Accelerating Multi-threaded Application Simulation Through Barrier-Interval Time-Parallelism

Paul D. Bryan, Jason A. Poovey, Jesse G. Beu, and Thomas M. Conte
College of Computing,
Georgia Institute of Technology
Atlanta, GA
{paul.bryan,japoovey,jbeu3,conte}@gatech.edu

Abstract— **In the last decade, the microprocessor industry has undergone a dramatic change, ushering in the new era of multi-/manycore processors. As new designs incorporate increasing core counts, simulation technology has not matched pace, resulting in simulation times that increasingly dominate the design cycle. Complexities associated with the execution of code and communication between simulated cores has presented new obstacles for the simulation of manycore designs. Hence, many techniques developed to accelerate uniprocessor simulation cannot be easily adapted to accelerate manycore simulation.**

In this work, a novel time-parallel barrier-interval simulation methodology is presented to rapidly accelerate the simulation of certain classes of multi-threaded workloads. A program delineated into intervals by barriers may be accurately simulated in parallel. This approach avoids challenges originating from unknown thread progressions, since the program location of each executing thread is known. For the workloads tested, wall-clock speedups range from 1.22x to 596x, with an average of 13.94x. Furthermore, this approach allows the estimation of stable performance metrics such as cycle counts with minimal losses in accuracy (2%, on average, for all tested workloads). The proposed technique provides a fast and accurate mechanism to rapidly accelerate particular classes of manycore simulations.

Keywords-simulation; computer architecture; parallel architectures; multicore processing; simulation

I. INTRODUCTION

Contemporary physical constraints, most notably the power wall, have necessitated a paradigm shift in architecture design. Although contemporary multi-core designs contain a small number of cores, it is expected that future systems may contain hundreds or even thousands of cores on a single die. In order for such systems to become a reality, the industrial and academic communities must first tackle a number of challenges, that include determining the fundamental hardware building blocks, designing efficient interconnection networks, and providing new programming models to effectively and efficiently use system resources [6]. In prototyping potential solutions to these problems, detailed time-step simulation is vital for exploring the design space of potential architectures.

During an iterative design cycle, long simulation times have been and remain to be one of the primary bottlenecks for architects [38]. The simulation of architectural designs is typically orders of magnitude slower than native execution. In uniprocessor systems, this has resulted in runtimes that are intractable for the complete simulation of many realistically sized workloads. The shift to manycore systems has only further exacerbated the problem of simulation intractability.

Numerous strategies have been proposed to reduce the simulation effort. Previous solutions include workload reduction (e.g., reduced input sets [13], statistically synthesized workloads [14], statistically sampled simulation [4], [9], [10], [11], [12], [21], [30], SimPoints [7], and benchmark subsetting [15]), optimizing simulation tasks (e.g., direct execution [22]), and parallelization of the simulator itself [2], [4]. Unfortunately, many of these proposed techniques cannot be easily applied to the simulation of shared memory multiprocessor designs. Single-application, multi-threaded workloads generally have higher degrees of inter-thread communication and inter-thread dependence, rendering many previous uniprocessor acceleration techniques ineffective. Because of this, statistical simulation, sampled simulation, and SimPoints, among others, have not been extended into the domain of multi-threaded applications and cannot be relied upon to safely reduce simulation times. Although certain accelerative techniques have been extended to multiprocessors, including SimPoints for multiprogrammed (i.e., multiple, non-interacting processes) workloads [29], and sampled simulation for throughput-oriented (i.e., multiple, non-interacting tasks) workloads [28], these acceleration techniques are not easily applied to the simulation of single-application, multi-threaded, parallel-algorithmic workloads. One exception is [27], which applies sampling to multi-threaded workloads. Unfortunately, the technique suffers from high error ("usually within 15%" [27]), and also cannot be used to estimate the execution time or speedup.

The design, verification, and maintenance of an architectural simulator are complicated tasks [25]. When the simulator is the target of parallelization, system complexity can increase significantly and introduce challenges of parallel programming debugging and performance tuning. Indeed, several contemporary manycore simulators currently execute sequentially even though they simulate parallel systems [16], [17], [18], [19]. This work presents a unique solution to parallel simulation that does not significantly increase the effort of simulator design, verification, or maintenance.

Simulator parallelization may be divided into two classes characterized by the target parallelism extracted. The first class is parallel discrete-event simulation (PDES), which parallelizes the simulator itself. Simulator tasks and state variables are decomposed into a number of parallel logical processes. Logical processes communicate via timestamped event messages when other logical processes need to be notified of a particular event. PDES techniques have been leveraged to obtain high levels of concurrency in architectural simulations [1], and are a promising method to accelerate multi-threaded

simulation. Several state-of-the-art simulation environments currently employ PDES [1], [23], [24]. PDES is completely compatible with the proposed technique in this paper. The second class of parallel simulation is time-parallel simulation, which parallelizes simulator inputs (i.e., the workloads) rather than the simulator. Time-parallel simulation separates simulation inputs into a number of temporally adjacent intervals, which are then simulated in parallel [5]. In order for time-parallel methods to obtain accurate measurements, the state-match problem must be overcome (see Section 2). Time-parallel simulations have been successfully applied to cache simulations [3], processor simulation [4][39], [38], and performance modeling [5].

This work proposes a novel time-parallel based simulation methodology to rapidly accelerate the simulation of an important class of multi-threaded workloads. We leverage the idea that barriers provide a natural, inter-thread independent point at which to split multi-threaded simulations into discrete time intervals. The proposed barrier interval simulation can also be used in conjunction with other approaches, such as PDES, to further parallelize simulation since the approaches are orthogonal and compatible. Specifically, this work makes the following contributions:

1) We quantitatively measure and define thread skew, a component of cold-start specific to multiprocessor simulation. Using the thread skew metric, we demonstrate why barriers are useful constructs that may be leveraged to accurately parallelize single-application, multi-threaded workloads.

2) Unlike prior work that focused on process-multi-programmed or independent-task, throughput-oriented workloads, our technique is the first to apply time-parallel techniques to the simulation of single-application multi-threaded, parallel-algorithmic workloads for manycore architectures.

3) Our technique achieves extremely high wall-clock speedups for multi-threaded, parallel simulations with minimal losses in simulation accuracy.

4) Speedup is the most commonly used figure of merit for parallel algorithms and parallel architectures. Our technique provides an accurate measurement of cycle counts (a stable performance metric) that can be used to calculate speedup across multiple machine configurations.

5) Our technique is the first to evaluate the effectiveness of detailed warming for single-application, multi-threaded workloads, which allows us to minimize the state match problem (Section 2).

The remainder of this paper is organized as follows: Section 2 provides a basic description of time-parallel simulation; Section 3 discusses related work and describes how barrier interval simulation avoids the obstacles presented by thread skew; Section 4 describes the barrier-interval time-parallel simulation methodology; and, Sections 5, 6, and 7 discuss the experimental methodology, results, and conclusion, respectively.

