ParTejas: A Parallel Simulator for Multicore Processors

GEETIKA MALHOTRA, RAJSHEKAR KALAYAPPAN, SEEP GOEL, POOJA AGGARWAL, ABHISHEK SAGAR, and SMRUTI R. SARANGI, Indian Institute of Technology Delhi

In this article, we present the design of a novel parallel architecture simulator called *ParTejas*. *ParTejas* is a timing simulation engine that gets its execution traces from instrumented binaries using a fast shared-memory-based mechanism. Subsequently, the waiting threads simulate the execution of multiple pipelines and an elaborate memory system with support for multilevel coherent caches. *ParTejas* is written in Java and primarily derives its speedups from the use of novel data structures. Specifically, it uses lock-free slot schedulers to design an entity called a *parallel port* that effectively models the contention at shared resources in the CPU and memory system. Parallel ports remove the need for fine-grained synchronization and allow each thread to use its local clock. Unlike conventional simulators that use barriers for synchronization at epoch boundaries, we use a sophisticated type of barrier, known as a phaser. A *phaser* allows threads to perform additional work without waiting for other threads to arrive at the barrier. Additionally, we use a host of Java-specific optimizations and use profiling to effectively schedule the threads. With all our optimizations, we demonstrate a speedup of 11.8× for a multi-issue in-order pipeline and 10.9× for an out-of-order pipeline with 64 threads, for a suite of seven Splash2 and Parsec benchmarks. The simulation error is limited to 2% to 4% as compared to strictly sequential simulation.

CCS Concepts: • **Computing methodologies → Discrete-event simulation**; **Distributed simulation**; **Simulation tools**; *Systems theory*;

Additional Key Words and Phrases: Parallel simulation, architectural simulator, Tejas, ParTejas, phasers, parallel ports, slot scheduling

ACM Reference format:
Geetika Malhotra, Rajshekar Kalayappan, Seep Goel, Pooja Aggarwal, Abhishek Sagar, and Smruti R. Sarangi. 2017. ParTejas: A parallel simulator for multicore processors. *ACM Trans. Model. Comput. Simul.* 27, 3, Article 19 (July 2017), 24 pages.
http://dx.doi.org/10.1145/3077582

1 INTRODUCTION

An architectural simulator is arguably the most important tool that a computer architect uses to design and evaluate novel processor architectures. Simulators such as SimpleScalar [4], Sesc [23], M5 [6], GEMS [17], and Multi2Sim [26] are widely used by computer architects in both industry and academia. Along with their use in industrial settings, they are also heavily used by instructors

This article is an extension of a poster presented at ISPASS 2014 with the same title [16].

Authors' addresses: G. Malhotra, R. Kalayappan, S. Goel, P. Aggarwal, A. Sagar, and S. R. Sarangi, Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi - 110016. India; emails: geetika.mlhtra@gmail.com, rajshekark@cse.iitd.ac.in, seep.goyal@gmail.com, csz112569@cse.iitd.ac.in, sagar23jan@gmail.com, srsarangi@cse.iitd.ac.in.

to teach courses on computer architecture. Consequently, it is necessary to design efficient architectural simulators that capture the needs of the computer architecture community.

As compared to detailed RTL-level simulators, architectural simulators use a fairly high-level description of a processor and are consequently at least two orders of magnitude faster. The price of this additional speed is accuracy. Architectural simulators are primarily used to estimate broad high-level trends and are typically not used for getting extremely fine-grained performance data. Also, as compared to other simulation technologies such as FPGA emulators, architectural simulators are much easier to design, and it is possible to create prototypes of futuristic architectures very quickly.

Architectural simulators were traditionally sequential in nature. This is because they were up till now being used to simulate multicore processors with at most four to 16 cores. The requirements of simulators are changing, however. There are predictions that in the near future we shall have hundreds and thousands of cores per chip. In line with this trend, researchers have begun proposing architectural techniques aimed at thousand-core systems [18, 24]. With sequential simulators [4, 12, 15, 22], such research is highly cumbersome. Consequently, there is a consensus view that *parallel simulators* are required. In the world of RTL-level simulators, parallel simulators are already commercially available, for example, the Synopsys VCS Cheetah.

In the last 4 years, several parallel architectural simulators have been proposed for modern processors such as Graphite [18], ZSim [24], Sniper [7], and Slacksim [8]. These C/C++-based parallel simulators have mostly been written from scratch, and are typically 10× faster than their sequential counterparts. The total error as compared to sequential simulation is typically limited to 10%. Since architectural simulation is a method for estimating the broad high-level trends, the research community has by and large found the amount of additional simulation error in parallel simulators acceptable.

In this article, we present the design of a soon-to-be-released, open-source simulator called *ParTejas*. *ParTejas* is the parallelized version of the popular *Tejas* simulator [25]. *Tejas* has been released under an open-source Apache 2 license. It is a fast, trace-driven, cycle-accurate simulator, which is platform independent. It supports a plethora of architectural features. It simulates out-of-order pipelines, nonuniform caches (NUCAs), complex networks on chip (NOCs), and CUDA-based GPGPUs. It also simulates relaxed memory models—specifically, processor consistency. It supports applications written for relaxed memory models by supporting the `fence` instruction, which requires write acknowledgments to the pipeline. *Tejas* has been validated against real hardware (Dell PowerEdge R620 server). It has been observed that *Tejas* is more accurate than most of the existing simulators (for which the validation results have been published). *Tejas* incurs an average absolute error of 11.45% for sequential workloads (SPEC CPU2006 benchmark suite) and 18.77% for parallel workloads (SPLASH2 benchmark suite). The reader is encouraged to read [25] for a detailed treatment of *Tejas*'s design, implementation, and evaluation.

The important differentiating factor of *ParTejas* is that unlike recent works that achieve their performance by relying on high-level models (of the cores and memory system) or advanced sampling-based approaches, we primarily rely on novel concurrent data structures. Specifically, we use the recently proposed lock-free slot scheduling technology [1] and flexible barriers (newly introduced in Java 6) known as *phasers* to derive our speedups. We demonstrate that by using these novel data structures, we can simulate tightly coupled multiprocessors. We view approaches such as sampled simulation and statistical simulations as complementary approaches that can always be added to our simulation setup.

Specifically, our contributions in this article are as follows:

(1) We observe that the pipelines operate in parallel. The rendezvous points between the threads are in the memory system. The crux of our technique for implementing a parallel

memory system is a novel lock-free data structure that we term as a *parallel port*. The parallel port helps us use local clocks in each Java thread yet model the contention at shared resources accurately.

(2) We propose a simulation algorithm that uses *phasers* (see Section 4.1) to dynamically reduce the time that threads spend in synchronization. This ensures that the time wasted in waiting for a barrier is significantly reduced.

(3) We design a fast shared-memory-based transfer mechanism between the native PIN application threads and our Java-based simulator threads.

(4) For improving the performance of Java programs, we propose a host of Java-specific optimizations, namely, selection of appropriate data structures, fine-grained locking mechanisms, and pooling techniques.

(5) We show 10 to 15× speedups on in-order and out-of-order pipelines with 64 threads.

1.1 Improvements Over Base Tejas

In the base *Tejas* simulator, the number of simulation threads, n_{sim}, is equal to 1. The efforts taken (1) to enable n_{sim} to be greater than 1, (2) to ensure that the induced error is modest, and (3) to ensure that the parallelization is maximized constitute *ParTejas*. These include spawning and scheduling of multiple simulator threads, updating the transfer engine to support n_{sim} consumers, equally dividing the simulation responsibilities among the simulator's threads, and accurately modeling the causality and contention in the memory system. These are done in a manner such that a minimal amount of synchronization overhead is introduced.

Additionally, *ParTejas* adds support for the simulation of various artifacts of parallel benchmarks. Specifically, *ParTejas* simulates the following synchronization functions of the *pthread* library: wait, signal, broadcast, mutex lock, mutex unlock, and barrier wait. Importantly, causality is maintained—the order in which the different application threads invoke the different synchronization functions are exactly mimicked by their simulated counterparts. This greatly improves the accuracy of simulation. *ParTejas* also supports different atomic instructions such as *compare-and-swap* (CAS). Such support requires modifications to the coherence protocol to ensure that the core performing the CAS holds the cache line exclusively until the operation is completed to ensure atomicity.

2 SYSTEM ARCHITECTURE

2.1 Overview

Figure 1 shows an overview of *ParTejas*. We instrument regular x86 or x86-64 binaries using an instrumentation engine such as Intel's PIN [14] to provide us with execution traces. An *execution trace* consists of a stream of packets, where each packet contains the details of an executed instruction such as its program counter, memory addresses accessed, and its branch outcome (if any).

This stream of packets is passed to waiting Java threads. Each Java thread simulates a set of cores, each having a separate pipeline (in order or out of order). A Java thread first translates the CISC instruction packet into a sequence of VISA (a custom RISC instruction set) instructions. It then sends the VISA instructions to the appropriate pipeline. The timing of the instructions is simulated by the individual pipelines and memory accesses are passed to the parallel memory system. The memory system supports private caches, shared caches, and coherence mechanisms.

