PriME: A Parallel and Distributed Simulator for Thousand-Core Chips

Yaosheng Fu
Princeton University
yfu@princeton.edu

David Wentzlaff
Princeton University
wentzlaf@princeton.edu

Abstract

Modern processors are integrating an increasing number of cores, which brings new design challenges. However, mainstream architectural simulators primarily focus on unicore or multicore systems with small core counts. In order to simulate emerging manycore architectures, more simulators designed for thousand-core systems are needed. In this paper, we introduce the Princeton Manycore Executor (PriME), a parallelized, MPI-based, x86 manycore simulator. The primary goal of PriME is to provide high performance simulation for manycore architectures, allowing fast exploration of architectural ideas including cache hierarchies, coherence protocols and NoCs in thousand-core systems. PriME supports multi-threaded workloads as well as multi-programmed workloads. Furthermore, it parallelizes the simulation of multi-threaded workloads inside of a host machine and parallelizes the simulation of multi-programmed workloads across multiple host machines by utilizing MPI to communicate between different simulator modules. By using two levels of parallelization (within a host machine and across host machines), PriME can improve simulation performance which is especially useful when simulating thousands of cores. Prime is especially adept at executing simulations which have large memory requirements as previous simulators which use multiple machines are unable to simulate more memory than is available in any single host machine. We demonstrate PriME simulating 2000+ core machines and show near-linear scaling on up to 108 host processors split across 9 machines. Finally we validate PriME against a real-world 40-core machine and show the average error to be 12%.

1. Introduction

The rapid shrinking of silicon feature size brings the opportunity to integrate more processor cores into a single chip. Over the past decade, multicore processors with an ever increasing number of cores have become mainstream in both industry and academic research. For example, the Intel Xeon Phi Coprocessors [10], have up to 61 cores, the Cavium Octeon III processors have up to 48 core [20], the MIT Raw processor has 16 core [37], and the TILE-Gx [30] processors support at most 72 cores. If core counts continue to increase, processors containing hundreds or even thousands of cores will be built in just a few years.

Simulation plays a vital role in exploring new architecture designs. However, accurate, detailed modeling and simulation is a time-consuming process. When detailed modeling is applied to thousand-core simulation, simulation speed may be untenably slow, thereby limiting the ability to search the architectural design space. For future processors with hundreds or thousands of cores, simulation not only needs to model the cores, but also their interaction through complex uncore components such as memory systems, Networks-on-Chip (NoC), and shared I/O devices. In order to achieve reasonable speeds, this work, like other recent parallel simulators [7, 17, 24, 32], focuses on detailed modeling of the uncore at the expense of detailed modeling of the core. As innovations in manycore processors are primarily concerned with how to connect cores and uncore design, it is a reasonable assumption to model each core in an undetailed manner, while modeling the uncore or module of interest in greater detail.

Along with the the emergence of massively manycore architectures, we have seen a shift in workloads. Community interest in sequential workloads or even multi-threaded workloads that scale up to a small number of processors (8 or 16) has waned due to the core count growth available in future designs. Instead, interest in workloads for current data centers and future thousand core chips have gained in popularity. Data center and future manycore workloads are typically large, complex, use hundreds of cores, and are more likely to be multi-programmed. Much of this shift has been driven by market needs, for example, Cloud Computing services have become increasingly popular, have offloaded much desktop computing, and are typically structured as many communicating multi-threaded servers. In essence, to enable software scalability, these applications are structured as multi-programmed workloads, each of which itself may be multi-threaded. Also, in Infrastructure as a Service (IaaS) Cloud Computing centers, each physical machine can be shared by many users or customers. This is becoming especially important for systems with high core count as thread-level parallelism alone may not be enough to take full advantage of all of the processor cores. Therefore, a combination of thread-level parallelism with application-level parallelism promises to make the best use of hardware resources. Supporting multi-programmed workloads is critical for future manycore simulators.

The Princeton Manycore Executor (PriME) is unique among parallel architecture simulators. Unlike most existing parallel simulators which only use parallelism within a host machine that contains modest parallelism, PriME is able to parallelize across machines when multi-programmed workloads are used. In particular, PriME is the only parallel simulator to accomplish all of the following:

- Simulate multi-programmed, multi-threaded workloads.
- Exploits parallelism within a host multicore machine.

- Exploits parallelism across multiple multicore host machines.
  - This provides the advantage of the guest system being able to utilize the aggregate of the memory of many host systems in addition to host CPUs.

