ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems

Daniel Sanchez
Massachusetts Institute of Technology
sanchez@csail.mit.edu

Christos Kozyrakis
Stanford University
christos@ee.stanford.edu

ABSTRACT

Architectural simulation is time-consuming, and the trend towards hundreds of cores is making sequential simulation even slower. Existing parallel simulation techniques either scale poorly due to excessive synchronization, or sacrifice accuracy by allowing event reordering and using simplistic contention models. As a result, most researchers use sequential simulators and model small-scale systems with 16-32 cores. With 100-core chips already available, developing simulators that scale to thousands of cores is crucial.

We present three novel techniques that, together, make thousand-core simulation practical. First, we speed up detailed core models (including OOO cores) with instruction-driven timing models that leverage dynamic binary translation. Second, we introduce bound-weave, a two-phase parallelization technique that scales parallel simulation on multicore hosts efficiently with minimal loss of accuracy. Third, we implement lightweight user-level virtualization to support complex workloads, including multiprogrammed, client-server, and managed-runtime applications, without the need for full-system simulation, sidestepping the lack of scalable OSs and ISAs that support thousands of cores.

We use these techniques to build zsim, a fast, scalable, and accurate simulator. On a 16-core host, zsim models a 1024-core chip at speeds of up to 1,500 MIPS using simple cores and up to 300 MIPS using detailed OOO cores, 2-3 orders of magnitude faster than existing parallel simulators. Simulator performance scales well with both the number of modeled cores and the number of host cores. We validate zsim against a real Westmere system on a wide variety of workloads, and find performance and microarchitectural events to be within a narrow range of the real system.

1. INTRODUCTION

Computer architects rely extensively on simulation to explore and evaluate future architectures. Unfortunately, accurate microarchitectural simulation is time-consuming, and the trend towards chip-multiprocessors with a large number of cores is making conventional sequential simulators even slower. For example, gem5, Flexus, and MARSS [4, 29, 51] achieve simulation speeds of about 200 KIPS. At this speed, it takes around eight hours to simulate one core for a second. Simulating a thousand-core chip for one second would take almost a year.

Ideally, a simulator should satisfy four desirable properties: it should be *accurate*, *fast*, *execute a wide range of workloads*, and be *easy to use and modify*. Sequential software simulators are not fast enough for large-scale multi-core chips. The two main alternatives, FPGA-based simulators and parallel software simulators, make different trade-offs, but none of them satisfies all four requirements. FPGA-based simulation frameworks are fast and accurate [12, 47], but they take significant effort to develop and use. Parallel simulation seems a natural fit for highly parallel chips, but poses significant difficulties as well. Simulators based on parallel discrete event simulation (PDES) [10, 34] suffer from poor scalability due to frequent synchronization. Other parallel simulators [8, 27] relax synchronization requirements to achieve scalability. In doing so, they allow microarchitectural events to occur out of order, sacrificing accuracy and making it difficult to model the actual behavior of contented resources such as shared caches and memory controllers. As a result, most researchers still use sequential simulators and limit their studies to architectures with 16-32 cores. With chips that feature hundreds of threads and cores already on the market [45, 48], developing simulation techniques that scale efficiently into the thousands of cores is critical.

In this paper, we introduce three techniques that, together, enable fast, accurate, and scalable simulation:

1. We speed up sequential simulation by 1-2 orders of magnitude by developing *instruction-driven timing models* that leverage *dynamic binary translation (DBT)* to perform most of the work in a core's timing model during program instrumentation, reducing the overheads of conventional cycle-driven or event-driven core models. As a result, we can simulate a Westmere-class OOO core including features such as branch prediction, limited fetch and issue widths, and µop fission, at 20 MIPS, 10-100× faster than conventional simulators. We can also run a simpler $IPC = 1$ core model at up to 90 MIPS.
2. We introduce *bound-weave*, a two-phase parallelization algorithm that scales parallel simulation without significant overheads or loss of accuracy. We observe that, over small intervals of a few hundreds to thousands of cycles, instructions from different cores interact timing-wise on shared resources, but their interactions rarely affect the microarchitectural components accessed by each instruction (e.g., the caches accessed to serve a miss). We exploit this in-

Simulator	Engine	Parallelization	Detailed core	Detailed uncore	Full system	Multiprocess apps	Managed apps
gem5/MARSS	Emulation	Sequential	OOO	Yes	Yes	Yes	Yes
CMPSim	DBT	Limited skew	No	MPKI only	No	Yes	No
Graphite	DBT	Limited skew	No	Approx contention	No	No	No
Sniper	DBT	Limited skew	Approx OOO	Approx contention	No	Trace-driven only	No
HORNET	Emulation	PDES (p)	No	Yes	No	Trace-driven only	No
SlackSim	Emulation	PDES (o+p)	OOO	Yes	No	No	No
ZSim	DBT	Bound-weave	DBT-based OOO	Yes	No	Yes	Yes

Table 1: Comparison of microarchitectural simulators: techniques, timing models, and supported workloads.

sight by dividing parallel simulation in two phases. We first simulate cores in parallel for an interval of a few thousand cycles, ignoring contention and using zero-load (i.e., minimum) latencies for all memory accesses, and record a trace of memory accesses (including caches accessed, invalidations, etc.). In the second phase, we perform parallel, event-driven simulation of the interval to determine the actual latencies. Since the possible interactions between memory accesses are known from the first phase, this timing simulation incurs much lower synchronization overheads than conventional PDES techniques, while still being highly accurate.

3. Due to the lack of scalable OS or ISA support beyond tens of cores, thousand-core simulations must be user-level for now. However, user-level simulators support a limited range of workloads. We implement *lightweight user-level virtualization*, a technique that uses DBT to give user processes a virtualized system view, enabling support for multiprogrammed, managed-runtime, and client-server workloads that are beyond the realm of conventional user-level simulators.

We use these three techniques to build *zsim*, a parallel multicore simulator designed from the ground up to leverage the capabilities of current multicore systems. ZSim is fast, accurate, and scalable. It can model in-order and out-of-order cores, heterogeneous systems with arbitrarily configured memory hierarchies, includes detailed timing models, and can run a wide range of x86 workloads. On a 16-core Xeon E5 host, zsim simulates a 1024-core chip at up to 1,500 MIPS with simple timings models, and up to 300 MIPS with detailed, validated core and uncore models. This is about 2-3 orders of magnitude faster than existing parallel simulators like Graphite and Sniper [8, 27], and around 4 orders of magnitude faster than sequential simulators like gem5 and MARSS [4, 29]. ZSim's performance scales well with the number of simulated and host cores. We validate zsim against existing systems and find its performance and microarchitectural events, such as MPKI, to be commonly within 10% of Westmere-class chips.

The rest of the paper is organized as follows. Section 2 presents an overview of simulation techniques. Section 3 describes the three techniques underpinning zsim. Section 4 presents a comprehensive evaluation of zsim's accuracy and performance. Finally, Section 5 concludes the paper.

2. SIMULATION TECHNIQUES

Despite recent progress on fast and general FPGA-based simulators [12, 19, 31, 47], these frameworks require substantial investment in specialized multi-FPGA boards [50] and are difficult to use for architectural exploration as they involve complex FPGA toolchains with hours-long compiles. Consequently, we focus on software-based simulation.

Simulation techniques can be classified along many dimensions: user-level vs full-system; functional vs timing; trace-driven vs execution-driven; cycle-driven vs event-driven [26]. We focus on *execution-driven*, *timing* simulators, which are the preeminent tool for evaluation of future designs [1]. With the goals of *high sequential performance*, *scalability*, *accuracy* and *usability* in mind, we categorize simulation techniques along the axes discussed in the following paragraphs. We focus on techniques instead of simulators; each simulator implements a different mix or variation of these techniques. Table 1 compares the key characteristics and features of representative, current simulators.

Emulation vs instrumentation: The choice between emulation and instrumentation is a crucial factor for simulation performance. An *emulation-based* simulator decodes each instruction and invokes both functional and timing models with some coordination between the two [26]. Emulation is the natural choice when the simulated ISA is different from the host's, and many simulators choose it to be portable. gem5 [4, 5, 25], Flexus [51] and MARSS [29] are current representative emulation-based systems. They can simulate up to 200 KIPS with detailed core and memory models and up to few MIPS with simpler timing models.