This decoupling of the instrumentation and the simulation engines has important advantages. First, since the simulation engine does not have to emulate the benchmark's functionality, it does not have to deal with register or memory values at all. The duty of the simulation engine now is to simply provide an accurate estimate of the time taken to execute the benchmark on the target architecture. It merely has to simulate accurately the times at which data is available for

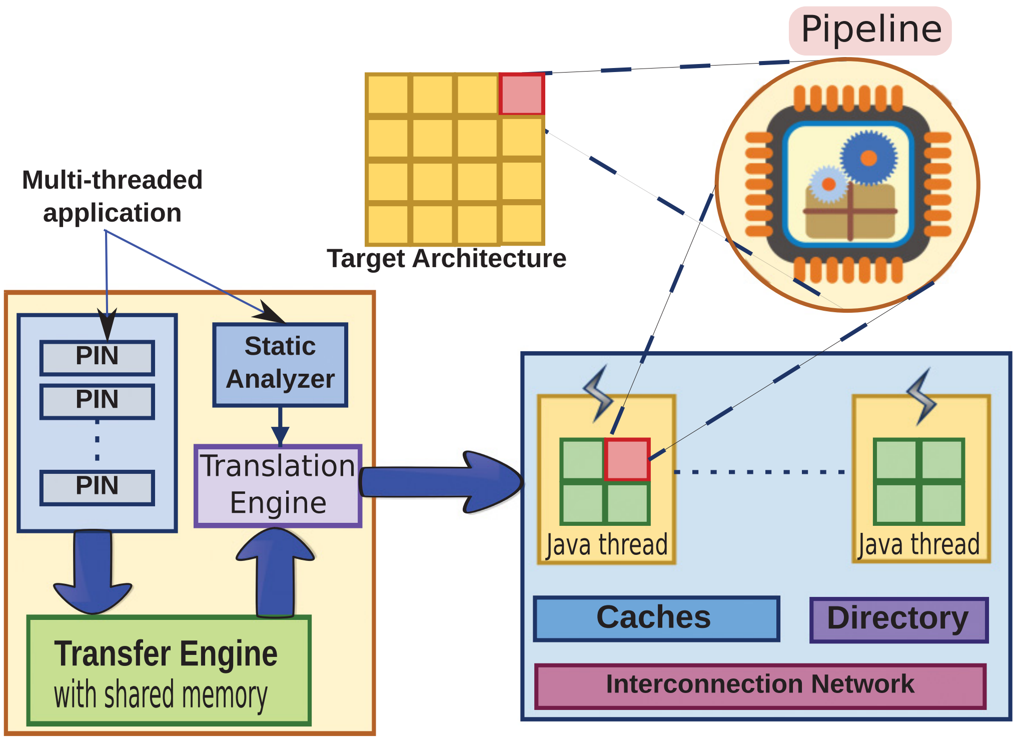


Fig. 1. Overview of *ParTejas*.

consumption, rather than the value of the data itself. Second, instrumentation engines such as PIN and QEMU are highly optimized and far outperform any custom emulator. Custom emulators also suffer from low coverage of the intended functionality, given the complexity of the x86 instruction set. Third, the decoupling allows the two engines to run on different machines, enabling faster simulations. Fourth, the instrumentation engine may be run once and the traces may be stored in files. These files can then be used to feed the simulation engine independently. Such an approach ensures a deterministic input trace across multiple simulation runs, something that is often useful when working with parallel benchmarks. In this article, we do not assume the file interface in our discussions, for the sake of simplicity. We assume that the emulator runs in parallel with the simulator, the former passing traces to the latter.

The Java threads do not operate in lock step. This creates issues in the memory system, which needs to cater to requests from multiple threads. We shall design novel solutions to model the causality (load-store order) and contention in the memory system.

2.2 PIN Instrumentation

PIN is a binary instrumentation engine that runs on Intel-based x86 (32/64-bit systems). It dynamically reads the contents of a binary and adds its instrumentation code to track events such as the execution of a memory or branch instruction. After it detects an event such as a branch event, PIN invokes a custom function. We use such custom functions to summarize the execution of an instruction in an *instruction packet* and send it to the Java threads. Every instruction packet contains the application's thread id and the program counter of the instruction. Additionally, in the case of

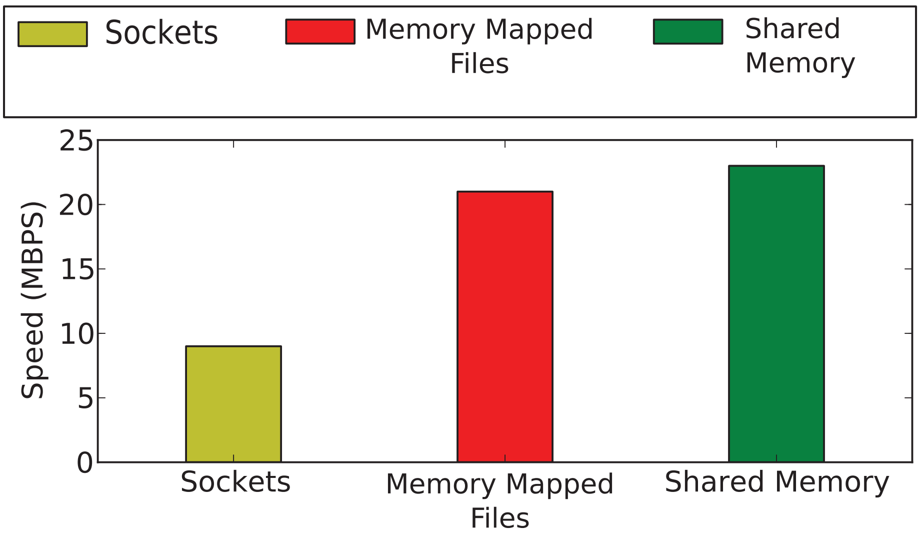


Fig. 2. Comparison of various interprocess communication mechanisms.

a memory instruction, the instruction packet contains the memory address, and in the case of a branch instruction, it contains the branch outcome. We send *synchronization packets* to indicate the beginning and end of synchronization events such as locks, unlocks, signal-wait events, and barriers. Such packets also contain the address of the synchronization variable, timestamp (real time), and nature of the event. *ParTejas* uses this information to simulate the effect of synchronization calls.

ParTejas can also employ QEMU as the emulator, as an alternative to PIN. This allows *ParTejas* to simulate the entire software stack, and not just the application. *ParTejas* also supports the saving of emulator trace information in files and reading from these to perform simulation. This helps achieve deterministic outcomes in experiments involving parallel benchmarks. In this article, however, we will assume the PIN emulator mode only. The proposed optimization techniques are independent of the emulator used.

2.3 Transfer Engine

Application threads run in parallel, and also dump their traces to small in-memory circular buffers. Gigabytes of trace data are generated per second. It is necessary to send all of this data to *ParTejas* using a high throughput channel. We evaluated several options: shared memory, memory-mapped files, network sockets, and Unix pipes. Unix pipes were found to be the slowest of all. Hence, we ceased to consider them early in our design process.

As an experiment, we wrote a small compute-intensive loop-based benchmark. For this experiment, we measured the time that it takes to transfer 1GB of data from PIN to Java on an Intel Core i7 desktop (2.4GHz processor, 4GB RAM, Ubuntu Linux 12.04). Since we have finite buffers, we need a method for the consumer (Java thread) to indicate to the producer (PIN) that it cannot accept more packets for the time being. In the case of sockets, we use a dedicated socket from the consumer to the producer, and for shared memory or memory-mapped files, we use shared variables to indicate the status of the consumer. Figure 2 shows the results. We observe that sockets are the slowest (10MBps). This is because they make costly system calls to transfer data across the processes, and buffering is done by the kernel. However, they are also very versatile. We can run

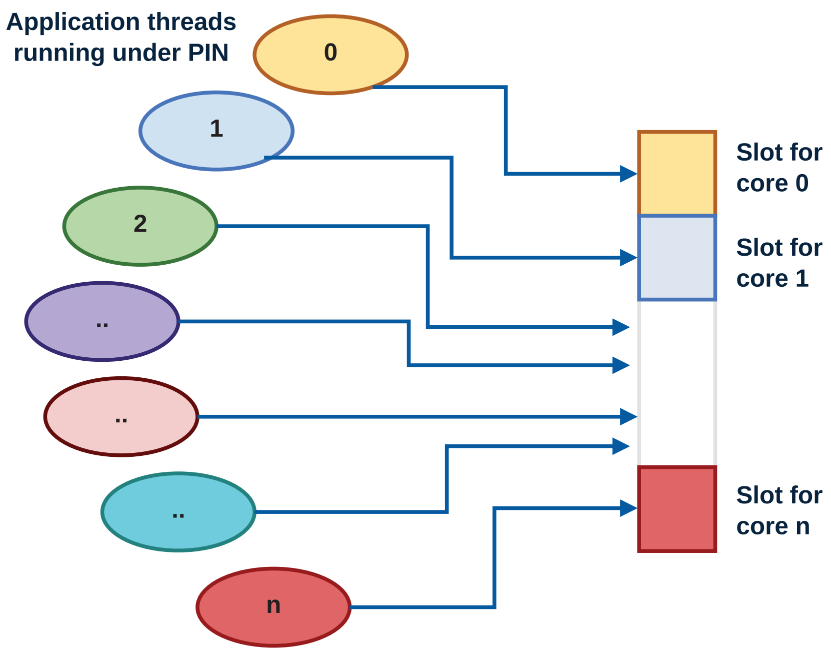


Fig. 3. Structure of the shared-memory segment.

the application threads on one machine and run simulator threads on another remote machine. For high throughput, shared memory is the best option (24MBps). Communication with memory-mapped files is slower, because we need to synchronize data with the hard disk or the disk cache in main memory.