II. Time Parallel Simulation

In traditional time-parallel simulation, the time axis is decomposed into a set of non-overlapping intervals. Although intervals are not required to be homogenous in size, homogeneity benefits load balancing and improves parallel speedup. Computation then consists of two phases: first, the initial phase simulates each interval with a speculative initial state (thus performance measurements obtained from the initial phase may be inaccurate); and, the second phase, or the fix-up computation phase, iteratively re-simulates each of the intervals. Subsequent fix-up iterations continue until an interval's initial state matches that of the predecessor's final state (i.e., the state-matching problem [3]).

This paper presents a framework based upon time-parallel simulation to speedup the simulation of single-application, multi-threaded workloads. Unlike traditional time-parallel simulation, we remove the iterative fix-up computation phase (which may limit wall-clock speedups), and instead use a warm-up based approach to approximate system state. As in time-parallel simulation, the proposed technique parallelizes the input workload. This work is based on the following intuition: that barriers provide a natural segmentation point to parallelize a workload.

III. The Circular Dependence Dilemma of Parallel Workload Simulation

Many strategies for accelerating simulation are only applicable to single-threaded applications. Identification of representative simulation points [7], benchmark subsetting [15], statistically sampled approaches [4], [9], [10], [11], [12], [21], reduced workload input sets and loop counts [13], and statistically synthesized benchmarks [14], have all been used with great success in the simulation of uniprocessor designs. However, multiprocessor systems exhibit a circular dependence dilemma, explained below, that introduce new challenges that must be overcome to accurately and effectively accelerate their simulation.

In multiprocessor systems, performance is a combination of individual thread executions, which depend upon system state. Thread interactions occur implicitly through shared resources (e.g., a shared Last Level Cache) or explicitly through synchronization constructs. Race conditions due to resource locking may not be predictably modeled unless detailed state information regarding cache contents, system coherence state, core proximity to the home node, network contention, etc., are known. For example, consider the common practice of skipping initialization code at the beginning of a workload, which leaves the system in a cold state at detailed simulation startup. For uniprocessor systems, solutions to the cold-start problem have been extensively studied and mitigated [8], [9], [12], [21]. In multiprocessor systems, previously studied solutions are limited to fast-forwarding over serial code regions. If fast-forwarding terminates in a region of parallel thread executions, not only is system state unknown, but the relative thread progression and thread interleavings are unknown as well. Effectively compensating for cold-start involves reconstructing system state, and requires precise knowledge of each individual thread's progress. But, the reconstructions of each thread's progress requires knowledge of system state to determine, for instance, the order that threads acquire and release critical sections. The approximation of system state, therefore, is dependent upon individual thread progressions, and the approximation of thread progressions are dependent upon system state, resulting in a circular dependence dilemma.

For each measurement, C_c:

1) After functional fast-forwarding:

- *Record* sys_fetch_C *and all* t_i *where,*

$t_{C,i} = fetch\ count\ of\ thread\ i$

$N = \#\ \text{threads}$

$sys\_fetch_C = \sum_{i=1}^{N} t_{C,i}$

2) From the full simulation (no fast-forwarding)

$t'_i = fetch\ count\ of\ thread\ i$

$sys\_fetch'_C = \sum_{i=1}^{N} t'_i$

when (sys_fetch_C' == sys_fetch_C):

$t'_{C,i} = t'_i\ \forall i$

3) Calculate thread skew from profiled data

$thread\_skew_{C,i} = t_{C,i} - t'_{C,i}$

When $sys\_fetch'_C = sys\_fetch_C$

$\sum_{i=1}^{N} t'_{C,i} = \sum_{i=1}^{N} t_{C,i}$

$\sum_{i=1}^{N} t_{C,i} - \sum_{i=1}^{N} t'_{C,i} = 0$

$\sum_{i=1}^{N} (t_{C,i} - t'_{C,i}) = 0$

$\sum_{i=1}^{N} thread\_skew_{C,i} = 0$

Furthermore, at barrier releases:

$thread\_skew_i = 0$, for all i

Figure 1. Thread skew is calculated using aggregate system and per-thread fetch counts. Simulations with functional fast-forwarding record fetch counts for all threads at the beginning of a simulation. Full simulations use these counts to determine when fetch counts are recorded. Since total system fetch counts are identical in the fast-forwarded and full simulations, the sum of thread skew for every measurement must be zero. Individual threads may lead or lag their counterpart in the full simulation.

In order to measure thread divergence quantitatively, and thus the impact of the circular dependence dilemma, we introduce *thread skew*. *Thread Skew* measures the divergence of thread progressions between two simulations: one simulation that uses functional fast-forwarding (where thread divergence is introduced through imprecise skipping), and another that performs full-simulation from the beginning of the program. A formal definition of thread skew is shown in Figure 1. Skew values are measured at the beginning of various program locations. For each location with imprecise skipping, the fetch counts[1] of all threads are summed to obtain a total system fetch count. Full simulations are performed to profile the fetch counts of all threads when the system observes the same system fetch count. The use of total fetched instructions provides a system-wide estimator of progress that is used to map divergent executions between the two simulations.

Thread skew is shown graphically in Figure 2 for *ocean contiguous* executing with 16 cores and for *lu contiguous* executing with 256 cores. Comparing thread progressions at a constant system fetch count causes skew values for all threads to sum to zero, since for every thread that leads true execution, another must lag. Threads leading true execution have positive thread skew, and those lagging have negative thread skew. Barriers cause thread skew of all threads to collapse to zero. This leads to an important observation: the circular dependence dilemma can be avoided by parallelizing the simulation at barrier events.

IV. Barrier Interval Simulation

Barriers are an important, and commonly used synchronization construct found in many parallel algorithm implementations. They are found within the SPLASH-2, PARSEC [32], SpecOMP, and NAS parallel benchmark suites, among others. The popularity of barrier based programs stems directly from the popular parallel programming paradigms. Directive-based languages, such as OpenMP, implicitly define barriers at parallel loop constructs. Barriers are also present in fork/join models of parallelism (e.g., CUDA [36]). What's more, they are used in next-generation programming language constructs such as Cilk's *synch* operation [33] and X10's *finish* [34] operation. Barriers are of particular importance within scientific applications, since many coarse-grained parallel programs execute in phases separated by barriers [37]. Others, such as Liu, et al. [31] leverage barriers to, for example, conserve power in CMP systems, whereas our work exploits barriers to accelerate architectural simulation.