An *instrumentation-based* or *direct-execution* simulator adds instrumentation calls to the simulated binary, e.g. instrumenting every basic block and memory operation to drive the timing model. Instrumentation leverages the host machine it runs on to perform functional simulation, and can be much faster than emulation if the timing models are also fast. Moreover, it eliminates the need for a functional model, which can be difficult for complex ISAs. Early instrumentation-based simulators such as WWT [33] used static binary translation, while Embra [52] was the first to leverage more powerful dynamic binary translation. Since x86 is prevalent in the desktop and server segments, and the impact of ISA is less relevant due to sophisticated CISC to RISC µop decoding [6], several recent simulators use instrumentation. CMPSim [18] (unreleased) and Graphite [27] achieve about 10 MIPS with simple timing models. Instrumentation-based systems are often extended with capabilities such as magic NOPs to model new instructions [25] and memory shadowing to simulate speculative techniques (OOO cores or transactional memory) and fault injection [27, 42].

Parallelization strategies: Given the widespread availability of multicore hosts, parallel simulation is an attractive way to improve performance. However, parallelizing a simulator without sacrificing its accuracy is challenging. Parallel microarchitectural simulators are specialized parallel discrete event simulators (PDES) [14]. In PDES, events are distributed among host cores and executed concurrently while maintaining the illusion of full order. PDES algorithms are either pessimistic, synchronizing every time an ordering violation may happen, or optimistic, executing speculatively and rolling back on ordering violations. HORNET is a pessimistic PDES simulator [34], while SlackSim (unreleased) uses both pessimistic and optimistic PDES [10].

Multicore timing models are extremely challenging to parallelize using pessimistic PDES, as cores and caches are a few cycles away and interact often. Hence, preserving full accuracy adds significant overheads. Pessimistic PDES only scales when the timing models for each component are so detailed that simulation is very slow to begin with [34]. Optimistic PDES simulators are hard to develop as timing models need to support abort and rollback, and incur large execution time overheads [9]. As a result, PDES simulators are not competitive with performant sequential simulators.

Approximation strategies: Since accurate PDES is hard, a scalable alternative is to relax accuracy and allow order violations. Graphite [27] simulates cores in parallel allowing memory accesses to be reordered, keeping their slack limited to a few thousand simulated cycles. HORNET and SlackSim can also operate in this mode. Unfortunately, contention cannot be modeled using accurate microarchitectural models because events arrive out of order. Graphite simulates contention using queuing theory models, which prior research has shown to be inaccurate [46], as will we in Section 4.

Approximation techniques are also used to accelerate the simulation of individual components. For instance, architects often use *IPC=1* core models instead of detailed OOO core models to evaluate caching optimizations. Sniper extends Graphite with an OOO model using interval simulation, an approximate technique that only models ROB stalls due to misses and mispredictions. Sniper achieves 1-2 MIPS on an 8-core host [8]. CMPSim reaches up to 10 MIPS on a 1-core P4, but it simulates cache MPKIs [18], not timing.

Sampling: Robust statistical sampling [51, 53] and automated techniques to simulate a small, representative portion of execution [44] are also widely used to reduce simulation time of long-running workloads. These techniques are complementary and orthogonal to the need for fast simulation. Even with sampling, conventional simulators would take months to warm up and simulate a thousand-core chip for tens of millions of cycles, the typical sample length.

Concurrent simulations: Architects scale out to parallel clusters using sequential simulators by running independent simulations. This is often insufficient, much like running multiple slow sequential programs does not eliminate the need for fast parallel programs. Concurrent simulations are practical for large design space explorations or sensitivity studies, but the design process often relies on running one or a few latency-critical simulations at a time.

Full-system vs user-level simulation: Full-system simulators model features such as privileged modes and emulate peripheral devices in order to support an OS. They can run complex multithreaded and multiprocess workloads, and applications that exercise networking and I/O. gem5, Flexus, and MARSS take this approach.

User-level simulators model faithfully only the user portion of applications. They are much easier to develop and use as there is no need for device timing models, large disk images, or booting an OS. However, they typically only support basic workloads. For example, Graphite can only simulate a single multithreaded application. Sniper supports multiple single-threaded applications but only in trace-driven mode. Applications often need to be modified or compiled against special libraries (e.g., Graphite). Apart from not faithfully modeling workloads with significant OS time, many user-level simulators cannot run workloads that barely use the OS. For example, they cannot run applications that launch more threads than cores, common in managed runtimes such as the JVM, or workloads that are sensitive to system time, e.g., client-server workloads would time out as simulated time advances much more slowly than real time.

Since zsim targets simulation of thousand-core chips, it has to be a user-level simulator for now. No current mainstream OS scales to thousands of cores, and ISAs also limit the number of cores. For example, x86's xAPIC only supports up to 256 cores. To achieve full-system simulation at that scale, we would first need to perform significant research on the OS and develop custom ISA modifications. Instead, we use virtualization techniques that allow us to run a much wider set of workloads than existing frameworks, including multithreaded, multiprogrammed, managed (Java, Scala, Python), and client-server applications.

Several trends suggest that user-level simulation is sufficient for many experiments with thousand-core systems. First, a prevailing approach for multicore OSes is to assign groups of cores to each application [7, 21], having user-level runtimes schedule work within each group [28, 40]. OS tasks execute on a separate group of cores to minimize interference with other workloads [32]. Second, user-level networking stacks [49] are increasingly popular due to their lower overheads, allowing user-level simulators to capture most networking effects. For example, the PARSEC suite [3] features workloads with user-level TCP/IP. Finally, important system-related performance effects such as TLB misses and page table walks can be modeled without running an OS.

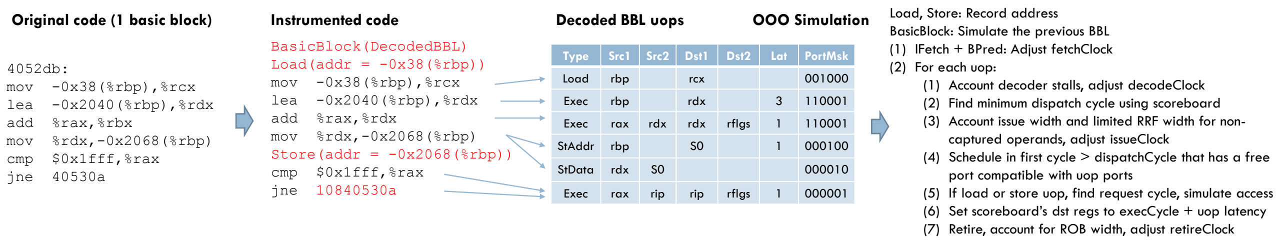
3. ZSIM TECHNIQUES

We now present the three simulation techniques that allow zsim to achieve high speed and high accuracy when modeling thousand-core chips. First, we accelerate the simulation of processor cores using dynamic binary translation and instruction-driven timing models. Next, we parallelize multicore simulation using the two-phase bound-weave algorithm, which breaks the trade-off between accuracy and scalability. Finally, we describe lightweight user-level virtualization, which allows us to accurately simulate most of the workloads that full-system simulators can handle.

3.1 Fast Sequential Simulation using DBT

To make modeling thousand-core chips practical, we need a 100-1,000× speedup over existing simulators. Parallelization alone is insufficient to reach such speedups, as current hosts have a few tens of cores. Fortunately, we can also speed up sequential simulation by 10-100×. ZSim uses an instrumentation-based approach to eliminate the need for functional modeling of x86. Specifically, we use Pin [22] to perform dynamic binary translation (DBT). With DBT, we can push most of the work done by timing models to the instrumentation phase, doing it once per instrumented instruction instead of every time the instruction is simulated. Our insight is to structure detailed timing models for OOO cores in a manner that maximizes this opportunity.

Simple core model: For a simple core model (*IPC=1* for all but load/store instructions), we instrument each load, store, and basic block to call into the core's timing model. The timing model simply keeps a cycle count, instruction count, and drives the memory hierarchy. Instruction fetches,



```
4052db:
mov   -0x38(%rbp),%rcx
lea   -0x2040(%rbp),%rdx
add   %rax,%rbx
mov   %rdx,-0x2068(%rbp)
cmp   $0x1fff,%rax
jne   40530a
```

```
BasicBlock(DecodedBBL)
Load(addr = -0x38(%rbp))
mov   -0x38(%rbp),%rcx
lea   -0x2040(%rbp),%rdx
add   %rax,%rdx
mov   %rdx,-0x2068(%rbp)
Store(addr = -0x2068(%rbp))
cmp   $0x1fff,%rax
jne   10840530a
```

Type	Src1	Src2	Dst1	Dst2	Lat	PortMsk
Load	rbp		rcx			001000
Exec	rbp		rdx		3	110001
Exec	rax	rdx	rdx	rflgs	1	110001
StAddr	rbp		S0		1	000100
StData	rdx	S0				000010
Exec	rax	rip	rip	rflgs	1	000001

Load, Store: Record address
BasicBlock: Simulate the previous BBL
(1) IFetch + BPred: Adjust fetchClock
(2) For each uop:
  (1) Account decoder stalls, adjust decodeClock
  (2) Find minimum dispatch cycle using scoreboard
  (3) Account issue width and limited RRF width for non-captured operands, adjust issueClock
  (4) Schedule in first cycle > dispatchCycle that has a free port compatible with uop ports
  (5) If load or store uop, find request cycle, simulate access
  (6) Set scoreboard's dst regs to execCycle + uop latency
  (7) Retire, account for ROB width, adjust retireClock

Figure 1: OOO core modeling in zsim using µop decoding at instrumentation time and instruction-driven timing simulation.

loads, and stores are simulated at their appropriate cycles by calling into the cache models, and their delays are accounted in the core's cycle count. This model can achieve up to 90 MIPS per simulated core (see Section 4).