2.3.1 Shared Memory.. We use the *shmget*and *shmat*calls in Linux to get and attach shared-memory segments to the PIN processes. We use a single shared-memory segment and avoid using a separate shared-memory segment for each application thread because underlying operating systems typically place a limit on the number of shared memory segments a process can use. Subsequently, we divide the segment into *n*contiguous regions, one for each of the *n*application threads. Each region contains a header and a circular queue. We fix the size of each packet to 192 bytes and allocate space for 50 packets in each circular queue. The header contains the status of the thread, the number of outstanding packets in the queue (*count*), and its start and end locations.

The important point to note here is that the custom functions in PIN are written in C++, and the simulator threads are written in Java. There is no support for Linux shared-memory segments in Java. Hence, we use the Java Native Interface (JNI) to access shared memory. It allows us to write code in C that can be linked to the JVM at runtime. The second issue is that of locking. We need to be able to get locks to update the pointers to the circular queue and the *count*variable. We use a Peterson lock [20] optimized for the Intel x86 TSO (total store order) memory model.

The code shown in Algorithm 1 implements a Peterson Lock for TSO systems with a single fence (hardware and compiler). Let us elaborate. A Peterson Lock without fences guarantees mutual exclusion between two processes only for a sequentially consistent system. However, if sequential consistency cannot be ensured, then the algorithm will fail to work because the ordering of memory accesses cannot be guaranteed. Now, in the TSO model, writes to different addresses are globally ordered. As a result, the writes to *flag* and *turn* will be globally ordered, and we do

Table 1. Time Taken by Different Types of Queues

Mechanism	Time
Compare-and-set-based lock	139s
Test-and-set-based lock	83.5s
Fetch-and-increment-based lock	23.4s
Peterson Lock	1.15s

ALGORITHM 1: The Peterson lock for x86 Systems

```
$flag[tid] \leftarrow$ true
$turn \leftarrow tid$
fence()
while $flag[1 - tid] = $ true and $turn = tid$ do
end
/* Critical Section Begins                                   */
...
/* Critical Section Ends                                     */
$flag[tid] \leftarrow$ false
```

not need to insert a fence between them. However, TSO relaxes the write-to-read ordering and allows a processor to read its own write early. Hence, it is necessary to have a global ordering between a write and a subsequent read. Therefore, we insert a fence between the update to *turn* and the while loop. Subsequently, we are only reading the values of $flag[1 - tid]$ and *turn*. Since TSO does not change the read-to-read ordering, there will be no violations in correctness. Similarly, while releasing a lock, there is no problem in not having a fence. We can argue that in the worst case, some other process may see that the process that held a lock has proceeded forward without changing $flag[tid]$ to $false$. However, this will not happen if we assume that variables accessed inside the critical section of the Peterson Lock are not accessed outside it (property of properly labeled programs [11]). For another process to know this fact, it needs to enter the critical section first, and this is not possible unless *flag*[tid] is set to **false**. Hence, our code for the Peterson Lock is correct.

Along with updating the header and its contents, we also read the execution packets that have already been transferred to *ParTejas* using dedicated JNI routines. JNI uses Intel x86 string instructions to transfer large amounts of data between memory locations in one go. Hence, we homogenized the structure of a packet, at the cost of space. Each packet contains three 64-byte fields. We use a single JNI call, $memcpy(void*, const\ void*, size\_t)$, to copy an entire memory area into the address space of Java threads. Also, note that Java is big endian, while Intel x86 is little endian. JNI handles this by providing its own data types such as *jint* and *jlong*. While transferring values to the JVM, JNI routines seamlessly convert little endian values to big endian values.

2.3.2 Comparison of the Peterson Lock with Other Atomic Instruction-Based Approaches.. To figure out what is the best synchronization mechanism, we conducted a simple experiment. We created a 100-element shared queue of integers and shared it between two processes (one written in C++ and one in Java). This mimics our PIN-Tejas setup. On an Intel Xeon 2.2GHz CPU with GCC 4.7.2, we compiled our C++ code, and we compiled our Java code with Sun Java 7. We then transferred 1 million integers between the processes. The time required for transfer is shown in Table 1 using different mechanisms: Peterson Lock optimized for the TSO memory model, and different implementations using atomic primitives.

We observe the Peterson lock to be 100× faster for this workload. This is per se not a new observation for a system with two threads. The reasons can easily be understood by looking at the data produced by David et al. [9] (SOSP 2013). An atomic operation takes somewhere between 120 and 310 cycles on a dual-socket Xeon server. This is because it involves locking a line in the modified state, and at least a single fence instruction. A fence imposes a total order on memory accesses, and significantly increases overall latency, because loads can be issued only after the fence is committed. Most of the advantages of out-of-order processors will be lost. In comparison, only loads and stores are used in the Peterson Lock, and thus are cheap. Most of them hit in the local cache, and are thus very fast (one to three cycles). In the waiting phase of a Peterson Lock (while loop), there are no fences, and thus it is much faster. There is only one fence before the while loop, and this does not determine the overall timing.

2.4 Translation Engine and Pipelines

We initialize the simulation by populating a static table called the *instruction table* by reading the contents of the executable that we wish to simulate. We use the GNU *objdump* command for this. The instruction table contains the contents of each instruction indexed by the program counter in a hashtable. To avoid issues with dynamically linked libraries, we statically compile our benchmarks. If this is not possible, we can use one of two options: use trace-driven simulation (file interface) or use programs such as statifier and Ermine to create statically linked binaries out of dynamically linked binaries.

During the simulation, we need to only transfer the details of memory and branch instructions from PIN. We have to transfer memory addresses because we need to simulate the memory system. For branch instructions, we transfer the branch outcome from PIN. The outcome of a branch instruction indicates the beginning of a basic block. Other than branch, memory, and synchronization instructions, it is not necessary to transfer any other type of instruction such as ALU instructions from the basic block, because we can infer its execution from the instruction table. We do not need operand values because we only simulate the timing of instructions. After this offline component, we start the application emulation with PIN.

Now, for n_{app} application threads and n_{sim} Java threads, each Java thread simulates n_{app}/n_{sim} parallel pipelines (see Figure 1). In *ParTejas*, each Java thread independently reads the circular queues of all the application threads that it simulates. We assume that each Java thread is physically mapped to a separate core. Subsequently, we translate each CISC instruction into a set of RISC instructions. Our instruction translation scheme is similar to PTLSim [29]. PTLSim has been validated with commercial x86 processors such as AMD K8. Next, instructions are passed to a data structure that simulates the pipeline. All the pipelines in *ParTejas* inherit from a generic *Pipeline* interface. Individual pipelines can be either in order or out of order.

For in-order pipelines, we support both single- and multiple-issue pipelines. For out-of-order pipelines, we model a 12-stage pipeline with load latency speculation and multilevel bypass. Since modeling pipelines is established technology, we shall avoid explaining the operation of our pipelines in detail. In our simulator, we distinguish between two types of events: regular and nondeterministic. *Regular events* need some processing in the current cycle and also need some processing in the subsequent cycle. Examples of such events are events generated by instructions moving from the fetch to the decode stage. Unless there is a pipeline stall, these instructions will progress from one pipeline stage to the next, every subsequent cycle. In comparison, nondeterministic events such as memory accesses do not have a fixed latency. It is possible that a memory event leaving the load-store queue might return after two cycles (L1 hit) or after hundreds of cycles if it goes to the main memory. For nondeterministic events, we use a per-thread event queue, whereas for regular events, the standard iterative simulation model is employed.

3 PARALLEL SIMULATION

We should note that the process of transferring packets, translation, and execution is inherently parallel. The challenges to parallelism arise because of interactions between threads in the form of synchronization operations and memory operations. Traditional parallel discrete event simulators follow one of the two alternatives: a pessimistic strategy where we explicitly enforce ordering among all interthread interactions or an optimistic strategy that detects causality violations and later recovers from them. When reasoning about *ParTejas* along these lines, we find that it adopts a mostly pessimistic strategy.

We assume that every Java thread has its local clock. The threads periodically synchronize using a barrier at every epoch boundary (clocks become the same). This periodic synchronization of clocks ensures that most interactions between threads are simulated in the same order as emulated, or at the very least reduces the timing skew. An *epoch* is defined as a given number of cycles measured using the thread's local clock. We discuss the tradeoffs for choosing an optimal epoch size in Section 6.5.

We now discuss further efforts at enforcing synchronization so as to minimize the simulation error while still maintaining the performance benefits of parallelization.

3.1 Synchronization Operations

The POSIX pthread library primarily uses three kinds of synchronization operations: $signal - wait$, $lock - unlock$, and $broadcast - wait$. The lock-unlock primitive is used to implement critical sections by establishing exclusive ownership of a lock variable. The signal-wait and broadcast-wait primitives are used to make a set of threads wait till another thread releases them. Barriers internally use the broadcast-wait primitive. Let us consider signal-wait. During simulation, it is possible that a waiting thread might move ahead of the signaling thread. This is because it might be the case that PIN was able to quickly send the packets for the waiting thread, and the packets for the signaling thread got delayed, possibly due to a context switch. Thus, *ParTejas* will see the waiting thread resume before seeing the signal event.

