variable. 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 up to a maximum slack bound value. On the other hand, if the violation rate becomes larger than the target, the slack bound is progressively decreased. In our current implementation the default slack bound is set as 10 cycles and the maximum slack bound is 5000 cycles. We increase the slack bound in increments of 5% over the current slack. When the slack bound needs to be decreased we decrease it by 35% until the slack eventually becomes zero. By rapidly decreasing the slack we quickly bring down the violation rate.

Frequent changes to the slack bound even if the violation rate deviates from the target rate by a very small amount may cause high frequency oscillations and large overheads. To avoid that, we explored one simple approach: 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 5 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. Each of these two series is made of 12 data points, with target violation rates of 0.01%, 0.06%, 0.10%, 0.14%, 0.18%, 0.20%, 0.22%, 0.26%, 0.30%, 0.34%, 0.38%, and 0.40%. Wider violation bands lead to shorter simulation times. This phe-



(a) Barnes (b) FFT LU

Figure 6. Correlation between simulation violation rate and relative error in execution time.

nomenon is caused by the overhead of slack bound adjustment.

For the purpose of comparison, Figure 5 also shows a third series of data points for CC and bounded slack simulations with slack bounds from 1 to 20 (S1-20). Adaptive slack simulations always run faster than cycle-by-cycle simulation but most of the times bounded slack simulations with similar violation rates run faster than adaptive slack with the same target violation rate. 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”.

Figure 6 shows the correlation between the simulation violation rate and the relative execution time error of the target system for a subset of benchmarks. We observe a general trend that relative execution time errors increase as the violation rates grow. In our evaluations, we considered only one target metric, namely target CMP execution time. However in architectural simulations many other metrics may be considered and more work is needed to evaluate the correlation with other metrics.

5 Speculative Slack Simulation

In speculative slack simulation, simulation violations are allowed to occur, but a recovery mechanism is provided to restore the simulation to a consistent state whenever they happen. Checkpointing 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 before the violation 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, excessive 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 components to make speculative slack simulation work: 1) checkpointing, 2) violation detection, 3) rollback, and 4) forward progress. Checkpointing affects the overall simulation time the most. Hence, we have implemented it in detail to assess the checkpointing overhead. We have yet to fully integrate the rollback mechanism triggered on a violation detection. Although we have not fully implemented speculative slack simulation in SlackSim, we have developed a simple analytical model to assess its performance. This model helps understand the overheads of speculative slack simulations.

5.1 Checkpointing

In order to make a global checkpoint of the simulation, all threads must synchronize, establish a consistent checkpoint, and then proceed [13]. We first overview the thread checkpointing mechanism we have implemented in each simulator thread.

The basic idea of memory-based checkpoints is built on the *fork()* system call [17]. *Fork()* is used to create processes in Unix or Unix-like operating systems, including Linux. Each simulator thread executes the *fork()* system call to create a checkpoint. The parent process suspends itself by executing *waitpid()* immediately. The child process proceeds. In the context of speculative slack simulations, 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 from its waitpid() call. On a violation, rollback is initiated by the child process. By using the fork() and waitpid() functions, we rely on the operating system to implicitly handle all the details of checkpoint and rollback. 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 of a future rollback. At the same time, old checkpoints become useless and should be discarded in order to release system resources. The 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 parent 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. When a core thread detects a violation, it communicates with the manager thread through shared variables. 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. The rollback mechanism followed by CC simulation is the only component that is not yet integrated into SlackSim.

5.2 Performance of Speculative 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 the simulation back to a previous checkpoint. This overhead is omitted in this discussion because it is a secondary factor, and it is hard to estimate without fully implementing it in SlackSim. 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 with checkpoint, 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. Simulation times of schemes.

	CC	SU	Adapt	5K	10K	50K	100K
Barnes	517	108	485	1063	795	542	513
FFT	484	105	320	811	625	391	348
LU	343	105	236	856	594	318	315
W-Nsq	390	94	277	673	512	315	294
Radix	690	89	398	1027	792	482	452

Table 2 compares the simulation times of cycle-by-cycle simulation (CC) and adaptive slack simulation (Adapt) with a target violation rate of 0.01% and a 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. 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 3 to 4 (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. Checkpoint violation rate and Rollback distance

	F			***D_r***		
	10K	**50K**	**100K**	**10K**	**50K**	**100K**
Barnes	83%	93%	94%	4.6k	6.0k	8.0k
FFT	37%	61%	88%	4.0k	16k	27k
LU	13%	30%	31%	4.3k	16k	25k
Water-Nsq	55%	97%	100%	4.9k	12k	12k
Radix	25%	65%	76%	4.5k	15k	16k

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. In Table 3, the columns under "F" display the fraction of checkpoint intervals that have at least one violation for various checkpointed intervals. 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 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. In Table 3 the columns under D_r show 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, Radix* 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.