OOO core model: Unfortunately, the simple core model is not representative of the OOO cores used in server, desktop, and mobile chips. Hence, we developed an OOO core model that closely follows the Westmere microarchitecture. We model branch prediction, instruction fetch including wrong-path fetches due to mispredictions; instruction length predecoder and decoder stalls, instruction to µop decoding, macro-op fusion; issue stalls, limited issue window width and size, register renaming with limited bandwidth; dataflow execution with accurate µop to execution port mappings, µop latencies, and functional unit contention; load-store ordering and serialization including load forwarding, fences, and TSO; and a reorder buffer of limited size and width. In the interest of speed, we do not model some features, mainly mispredicted instruction execution beyond instruction fetches. Since Westmere recovers from mispredictions in a fixed number of cycles and seems to cancel in-flight data misses, we can model the primary effect of mispredictions without executing wrong-path instructions. We also do not model a few components that would not add major overheads, but that seem to have a minor performance effect in Westmere for most or all of the workloads we have studied: the BTB, the loop stream detector, an RRF of limited size, and micro-sequenced instructions. To closely match the Westmere core, we used Fog's microarchitectural analysis [13] and an extensive suite of microbenchmarks that use performance counters to pinpoint inaccuracies and derive exact instruction-µop decodings, ports, and latencies. After extensive validation (see Section 4), we believe that most of the small discrepancies between the modeled core and the real core stem from unknown Westmere details, such as the branch predictor organization or the exact decoder organization. Note that the parameters of the OOO model can be configured to simulate other existing or proposed cores.

Conventional simulators with detailed OOO core models execute at around 100 KIPS [4, 8]. ZSim accelerates OOO core models by pushing most of the work into the instrumentation phase, as shown in Figure 1. We perform instruction-µop decoding at instrumentation time, and create a basic block descriptor that includes a µop representation in a format optimized for the timing model. The format includes µop type, feasible execution ports, dependencies, and latency. For simplicity, we produce a generic, approximate decoding of x86 instructions that are barely used (e.g., x87). These are 0.01% of dynamic instructions in our benchmarks. Additionally, we find the delays introduced by the in-order frontend (instruction length predecoder and 4-1-1-1 decoders) at instrumentation time. We interface with Pin's translation cache so that, when Pin invalidates a code trace, we free the corresponding translated basic blocks.

Moreover, we structure the OOO core model in an *instruction-driven* manner. We call the core model once per µop, simulating all stages for the µop. Each stage (fetch, decode, issue, retire) maintains a separate clock. As each µop passes through each stage, it alters the stage clocks. To model OOO execution, we maintain a window of future cycles relative to the issue clock, keeping track of the execution ports used by previously scheduled µops. Accuracy is maintained by tracking interdependencies between stage clocks. For example, when a µop fills the ROB, the issue stage clock is increased to the cycle when the head-of-line µop is retired. We essentially stall the next µop at the issue stage until an ROB entry is available. We track similar dependencies between stage clocks to model mispredictions, I-cache misses, decoder stalls, and issue stalls.

The instruction-driven approach uses the µop stream to drive the timing model directly, avoiding the overheads of conventional cycle-driven models, where each pipeline stage maintains its per-cycle microarchitectural state and is simulated every cycle to advance its state. It also avoids the overheads of event-driven models, where cores or individual stages control their timing by enqueuing timestamped events into a priority queue. The combination of DBT and instruction-driven models allows us to achieve 20 MIPS per simulated core, and slowdowns of about 200×. This is faster than Sniper [8], which uses approximate OOO models (though slower memory system simulation likely plays a role in the difference), and is comparable to FastSim, a cycle-accurate OOO model based on pipeline state memoization [42]. However, FastSim is significantly more complex: pipeline state must be carefully encoded to make memoization practical, requiring a domain-specific language to do so with reasonable effort [43].

The instruction-driven approach imposes only two restrictions. First, the schedule of a given µop must not depend on future µops. In other words, we assume that the instruction window prioritizes the µop that comes first in program order, which is what OOO cores typically do. Second, the execution time of every µop must be known in advance. This is not the case for memory accesses, due to cache misses and contention on shared caches. For now, we assume uncontended memory accesses, and address contention in Section 3.2.

3.2 Scalable and Accurate Parallel Simulation

Even though parallel simulation is a natural fit for modeling multi-core chips, conventional parallelization schemes suffer from limited scalability or limited accuracy, depending on how they model the interactions between concurrent ac-

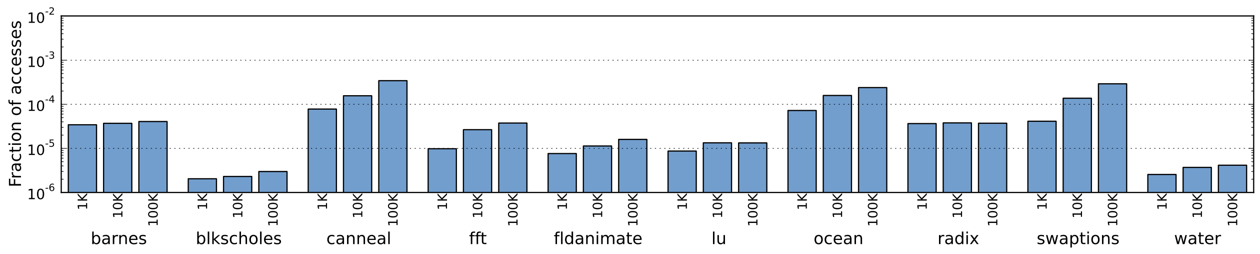


Figure 2: Fraction of memory accesses that cause path-altering interference for 1, 10 and 100 Kcycle intervals. The simulated system has 64 cores, private split 32KB L1s, an unified private L2, and a 16-bank 16-way shared L3. Note the log scale.

cesses from different cores. We present an event-driven parallelization technique that breaks this trade-off. Our main insight is that, at a small timescale (∼1,000 cycles), *most concurrent accesses happen to unrelated cache lines.* When accesses are unrelated, simulating them approximately and out of order first, then simulating their detailed timing in order, is equivalent to simulating them completely in order.

Characterizing interference: We categorize concurrent interfering accesses in two classes. Two accesses suffer from *path-altering interference* if simulating them out of order changes their paths through the memory hierarchy. For example, two writes to the same line from different cores suffer from path-altering interference, as simulating them out of order affects the misses and invalidations simulated. Path-altering interference occurs when either (a) two accesses address the same line, except if they are both read hits, or (b) the second access, if simulated out of order, evicts the first access's line, causing it to miss. In contrast, two accesses suffer from *path-preserving interference* if simulating them out of order changes their timing, but does not affect their paths through the memory hierarchy, including the resulting messages and transitions such as evictions and invalidations. For example, two accesses to different cache sets in the same bank suffer from path-preserving interference.

Our insight is that, if we allow reorderings only within a small interval, e.g., within 1,000 cycles, path-altering interference is exceedingly rare. Figure 2 shows the fraction of accesses that suffer from path-altering interference when simulating a 64-core chip with private L2s and a shared 16-way L3, using intervals of 1K, 10K, and 100K cycles on PARSEC and SPLASH-2 applications, many with frequent read-write sharing and contended critical sections. Path-altering interference is negligible with small intervals (note the logarithmic scale). Practically all path-altering interference is due to same-line accesses; interference due to evictions is extremely rare unless we use shared caches with unrealistically low associativity (1 or 2 ways). Finally, only a subset of the accesses with path-altering interference can affect the *functionality* of the simulated program (e.g., by altering the lock acquisition order). Consequently, this functional interference is also rare. Prior work has observed this effect, and exploited it to e.g., speed up deterministic replay [15].