We have two modes. In the first mode, *nosync*, we allow this situation to happen in the interest of simulation speed. We shall see that the jitter introduced by PIN is not very large. However, we have a more accurate (but pessimistic, in terms of its effect on performance) mode, $fullsync$, which tracks dependencies across threads and enforces them. We track the following events: l_e (lock enter), l_x (lock exit), u_e (unlock enter), u_x (unlock exit), s_e (signal enter), s_x (signal exit), b_e (broadcast enter), b_x (broadcast exit), w_e (wait enter), and w_x (wait exit) using the emulator. For each event, the emulator appends its timestamp (real time) and address in a synchronization packet, and sends it through the transfer engine. The simulator process has a table called *syncops* that maintains a list of all the synchronization operations indexed by the address of the synchronization variable. *syncops* is protected by a single lock.

Let us first define some terminology. $w_e(t_A)$ refers to the wait enter event of thread A. We further say that $w_e(t_A) < w_e(t_B)$ if the timestamp of the former event, as recorded by the emulator, is less than the timestamp of the latter. Based on these timestamps, we want to establish a partial order $\rightarrow$ across events in the simulation. $A \rightarrow B$ means that A needs to happen before B. Table 2 shows the timing condition, as recorded by the emulator, on a synchronization variable for different synchronization operations and the consequent invariant that we need to maintain in the simulation.

For the sake of brevity, let us explain the most complex lock-unlock condition (the others follow the same pattern). The first subcondition says that if a lock-enter for thread B arrives when the same lock is held by thread A, B needs to wait till the lock is released. We thus have $u_e(t_A) \rightarrow l_x(t_B)$. Second, we need to ensure that the lock-enter call for B is processed by *ParTejas* during the duration

Table 2. Conditions for Maintaining the Synchronization Order Across Threads

Condition	Invariant Enforced in *ParTejas*
Signal-Wait	
$w_e(t_B) < s_e(t_A) < w_x(t_B)$	$s_e(t_A) \rightarrow w_x(t_B)$
	$w_e(t_B) \rightarrow s_e(t_A)$
Broadcast-Wait	
$w_e(t_B) < b_e(t_A) < w_x(t_B)$	$\forall t_B, b_e(t_A) \rightarrow w_x(t_B)$
	$\forall t_B, w_e(t_B) \rightarrow b_e(t_A)$
Lock-Unlock	
$(l_e(t_A) < l_e(t_B) < u_e(t_A)) \wedge$ $(u_e(t_A) < l_x(t_B))$	$u_e(t_A) \rightarrow l_x(t_B)$ $l_e(t_A) \rightarrow l_e(t_B) \rightarrow u_e(t_A)$

that A holds the lock. Hence, we have $l_e(t_A) \rightarrow l_e(t_B) \rightarrow u_e(t_A)$. Note that the right column is the invariant we need to enforce by stalling the pipeline of either A or B, and the left column is the set of conditions that the timestamps of the events must satisfy. To implement any dependency of the type $A \rightarrow B$, whenever we receive B, we stall all the threads till they have received events with greater timestamps. This ensures that A has reached one of the Java threads. B needs to retire from its pipeline after A has retired from its pipeline. After B retires, we can free the entry for A in the *syncops* table.

3.2 The Memory System

3.2.1 Parallel Ports.. Implementing the memory system (both shared and coherent caches) is difficult because there are happens-before relationships in memory (store → load), and it is necessary to model the contention in memory banks and directory structures. The interactions between the threads in the memory system can be classified into three categories: (1) contending for access to the shared resource (cache, directory, network), (2) memory behavior when they access the same shared data, and (3) memory behavior when they access private blocks that happen to map to the same set in the shared cache. In the third case, changing the order of accesses may change the eviction pattern in the shared cache and the directory, leading to results different from the native run. Fortunately, prior work [24] and our experiments (Section 6.5) demonstrate that the number of such conflicting accesses between threads in a small window of time is very small. The second type of interaction, that is, accessing the same shared data, is much more frequent. Since synchronization primitives are modeled accurately in *ParTejas* (see Section 3.1), and most programs are free of data races (accesses to shared data are wrapped in critical sections), there is no possibility for any deviation due to shared data access.

We thus primarily focus in this section on the first type of interaction—modeling contention. The most important assumption that we make is that every thread issues requests using its local clock. The memory system does not have a separate clock. For example, if a thread with its local clock equal to τ records a miss in the L1 cache, then it accesses the shared L2 cache at time τ. At this point, it essentially assumes that its local clock is equal to the global clock. It performs the L2 access using its local clock. As discussed, we need to model contention at the L2 cache. For example, if $\tau = 10$, we need to consider the state of the L2 cache at the 10^{th} clock cycle and delay the current request if there is contention at that cycle. Unlike other simulators, there is no separate thread that simulates the memory system.

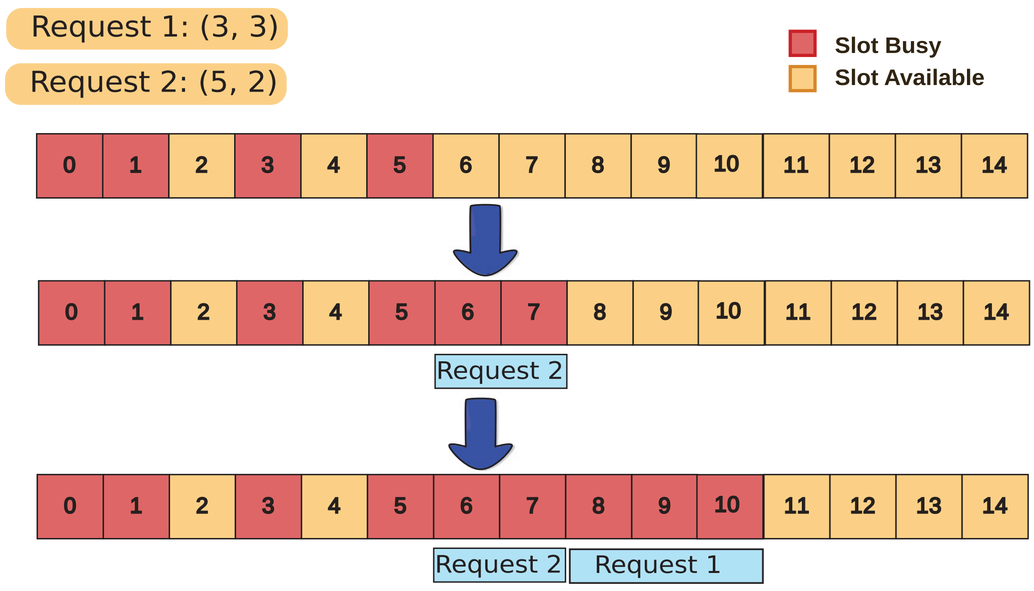


Fig. 4. Operation of a parallel port.

To model contention, we assume that every structure (cache bank, controller, directory, NOC element) has a parallel port at its inputs, which acts as a gatekeeper. The *parallel port* is implemented as a matrix of slots, where the number of rows is equal to the number of simultaneous requests that can be processed in the same cycle (capacity), and each column represents the number of the cycle (as per the local clock of a thread) in the current epoch. Each slot can either be *available* or *busy* (if a request is scheduled). For example, if an L2 request requires three clock cycles, it needs to reserve three contiguous slots.

A parallel port that can accept only one request per cycle is shown in Figure 4. Let us assume that two requests arrive simultaneously. The request is in the form of a tuple having the requested starting slot (cycle) number and the number of consecutive slots as its two parameters. Since slots 3 and 5 are busy, we need to start scheduling from slot 6. The parallel port arbitrarily chooses request 2 to be scheduled in slots 6 and 7. Henceforth, it schedules request 1 in slots 8, 9, and 10. The main aim of a parallel port is to provide a linearizable (all requests appear to execute instantaneously) data structure that tries to schedule requests as close as possible to the requested time. It can effectively model contention because it never schedules more requests in a cycle than the capacity of the structure.

We implement the parallel port using lock-free algorithms based on the work on nonblocking slot schedulers by Aggarwal and Sarangi [1]. They propose a method to implement large 10,000- to 100,000-entry parallel ports using algorithms that do not use any kind of locks. Each cell is an atomic integer, and we need to only use the compare-and-set primitive. This approach is at least two orders of magnitude faster than approaches that use locks. The main idea of their approach is to first temporarily reserve the earliest available slots. If there is a conflict with another thread while performing temporary reservations, then one thread helps the other thread in completing its request. After a thread completes its request either on its own or with the help of other threads, it proceeds to permanently mark them as reserved. This data structure is linearizable; that is, operations appear to take effect instantaneously at some point of time between its invocation and completion. Our implementation of the parallel port can typically process a request in less than 2 to 3μs.

Since lock-free algorithms can sometimes suffer from starvation, we have the option of using a wait-free parallel port that guarantees completion of the request in a finite amount of time. This is again based on the work by Aggarwal and Sarangi [1]. A thread desirous of booking some slots first places its request in an array of requests (one entry per thread). Subsequently, it traverses the older entries of the list to see if there is any request that has been waiting for a long time (determined by a threshold). If there is such a request, then the current request first *helps* this request. This is done by taking the parameters of the request (starting slot, number of slots) and then trying to mark the slots in the parallel port. This is again a two-step process. First, the thread needs to temporarily reserve the slots, and then tries to permanently reserve them. At any point of time in the temporary reservation process, it is possible that another thread might take a slot away. In this case, the thread needs to undo its work and start from another position. It is also possible that another helper completes the work. This also needs to be checked periodically. This approach is much slower than the lock-free algorithm, yet it guarantees that every thread makes progress and there is no starvation. The wait-free implementation was measured to be 3 to 5× slower than the lock-free implementation and is only suitable for a system with a very large number of simulation threads or a system that has a lot of jitter. The rest of this article is based on the lock-free parallel port.