PriME is structured as a MPI-based simulator which enables it to be easily run on any large cluster which supports MPI, including clusters larger than we show results for. PriME executes x86 programs. Each guest program can execute on a different host node, but we restrict a single multicore program to being simulated on a single multicore host machine. This restriction reduces communication cost while still enabling the parallelization of multi-threaded applications. By straddling the boundary of multiple host machines, PriME is able to scale up the number of host CPUs and physical memory available to be used for simulation which is of the utmost importance when simulating 1000+ core chips.

PriME is explicitly and purposefully not designed to be cycle-accurate as cycle accuracy will hinder performance. Rather, PriME is designed to investigate thousand-core architectures at a reasonable speed, and therefore we have put significant effort into modeling of uncore components such as the memory subsystem and NoCs, but provide only a very simple core model with a constant cycles-per-instruction (CPI) for non-memory instructions. Although such a core model seems too simple to capture the behavior of modern out-of-order processors such as Intel Westmere, we argue that future manycore architectures will likely use relatively simple in-order cores such as Tilera TILE-Gx [30]. Moreover, in the results section, we show that even our simple core model is reasonably accurate when modeling a current day mainstream out-of-order processor.

We evaluated PriME using the PARSEC benchmark suite version 3.0 [4] and augmented it to include the concurrent execution of multiple benchmarks from the PARSEC suite. We validated PriME against a quad-socket 40-core Intel Xeon E7 machine and show that PriME on average has 12% error and maintains a speed of approximately 20MIPS across all PARSEC benchmarks. We then go on to demonstrate the scalability of PriME and measure the scalability across a varying number of host machines and cores within a host machine. We demonstrate multiple 2048 core simulations, showing that it is wholly reasonable to execute 2048 core simulations with the aid of multiple multicore host machines. We provide simulation results from multi-programmed workloads which use up to nine, 12-core machines for a total of 108 host cores.

The balance of this paper is organized as follows. Section 2 describes the high-level design of PriME, Section 3 describes the implementation details, the techniques we developed in order to scale PriME, and how we derived the models for the uncore. Section 4 presents detailed results which include multiple CPU-compute years worth of simulation which we gathered across a 1536 core cluster. Section 5 relates our work to others and finally we conclude.

2. Simulator Design

PriME is an execution-driven simulator for multicore and emerging manycore architectures. As shown in Figure 1 PriME is capable of running one or more multi-threaded applications concurrently on one or more host machines. However, this distribution can only be done at the process level, so different threads within one application must stay in the same host machine in order to maintain a unified shared address space at the functional modeling level. Our performance model enables modeling of global shared memory across all applications. Core modeling is embedded into the application threads themselves through dynamic binary translation (DBT) to form a one-to-one map from application threads to simulated cores, so no additional threads are used for core modeling alone. A separate process contains the modeling of all uncore components. Therefore, when simulating N applications concurrently, the total number of processes is N+1 which can be distributed on up to N+1 host machines. We rely on the operating system to schedule threads and processes of the simulator. Guest system calls are handled by the host OS.

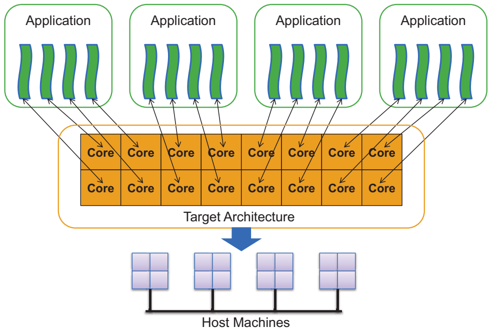


Figure 1: The high-level architecture of PriME.

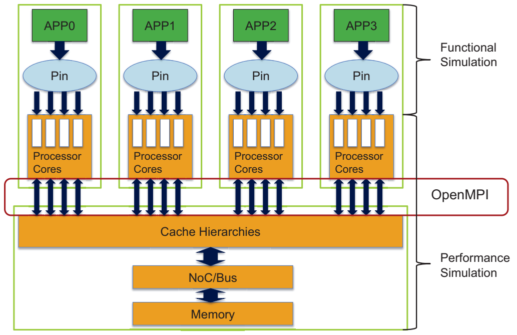
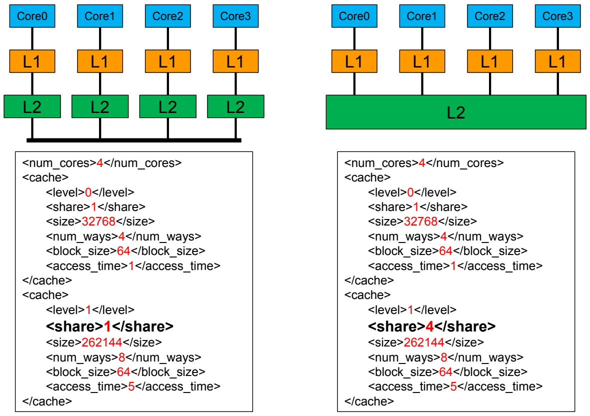