By plugging values from Tables 2 and 3 into our analytical model above, we are able to calculate good estimates of the simulation times of a fully functional speculative slack simulation. Table 4 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. However, 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.

Table 4. Estimated overall simulation time

	CC	**50K**	**100K**
Barnes	517	578	555
FFT	484	528	550
LU	343	363	355
Water-Nsq	390	463	422
Radix	690	698	693

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 core and 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 core cache map and bus violations. Evaluating the speculative schemes for less ordered interconnects (like mesh interconnect) and multiprogrammed workloads is interesting as the violation count in these scenarios should be less. Finally, the overhead of checkpointing itself can be reduced by reducing the involvement of the operating system in managing checkpoints. For instance, if we can create a customized checkpoint that saves the minimal amount of simulation and simulated system state it is possible to reduce checkpointing overhead.

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* [5][11]. The best known framework for optimistic simulation is Jefferson's Time Warp [14]. 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 [18]. WWT II uses a conservative approach with barrier synchronizations, an approach referred to as quantum simulation. The simulation accuracy 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 [3].

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 [8][9], 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 control errors in slack simulations.

Parallel simulation has been applied to CMP simulation in [5]. 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. Slack simulations take advantage of fast R/W accesses to shared variables, instead of exchanging messages through MPI.

An adaptive quantum simulation scheme is proposed in [10] in a message-passing parallel server environment. 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 [16] 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 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 few and 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 on SlackSim.

7 Conclusions and Future work

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 simulation violations. A method to detect and track violations is given and the measured simulation violation rate is used to control and ameliorate the error rate. We have introduced an adaptive slack simulation scheme that adjusts the simulation violation rate to keep simulation violations under a given threshold. 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 rollback 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. More importantly efficient checkpointing is key. 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. Another avenue for improvement is to minimize the checkpointing overhead. In this research we have used a simple general-purpose memory-based checkpointing method. The whole simulator is checkpointed. A tailor-made checkpointing scheme which does not involve the operating system as much and which does not save the entire virtual space of the simulation would certainly improve performance.

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 definitive 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. Future work also includes exploring slack simulations for heterogeneous multicores (as a target CMP) and integrating SlackSim with power [2], thermal [19] and reliability [20] models.

Acknowledgments

This material is based upon work supported by the National Science CAREER-0954211, CSR-0615428 and CCF-0834798.

References

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

[2] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A Framework for Architectural Level Power Analysis and Optimizations,” In *Proceedings of 27th Annual International. Symposium on Computer Architecture*, pp. 83-94, 2000.

[3] D. Burger and D. Wood, “Accuracy vs. Performance in Parallel Simulation of Interconnection Networks,” In *Proceedings of 9th International Symposium on Parallel Processing*, pp. 22-31, 1995.

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

[5] M. Chidester and A. George, “Parallel simulation of chip-multiprocessor architectures,” In *ACM Transactions on Modeling and Computer Simulation*, Vol. 12, No. 3, pp. 176-200, July 2002.

[6] 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.

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

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

[9] J. Chen, M.Annavaram, and M. Dubois, "Exploiting Simulation Slack to Improve Parallel Simulation Speed," In *38th International Conference on Parallel Processing*, pp 371-378, September 2009.

[10] A. Falcon, P. Faraboschi, and 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.

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

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousell, ”The Microarchitecture of the Pentium 4 Processor,” In *Intel Technology Journal*, Q1, 2001.

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

[14] D. R. Jefferson, B. Beckman, F. Wieland, and L. Blume, “Distributed Simulation and the Time Warp Operating System,” In *Operating Systems Review*, Vol. 21, pp. 77-93, 1987.

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

[16] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, “Graphite: A Distributed Parallel Simulator for Multicores,” In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, January 2010.

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

[18] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis and D. A. Wood, “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,” In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 48-60, May 1993.

[19] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, “Temperature aware microarchitecture,” In *International Symposium on Computer Architecture*, June 2003.

[20] J. Suh, M. Annavaram, and M. Dubois, “Soft Error Benchmarking for L2-cache with PARMA,” In *Sixth Annual Workshop on Modeling, Benchmarking and Simulation*, June 2010

[21] 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.