Another subtle point to note is that as we reach the end of an epoch, we need to place requests in the next epoch. We thus use a pair of parallel ports. When we reach the end of the epoch, we place requests in the other parallel port. Thus, for a short duration of time, both the parallel ports are used, and finally when all the threads reach the end of the epoch, the old port is not used anymore. The old port will be reused at the end of the next epoch. To avoid cleaning up entries eagerly and thus wasting time, we lazily clear them by prefixing the least significant bit of the epoch number with the state bit (vacant or reserved).

3.2.2 Directory Coherence.. The moment there is a read/write miss in a private cache, the cache controller of the requesting core, C, sends a message timestamped with its local clock. After passing through the parallel port, the Java thread looks up the list of sharers. The directory implements a MOESI protocol and subsequently sends a request to the owner core. The owner core uses the clock domain (local clock) of C to access its private cache and sends the block directly to the cache controller of C. The reader should note that all these operations happen in the clock domain of the requesting core, C. All the relevant parallel ports are accessed using this clock domain. Also, the Java thread simulating Cperforms all of these accesses. A memory request never passes to another thread.

We admit that this approach introduces some error, as the state of the cache being accessed corresponds to a time different from that of the requesting cache. However, the error introduced is low, as our results confirm (see Section 6.5). The low error is due to two reasons: (1) accesses to shared data, which are more critical when determining performance, are simulated accurately as synchronization operations are captured meticulously (see Section 3.1), and (2) accesses to shared structures like the directory are modeled using the parallel port, thereby capturing the contention among accesses (see Section 3.2.1). Our aim in following this coherence approach is to provide an abstraction that a simulator thread operates independently of others. At a small price in terms of error, significant gains are attained in simulator performance by not forcing a synchronization between simulator threads for every directory access.

4 OPTIMIZATIONS

Aside from the high-performance parallel ports, we also propose the following optimizations.

4.1 Relaxed Synchronization Using Phasers

We shall observe in Section 6.3 that the periodic synchronization at the end of an epoch is one of the largest limits to scalability. The reason for the lack of scalability is that a considerable imbalance exists between the slowest thread and some of the fastest threads. The faster threads wait for the slowest thread to catch up at the barrier, resulting in a considerable amount of idling in the system. For example, it is possible that one thread simulates cores that have high IPC, and thus it needs to do a lot of work to reach the end of the epoch. In comparison, another thread might simulate cores that suffer a series of L2 cache misses. Most of the time, its pipelines will be idle, and thus the simulation will proceed quickly. When both of the threads synchronize at a barrier, one of the threads has to remain idle and wait for the other thread to finish its epoch. We have observed in our experiments that most of the time the interaction between threads is relatively low, and thus it does not make sense for threads to wait for each other all the time.

Consequently, we propose to replace barriers with *phasers* to cater to programs that have long phases of low interaction between threads. A phaser is a special kind of barrier that has two distinct points. After a thread arrives at the first point, it informs the rest of the threads. No thread can cross the second point unless all the threads have reached the first point. Note that a regular barrier is a special kind of phaser that merges both the points. Java 6 (with backports) or Java 7 has native support for phasers. We use phasers to consider two epochs at a time. After the end of the first epoch, a thread enters the first point of the phaser. Instead of suspending itself, the thread continues till the end of the second epoch or till the time it has to access a shared resource such as the shared cache or the communication interface. At this point, it enters the second point of the phaser. Now, if there is any thread that has not completed the first epoch, then the thread waits for it to finish the first epoch. Thus, *phasers* give us the same synchronization benefit as barriers and at the same time we are able to reduce the idle time by a considerable amount (see Section 6.3). The reason we can reduce idling with a phaser is because it is better suited for adjusting to timing jitter in simulation. Since we allow a limited amount of divergence between the threads, the jitter can get balanced across two epochs. On the other hand, a phaser does not allow the divergence between the cycle count of two threads to be arbitrarily large. Any two threads can at the most be one epoch away. Thus, there is a tradeoff between accuracy and the amount of idle time here.

The addition of phasers requires us to modify the structure of our parallel ports. Previously, the size of a parallel port was equal to the size of an epoch, or *phase*, and we used two parallel ports to seamlessly place requests at the epoch boundary. Now, we maintain three parallel ports (one for each phase and one buffer). To reuse our ports, we again lazily clear them by appropriately encoding our states. We can extend this idea, to consider phases with n points, and use $n + 1$ copies of parallel ports for each structure. There is a space, speed, and accuracy tradeoff here. We can switch between barriers and phasers depending on the amount of interthread communication. If the load at all the parallel ports is higher than a threshold, then we may switch to using regular barriers at epoch boundaries to ensure additional accuracy.

4.2 Core Partitioning and Mapping

We have n_{app} application threads. In comparison, we have n_{sim} Java simulator threads, where each thread typically simulates more than one application thread. We need to carefully choose the number of cores that we assign to the application and the number of cores that we assign to the Java threads. We observed in our experiments that statically partitioning the cores among these two classes of threads was better in terms of performance than allowing the OS scheduler to manage application and Java threads.

To maximize performance, we need to carefully compute the right sizes of the two partitions. For our workloads, we did not observe a lot of intra-application phases. Hence, we divide the execution into two phases. The first is a short profiling run that runs a benchmark for a predetermined number of instructions with varying partition sizes. We choose the best result for the partition and use the `sched_affinity()` call in Linux to create the partitions.

4.3 Java-Specific Optimizations

A key principle in our design is to avoid locks as much as possible. Most of our structures are nonblocking. We use Java's built-in concurrent data structures that are known to have lock-free implementations whenever possible. We observed that the appropriate choice of the data structure is important.

For example, some read-only structures such as the instruction table should be implemented using HashMaps rather than HashTables. HashTables are synchronized (use locks) by default, whereas HashMaps are not synchronized (do not use locks). Similarly, Vectors are synchronized by default, whereas the ArrayList does not use locks. Some data structures such as the cache bank's tag array has both read and write accesses. The probability of a simultaneous access to the same entry is very low. Hence, in the interest of speed, we implement it with a HashMap. The chances of an occasional error are very small. Similarly, thread-specific state in shared data structures such as the directory are implemented using Array Lists.

To avoid *false sharing*, we try to minimize the number of instances in which we place data belonging to different threads in the same cache line, even if the threads access disjoint sets of data. Instead, we use $ThreadLocal$ variables to localize the data of each thread.

Lastly, there is a tradeoff between manual memory allocation and garbage collection. Garbage collection is fairly efficient and boosts programmer productivity for data structures that are not instantiated very frequently. However, classes such as events and instructions are instantiated extremely frequently. For events, we follow a policy of *object reuse*. The event class is so engineered that it can be used for a variety of communication scenarios between different architectural elements. Therefore, instead of instantiating new events every time, existing objects are reused. Instructions, in *ParTejas*, have elegant points of entry (fetch stage) and exit (commit stage). So for them, we adopt *object pooling*. A pool of instruction objects is created. Objects are drawn from the pool when required during the fetch stage and returned to the pool at commit. Per-thread pools are maintained to avoid expensive synchronization.

5 RELATED WORK

For the sake of brevity, we shall only discuss parallel architecture simulators in this section. For a background of sequential simulators, the reader can look at the following simulators: Simics [15], SimFlex [27], GEMS [17], COTSon [3], SESC [23], ESESC [2], Multi2Sim [26], M5 [6], and SimpleScalar [4].

Wisconsin Wind Tunnel [21] was one of the earliest parallel simulators. It was designed to run on CM-5 machines and used barriers for synchronization similar to *ParTejas*. Wisconsin Wind Tunnel II [19] was later proposed as an extension to Wisconsion Wind Tunnel. This proposal relied on extensive binary instrumentation to estimate the execution time of basic blocks rather than instructions. Also, for simulating memory, the authors proposed to replace memory instructions by small code snippets that quickly estimate the time an instruction will take to execute. The main limitation of these two projects was that they could not be used to simulate complex architectures or complex cache coherence and were not very customizable. On similar lines, BigSim [30] and FastMP [13] were designed to simulate large message-passing computers such as the IBM

Blue Gene. They primarily simulate the message complexity across processors. In comparison, we simulate complex out-of-order pipelines and elaborate cache coherence protocols as well.

SlackSim [8] uses the pthreads API to parallelize the Simplescalar simulator. Each core of the target CMP is simulated by one pthread and the L2 cache is simulated by another dedicated manager thread. Unlike *ParTejas*, a single thread (the manager thread) can become a bottleneck (also pointed out by the authors of the paper). Graphite [18] uses PIN-based instrumentation and runs a single binary on a cluster of machines. A major focus of Graphite is to provide the abstraction of a single address space for all the threads running on different machines. This is achieved by trapping each memory access and converting it into a message that is sent over the network to another node that contains the value of the memory address. Along with functionally simulating a single binary on multiple machines, Graphite simulates cores and the memory system at each node. The authors use Graphite to simulate private caches and private DRAM controllers using Graphite. In our opinion, it is difficult to use Graphite to simulate a CMP with a tightly coupled cache coherence system, a complex NOC, and a multitude of shared structures, because each message needs to be sent over the network. *ParTejas*, on the other hand, is designed to simulate more tightly coupled systems.