Figure 2: Partitioning of PriME across systems

Figure 2 shows the structure of PriME. The design consists of two parts: Pin [21], a DBT tool developed by Intel, performs the fast, frontend, functional simulation while an event-driven backend simulator models the performance. For each application in a multi-programmed workload, a separate Pin instance is used to do dynamic binary translation, extracting instructions and other useful information such as memory addresses which are fed into the performance model. All uncore components, including cache hierarchies, NoC/Bus and memory, are contained in a separate process which we call the uncore process to distinguish it from application processes. Application processes do not communicate with each other directly, but send memory requests to and receive responses from the uncore process through



```
<num_cores>4</num_cores>
<cache>
      <level>0</level>
      <share>1</share>
      <size>32768</size>
      <num_ways>4</num_ways>
      <block_size>64</block_size>
      <access_time>1</access_time>
</cache>
<cache>
      <level>1</level>
      <share>1</share>
      <size>262144</size>
      <num_ways>8</num_ways>
      <block_size>64</block_size>
      <access_time>5</access_time>
</cache>
```

```
<num_cores>4</num_cores>
<cache>
      <level>0</level>
      <share>1</share>
      <size>32768</size>
      <num_ways>4</num_ways>
      <block_size>64</block_size>
      <access_time>1</access_time>
</cache>
<cache>
      <level>1</level>
      <share>4</share>
      <size>262144</size>
      <num_ways>8</num_ways>
      <block_size>64</block_size>
      <access_time>5</access_time>
</cache>
```

Figure 3: Two cache configurations and corresponding XML inputs. The first configuration chooses private L2 caches, while the second one chooses a shared L2 cache simply by modifying the <share> parameter of the L2 cache.

OpenMPI [14].

As shown in Figure 2, we can see that although processor cores are highly parallelized and can be distributed on multiple host machines, the uncore subsystem is centralized in one process which could potentially become the bottleneck of simulation performance. In order to improve the parallelism of the uncore process, it is built as a multi-threaded process with the number of threads configurable by the user. Memory requests to the uncore process from different cores can be handled by different threads concurrently, enabling multiple simultaneous memory requests. Another technique we adopt is using nonblocking MPI function calls to overlap communication and computation time, so that the uncore process can receive MPI messages while accessing the memory subsystem performance model.

PriME offers a highly configurable memory subsystem. Users can select an arbitrary level of caches and arbitrary sharing patterns for each level. In addition, we provide both bus-based and directory-based coherence protocols. The memory system only keeps meta-data for performance modeling without storing and transferring the underlying data for correctness. This feature improves performance and minimizes memory storage on host machines. In PriME, all configurations can be done by editing an input XML file. Figure 3 shows two example cache configurations with different L2 cache sharing patterns. PriME models different interconnection topologies including buses, hierarchical buses, and meshes with congestion models, all of which can be accessed in parallel by multiple threads.

3. Simulator Implementation

PriME is different from many other simulators in two main aspects: First, PriME is primarily designed for multicore and emerging manycore architectures. Therefore, PriME favors simulation performance and scalability over cycle accuracy. Second, PriME leverages OpenMPI to communicate between different host processes during simulation. The use of MPI brings up a new interaction mode between modules and also some design challenges that need to be addressed. This section describes the design and implementation along with design challenges.

3.1. Constant CPI Core Model

The core model is a simple in-order performance model. It consumes instruction streams generated by Pin and calculates timing costs per instruction. For memory instructions, memory requests are sent to the uncore process and instruction costs are returned. All non-memory instructions have a constant, but configurable, cycle cost. We provide two different sub-models to determine the value of this constant.