The proposed barrier-interval simulation methodology is illustrated in Figure 3. First, the input workload is instrumented to identify, at runtime, barrier release events to define discrete time intervals for parallelization. Barrier release events are triggered following the last thread's arrival at a barrier, when all threads are allowed to continue execution. Each workload, comprising a parallel algorithm, is functionally executed to completion to determine the number of emulated instructions before each barrier release. These functional instruction counts provide the functional fast-forwarding values necessary to begin each simulation at the appropriate barrier release event. The functional profiling of barrier interval locations is necessary only once per workload and core count, irrespective of changes to the detailed simulator. Every interval is then simulated in parallel with a specified warm-up length. If a warm-up of W instructions were desired before an interval occurring at instruction I, fast-forwarding would be performed for I-W instructions. Detailed warming simulation continues until the first barrier release, where simulator statistics are reset. Execution of the interval then commences until the subsequent barrier release, which terminates the interval.

The extensions necessary for a sequential simulator to support barrier-interval simulation are outlined below. In addition to functional fast-forwarding, the simulator must be notified of barrier release events to clear system statistics and precisely terminate intervals. The clearing of system statistics

[1] The fetch counts used in the calculation of the thread skew metric exclude instructions that occur within thread synchronization functions.

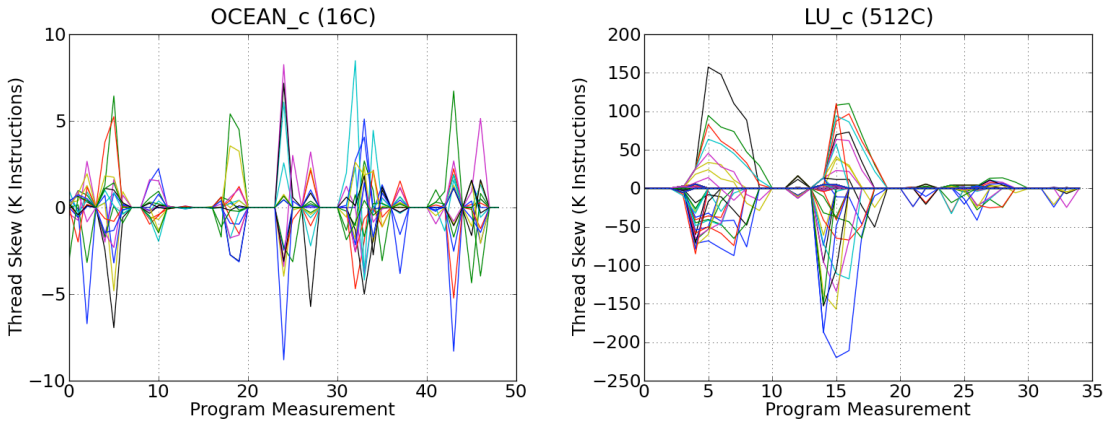

Figure 2. An illustration of thread skew. This is a time sequence showing the difference of thread progressions between various program measurements with imprecise fast-forwarding and the full-simulation. Barriers cause thread skew to collapse to zero, and may be exploited to accurately parallelize the target workload.

is present in many simulators since many studies include a detailed warming period after the functional skipping of initialization code. Warm-up can be applied either before or after an interval's starting point. However, if detailed warm-up consumes instructions after an interval's starting point, then errors associated with accumulative metrics such as cycle counts grow proportionally with the amount of warm-up. Although increased warm-up prior to the starting point generally improves accuracy, it does so at the expense of speedup since extra work is introduced into the simulation effort by overlapping particular instruction streams (i.e., from two or more barrier-intervals).

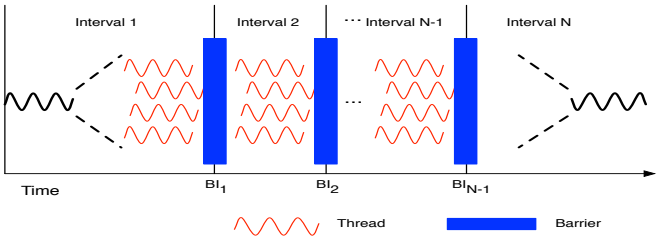

Figure 3. An illustration of simulation parallelization via the barrier-interval simulation method. A target workload is divided into intervals delineated by barrier releases, all of which are then simulated in parallel.

Barrier release events are also necessary to precisely simulate the targeted barrier-interval boundaries. Profiled interval boundaries are imprecise since they are not guaranteed to be exact locators in the instruction stream, unless fast-forwarding is performed for all previous instructions (thus reproducing the profiled thread schedule). Simulating instructions in full cycle-accurate detail can cause divergent thread behaviors within synchronization events, such as the number of times a thread spins in a test-and-set operation waiting to acquire a lock. Thus, potential divergent thread behaviors create unknown interval boundaries, which may only be identified at runtime.

The methodology employed by barrier-interval time-parallel simulation eliminates thread skew, since the simulated intervals are guaranteed to be at boundaries where thread progressions are known (e.g., convergence points in Figure 2). By applying detailed warm-up heuristics adopted from sampled simulation, cache state and coherence information may be reconstructed to obtain highly accurate measurements over the defined intervals, producing measurements that closely resemble those of sequential simulation. Measurements obtained from individual intervals can then be aggregated to form estimated system metrics of the simulated program. For accumulative metrics, such as simulated runtime, individual measurements can simply be summed. For rate-based metrics, system metrics can be formed through the appropriate means (e.g., harmonic, arithmetic, geometric).

V. Experimental Methodology

Experiments in this study were conducted using the Manifold shared-memory manycore simulator, which is part of a larger, multi-agency-funded simulation framework being developed by the authors and other collaborators. The simulator is execution-driven, using the SESC front-end framework to perform functional emulation of RISC instructions, and to provide input instructions to the detailed simulator back-end. During SESC functional emulation, threads are assigned instructions in a two-dimensional queue based upon the thread ID. During fast-forwarding, each thread is emulated by a constant number of instructions in a round-robin fashion. The detailed back-end consists of a number of architectural nodes, each containing a processor, a private L1 cache, a distributed, shared L2 cache-slice, and a network interface. The system implements a directory-based MESI coherence protocol. Nodes are connected via a network-on-chip incorporating a mesh topology that implements wormhole routing. Table 1 shows a summary of the simulation parameters. Experimental workloads consist of SPLASH-2 benchmarks cross-compiled to the target ISA using the GNU C compiler (gcc) version 4.2.2.

Evaluation of the barrier-interval simulation approach was performed on the following SPLASH-2 workloads: *lu contiguous*, *ocean contiguous*, *radix*, *fft*, and *water spatial*. Each workload was simulated by varying the number of cores between 1 and 512, resulting in 10 distinct simulations for each workload. For each (core count, workload) pairing, multiple detailed warming lengths were applied: none, 10k, 100k, 1M, and 10M pre-interval instructions. Although implementing

fast-functional warming [12], instead of detailed warming, might produce further speedups, its use is reserved for future work. For the workloads evaluated, 181,000 simulations were performed to evaluate the trade-offs of the proposed technique in terms of speed and accuracy.