Sniper [7] speeds up simulation by using high-level models. The authors define an *interval* to be a compute-bound phase of a thread delimited by misses in the data or instruction caches. Sniper uses high-level models of cores to speed up the simulation time of intervals, instead of performing cycle-accurate simulation. Between intervals, it simulates the misses in the memory system and computes their latency. It simulates an approximate OOO model. Only the functionality of an OOO pipeline is simulated, whereas the structures such as the load-store queue, rename buffer, and register allocation table are not simulated. This makes Sniper unfit to study and design novel structures that are part of a detailed architectural pipeline. The authors also suggest that interval simulation is not a substitute to cycle-accurate simulation [10]. It is rather a complementary approach, which can be used when the need for accuracy is not paramount and the focus is on the speed of simulation.

Similarly, ZSim [24] uses high-level models for in-order and out-of-order cores. It quickly estimates the execution time for a sequence of instructions. It introduces a novel method called "bound and weave." The *bound* phase runs the instructions assuming zero cycle latency for memory instructions, and it simultaneously collects traces. The *weave* phase executes the traces on dedicated nodes that simulate the memory system. The final latency is obtained by a combination of the results in the bound and weave phases. Sampling, modeling, and other approximate simulation approaches are orthogonal to our approach. It is possible to seamlessly augment our simulator to support a variety of high-level models and sampling approaches.

6 RESULTS

In this section, we shall discuss the performance gains achieved through our parallelization techniques. We shall also discuss the error, or rather the deviation, induced due to parallelization.

6.1 Experimental Setup

We evaluated the performance of *ParTejas* on a four-socket, 64-bit, Dell PowerEdge R820 server. It had four 8-core 2.20GHz Intel Xeon CPUs (with hyperthreading enabled), 16MB L2 cache, and 64GB of main memory. We thus had a maximum of 64 cores visible to software. This server runs Ubuntu Linux 12.10 using the generic 3.5.0-17 kernel. All our code is written in Java 6 using Sun OpenJDK 1.6.0_27 with the latest patches. We use Intel PIN (rev:49306) [14], with gcc 4.7.2. Table 3 shows the details of the simulated architecture. We show our results with multi-issue in-order (MII) and out-of-order (OOO) pipelines. We use private L1 caches (instruction and data) and a large shared

Table 3. Parameters of the Simulated Architecture

Architectural Parameters	
Frequency	3.6GHz
Cores (4 per cluster)	64
NOC	mesh (XY routing)
Issue width	4
Decode width	4
ROB size	128
Branch predictor	GShare
L1 size (I and D) (per core) L2 size (per cluster)	16KB (private) 2MB (shared)
Associativity	8(L1)/8(L2)
Latency	2(L1)/22(L2)
Cache coherence	MOESI, directory
Directory	address interleaved, 1 per 4 clusters, fully mapped, 4-way associative
Memory controllers	1(per cluster)
Memory latency	200 cycles

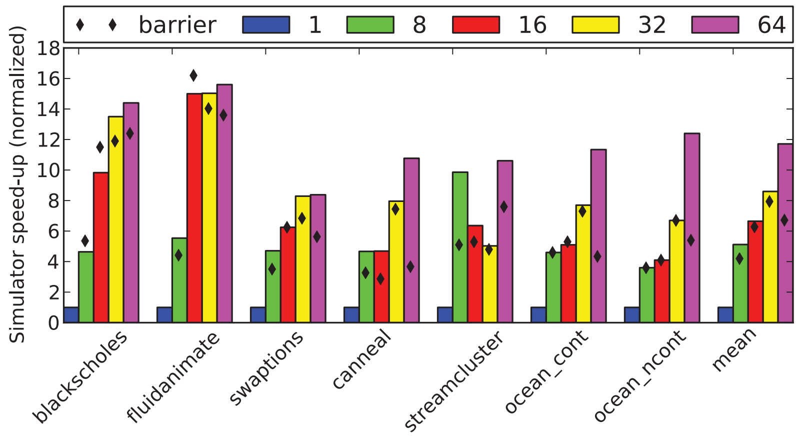
L2 cache. We group four cores into a cluster (also referred to as *tile* in the literature). Each cluster has a *slice* of the shared L2 cache. A 64-core system, divided into 16 clusters, is simulated for all our experiments.

We simulate seven benchmarks from the Parsec [5] (*blackscholes*, *fluidanimate*, *swaptions*, *canneal*, *streamcluster*)and Splash2 [28] (*ocean_cont*, *ocean_ncont*) benchmark suites. We chose benchmarks that have large parallel sections and that spawn parallel threads early in their execution (for quicker simulation). For Parsec, we use simsmall inputs (trends were the same across all the input sets), and for Splash2, we use the default inputs. An epoch size of 1,000 cycles is used for all our simulations.

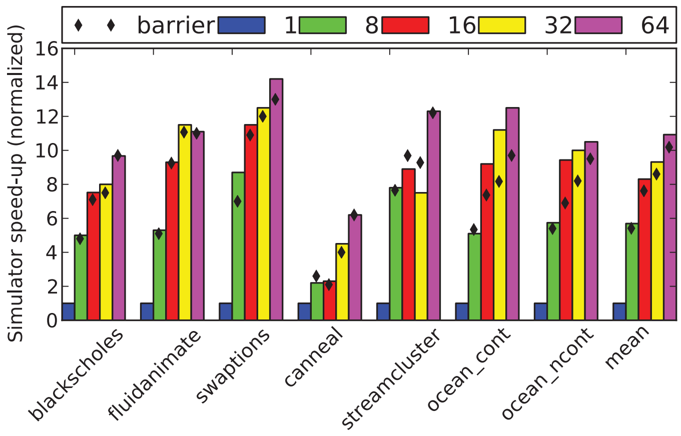
6.2 Performance Results

For all our experiments, we simulate a 64-core processor (details shown in Table 3); that is, the application has 64 threads, each mapped to a different *simulated* core. The number of Java threads, n_{java}, that simulate the application's execution on the target architecture is varied: eight, 16, 32, and 64. Each Java thread is responsible for the simulation of $64/n_{java}$ application threads. As elaborated in Section 6.1, our simulation infrastructure has 64 *physical* cores. These *physical* cores are statically partitioned between the application threads and the Java simulation threads. That is, n_{java} physical cores are allocated to the Java threads, and the remaining are allocated to the application threads. For each benchmark, we show the normalized speedup with respect to the sequential execution time on one core. The bar chart shows the results with phasers, and the results with simple barriers are shown using the diamond-shaped dots. Note that for some benchmarks, such as *streamcluster*,intermittently some more threads (up to 69) are created. We assign each such additional thread (that is short lived) to a new virtual pipeline with the same configuration as a regular pipeline and new Java threads.

Our performance results are shown in Figure 5. Let us first discuss the general trends with phasers. With the MII pipeline, *blackscholes* and *fluidanimate* show speedups of 4.2 to 5.1× for eight threads and 10 to 15× for 16 threads. However, the scalability tapers off beyond 32 threads.



(a) Multi-issue inorder Pipeline



(b) Out-of-order Pipeline

Fig. 5. Performance results (speedups with respect to one core).

In case of the OOO pipeline, the speedup is 5 to 5.5× for eight threads, 7.5 to 9.3× for 16 threads, and 9.7 to 11.5× for 64 threads. *swaptions* has an incremental benefit beyond 32 threads for the in-order pipeline. *canneal*, *ocean_cont*, and *ocean_ncont* show a steady rise in speedup as we move from 16 to 64 cores. For 16 threads, their speedups are between 2.2 and 6.6×; for 32 threads, their speedups are between 2.3 and 9.43×; and for 64 threads, their speedups are in the range of 6.2 to 12.5×, respectively, for both the pipelines. *streamcluster* is an outlier. We see a speedup of 7.8 to 9.8× for eight threads. The performance goes down for 16 and 32 threads in the case of the MII pipeline. With the OOO pipeline, we observe a slowdown for 32 threads. The speedup increases to 10.6 to 12.3× for 64 cores. *streamcluster* creates a large number of auxiliary threads, and there is a sizeable amount of interaction between the threads. However, as we increase the number of hardware threads, it is easier to schedule the execution of these auxiliary threads. Hence, the performance improves with 64 threads.