Bound-weave overview: Since path-altering interference is rare, we maintain accuracy only on path-preserving interference. To this end, we develop the *bound-weave algorithm*, which operates in two phases. The simulation proceeds in small intervals of a few thousand cycles each. In the *bound phase* of the interval, each core is simulated in parallel assuming no interference at all, injecting zero-load latencies into all memory accesses and recording their paths through the memory hierarchy. The bound phase places a lower bound on the cycle of each microarchitectural event. The second phase, called the *weave phase*, performs parallel microarchitectural event-driven simulation of these accesses, weaving together the per-core traces from the bound phase, and leveraging prior knowledge of the events to scale efficiently. Figure 3 gives an overview of this scheme. We also profile accesses with path-altering interference that are incorrectly reordered. If this count is not negligible, we (for now, manually) select a shorter interval.

3.2.1 Bound Phase

In the bound phase, zsim uses one host thread per simulated core. For each thread, it instruments all loads, stores and basic blocks to drive the timing models. Threads execute in parallel and sync on an *interval barrier* when their cores' simulated cycles reach the interval length (e.g., 10K cycles). The barrier serves three purposes:

1. *Limit the skew between simulated cores:* Barriers are the simplest and most common approach to bound the skew between cores [11, 33]. While Graphite uses peer-to-peer synchronization to limit skew [27], this is only advantageous for simulations spread across loosely coupled clusters, where synchronization is much more expensive. On a multicore host, barriers barely constrain parallelism.
2. *Moderate parallelism:* The barrier only lets as many threads as host hardware threads run concurrently. For a 1024-core simulation on a host with 32 hardware threads, the barrier only wakes up 32 threads at the beginning of each interval. When a thread finishes its interval, it wakes up the next available thread for that interval, until no more threads are left. This greatly improves scalability since it avoids overwhelming the host OS scheduler.
3. *Avoid systematic bias:* At the end of each interval, the barrier randomly shuffles the thread wake-up order. This avoids consistently prioritizing a few threads, which in pathological cases can cause small errors that add up to significant inaccuracies. It is also a natural way to add non-determinism, which improves robustness [2].

Memory hierarchy: While the bound phase allows skews among cores, we still need a model for the memory hierarchy that maintains coherence in the presence of concurrent accesses to potentially conflicting addresses. Other parallel simulators use message passing between concurrent timing models to address this issue [27, 33]. We develop a shared-memory approach that is better suited to multicore hosts.

Each cache model contains a fully decoupled associative array, replacement policy, coherence controller, and weave timing model. This sacrifices some performance, but enhances modularity and ease of use. We maintain coherence

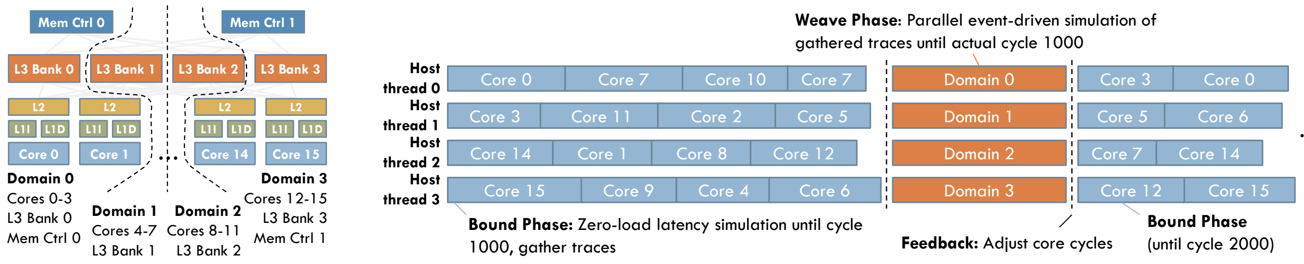


Figure 3: Bound-weave simulation of a 16-core chip using 4 host cores. Microarchitectural components (shown left) are divided across four domains. In the bound phase of each interval, cores are simulated in parallel assuming no contention. The weave phase uses one thread per domain to simulate detailed timing. At the end of the interval, the cycle counts of each core are adjusted to account for the additional latency modeled in the weave phase (which is always ≥ 0).

in the order we simulate the accesses in the bound phase. This is inaccurate only for requests to the same line, which we have shown are rare. On an access, the core first calls into its L1, which, on a miss, calls into the next level of the hierarchy. Each cache level can trigger invalidations to lower levels and evictions to upper levels. Because accesses go both up and down the hierarchy, conventional locking schemes that rely on maintaining a fixed order for lock acquisitions are not applicable, and would deadlock. Instead, we develop a custom protocol that relies on the communication patterns through the memory hierarchy. Each cache has two locks, one for accesses up (fetches and writebacks), and another for accesses down (invalidations and downgrades). Accesses acquire these locks using a scheme similar to spider locking that prioritizes accesses down over accesses up. Accesses up are exposed to only one kind of race, receiving an invalidation while traversing cache levels. This scheme is deadlock- and livelock-free, and avoids global or coarse-grained locks, allowing accesses to different caches and banks to proceed concurrently.

Tracing: As accesses are simulated, each core, cache and memory controller model records an event trace to be used in the weave phase. We only record accesses that miss beyond the private levels of the cache hierarchy. Contention in those levels is predominantly due to the core itself, so it is modeled in the bound phase. Events are generated frequently and dynamically, so we use per-core slab allocators and recycle slabs in LIFO order as soon as their interval is fully simulated to avoid the performance overheads and scalability issues of generic dynamic memory allocation.

3.2.2 Weave Phase

The weave phase performs parallel event-driven simulation of the per-core event traces from the bound phase, dividing events among parallel event queues, and simulating them in full order. The basic problem of conventional PDES-based engines is the need for very frequent synchronization between event queues, as any event may in principle produce an event for another queue, for a time only a few cycles ahead. By exploiting prior knowledge about event dependencies, *we synchronize only when needed.*

Events: Figure 4 shows an example event trace for core 0, including events from different L3 banks and memory controllers. Each event is associated with a specific component (e.g., a cache bank), and its dependencies are fully specified: each event has one or more parents, and zero or more children. An event cannot be executed until all its parents have finished. When the last parent finishes, the event is enqueued into a priority queue, to be executed at its appropriate cycle, determined by the parent's finish cycle and a fixed delay between the parent and child. Once the event is dequeued for execution, it can either be requeued at a later cycle, or marked as completed. At that point, it notifies all its children, which may become available for execution. Each timing model defines its own events. This is similar to a conventional event-driven framework, except that all events and dependencies are pre-specified in the bound phase, and each event has a lower bound on the cycle it can be executed.

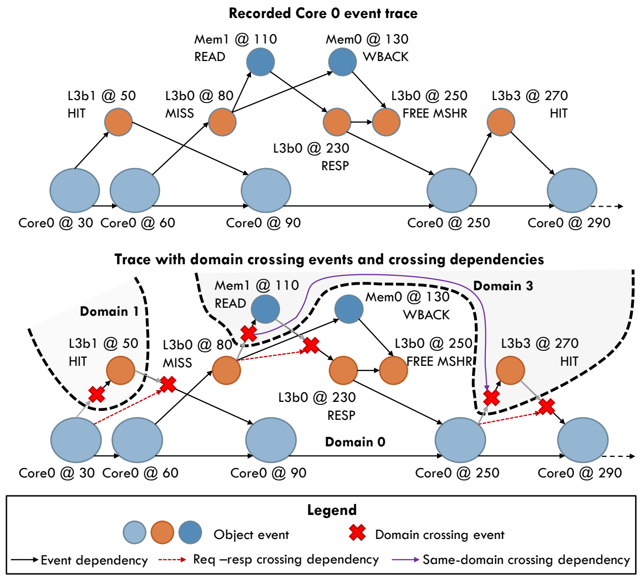


Figure 4: Example event trace from bound phase (of core 0 in Figure 3), and its division in domains with domain crossing events inserted for simulation in the weave phase.

Domains: We partition all simulated components (caches, cores, memory controllers) and their events in multiple *domains*. Currently, this assignment is static, and simply partitions the system in vertical slices. Figure 3 shows an example partitioning of a chip with 16 cores, four L3 banks and two memory controllers in four domains. We parallelize event-driven execution across domains: each domain has its own event priority queue, and is driven by a separate thread. Each thread simply executes the events in its domain's priority queue, enqueuing their children when they terminate.

Domain crossings: Domains synchronize when a parent and a child are in different domains (e.g., a core in one domain issued a request to a cache bank in a different domain). When this happens, we introduce a *domain crossing event* and enqueue it in the child's domain. Figure 4 shows the original trace augmented with domain crossing events (red X's). When the crossing is executed in the child's domain, it checks whether the parent has finished. If so, the crossing just enqueues the child, which can now be run. Otherwise, the crossing requeues itself on a later cycle, the current cycle of the parent's domain plus the delay between parent and child. Since we have lower bounds on the execution times of all events, this is accurate, as we are guaranteed that the child will never be available before the crossing runs and checks for the parent again. This way, delays introduced by contention propagate across domains only when there is an actual dependency.