TABLE 1. ARCHITECTURE PARAMETERS OF THE SIMULATED SYSTEM

# Cores	1, 2, 4, 8, 16, 32, 64,128, 256, 512	Coherence / Tracking	Directory-based MESI Protocol w/ Full Presence Bits
Core Model	2-issue in-order 2 MSHRs	NOC Topology	Mesh 4-node express links
Per-node L1 Cache	32 KB set associative 4-way (WBWA) 2-cycle hit latency	NOC Router Architecture	3-stage pipeline 4 VCs / connection 2 buffers / VC
Per-node L2 Shared Last-level Cache	256KB set associative 8-way (WBWA) 8-cycle hit latency	Cache line size	64B
		Cache replacement	LRU
		Main Memory Latency	200 cycles
System L2 size	# Cores * 256KB		

VI. RESULTS

A. Parallel Simulation Accuracy

The accuracy of interval estimates are dependent upon overcoming cold-start effects. For multi-threaded simulation, cold-start components consist of thread skew, unknown cache, network, and directory state. Through the use of detailed warming, error components associated with unknown cache state and network state are sufficiently reduced. Error results collected for individual (core count, workload) pairs for the tested warm-up lengths, and their summaries, are shown in Figure 4. Error summaries are obtained by calculating the harmonic mean of error percentages for each warm-up length. Cycle counts of the barrier-intervals are summed for the parallel simulations, and then compared to the sequential simulation using absolute relative error. On average, the error rates of the five warm-up lengths are 0.81%, 0.79%, 0.62%, 0.09%, and 0.01% for none, 10k, 100k, 1M, and 10M, respectively. This demonstrates that, if cold-start effects associated with thread skew are sufficiently reduced, then cache state, network state and cache coherence information of multi-threaded workloads may be accurately approximated through the application of warm-up methods.

Larger warm-ups intuitively, and often empirically, lead to increased accuracy for interval measurements. However, certain data points, such as *lu contiguous* for 512 cores, observe higher error when a warm-up of 10k instructions is used vs. no warm-up. Error rates occasionally increase with more warm-up, but eventually converge to their expected values once sufficient warm-up is performed. One reason for this effect involves the incorrect partial warming of the caches and the on-chip network. Even though system statistics are cleared at the start of an interval, network packets generated from cache misses are still in-flight when the new interval begins. In general, this is desirable for reducing cold-start effects. However, in some cases high network contention caused by detailed warming can affect cache request latencies at the beginning of the interval. For example, no warm-up results in a cold network without any contention. Increasing warm-up to 10k-instructions can create a large burst of cache accesses, resulting in miss-traffic and corresponding network contention that spills into the interval execution. If warm-up is increased to 100k- instructions, however, the accesses in the shorter 10k-instruction warm-up reveal themselves to actually be cache hits due to earlier accesses in the larger 100k-instruction window. As a result, correct network contention is achieved with both the lowest and highest warm-up lengths, whereas the mid-range warm-up length creates additional bias from incorrect miss-traffic on the network. The important observation is that larger warm-ups are not always guaranteed to increase accuracy, and can even introduce additional bias.

The effect that initial state has upon measurement error is also impacted by individual thread performance. Performance is measured as the speedup relative to a single core machine. As cores are added to the simulated machine, the performance of a multi-threaded workload increases until a saturation point. Once the saturation point is reached, the addition of cores to the simulated machine begins to erode performance gains due to the increased traffic and overheads associated with thread synchronization. For the SPLASH-2 workloads, computation is divided among all the available cores. The overheads to obtain work eventually dominate useful computation, and result in system slowdown. Thus, computation performed by threads after saturation becomes increasingly non-useful. For ocean contiguous, the point of saturation occurs at eight cores, and has the highest error rate of all experiments. Increasing the number of cores past saturation causes long chains of requests to form, where each thread must wait to access semaphores. As more threads are queued waiting to receive work, the relative importance of warm-up towards measurement accuracy diminishes.

B. Error Rates vs. Interval Size

In the single-threaded domain where sampling is viable, a common metric is the relationship between sampled measurements and error rates [20]. If we consider a barrier interval to be a sample of the full execution we can perform a similar study. Past work in the single-threaded domain found an inverse relationship between an interval's size and the measured error rates when no warm-up has been applied. The intuition behind this trend is that cold-start effects are amortized across the interval. The larger the interval, the less impact that cold-start has upon measurement error. Therefore, measurements obtained from small intervals may not be reliable if warm-up is not incorporated.

The relationship between barrier interval sizes and associated error rates for the barrier-interval simulation of multi-threaded parallel workloads is also explored. To determine if the single-threaded trend between interval size and error holds for barrier intervals, we show the average normalized interval sizes (measured in cycles) as the number of cores increases. Measurements are normalized such that the core count with the largest interval size is assigned a value of one. All experiments incorporate no warm-up. As shown in Figure 5a, the interval sizes vary dramatically. In all tested workloads increasing the number of cores causes interval sizes to follow a parabolic shape, where the average size decreases to