3.1.1. One-CPI Core Model This core model assumes the cycles-per-instruction (CPI) equals one for all non-memory instructions, and is widely used as a simple core abstraction. This model is appropriate to model very simple in-order cores, but is not capable of capturing the behavior of modern out-of-order (OOO) or superscalar cores.

3.1.2. Profiling-Based Core Model In order to best model modern high-performance processor cores which may be OOO or superscalar, we rely on hardware profiling tools to estimate the value of this constant CPI. To be more precise, we run each application natively on a real machine and utilize the hardware performance counters to calculate the average CPI for non-memory instructions as follows:

$$CPI_{non-mem} = \frac{Cycles_{total} - \sum_{i=1}^{n+1} Hits_{Li} \times AccessTime_{Li}}{Instructions_{total} - Instructions_{mem}} \qquad (1)$$

In (1) n represents the number of cache levels beginning with 1 (L_{n+1} means DRAM). For simplicity, the influence of instruction caches is not taken into consideration. This value is fed into PriME as the constant CPI for non-memory instructions during simulation. For multi-threaded applications with an adjustable number of threads such as PARSEC [4] and SPLASH2 [40], we found that the thread count has little impact on the non-memory CPI value because the code structure and composition remain almost the same. Hence, a sequential profiling is used when simulating applications with a different number of threads. Although this model is simple compared to real processor cores, it does reflect core performance simply with no simulation speed overhead. Moreover, this core model provides surprisingly good accuracy as shown in Section 4, taking into consideration how simple this core model is versus other detailed OOO core models.

3.2. Memory Subsystem

PriME supports very flexible configurations of cache hierarchies to allow large design space exploration for memory subsystems. Cache hierarchies can be composed of an arbitrary number of levels as long as they can fit in the memory of the host machines. Each level can contain either private or shared caches. We only maintain meta-data in each cache line for performance modeling, thereby removing the need to store and transfer the underlying data for correctness. This simplifies our modeling of memory systems, while bringing performance gain and less pressure on memory storage of host machines.

3.2.1. Multi-level Cache Coherence In PriME, cache coherence is maintained across multiple cache levels. Multi-level

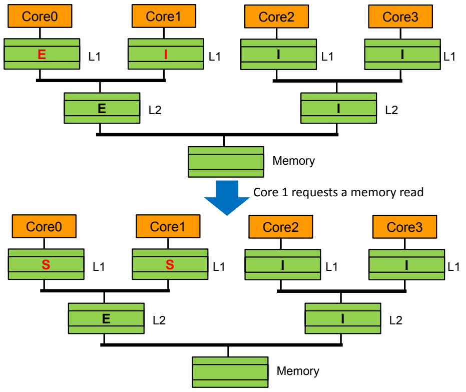


Figure 4: In a multi-level MESI coherence protocol, a data block is kept exclusively in Core 0's private L1 and the shared L2 at the beginning. When a read request from Core 1 causes L1 caches in both cores to switch to the shared state, the L2 cache remains in the exclusive state. Shared state and Exclusive state exist in different cache levels at the same time.

coherence protocols are more complicated than single-level coherence protocols because coherence states need to be consistent not only among all caches within the same level, but also across multiple cache levels. For instance, exclusive state and shared state cannot appear concurrently for the same cache line in single-level MESI coherence protocols, but this may happen in multi-level MESI protocols as long as they are kept in different cache levels as shown in Figure 4. Cache coherence protocols in PriME are carefully designed to propagate coherence messages automatically through the entire system in an iterative way, so there is no limit on the maximum number of cache levels they can operate on. Upon a cache access, the L1 cache is accessed at first and further access to high-level caches may be triggered upon miss. Each cache may propagate messages either down to lower-level caches or up to higher-level caches. In order to reduce the complexity of multi-level coherence protocols, an inclusive property is maintained for all cache levels. This inclusion ensures that data blocks lying in a low-level cache can always be found in corresponding high-level caches. Currently both bus-based and directory-based coherence MESI protocols are supported and can be used at any level of the cache hierarchy.

3.2.2. Set-Based Bidirectional Cache Locks Since the uncore process is multi-threaded itself, cache hierarchies may be accessed by multiple threads concurrently. To avoid race conditions, the simplest solution would be to lock the entire cache hierarchy with one lock upon access. This method is easy to implement but incurs significant lock contention. ZSim [34], uses two locks per cache, one for accesses up to higher-level caches and another for accesses down to lower-level caches. This is much more efficient than the first method, but can still cause unnecessary contention when two low-level caches are trying to access different lines of the shared high-level cache. In order to minimize unnecessary contention, we maintain two different locks for each **cache set** rather than each cache, which provides finer-grain locking than ZSim. Our solution avoids access conflict on different cache sets of shared caches. Figure 5 illustrates an example of a shared L2 cache utilizing our design, which can handle two memory requests from two L1 caches and one invalidation from another L2 cache concurrently because they all acquire different locks within different sets.