To improve performance, we introduce dependencies between crossings, delaying when they become executable to avoid premature synchronization between domains. Figure 4 shows example dependencies with red and purple arrows. First, for request-response traffic, we make the response's crossing depend on the event that generates the request. In our example, when the L3's bank 0 (L3b0, domain 0) produces a read request for memory controller 1 (Mem1, domain 3), we insert crossing events for the request (0→3) and the response (3→0), and make the response crossing a child of the miss event. If, due to contention, domain 0 gets delayed to the point where L3b0's miss happens after cycle 250, by doing this we avoid needlessly executing the response's crossing starting at cycle 250. Second, we insert dependencies on crossings in the same domain caused by serialized accesses from the same core. In our example, this introduces a dependency between Mem1's and L3b3's crossings. If core 0 suffers a large delay, this avoids having multiple domain 3 crossings synchronizing with domain 0 at the same time.

Termination and feedback: Each domain terminates the current interval when the first event in its queue is over the interval's limit (e.g., 1K cycles). Domains stop at the limit instead of running through their event queue. Because events are lower-bounded, no domain runs out of events, and in the next interval, the bound phase cannot produce events for a clock cycle that precedes the stopping point. When the interval ends, the core's internal cycle counters are adjusted to account for the delay caused by contention simulation, i.e., the difference between the initial lower bound on the execution cycle of the last simulated event at each core and its actual execution cycle. In our example, if the last consumed event from core 0 was simulated at cycle 750 in the bound phase, and at cycle 900 in the weave phase, we advance its clock 150 cycles.

Models: We develop detailed event-driven weave models for memory controllers (including DDR3 timings, access scheduling, and conflicts), pipelined caches (including address and data port contention, and limited MSHRs), and IPC1 and OOO core models that account for increased latencies due to contention to retime their accesses. The only component without a weave phase model is the network, since well-provisioned NoCs can be implemented at modest cost, and zero-load latencies model most of their performance impact in real workloads [41]. We leave weave phase NoC models to future work.

Limitations: Bound-weave relies on path-altering interference being rare. While this is true for cores that communicate implicitly through the cache hierarchy, other communication styles, e.g., extremely fine-grain message passing, will require significant changes or a new parallelization scheme.

3.3 Simulating Complex Workloads

As explained in Section 2, developing a full-system simulator for thousand-core chips would run into limitations of current mainstream OSes and ISAs. This leaves user-level simulation as the only viable choice. However, conventional user-level simulators cannot support many modern workloads, such as client-server workloads, multiprocess applications, or managed language runtimes that use more threads than the number of simulated cores. We extend zsim with a set of mechanisms that collectively provide *lightweight user-level virtualization*, enabling most workloads to run unmodified on zsim, bridging the gap with full-system simulators.

Multiprocess simulation: ZSim naturally supports multithreaded programs. To support multiple processes, each process attaches to a shared memory segment, and uses it as a global heap. All simulation state is allocated in this heap, and we align the mappings of both the shared heap and the library code segments across all processes. With this technique, zsim can still be programmed as a single multithreaded application, even though the simulated threads come from different processes. We also instrument fork() and exec() system calls to capture full process hierarchies (e.g., bash executing a Java program that launches several other commands).

Scheduler: Many applications launch more threads than cores, but are well tuned and do not rely on the OS scheduler for performance. For instance, the JVM runs garbage collection on a different set of threads, but does not overcommit the system. We implement a simple round-robin scheduler that supports per-process and per-thread affinities, allowing applications to launch an arbitrary number of threads [16].

Avoiding simulator-OS deadlock: Barrier synchronization alone may cause deadlock if the thread takes a syscall that blocks on the kernel waiting on another thread (such as futex_wait). All other threads will block on the interval barrier, waiting for the OS-blocked thread to return. The typical approach is to avoid those syscalls, often by modifying the workload [27]. Instead, we identify the small subset of blocking syscalls, and change our interval barrier to support join and leave operations. Threads that execute a blocking syscall leave the interval barrier, and join when they return to user-level code. Meanwhile, the simulation can advance. Non-blocking syscalls appear to execute instantaneously.

Timing virtualization: Many applications depend on accurate timing. These include self-profiling adaptive algorithms (e.g., adaptive back-off locks) or client-server applications with protocols that rely on timeouts. We virtualize the rdtsc (read timestamp counter) instructions, the few kernel interfaces (syscalls and vsyscalls) that return timing information, as well as sleep syscalls and syscalls with timeouts. Together, these isolate the timing of the instrumented process from the host, linking it to simulated time.

System virtualization: Even when we isolate their timing, instrumented processes can still see the host's hardware configuration. This is problematic for applications that self-tune to the system's resources (e.g., gcc's OpenMP and Oracle's JVM tune to the number of cores; Intel's MKL tunes to SSE extensions). We virtualize the process's system view by redirecting /proc and /sys open() syscalls to a pre-generated tree, and virtualizing the CPUID instruction and the few syscalls that give system-specific information, like getcpu().

Fast-forwarding and control: We use DBT to perform per-process fast-forwarding at close-to-native speeds, running trillions of instructions before starting the simulation. As in GEMS [25], simulated code can communicate with zsim via magic ops, special NOP sequences never emitted by compilers that are identified at instrumentation time.

Discussion: These virtualization mechanisms allow us to run a wide range of modern workloads. For example, we have used zsim to simulate JVM workloads like SPECJBB; h-store, a multiprocess, client-server workload that includes Java, C++ and Python components; and memcached with user-level TCP/IP. The remaining shortcoming of our approach is accounting for OS execution time. As we discussed in Section 2, large system time is mostly caused by either OS scalability artifacts (e.g., the VM subsystem), poor scheduling decisions (addressed with user-level scheduling and coarser-grain OS management [21]), or frequent networking (easily modeled with user-level networking stacks).

3.4 Flexibility and Usability

We architected zsim to be modular, highly configurable, and support heterogeneous systems. We support multiple core types running at the same time, with heterogeneous, multi-level cache hierarchies. For instance, we can model a multi-core chip with a few large OOO cores with private L1s and L2 plus a larger set of simple, Atom-like cores with small L1 caches, all connected to a shared L3 cache. We are currently working on fixed-function core models.

To help usability, we keep the code small by leveraging external infrastructure: Pin for DBT, XED2 for programmatic x86 instruction decoding, and HDF5 for stats. ZSim consists of 15K lines of C++, fewer than other simulators [4, 27]. While usability is hard to quantify, we have successfully used zsim in two computer architecture courses at Stanford, and students had little trouble using and modifying it.

4. EVALUATION

4.1 Accuracy

Methodology: We validate zsim against a real Westmere system. Table 2 shows the real system's configuration and the corresponding zsim configuration. We developed a profiler that uses ptrace and libpfm4 to run each application multiple times, record several relevant performance counters, and compute microarchitectural metrics (e.g., IPC, µops per cycle, cache MPKIs, etc.). We then compare these metrics against zsim's results. We execute profiling runs and simulations multiple times until every relevant metric has a 95% confidence interval of at most 1%. Since our infrastructure can only reliably profile one application at a time, we validate zsim using single- and multi-threaded applications.

Single-threaded validation: We validate zsim's OOO core model with the full SPEC CPU2006 suite. We run each application for 50 billion instructions using the reference (largest) input set. Figure 5 shows both real and simulated IPCs with applications sorted by *absolute performance error*, where $\mathrm{perf}_{error} = {}^{(IPC_{zsim} - IPC_{real})}/_{IPC_{real}}$. A positive error means zsim is overestimating the performance of the real system. As we can see, IPC differences are small: the average absolute perf_{error} is 9.7%, and in 18 out of the 29 benchmarks, zsim is within 10% of the real system.

HW	Xeon L5640 (6-core Westmere), 24GB DDR3-1333, no hyperthreading, turbo/DVFS disabled
SW	Linux 3.5 x86-64, gcc 4.6.2, Pin 2.12
Bound-weave	1000-cycle intervals, 6 weave threads
Cores	6 x86-64 OOO cores at 2.27GHz
L1I caches	32KB, 4-way, LRU, 3-cycle latency
L1D caches	32KB, 8-way, LRU, 4-cycle latency
L2 caches	256KB, 8-way, LRU, 7-cycle latency, private
L3 cache	12MB, 16-way, hashed, 6 2MB banks, 14-cycle bank lat, shared, inclusive, MESI coherence w/ in-cache directory, 16 MSHRs
Network	Ring, 1-cycle/hop, 5-cycle injection latency
Mem ctrl	1 controller, 3 DDR3 channels, closed page, FCFS scheduling, fast powerdown with threshold timer = 15 mem cycles [17]
DRAM	24GB, DDR3-1333, 2 4GB RDIMMs per channel

Table 2: Hardware and software configuration of the real system, and corresponding zsim configuration.