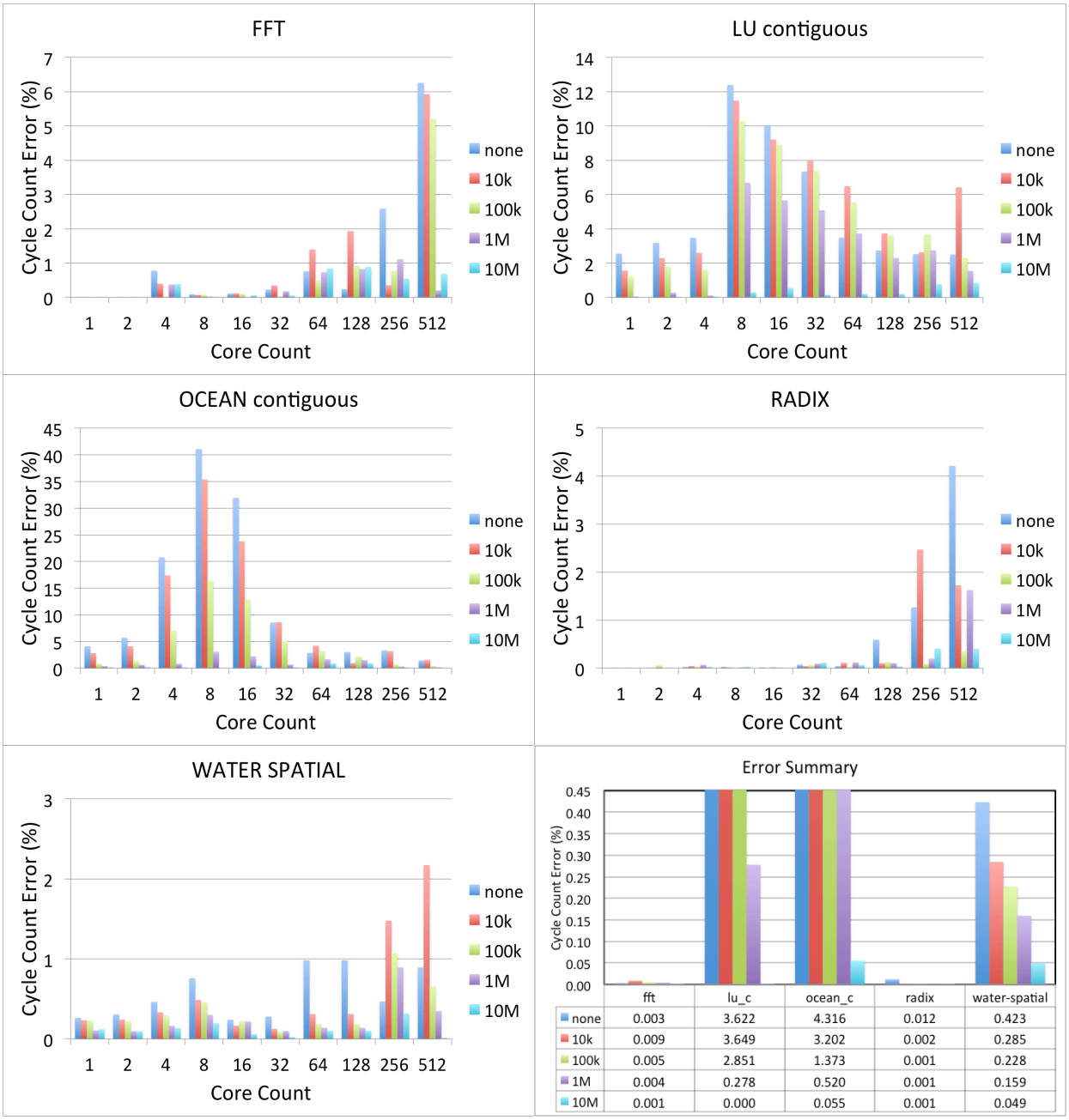

Figure 4. Accuracy measurements of barrier-interval time-parallel simulation. Absolute relative errors are computed for the differences in simulated cycles times between the parallel and sequential simulations

a minima before eventually increasing. The intuition behind these results is also related to the per-thread performance. Prior to saturation, additional threads cause more work to be performed in parallel, resulting in higher system performance, and a reduction in average interval sizes. After saturation, thread overheads causes additional threads to cause performance degradation of all threads, and result in larger interval sizes. This is interesting since even without warm-up, where measurements may be the most suspect, the saturation point is correctly identified for all tested workloads. Comparisons with baseline experiments confirm that saturation occurs for all of the workloads at the smallest interval size. Saturation for *fft* occurs at 64 cores, *lu contiguous* at 16 cores, *radix* at 256 cores, *ocean contiguous* at 8 cores, and *water spatial* at 8 cores. Similar speedup limitations have been observed for SPLASH-2 in the past (see, e.g., [26]).

If multi-threaded simulations exhibit a similar relationship to interval size and error rates as single-threaded workloads, then it would have been expected that experiments containing the highest interval sizes would have the lowest interval errors, and vice versa. This was not the case, and is explained by the central limit theorem (CLT) of statistics. Average error rates for interval measurements for all workloads without warm-up are shown in Figure 5b. Error rates are higher in this graph than in Figure 4 since the errors are based upon per-interval measurements rather than cumulative statistics. Even without

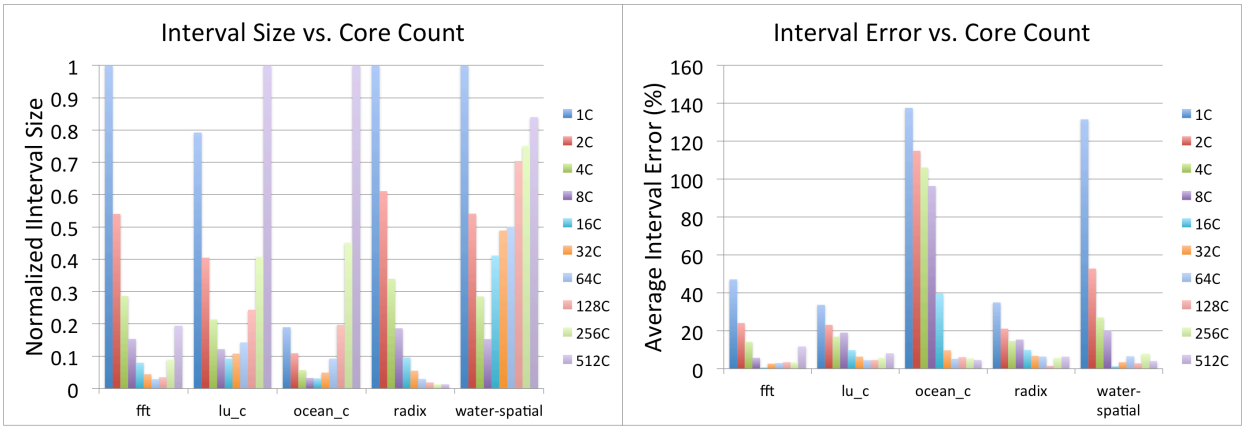

Figure 5. (a) Average normalized interval size and (b) average interval error as the number of cores increase.

any warm-up, increasing the number of cores causes interval errors to drop.

The distribution of interval errors with no warm-up at varying core counts for *ocean contiguous* is shown in Figure 5. Due to space constraints, only this workload is shown; however, other workloads exhibit similar behaviors. At one, two, and four cores, the distributions of errors closely follow the inverse relationship of error rates and interval sizes found in single-threaded sampling. Prior to saturation, as the number of cores increased, interval measurements begin forming clusters in the error space. These clusters of measurements decrease in size until the point of saturation, and then increase in size as the intervals become larger. At the same time, maximum interval error rates decrease due to CLT effects. The CLT dictates that the distribution of an average appears to be normal, even if the underlying distribution from which samples are taken is decidedly non-normal. The performance of individual threads may be considered as forming a distribution from which overall system performance is determined. Thus, overall system error becomes a function of component errors of the individual threads, which tends towards lower error as the number of threads increases (shown in Figure 5b and Figure 6).

C. *Parallel Simulation Speedup*

For these experiments, wall-clock speedup values were calculated from repeated measurements of the sequential and time-parallel workloads. Simulations were performed on identical Intel Xeon X5450 (12MB L2Cache, 3.00 GHz, 1,333 MHz FSB) machines, with 16GB of physical memory. Since distribution outliers have large effects upon the arithmetic mean, wall-clock speedups were calculated as the ratio of median values for both the sequential and parallel simulations. Wall-clock speedup results for the five workloads are shown in Figure 7. Although increasing warm-up generally improves the accuracy of interval measurements, it does so at the expense of speedup.