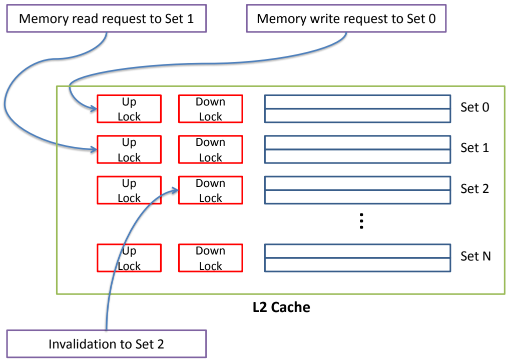


Figure 5: An example two-way set-associative L2 cache. With set-based bidirectional locks, each set contains an up lock and a down lock. Two memory requests from L1 caches and one invalidation from another L2 cache can all be concurrently satisfied because they acquire different locks.

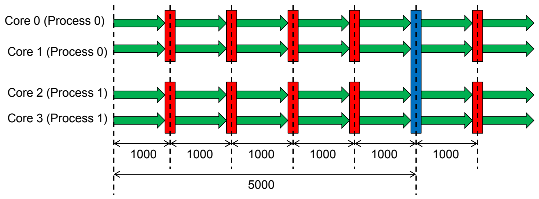


Figure 6: An example of two-level barriers in PriME. Cores within one process encounter barriers every 1000 cycles, while cores across all processes encounter barriers every 5000 cycles.

We use the cache set as the minimum lock unit for two reasons: First, different cache sets in the same cache do not interfere with each other while accesses to one cache set may cause replacement in any cache line within the set; Second, it is easy to identify which cache set to access simply by checking the address index without reading cache content. The set-based bidirectional cache lock scheme avoids deadlock and minimizes critical section size when accessing caches. This enables concurrent cache accesses with minimal lock contention time.

3.3. On-Chip Interconnects

PriME provides three different interconnect networks: Buses, a hierarchy of buses, and 2D mesh NoCs. We model these by using queuing-theory-based contention models adopted from the Graphite simulator [24]. For NoCs, we model routers and links with various input parameters. Upon a NoC access, a step-by-step walk through all routers and links along the routing path is conducted. Currently only link contention is modeled and each link is associated with a lock to avoid races when modeling contention. This allows multiple threads to access the NoC concurrently with lock contention only occurring when two threads are trying to access the same link at the same time.

3.4. Two-level Barrier Synchronization

In PriME, each guest core maintains a local clock which advances based on its own events. In order to limit clock skew between different cores, we insert periodic barriers to force all cores to synchronize after a certain amount of time. Without barriers, one simulated core can run ahead thereby causing simulation accuracy problems as lock acquisition ordering between threads can change. For multi-programmed workloads, since different applications generally are independent from each other and have less interactions than threads within the same application, we use two-level barriers for synchronization as shown in Figure 6: Thread-level barriers are used for threads within the same application and have a relatively small period; Process-level barriers are used to synchronize different processes and have a larger period. Thread-level barriers are implemented inside an application's simulation process and do not need to communicate with other processes. For process-level barriers, applications' simulation processes send MPI messages to the uncore process upon encountering a barrier and then wait for the uncore process to release all applications' simulation processes after all of them have reached the barrier. During our evaluation, we chose a period of 10K cycles for thread-level barriers, and 1M cycles for process-level barriers.

With thread-level barrier synchronization, dependency deadlocks may occur if one thread is trapped in a system call waiting on resources acquired by another thread while the second thread is stuck in the barrier waiting for the first thread to reach the barrier. In order to avoid deadlocks, we keep track of system calls which block in the kernel such as futex wait as they can cause dependence between threads. When a thread enters these system calls, it is excluded from barrier synchronization until it returns from the system call.