Figure 5 also shows scatter plots of MPKI errors for all the cache levels (MPKI = misses per thousand instructions). We define $MPKI_{error} = MPKI_{zsim} - MPKI_{real}$. Each dot represents the ($MPKI_{real}$, $MPKI_{error}$) of a single application. Note that the y-axis scale is much smaller than the x-axis. We observe that simulated cache MPKIs closely track the real MPKIs at all levels. Moreover, errors decrease as we move up in the hierarchy: while the L1D has an average $|MPKI_{error}|$ of 1.14, the L3 error is 0.30 MPKI. The few non-trivial errors are easily explained: all the workloads with L2 or L3 errors $\geq$ 0.1 MPKI have a non-negligible number of TLB misses, which we do not currently model. Page table walk accesses are also cached, affecting the reference stream and producing these errors. These results provide strong evidence for the correctness of our cache models.

Overall, while IPC errors are small, zsim has a certain tendency to overestimate performance. This is due to two main factors: the lack of TLB and page table walker models (which we will add soon), and inaccuracies in the frontend model. The modeled 2-level branch predictor with an idealized BTB has significant errors in some cases (see Figure 5), and we do not model the loop stream detector. Fortunately, these issues explain the larger errors: all applications with $\mathrm{perf}_{error} \leq -10\%$ either have TLB MPKI $\geq$ 1.0 or lose $\geq$ 30% of cycles to frontend stalls.

Finally, we observe that most instructions are decoded accurately: on average, only 0.01% of executed instructions have an approximate dataflow decoding, and the average absolute $\mathrm{\mu op}_{error} = {}^{(\mathrm{\mu ops}_{zsim} - \mathrm{\mu ops}_{real})}/_{\mathrm{\mu ops}_{real}}$ is 1.3%. This shows that implementing decoding for the most used opcodes and ignoring micro-sequenced instructions is accurate, as modern compilers only produce a fraction of the x86 ISA.

Multi-threaded validation: We use 23 multithreaded applications from multiple benchmark suites to validate the bound-weave algorithm and the coherence aspects of our cache hierarchy: 6 from PARSEC [3], 7 from SPLASH2, 9 from SPEC OMP2001, and the STREAM memory bandwidth benchmark. For SPLASH2 and PARSEC, we run with the configuration that takes closest to 20 billion instructions. For SPEC OMP, we run workloads for 100 billion instructions using the reference input sets. STREAM is run with $N = 10^8$ (a 2.2GB footprint). Figure 6 shows the difference between real and zsim performance. In this experiment, we run most workloads with 6 threads (fully loading the real system), but those that need a power-of-two threads run with 4 threads. We measure perf = ${}^{1}/_{time}$ (not IPC [2])

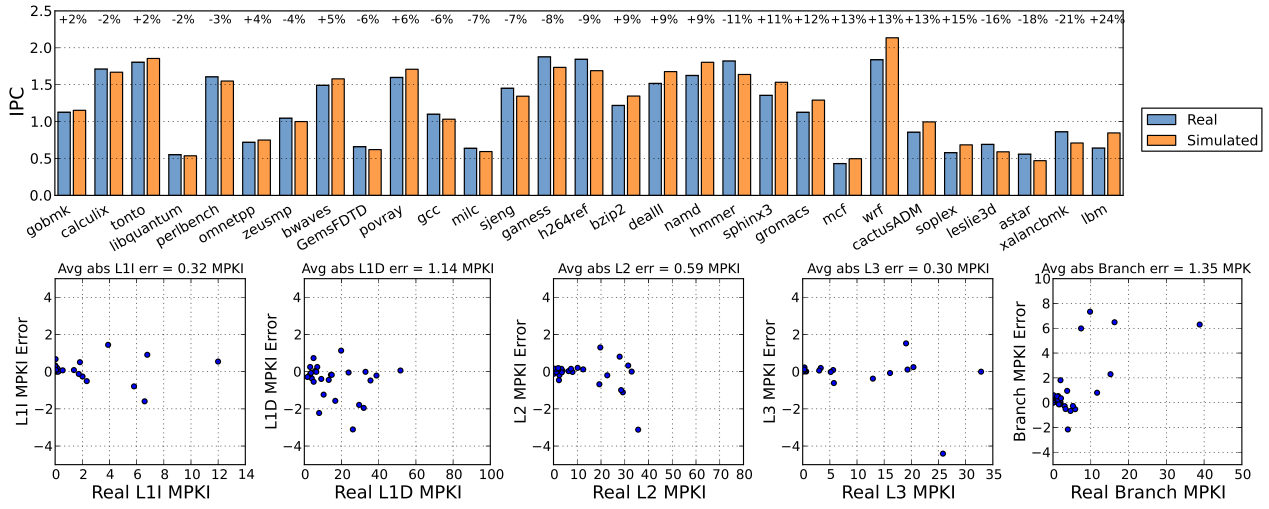


Figure 5: Validation of the OOO core model against a real Westmere machine on SPEC CPU2006: Simulated vs real IPC, and scatter plots of per-application MPKI errors (simulated - real MPKI) for all the caches and the branch predictor.

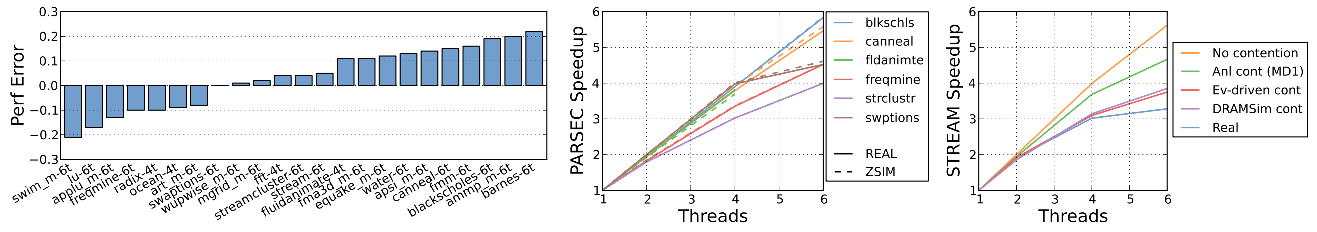


Figure 6: Validation of zsim with multithreaded apps: performance difference between zsim and the real system, real and simulated speedups of PARSEC benchmarks, and speedup of bandwidth-saturating STREAM with different contention models.

Tiles	16 cores/tile, 4/16/64 tiles (64/256/1024 cores)
Cores	x86 IPC1/OOO cores at 2GHz
L1I caches	32KB, 4-way, LRU, 3-cycle latency
L1D caches	32KB, 8-way, LRU, 4-cycle latency
L2 caches	4MB, 8-way, LRU, 8-cycle latency, shared per tile (16 cores), MESI coherence w/ in-cache directory
L3 cache	8MB bank/tile (32/128/512MB), 16-way, hashed, 12-cycle bank lat, fully shared, inclusive, MESI coherence w/ in-cache directory, 16 MSHRs
Network	Mesh, 1 router/tile, 1-cycle/hop, 2-stage routers
Mem ctrl	1 controller/tile , 2 DDR3 channels, same as Table 2 (assumes optical off-chip links as in [20])

Table 3: Parameters of the simulated tiled multicore chip.

and plot $\text{perf}_{error} = {}^{(\text{perf}_{zsim} - \text{perf}_{real})}/{\text{perf}_{real}}$ with applications sorted by performance error. Accuracy is similar to the single-threaded case: the average absolute perf_{error} is 11.2%, and 10 out of the 23 workloads are within 10%. Cache MPKI errors are also small, with L1I, L1D, L2, and L3 errors of 0.08, 1.18, 0.34, and 0.28 MPKI, respectively.

Much like in the single-threaded case, performance errors are mainly due to the lack of TLBs and frontend inaccuracies. Fortunately, these two effects often remain constant with the number of threads, so we can accurately predict parallel speedups. Figure 6 shows the speedups of all the PARSEC workloads from 1 to 6 threads. We observe an accurate match both when their scalability is limited due to lock contention (e.g., swaptions) or sequential portions (e.g., freqmine).

Contention models: The bound-weave algorithm allows for accurate modeling of contention using detailed microarchitectural models. Figure 6 illustrates this by showing STREAM's scalability on the real machine, as well as when simulated under several timing models: fully ignoring contention, an approximate queuing theory model, our detailed event-driven memory controller model, and DRAMSim2's model [35]. STREAM saturates memory bandwidth, scaling sublinearly. Ignoring contention makes STREAM scale almost linearly. The queuing theory model, which computes latency in the bound phase using the current load and the M/D/1 load-latency profile, tolerates reordered accesses but is still inaccurate. This matches the findings of prior work [46]. In contrast, using either our event-driven model or DRAMSim2 in the weave phase closely approximates the real machine, since both model details such as bank conflicts and DRAM timing constraints. Finally, note that the bound-weave algorithm makes it easy to integrate zsim with existing timing models. We integrated zsim with DRAMSim2 with 110 lines of glue code. In fact, we interfaced with DRAMSim2 before deciding to develop our own model for faster simulations. DRAMSim2 is cycle-driven, and limits zsim's performance to about 3 MIPS even with workloads that do not stress memory.