Measured speedups for each workload at each core count generally fluctuate as the interval size and interval homogeneity vary. Since each workload inherently contains a different number of barriers, expected speedups may differ significantly from one workload to another. Table 2 shows the number of barriers contained within the simulated workloads, along with the maximum obtained speedup. The computed relative efficiency is the ratio of obtained speedup to the maximum theoretical speedup. Coefficient of Variation (CV) values were computed for each workload, which is the ratio of the standard deviation, σ, to the absolute value of the mean, |μ|. As shown, there is a strong correlation between calculated CV values and relative efficiency. Lower CV values result in higher relative efficiencies. Minimum and maximum speedup values are taken from runtimes across all warm-up lengths. Even at the largest warm-up lengths, no simulation experienced slowdown over its sequential simulation.

The barrier-interval simulation methodology improves simulation times dramatically compared to their sequential

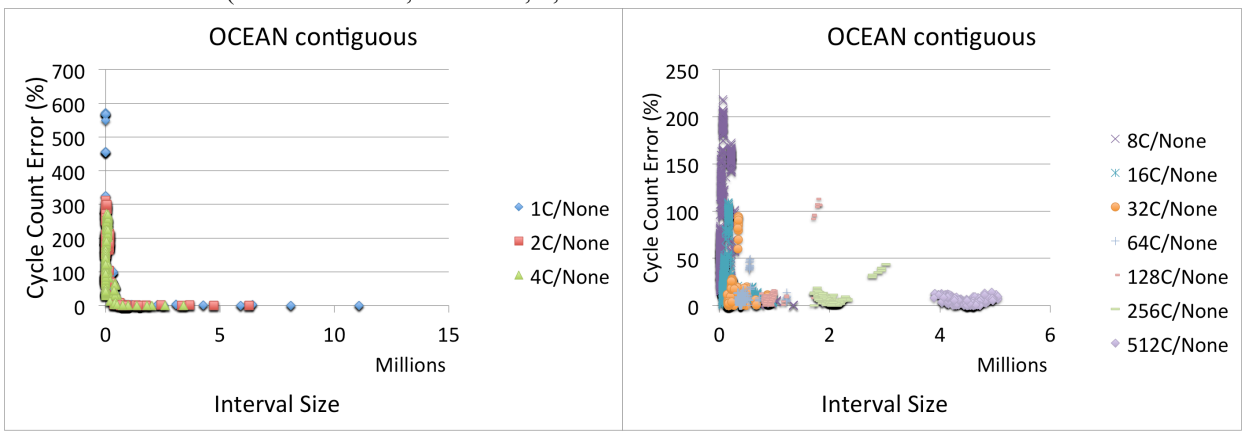

Figure 6. Distribution of interval errors for *ocean contiguous* as a function of the core count. No warm-up is applied before interval measurements.

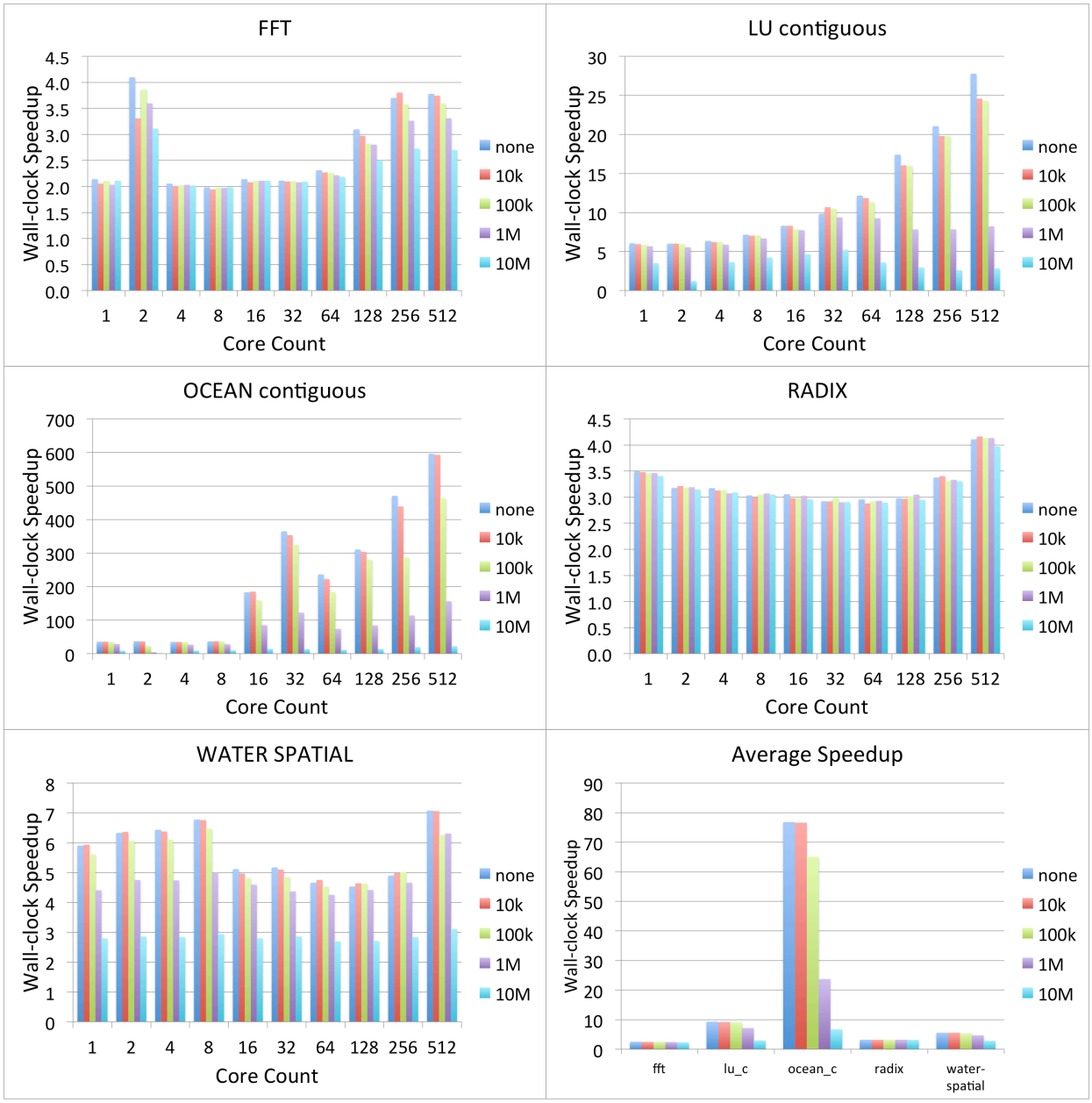

Figure 7. Wall-clock simulation speedup measurements of barrier-interval time-parallel simulation. Speedup is relative to the sequential simulation environment.

simulation. On average, detailed warming using none, 10k, 100k, 1M, and 10M instruction lengths had speedups of 20.13x, 19.95x, 17.56x, 8.32x, and 3.70x, respectively. The smallest speedup of 1.22x was obtained for lu contiguous 10M instruction warm-up for 2 cores. The highest speedup of 596x was obtained for ocean contiguous with no warm-up for 512 cores.