3.5. Message Passing Techniques

Several message types are sent to the uncore process, including memory requests, process state changes, and process-level barriers. Of these, memory request messages compose the bulk of the communication traffic. Since all uncore components are integrated in a single process, each memory instruction leads to an MPI request to the uncore process to model an access to the memory subsystem. This results in a large amount of communication between processes and can severely degrade simulation performance, especially when the application process and the uncore process lie in different host machines far from each other. In order to achieve high performance, we apply two major optimization techniques to reduce message passing cost: message batching and overlapping communication with computation.

3.5.1. Message Batching Although the number of memory requests may be very large, each memory request itself typically contains only tens of bytes. Based on this observation, we gather a number of memory requests into one single message before sending them to the uncore process. This batching mechanism greatly reduces the total number of messages sent through MPI.

3.5.2. Overlapping Communication with Computation Non-blocking MPI_Irecv function calls are used in the uncore process to overlap communication and computation time. As a result, one memory request can access the memory subsystem while the next memory request is being received simultaneously. When used with message batching, overlapping enables the processing time of a memory request to be overlapped with that of receiving many subsequent memory transactions.

System	40-core machine	128-node cluster (1536 core in total)
CPU	4 sockets with a 10-core Intel Xeon E7-4870 processor per socket	2 sockets with a 6-core Intel Xeon X5650 processor per socket
DRAM	128 GB NUMA	8GB/node
OS	Linux 3.5.0	Linux 2.6.18
Software	gcc 4.7.2, Pin 2.11	gcc 4.1.2 Pin 2.12

Table 1: Configuration of real systems. The 40-core machine is used as a performance reference comparison and hosting intra-machine experiments while the cluster is used to host inter-machine experiments.

Guest system	40-core machine	Tiled manycore
Cores	Profiling-based model, 2.4GHz	One-CPI model, 2.66GHz
L1 Caches	32KB, private per core, 8-way set associative, 1-cycle latency	32KB, private per core, 8-way set associative, 1-cycle latency
L2 Caches	256KB, private per core, 8-way set associative, 5-cycle latency	256KB, private per core, 16-way set associative, 7-cycle latency
L3 Caches	30MB, shared by 10 cores, 24-way set associative, 10-cycle latency	N/A
NoC/Bus	Bus	2D mesh network, X-Y routing, 32-bit flits and links
Coherence Protocol	Bus-based MESI protocol	Directory-based MESI protocol
Memory	120-cycle latency	100-cycle latency

Table 2: Parameters of guest systems. The 40-core machine is used to compare against native execution and the tiled many-core system is used as a guest architecture for multi-machine simulations.

4. Evaluation

In this section we first validate the accuracy of PriME by comparing with a 40-core real machine. We demonstrate that PriME, although using a simple core model, achieves good accuracy in modeling modern Intel Processors. We then present simulation speed results and find that PriME scales up well both on a single machine and across multiple machines. We also evaluate the sensitivity of PriME to different host machine configurations and simulator design parameters.

4.1. Methodology

Our simulations were executed on two types of host machines as shown in Table 1. All intra-machine experiments were simulated on a single 40-core machine running at 2.4GHz. Inter-machine experiment results were collected from a cluster of servers each of which contains 12 cores running at 2.66GHz. For validation against real hardware, we simulated the 40-core machine as described in Table 2. Table 2 also presents a tiled manycore architecture which we use as the guest architecture for inter-machine experiments.

We use the entire PARSEC benchmark suite version 3.0 containing 13 multi-threaded applications during evaluation. All benchmarks are executed in their entirety without fast-forwarding. Benchmarks on the 40-core machine are simulated with the medium input sets and those on the cluster are simulated with the small input sets. We simulate PARSEC benchmarks

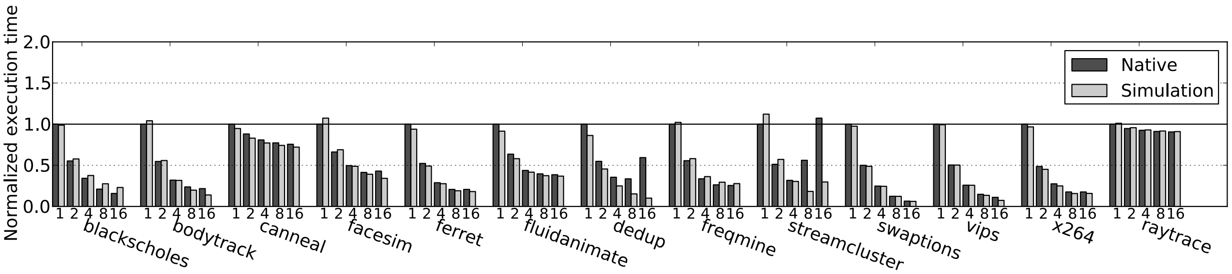


Figure 7: Validation of PriME against native execution results from a 40-core machine for all PARSEC benchmarks. Each benchmark is simulated with multiple thread parameters: 1,2,4,8,16. All results are normalized to native execution results with one thread input.