Comparison with other simulators: M5, the predecessor of gem5, was validated against an Alpha 21264 [5]. MARSS, a PTLSim-based simulator with a cycle-accurate OOO x86 model, has performance differences that range from -59% to +50% on SPEC CPU2006, with only 5 benchmarks being within 10% [30, Slide 32]. Sniper, which has an

	blackscholes	water	fluidanimate	canneal	wupwise_m	swim_m	stream	applu_m	barnes	ocean	fft	radix	mgrid_m	avg
IPC1-NC MIPS	1585.6	1336.2	825.2	169.4	777.6	87.7	67.7	307.5	470.0	109.6	286.6	358.0	111.1	197.8
Slowdown	49×	74×	120×	147×	160×	218×	245×	270×	318×	394×	406×	487×	524×	246×
IPC1-C MIPS	1123.3	936.0	629.7	97.8	400.2	27.8	27.7	191.0	403.4	69.7	117.4	231.1	75.0	95.2
Slowdown	70×	106×	158×	255×	311×	687×	600×	435×	371×	620×	991×	755×	776×	512×
OOO-NC MIPS	351.9	347.1	288.1	122.4	316.8	79.3	62.6	189.0	229.7	77.0	194.6	206.5	81.4	138.3
Slowdown	224×	287×	346×	203×	393×	241×	265×	440×	651×	561×	598×	846×	715×	352×
OOO-C MIPS	293.9	314.2	215.1	37.9	162.7	12.2	9.7	79.6	191.6	42.2	89.7	111.3	38.9	41.1
Slowdown	268×	317×	464×	657×	767×	1567×	1713×	1044×	781×	1024×	1298×	1570×	1498×	1186×

Table 4: ZSim performance on the simulated 1024-core chip. Each row shows a different set of models: IPC1 or OOO cores, with and without contention (-C and -NC). We report both simulated MIPS and slowdown vs (parallel) native workload execution on our host machine. The last column summarizes average MIPS (hmean) and slowdown (hmean($MIPS_{host}$)/hmean($MIPS_{zsim}$)).

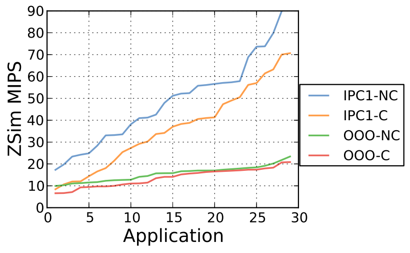


Figure 7: Performance distribution of single-thread zsim on SPEC CPU2006, using four models: IPC1 or OOO cores, with and without contention (C/NC).

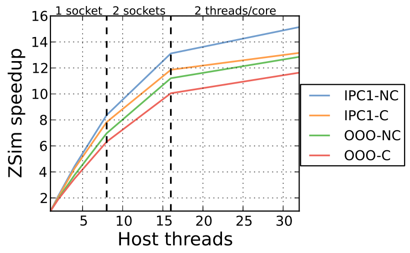


Figure 8: Average zsim speedup on the workloads in Table 4, as we increase the host threads from 1 to 32. The host has 16 cores and 2 hardware threads/core.

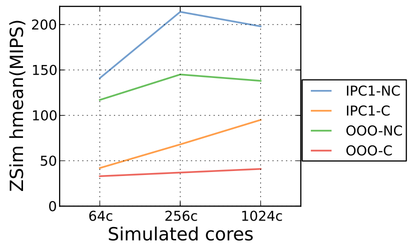


Figure 9: Average zsim performance (hmean(MIPS)) over the workloads in Table 4, with simulated CMPs of 4, 16, and 64 tiles (64, 256 and 1024 cores).

approximate OOO model, was compared against a real x86 machine using SPLASH2, with errors over ±50% [8, Figure 5]. Graphite, HORNET, and SlackSim have not been validated as far as we know.

4.2 Performance

Methodology: We run zsim on a 2-socket Xeon E5-2670 to evaluate its performance. This 2.6GHz Sandy Bridge-EP system has 16 cores, 32 threads, and 64 GB of DDR3-1600 memory over 8 channels. For single-thread performance, we use the same configuration and benchmarks as in Section 4.1. For parallel performance and scalability, we use a tiled architecture with a 3-level cache hierarchy, detailed in Table 3. Each tile has 16 cores, and we use 4 to 64 tiles to simulate chips with 64 to 1024 cores. We use PARSEC, SPLASH2 and SPECOMP workloads, and simulate parallel regions only. For SPLASH2 and PARSEC, we run with the configuration that takes closest to 100 billion instructions. We drop the workloads with an average core utilization below 50% on the 1024-core chip. We report performance in MIPS, and use harmonic means to report aggregate performance. For multithreaded workloads, we also report estimated slowdown = $^{MIPS_{host}}/_{MIPS_{zsim}}$, where $MIPS_{host}$ is the aggregate MIPS the host achieves on the parallel region when using 32 application threads (i.e., the host is not overcommitted).

Single-thread performance: Figure 7 plots zsim's performance distribution on the 29 SPEC CPU workloads. We plot performance for four sets of models: IPC1 or OOO cores, with and without contention (-C and -NC, respectively). Models without contention only run the bound phase. Models with contention run the full bound-weave algorithm with event-driven simulation of the core, L3, and memory controller. The most detailed model, OOO-C, is the one used in Section 4.1. More detailed models impose heavier overheads. Performance ranges between 6.7 MIPS (leslie3d, OOO-C) and 89.9 MIPS (namd, IPC1-NC). The harmonic means are 40.2 MIPS on IPC1-NC, 25.1 MIPS on IPC1-C, 14.8 MIPS on OOO-NC, and 12.4 MIPS on OOO-C. The main factor affecting performance is memory intensity: applications with very frequent L1 and L2 misses take longer to simulate. These results show that, by leveraging DBT and building timing models for performance from the ground up, we can achieve high simulation speed without sacrificing accuracy.

Thousand-core performance: Table 4 shows simulation performance (MIPS and slowdown vs fully loaded host) for the 1024-core chip with each of the four timing models: IPC1 or OOO cores, with and without contention. ZSim achieves 1,585 MIPS with the simplest models (blackscholes, IPC1-NC), and 314 MIPS with the most detailed (water, OOO-C). ZSim achieves lower MIPS on memory-intensive applications, especially swim and stream, which have > 50 L2 and L3 MPKIs. Nevertheless, memory-intensive application slowdowns are not that different from compute-bound applications, since the host also achieves lower MIPS on these workloads. Memory-intensive workloads both stress the weave phase and take more simulated cycles with contention models, so they show the highest performance differences between C and NC models. Since we use harmonic means to summarize performance, these slower simulations have a larger weight, so we see that simulating detailed core models without contention is on average more efficient than simulating IPC1 cores with contention. To put these results in context, a single system running these workloads in sequence simulates 1.32 trillion instructions, taking anywhere from 1.8 hours (IPC1-NC) to 8.9 hours (OOO-C) to simulate the whole suite.