Table 4. Simulator Speed, Breakup of Time Taken, and Optimal Partition Sizes for 64 Threads

Application	MIPS 16	MIPS 64	Barrier Wait (%)	Par. Port (%)	Mem. Sys. (%)	Pipeline (%)	Phaser Wait (%)	Par. Port (%)	Mem. Sys. (%)	Pipeline (%)	PIN	Java	Synch. Error (%)
			Barrier				Phaser				Threads(64)		
Multi-issue In-Order Pipeline													
blackscholes	3.5	5.1	76.99	1.98	0.01	21.02	40.56	35.03	0.01	24.40	5	59	0
fluidanimate	4.0	4.2	73.84	1.28	0.60	24.28	52.36	19.19	0.60	27.85	8	56	0.02
swaptions	3.1	4.1	83.27	1.23	1.07	14.43	39.53	37.92	1.07	21.48	15	39	0.04
canneal	1.7	3.9	88.52	4.83	0.10	6.55	32.98	47.17	0.62	19.23	8	56	0
streamcluster	4.9	8.2	85.60	1.29	0.01	13.10	61.45	19.84	0.42	18.29	6	58	0
ocean_cont	1.6	3.6	91.37	0.94	0.08	7.61	35.47	44.63	0.01	19.89	7	57	0.05
ocean_ncont	1.2	3.5	89.17	0.77	0.06	10.00	29.40	46.74	0.90	22.96	10	54	0.06
mean	2.8	4.6	84.11	1.76	0.28	13.85	41.68	35.78	0.52	22.02	9	55	0.02
Out-of-Order Pipeline													
blackscholes	2.40	2.80	83.66	0.00	0.30	16.04	72.13	1.42	0.20	26.25	8	56	0
fluidanimate	1.64	2.10	77.10	0.00	0.74	22.16	77.00	0.10	0.34	22.56	7	57	0.15
swaptions	1.41	1.90	71.56	2.43	1.01	25.00	61.20	5.24	1.23	32.33	7	57	0.10
canneal	0.06	0.20	35.04	0.00	0.25	64.71	33.78	3.48	0.74	62.00	4	60	0
streamcluster	3.52	4.00	83.53	0.00	0.42	16.05	85.51	6.59	0.61	7.29	2	62	0
ocean_cont	2.14	2.80	80.48	3.60	0.32	15.60	72.42	4.53	1.03	22.02	6	58	0.12
ocean_ncont	2.48	2.60	76.33	2.74	0.42	20.51	73.15	5.06	0.68	21.11	19	45	0.23
mean	1.95	2.34	72.53	1.25	0.49	25.72	67.88	3.77	0.69	27.65	8	56	0.08

Let us now look at the results with barriers (diamonds in the graph) for the MII pipeline. The first trend is that on average, barriers are 42.6% slower than phasers in terms of the speedup for 64 threads. The second point to note is that other than *blackscholes* and *streamcluster*, simulations with 32 or 16 threads are faster than those that use 64 threads. The synchronization overheads dominate in this case. We can thus infer a net loss in scalability with the use of barriers in this case.

In the case of the OOO pipeline, the difference in performance between barriers and phasers is not so large. On average, phasers are 7.23% better than barriers for 64 threads. Some benchmarks such as *swaptions* and *ocean_cont* show 9.23% and 28.86% improvement, respectively, for 64 threads. The reason for the diminished difference is as follows. The OOO pipeline is fairly complicated and slow as compared to in-order pipelines. Consequently, there are fewer idle cycles and rarely any outlier threads that have a lot of idle time at an epoch boundary (refer to Section 4.1).

Table 4 shows the simulation speed in MIPS (millions of instructions per second). For 64 cores, we observe simulation speeds of 3.5 to 8.2 MIPS with the MII pipeline and simulation speeds of 0.2 to 4.0 MIPS with the OOO pipeline across seven benchmarks. Note that *canneal* is an exception (0.06 MIPS (16 cores) and 0.2 MIPS (64 cores)) because of the large sequential sections in the benchmark that slowed down the OOO pipeline. For 16 cores, the simulation speeds for both pipelines are of the order of 2.8 MIPS and 1.95 MIPS, respectively. In comparison, Sniper [7], which uses high-level models, reports a simulation speed of 2.2 MIPS for simulating 16 cores on a 24-core SMP machine.

6.3 Analysis of the Speedups

Table 4 shows the breakup of the total time in terms of the average (across all threads) time taken to wait for barriers (or phasers), time spent in the parallel ports (and NOC), time taken to access memory structures (caches, directory), and time the pipeline takes to execute. Let us explain with an example. Consider a system with two threads that runs for one epoch (4s long). Let the first thread take 2s and the second thread take 4s. Assume that there was no accesses to the memory or the parallel ports. In this case, the average time in the pipeline is 3s and the average time to wait for a barrier is 1s.

To measure these parameters, we instrument the code with calls to read the current time. For example, when we start a function that simulates the pipeline, we record the time at which we enter and leave the function. We have similar instrumentations for the core functions of the memory system, parallel ports, and barrier/phaser. We use Java's built-in routines that access the CPU's built-in high-resolution timer. An astute reader may argue that these instrumentations may skew the readings and change the behavior of the simulator. However, we took care to minimize their use, and since we have well-defined access points for each module, we did not require a lot of instrumentation. At the end, we collate all the data.

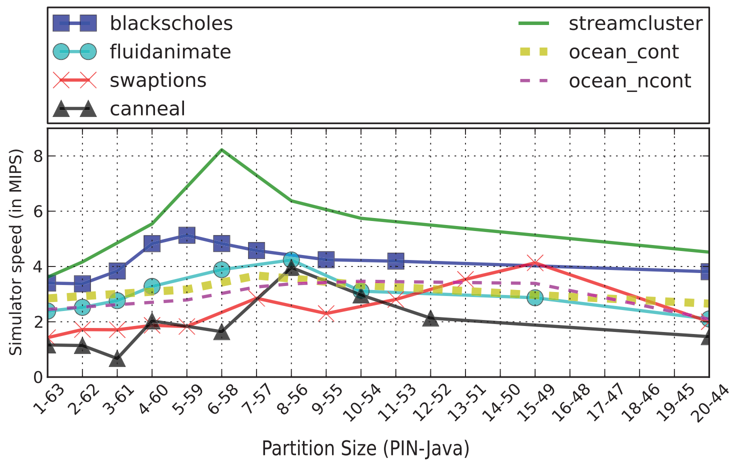
As we discussed in Section 6.2, the OOO pipeline shows similar trends with both phasers and barriers. A large amount of time (70% to 90%) is spent waiting at barriers or phasers, respectively. On the other hand, phasers perform better than barriers by 29.87% with the MII pipeline. With phasers, we spend much less time waiting for a phaser to release (32% to 61%). This is because phasers allow threads to move forward between the first and second points. Note that *streamcluster* is an exception (61%) because it spawns a large number of additional threads. We start spending higher amounts of time at the parallel ports (19% to 47%). Since all the shared structures (cache banks, NOC routers, directories) have parallel ports, the delay becomes large when there is a higher degree of parallelism enabled by phasers. In both cases, memory structures are accessed very quickly (because we use HashMaps).

6.4 Core Partitioning

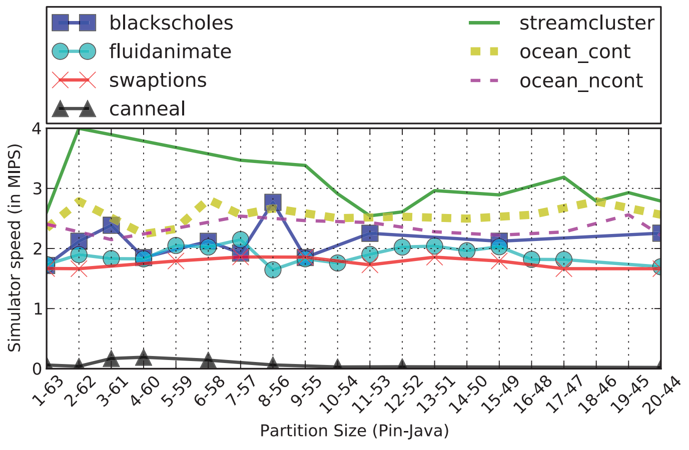
For each benchmark, Figure 6 shows the simulation speed (in MIPS) as a function of the partition sizes with 64 cores. The x-axis shows the number of cores assigned to PIN and the number of cores assigned to the Java threads. The format, $m - n$,means that the PIN threads are assigned m cores, and the Java threads are assigned n cores. From Figure 6, we can infer that there is an optimal partition size beyond which the speedup decreases. For example, for the MII pipeline, the optimal partition size for *streamcluster* is 6 to 58, and for *blackscholes* it is 5 to 59. The second important point to infer from Figure 6 is that choosing an optimal partition can increase the speedup by as much as 50% to 100%. Table 4 shows the optimal partition sizes for each benchmark for the experiments with 64 cores.

6.5 Simulation Errors and Deviations

Let us divide the simulation error into two categories: errors in enforcing proper synchronization (lock-unlock, signal-wait, broadcast-wait) and errors introduced due to the altered nature of the memory system. For the former case, let us compare the mean simulated execution time across the two configurations and report the error (relative difference). The first configuration uses a strict synchronization order as explained in Section 3.1. The second configuration does not obey any



(a) Multi-issue Inorder Pipeline



(b) Out-of-order Pipeline

Fig. 6. Speedup as a function of the sizes of partitions (64 threads).

synchronization constraints at all. The error is shown in the last column of Table 4. The error is limited to 0.23%, which is negligible. This is because PIN enforces synchronization strictly and the additional jitter introduced in the transfer and translation engines is minimal. Hence, it is not necessary to strictly enforce synchronization for Parsec/Splash benchmarks. The packets come to *ParTejas* (Java side) in the proper order.