Since there are no dependencies between barrier intervals, all intervals may be simulated in parallel. Thus, the potential simulation speedup is determined by two factors: (1) the number of barriers, and therefore barrier intervals, that are contained in the workload; and, (2) the homogeneity of barrier-interval sizes. The more barriers there are in the workload, the greater the opportunity for parallelization. However, since parallelization speedup is dominated by the slowest executing interval, it is also beneficial if intervals are approximately equivalent in size. The artificial introduction of additional barriers into the workload is a possible technique that could improve the parallelization effort, however it must first be proven that additional barriers do not change fundamental properties of the simulation (both in terms of runtime characteristics and correctness), and this is a topic reserved for future research. Barrier intervals could also be melded to achieve heterogeneously sized intervals, but this too is left for future research.

As shown in Section 6.2, barrier-interval sizes vary dramatically with the number of threads. Interval size homogeneity was measured using the coefficient of variation, which is a normalized measure of dispersion for a distribution and allows CV values to be compared across different

TABLE 2. RELATIVE SPEEDUP EFFICIENCY VS. COEFFICIENT OF VARIATION.

Workload	Barriers	Min Speedup	Max Speedup	Rel. Efficiency	CV
fft	5	1.94x	4.10x	82%	0.3939
lu contiguous	33	1.22x	27.78x	84%	0.1025
ocean	654	1.29x	596.04x	91%	0.0564
radix	13	2.88x	4.16x	32%	0.6953
water spatial	18	2.69x	7.08x	39%	0.7070

distributions. Distributions with CV values greater than one are considered high-variance, and those below one are considered low-variance. For each experiment, the CV is calculated using the interval sizes measured in cycles. As expected, lower CV values correspond to higher speedups. For example, for lu contiguous: 512 cores has a CV of 0.10 with a speedup of 27.8x; and, 2 cores has a CV of 1.45 with a speedup of 6x. CV values exhibit an inverse relationship with observed speedup for all tested workloads. Interestingly, CV values for all workloads are the smallest at the highest core counts where interval homogeneity is improved, despite increased interval size caused by thread saturation.

Larger warm-up generally results in increased accuracy, but rapidly diminishes speedup opportunities for certain workloads. Workloads with fewer barriers (i.e., *fft*, *radix*, and *water-spatial*) are more robust towards speedup losses, and can incorporate larger warm-ups without significant penalties in performance. Since speedup losses are more prevalent in workloads containing high numbers of barriers, an analysis of *lu contiguous* and *ocean contiguous* was performed to show the speedups lost due to increased warm-up, and are shown in Figure 8. The normalized speedup loss refers to the percentage of speedup (relative to no warm-up) that was eroded by increased warm-up. Since error rates significantly differed for these workloads at the various core counts, error and speedup values are classified into two groups: 1 to 16 cores (Figure 8a) and 32 to 512 processors (Figure 8b). Although higher core counts generally exhibit lower error rates even in the absence of warm-up, certain outliers exhibited non-negligible error rates (see, FFT at 512 cores). Thus, a conservative estimation of the necessary warm-up to obtain extremely high levels of accuracy results in a recommendation of 1M pre-interval instructions. At this warm-up length, the maximum error rate for all tested workloads was 6.7%, with an average error rate of 0.09%. Although a warm-up of 1M instructions diminishes attainable speedup between 28% and 41%, the actual performance loses are not as severe if a limited context environment is assumed.

VII. CONCLUSION

In this study, a novel simulation acceleration strategy was presented to rapidly simulate certain important classes of multi-threaded, parallel-algorithm, applications with minimal losses in accuracy. The strategy can be readily implemented by architects to obtain good speedups, at low cost. Using time-parallel barrier-interval simulation, wall-clock runtimes of a number of SPLASH-2 simulations were sped up by 13.94x on average, with a maximum speedup of 596x. These speedups were obtained using a technique that can be incorporated into a number of simulation environments, including PDES based approaches. By exploiting barriers, challenges associated with the circular dependence dilemma (Section 3) that currently hinder the applicability of other uniprocessor accelerative techniques are avoided. Additionally, we investigated the relationship between error rates associated with state-loss obtained from interval measurements in a multi-threaded context, which may be applied towards other time-parallel or even sampled simulation domains. Our results showed that for parallel workloads with barriers, dramatic simulator performance gains are possible, thus shortening the design process and enabling larger workloads and input sets to be simulated efficiently.

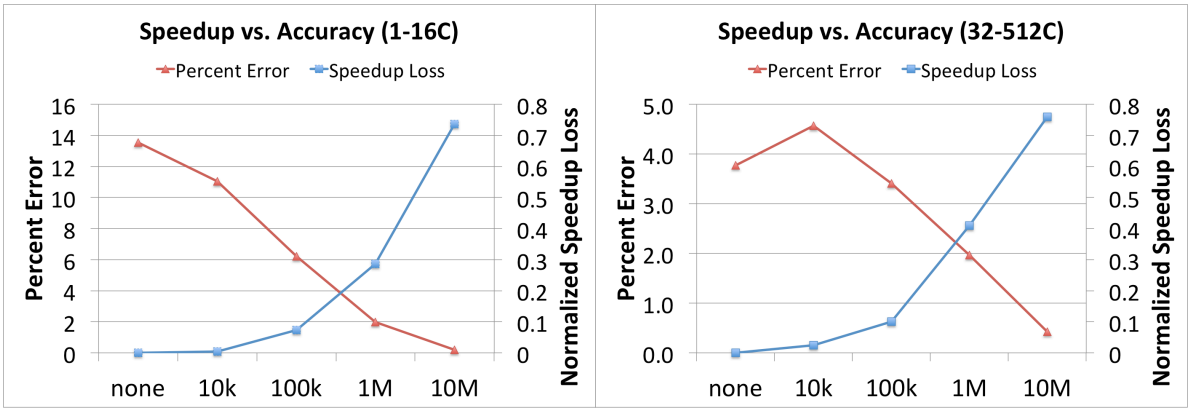

Figure 8. A comparison of accuracy and speedup for *lu contiguous* and *ocean contiguous*. Error rates are shown along with normalized speedup losses as warm-up lengths increase for 1 to 16 cores (a), and 32 to 512 cores (b).