Host scalability: Figure 8 shows scalability of the four different timing models simulating the 1024-core chip when using 1 to 32 host threads. To isolate simulator scalability from the host's Turbo gains, we disable Turbo in this exper-

iment. With Turbo, clock frequency ranges from 2.6GHz to 3.3GHz, introducing a 27% speedup with one or few cores active. As we can see, the no-contention models scale almost linearly up to 16 cores, while the contention models somewhat reduce scalability because the weave phase scales sublinearly. However, scalability is overall good, ranging from 10.1× to 13.6× at 16 threads. Using 32 threads yields an extra 16% speedup due to SMT. Note that we're not fully exploiting weave phase parallelism. In future work, we will pipeline the bound and weave phases and use better scheduling algorithms to assign multiple domains across host cores.
Target scalability: Figure 9 shows average simulation performance as the simulated chip scales from 64 to 1024 threads for models with and without contention. Without contention, the trade-offs are simple. On one hand, larger chips have more parallelism, reducing memory system contention in the simulation (e.g., due to concurrent same-bank L3 accesses), as well as more work per interval. On the other hand, larger chips also increase the simulator's working set, hurting locality. The second effect completely dominates in conventional simulators, which step through all the cores at almost cycle-by-cycle granularity. This is one of the main reasons conventional simulators scale poorly — with hundreds of cores, not even the simulated core's registers fit in the host's L2, resulting in poor locality. In contrast, zsim simulates each core for several hundred cycles, largely sidestepping this issue. Consequently, the first effect is more important at 64 cores (with just 4 L3 banks), and simulation performance peaks at 256 cores and slightly decreases at 1024 cores. With contention models, each domain has more components (cores, L3 banks, memory controllers) when going from 64 to 1024 cores, so the weave phase has more parallelism, and zsim *speeds up* with system size.
Sensitivity to interval length: The interval length allows us to trade off accuracy for performance. We run the 1024-core workloads in Table 4 with 1K, 10K and 100K cycles/interval. Compared with the 1Kcycle results, runs with 10Kcycle intervals show an average absolute error in simulated performance of 0.45% (max 1.9%, fluidanimate), and are on average 42% faster. Runs with 100Kcycle intervals show an average absolute error in simulated performance of 1.1% (max 4.7%, fft), and are on average 53% faster. Other architectural events (e.g., L3 misses) show similar deviations. Overall, an interval of 1-10K cycles causes small errors, but gives somewhat higher performance. In contrast, intervals beyond 10Kcycles yield little benefit and may introduce excessive error.
Comparison with other simulators: As discussed Section 2, parallel simulators typically report best-case performances of 1–10 MIPS and typical-case performances of hundreds of KIPS [8, 10, 18, 27]. Based on our results, zsim is 2-3 orders of magnitude faster than other simulators, depending on the timing models used. Since many factors affect simulator performance (host, workloads, and metrics used), we refrain from making exact comparisons, and leave such task to potential users.

5. CONCLUSIONS

We presented three techniques that break the trade-off between speed and accuracy for parallel simulators with detailed architectural models: DBT-based instruction-driven core timing models, the bound-weave parallelization algorithm, and lightweight virtualization for complex workload support. We used these techniques to build zsim, a validated simulator that reaches speeds up to 1,500 MIPS on thousand-core simulations and supports a wide range of workloads. ZSim is currently used by several research groups, has been used in multiple publications [36, 37, 38, 39, 23, 24], and is publicly available under a free software license.

6. ACKNOWLEDGMENTS

We thank Krishna Malladi for providing the initial version of the detailed memory controller. We thank Xiao Zhang and the Platforms Group at Google for allowing us to incorporate improvements to zsim developed as part of an internship. We are grateful to Nathan Beckmann, Christina Delimitrou, Babak Falsafi, Harshad Kasture, Mikko Lipasti, and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. This work was supported in part by the Stanford Pervasive Parallelism Laboratory, DARPA contract HR0011-11-C-0007, and DARPA PERFECT program contract HR0011-13-2-0005. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

7. REFERENCES

[1] Computer architecture simulation and modeling. *IEEE Micro Special Issue*, 26(4), 2006.
[2] A. Alameldeen and D. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.
[3] C. Bienia, S. Kumar, J. P. Singh, et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT-17*, 2008.
[4] N. Binkert, B. Beckmann, G. Black, et al. The gem5 simulator. *SIGARCH Comp. Arch. News*, 39(2), 2011.
[5] N. Binkert, R. Dreslinski, L. Hsu, et al. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4), 2006.
[6] E. Blem, J. Menon, and K. Sankaralingam. Power Struggles: Revisiting the RISC vs CISC Debate on Contemporary ARM and x86 Architectures. In *HPCA-19*, 2013.
[7] S. Boyd-Wickizer, H. Chen, R. Chen, et al. Corey: An operating system for many cores. In *OSDI-8*, 2008.
[8] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Supercomputing*, 2011.
[9] S. Chandrasekaran and M. D. Hill. Optimistic simulation of parallel architectures using program executables. In *PADS*, 1996.
[10] J. Chen, L. K. Dabbiru, D. Wong, et al. Adaptive and speculative slack simulations of CMPs on CMPs. In *MICRO-43*, 2010.
[11] M. Chidester and A. George. Parallel simulation of chip-multiprocessor architectures. *TOMACS*, 12(3), 2002.
[12] D. Chiou, D. Sunwoo, J. Kim, et al. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *MICRO-40*, 2007.
[13] A. Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, http://www.agner.org/optimize/.
[14] R. Fujimoto. Parallel discrete event simulation. *CACM*, 33-10, 1990.
[15] D. R. Hower, P. Montesinos, L. Ceze, et al. Two hardware-based approaches for deterministic multiprocessor replay. *CACM*, 52-6, 2009.
[16] X. Huang, J. Moss, K. McKinley, et al. Dynamic simplescalar: Simulating java virtual machines. Technical

report, UT Austin, 2003.
[17] Intel. Intel Xeon E3-1200 Family. Datasheet, 2011.
[18] A. Jaleel, R. Cohn, C. Luk, and B. Jacob. CMPSim: A Pin-based on-the-fly multi-core cache simulator. In *MoBS-4*, 2008.
[19] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind. Fast cycle-accurate modeling of a multicore processor. In *ISPASS*, 2012.
[20] G. Kurian, J. Miller, J. Psota, et al. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *PACT-19*, 2010.
[21] R. Liu, K. Klues, S. Bird, et al. Tessellation: Space-time partitioning in a manycore client os. In *HotPar*, 2009.
[22] C.-K. Luk, R. Cohn, R. Muth, et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[23] K. T. Malladi, B. C. Lee, F. A. Nothaft, et al. Towards energy-proportional datacenter memory with mobile DRAM. In *ISCA-39*, 2012.
[24] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, et al. Rethinking DRAM power modes for energy proportionality. In *MICRO-45*, 2012.
[25] M. Martin, D. Sorin, B. Beckmann, et al. Multifacet's general execution driven multiprocessor simulator (gems) toolset. *Comp. Arch. News*, 33-4, 2005.
[26] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS conf.*, 2002.
[27] J. Miller, H. Kasture, G. Kurian, et al. Graphite: A distributed parallel simulator for multicores. In *HPCA-16*, 2010.
[28] H. Pan, B. Hindman, and K. Asanovic. Lithe: Enabling efficient composition of parallel libraries. *HotPar*, 2009.
[29] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A full system simulator for multicore x86 CPUs. In *DAC-48*, 2011.
[30] A. Patel, F. Afram, K. Ghose, et al. MARSS: Micro Architectural Systems Simulator. In *ISCA tutorial 6*, 2012.
[31] M. Pellauer, M. Adler, M. Kinsy, et al. HAsim: FPGA-based high detail multicore simulation using time-division multiplexing. In *HPCA-17*, 2011.
[32] A. Pesterev, J. Strauss, N. Zeldovich, and R. Morris. Improving network connection locality on multicore systems. In *EuroSys-7*, 2012.
[33] S. K. Reinhardt, M. D. Hill, J. R. Larus, et al. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In *SIGMETRICS conf.*, 1993.
[34] P. Ren, M. Lis, M. Cho, et al. HORNET: A Cycle-Level Multicore Simulator. *IEEE TCAD*, 31(6), 2012.
[35] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *CAL*, 10(1), 2011.
[36] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *MICRO-43*, 2010.
[37] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *ISCA-38*, 2011.
[38] D. Sanchez and C. Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. *IEEE Micro's Top Picks*, 32(3), 2012.
[39] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *HPCA-18*, 2012.
[40] D. Sanchez, D. Lo, R. Yoo, et al. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *PACT-20*, 2011.
[41] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis. An Analysis of Interconnection Networks for Large Scale Chip-Multiprocessors. *TACO*, 7(1), 2010.
[42] E. Schnarr and J. R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-8*, 1998.
[43] E. C. Schnarr, M. D. Hill, and J. R. Larus. Facile: A language and compiler for high-performance processor simulators. In *PLDI*, 2001.
[44] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-10*, 2002.
[45] J. Shin, K. Tam, D. Huang, et al. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *ISSCC*, 2010.
[46] S. Srinivasan, L. Zhao, B. Ganesh, et al. CMP Memory Modeling: How Much Does Accuracy Matter? In *MoBS-5*, 2009.
[47] Z. Tan, A. Waterman, R. Avizienis, et al. RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In *DAC-47*, 2010.
[48] Tilera. TILE-Gx 3000 Series Overview. Technical report, 2011.
[49] T. von Eicken, A. Basu, V. Buch, et al. U-net: a user-level network interface for parallel and distributed computing. In *SOSP-15*, 1995.
[50] J. Wawrzynek, D. Patterson, M. Oskin, et al. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2), 2007.
[51] T. Wenisch, R. Wunderlich, M. Ferdman, et al. Simflex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4), 2006.
[52] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS Perf. Eval. Review*, volume 24, 1996.
[53] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA-30*, 2003.