Let us now evaluate the deviations due to the altered nature of the memory system. Let us first evaluate the nature of sharing of memory addresses between threads in an epoch. Figures 7, 8, and 9 show representative address diagrams for an epoch from the *fluidanimate* benchmark. Owing to the fact that the *fluidanimate* benchmarks have a high level of synchronization [5], it serves as a good representative for this evaluation. In an *address diagram*, the application threads are arranged in concentric circles. Thread 1 is assigned the innermost circle, and thread 64 is assigned

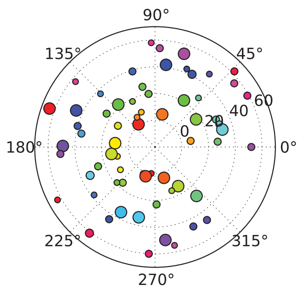


Fig. 7. Epoch size = 500.

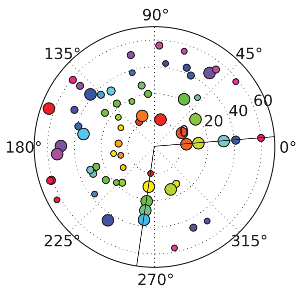


Fig. 8. Epoch size = 1,000.

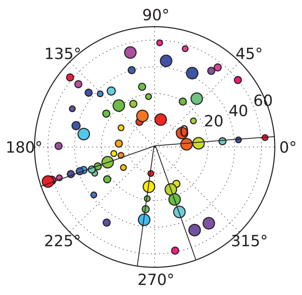
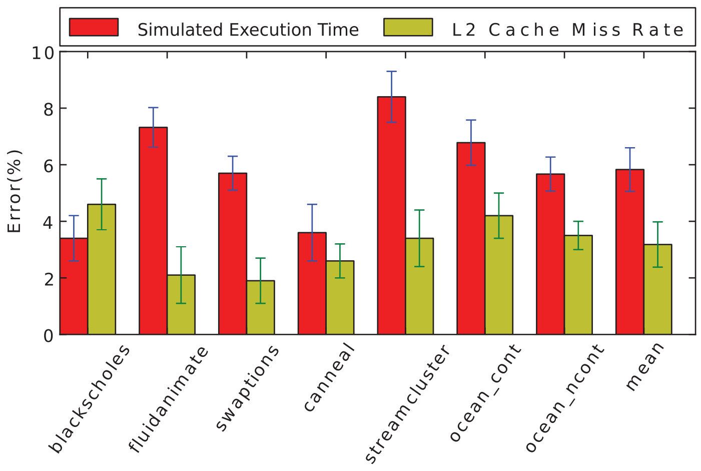


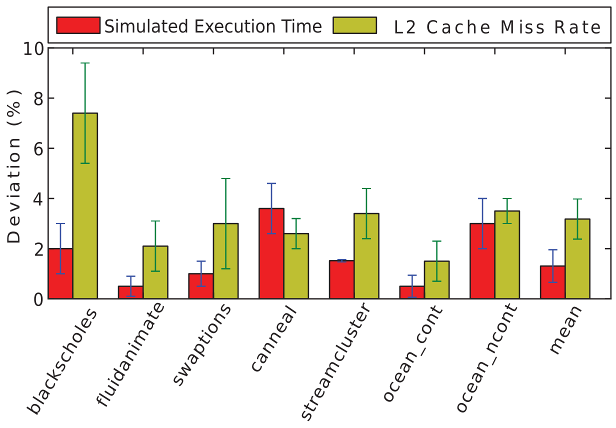
Fig. 9. Epoch size = 10,000.

the outermost circle. The entire physical address space is split radially. Now, for each thread, we find out all the non-read-only addresses that it has accessed in an epoch and mark them in the address diagram. For addresses that are close by and map to the same cache line, we create a bigger circle instead of a cloud of points. Also, if the same address is accessed more than once, we increase the size of the circle. Thus, in each of the address diagrams, we see a multitude of circles of various sizes. Now, we can infer memory contention if we have multiple circles along the same radial line (line passing through the center of a circle). When we use an epoch size of 500 cycles (Figure 7), the contention is minimal. There are at most two circles on any radial line. In comparison, we have two highly contended memory regions with an epoch size of 1,000 (Figure 8), and for an epoch size of 10,000, we have multiple (more than four) highly contended memory regions. There is a tradeoff between the epoch size and the interaction between threads. We choose an epoch size of 1,000 aiming to balance speed and the level of interaction between threads.

If there is a lot of interaction between threads in the memory system, then the order of simulating the accesses is important. Changes in the order of simulating memory accesses can change the simulated execution time of the benchmark as follows. Let us assume that thread T_1 has two writes to the same address W_1 and W_2 interspersed in time. Similarly, thread T_2 has two reads to that address (R_1 and R_2). The order $W_1 \rightarrow W_2 \rightarrow R_1 \rightarrow R_2$ will lead to lower simulated execution time because of higher locality. In comparison, $W_1 \rightarrow R_1 \rightarrow W_2 \rightarrow R_2$ will result in additional transfers between caches, leading to a higher simulated execution time. Hence, it is necessary to choose the epoch size carefully. Let us take a look at the average deviation as compared to a sequential execution in Figure 10. The deviation in mean simulated execution time varies from 0.5% to 8.6%, and the deviation in the total L2 miss rate varies from 2% to 7.4% for both the pipelines. There is some variance (<2%) for different runs of the same benchmark (shown in error bars). The reason we use the word **deviation** here instead of **error** is that once we enforce synchronization and the program is data race free, the simulator does not execute incorrectly. The order of memory accesses might differ as compared to a sequential execution, and the delays of some memory operations might appear to be unusually large because of the parallel port; however, the execution of memory accesses still follows a valid ordering. Recall that the parallel port is a linearizable (stronger than sequential consistency) structure and tries to schedule requests as close as possible to the requested time. Nonetheless, the consensus view in the parallel simulator community is to have simulation results as close as possible to that of a sequential run. In this regard, FluidCheck achieves mean deviations of 6% and 1.3% in the simulated execution time, for the MII and the OOO pipeline cases, respectively, and a mean deviation of 3% in the L2 miss rate (in both pipeline cases). This is similar to competing simulators [18, 24].



(a) Multi-issue Inorder Pipeline



(b) Out-of-order Pipeline

Fig. 10. Average deviation from sequential execution.

7 CONCLUSION

We outlined the design of a novel Java-based parallel simulator called *ParTejas* that can simulate a suite of Splash2 and Parsec benchmarks, which can provide a mean speedup of 11.8× for a multi-issue in-order pipeline and 10.9× for an out-of-order pipeline with 64 threads, as compared to sequential execution. *ParTejas* is different from the other conventional proposals that rely on high-level models and sampling. Unlike others, we base our performance on novel concurrent data structures such as the parallel port and phasers, along with Java-specific features and intelligent core partitioning. As we observed from the results, 70% to 80% of the total simulation time is consumed in waiting at the barriers, and hence we might be able to find improvements in the near future using better simulation techniques and data structures.

REFERENCES

[1] P. Aggarwal and S. R. Sarangi. 2013. Lock-free and wait-free slot scheduling algorithms. In *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2013), 1387–1400.

[2] E. K. Ardestani and J. Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE, 448–459.

[3] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. 2009. COTSon: Infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.* 43, 1 (2009), 52–61.

[4] T. Austin, E. Larson, and D. Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Comput.*, 35, 2 (2002), 59–67.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 72–81.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. 2006. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26, 4 (2006), 52–60.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *2011 IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–12.

[8] J. Chen, M. Annavaram, and M. Dubois. 2009. SlackSim: A platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37, 2 (2009), 20–29.

[9] T. David, R. Guerraoui, and V. Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 33–48.

[10] D. Genbrugge, S. Eyerman, and L. Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE, 1–12.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. 1990. *Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors*. Vol. 18, No. 2SI, ACM, 15–26.

[12] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. 2002. Rsim: Simulating shared-memory multiprocessors with ilp processors. *Computer*, 35, 2 (2002), 40–49.

[13] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter. 2006. FastMP: A multi-core simulation methodology. In *The Annual Workshop on Modeling, Benchmarking and Simulation (MOBS)*.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*. Vol. 40. ACM, 190–200.

[15] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *IEEE Comput.* 35, 2 (2002), 50–58.

[16] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi. 2014. ParTejas: A parallel simulator for multicore processors. In *ISPASS (Poster)*.

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News* 33, 4 (2005), 92–99.

[18] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA*. 1–12.

[19] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, M. D. Hill, D. A. Wood, S. Huss-Lederman, and J. R. Larus. 2000. Wisconsin wind tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8, 4 (2000), 12–20.

[20] G. L. Peterson. 1981. Myths about the mutual exclusion problem. *Inform. Process. Lett.* 12, 3 (1981), 115–116.

[21] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. 1993. The Wisconsin wind tunnel: Virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.* 21, 1 (1993), 48–60.

[22] M. Rosenblum, S. A. Herod, E. Witchel, and A. Gupta. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel Distrib. Technol. Syst. Appl.* 3, 4 (1995), 34–43.

[23] P. Sack. 2004. SESC: SuperESCalar SimulatorRetrieved from *http://iacoma.cs.uiuc.edu/ paulsack/sescdoc/*.

[24] D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 475–486.

[25] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. 2015. Tejas: A java based versatile micro-architectural simulator. In *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 47–54.

[26] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. R. Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. ACM, 335–344.

[27] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. 2006. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26, 4 (2006), 18–31.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23 (May 1995), 24–36.

[29] M. T. Yourst. 2007. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*. 23–34.

[30] G. Zheng, G. Kakulapati, and L. V. Kale. 2004. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium*. IEEE, 78.

Received March 2016; revised March 2017; accepted April 2017