REFERENCES

[1] J. E. Miller, et al. "*Graphite: A Distributed Parallel Simulator for Multicores*," in the International Symposium on High Performance Computer Architecture, 2010.

[2] R. M. Fujimoto. "*Parallel Discrete Event Simulation*," in Conference on Winter Simulation*, 1989*.

[3] T. Kiesling. "*Approximate Time-parallel Cache Simulation*," in Conference on Winter Simulation, *2004*.

[4] G. Lauterbach. "*Accelerating Architectural Simulation by Parallel Execution of Trace Samples*," Hawaii International Conference on System Sciences, 1994.

[5] T. Kiesling and S. Pohl. "*Time-Parallel Simulation with Approximative State Matching*," Workshop on Parallel and Distributed Simulation, 2004.

[6] K. Asanovic, et al. "*The Landscape of Parallel Computing Research: A View from Berkeley*," University of California, Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2006-183. Dec. 18, 2006.

[7] E. Perelman, G. Hamerly, and B. Calder. "*Picking Statistically Valid and Early Simulation Points*," in the International Symposium on Parallel Architecture and Compilation Techniques*, 2003*.

[8] R. E. Wunderlich, et al. "*Statistical Sampling of Microarchitecture Simulation*," ACM Transactions on Modeling and Computer Simulation, vol. 16, pp. 197-224, 2006.

[9] P. D. Bryan, M. C. Rosier, and T. M. Conte. "*Reverse State Reconstruction for Sampled Microarchitectural Simulation*," in the International Symposium on Performance Analysis of Systems and Software, 2007.

[10] P. D. Bryan and T. M. Conte. "*Combining Cluster Sampling with Single Pass Methods for Efficient Sampling Regimen Design*," in the International Conference on Computer Design, 2007.

[11] L. V. Ertvelde, et al. "*NSL-BLRL: Efficient Cache Warmup for Sampled Processor Simulation*," in the Annual Symposium on Simulation, 2006.

[12] R. E. Wunderlich, et al. "*SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling*," in the International Symposium on Computer Architecture, 2003.

[13] A. J. KleinOsowski, et al. "Adapting the SPEC 2000 Benchmark Suite for Simulation-based Computer Architecture Research," *Workload Characterization of Emerging Computer Applications*: Kluwer Academic Publishers, 2001, pp. 83-100.

[14] J. Ajay, et al. "*Distilling the Essence of Proprietary Workloads into Miniature Benchmarks*," *ACM* Transactions on Architecture and Code Optimization. vol. 5, pp. 1-33, 2008.

[15] A. Joshi, et al. "*Measuring Benchmark Similarity Using Inherent Program Characteristics*," IEEE Transactions on Computers. vol. 55, pp. 769-782, 2006.

[16] T. Austin, E. Larson, and D. Ernst. "*SimpleScalar: An Infrastructure for Computer System Modeling*," IEEE Computer, vol. 35, no. 2, pp. 59–67, 2002.

[17] M. Rosenblum, et al. "*Complete Computer System Simulation: The SimOS approach*," IEEE Parallel and Distributed Technology: Systems and Applications, vol. 3, no. 4, pp. 34– 43, 1995.

[18] P. Magnusson, et al. "*Simics: A full System Simulation Platform*," IEEE Computer, vol. 35, no. 2, pp. 50–58, Feb 2002.

[19] F. Bellard. "*QEMU, A Fast and Portable Dynamic Translator*," USENIX Annual Technical Conference, 2005.

[20] T. M. Conte, M. A. Hirsch, and K. N. Menezes. "*Reducing State Loss for Effective Trace Sampling of Superscalar Processors*," in the International Conference on Computer Design, 1996.

[21] L. Eeckhout, et al. "*BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation*," The Computer Journal., vol. 48, pp. 451-459, 2005.

[22] R. C. Covington, et al. "*The Rice Parallel Processing Testbed*," in Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 4–11, May 1988.

[23] S. S. Mukherjee, et al. "*Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator*," in the Workshop on Performance Analysis and its Impact on Design, 1997.

[24] H. Lv, et al. "P-GAS: Parallelizing a Cycle-Accurate Event-Driven Many-Core Processor Simulator Using Parallel Discrete Event Simulation," in the Workshop on Principle of Advanced and Distributed Simulation, 2010.

[25] B. Black and J. P. Shen. "*Calibration of Microprocessor Performance Models*," in Computer, 31(5), 59-65. 1988.

[26] A. Chauhan, C. Ding, and B. Sheraw. "*Scalability and Data Placement on SGI Origin*". Tech Rep. TR98-305, 1998.

[27] K. C. Barr, et al. "*Accelerating Multiprocessor Simulation with a Memory Timestamp Record*," in the International Symposium on Performance Analysis of Systems and Software, 2005.

[28] T. F. Wenisch, et al. "*SimFlex: Statistical Sampling of Computer System Simulation*," IEEE Micro, Vol 26, Issue 4, July-Aug. 2006.

[29] M. Van Biesbrouck, T. Sherwood, and B. Calder. "*A Co-phase Matrix to Guide Simultaneous Multithreading Simulation*," in the International Symposium on Performance Analysis of Systems and Software, 2004.

[30] T. F. Wenisch, et al. "*TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes*," in the International Conference on Measurement and Modeling of Computer Systems, 2005.

[31] C. Liu, et al. "*Exploiting Barriers to Optimize Power Consumption of CMPs*," in the International Symposium on Parallel and Distributed Processing, 2005.

[32] C. Bienia, et al. "*The PARSEC Benchmark Suite: Characterization and Architectural Implications*," in the International Conference on Parallel Architecture and Compiler Techniques*, 2008.*

[33] Cilk-5.2 Reference Manual. Available at: `http://supertech.lcs.mit.edu/cilk`.

[34] X10 release on SourceForge. Available at: `http://x10.sf.net`

[35] A. R. Alameldeen, and D. A. Wood. "*Addressing Workload Variability in Architectural Simulations*," Micro, IEEE, 23(6), 94-98, 2003.

[36] NVIDIA CUDA, "Compute Unified Device Architecture," `http://developer.nvidia.com/object/cuda.html`.

[37] T. E. Jeremiassen and S. J. Eggers. "*Static Analysis of Barrier Synchronization in Explicitly Parallel Programs*," in the International Conference on Performance Analysis and Compilation Techniques, 1994.

[38] S. Girbal, et al. "DiST: A Simple, Reliable and Scalable Method to Significantly Reduce Processor Architecture Simulation Time," in ACM SIGMETRICS, 2003.

[39] Rico, A.; Duran, A.; Cabarcas, F.; Etsion, Y.; Ramirez, A.; Valero, M.; , "Trace-driven simulation of multithreaded applications," Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on , vol., no., pp.87-96, 10-12 April 2011