with varying numbers of threads by modifying the thread parameter. However, this parameter is not a hard thread-count constraint and the actual number of threads may be higher. For simulating the 40-core machine, a profiling-based core model is adopted to model OOO cores as described in Section 3.1.2. Each application is profiled once on the serial code, and the CPI results for non-memory instructions are fed into the multi-threaded simulations.

4.2. Accuracy

We validate the accuracy of PriME against the 40-core machine and present the results in Figure 7. Each benchmark is simulated with different thread parameters and all results are normalized to native execution results with the thread parameter of 1. The simulation results obtained from PriME are close to native execution results under most cases with a total average error of 12.1%. There are two configurations of *streamcluster* and *dedup* with the thread parameter of 16 in which native execution time is much higher than simulated execution time. This is because these two applications spend a significant amount of time in the kernel, especially when running with a large number of threads. Since Pin only instruments user-level code, PriME is not accurate in simulating applications with heavy OS involvement leading to the inaccuracy in *streamcluster* and *dedup* simulation. Overall, PriME shows good accuracy against state-of-the-art real systems even though it is not a cycle-accurate simulator. More importantly, PriME shows the same performance scaling trends as real hardware for most cases.

4.3. Performance

4.3.1. Simulation Speed In Figure 8,we present the simulation speed of PriME on the 40-core machine from the same experiment in Section 4.2. According to Figure 8, simulation speed varies from benchmark to benchmark. In general, benchmarks with a lower percentage of memory instructions lead to higher simulation speed because fewer accesses to the guest memory subsystem greatly simplifies the simulation. For instance, *bodytrack* achieves the best performance and has about 23% memory instructions while *canneal* performs the worst and contains 57% memory instructions. On average, PriME runs at around 20 MIPS across all PARSEC benchmarks for complete execution without fast-forwarding.

4.3.2. Intra-Machine Scalability Figure 9 shows the performance scalability of PriME with different numbers of host cores on a single machine. All benchmarks are simulated with a thread parameter of 16, and the number of host cores varies from 2 to 16. We choose to start with 2 host cores instead of 1 because there are a minimum of two processes required for all PriME runs (one application process and one uncore process) so it is really slow if all processes are put on a single core. As seen in Figure 9, all applications scale up well with more host cores. Moreover, some applications (such as *canneal* and *raytrace*) even show super-linear performance speedup at certain points. This is because simulating more cores on a small number of host cores not only leads to more computation time, but also causes more frequent context switches and more cache pressure. Both of these cause additional performance degradation.

4.3.3. Inter-Machine Scalability We analyze the inter-machine scalability of PriME by simulating multi-programmed workloads across a varied number of host machines. Each multi-programmed workload is composed of 8 concurrent *blackscholes* applications chosen due to their good scalability, each of which contains 32/64/128/256 threads corresponding to 256/512/1024/2048 tiled, guest cores in total. The uncore process also has 8 computation threads. Figure 10 shows the scalability of simulation performance when scaling from 1 host machine (12 host cores) to 9 host machines (108 host cores in total). Despite small variations, PriME achieves linear performance speedup as more host machines are added under all cases. This demonstrates that PriME is capable of breaking the single machine boundary and taking advantage of hardware resources on multiple machines to accelerate simulation. For 256 simulated cores and 9 host machines, the ratio of guest core count to host core count is as low as 2.37:1 (256:108), which means each host core is assigned with less than 3 guest cores on average.

In Figure 10, we see that for a given number of host machines, the simulation speed does not vary much as we vary the number of guest cores. The reason is that although we use different workloads, the overall computation being performed stays the same: 8 *blackscholes* applications with the same input set. So while each host core may be time-multiplexed among more guest threads, the amount of work being done stays constant which results in similar performance.

For multi-machine simulation with PriME, the uncore process can easily become the bottleneck because of frequent communication with all other processes and the centralized modeling of memory access performance. Therefore, it is crucial to exploit as much parallelism as possible in the uncore process. Figure 11

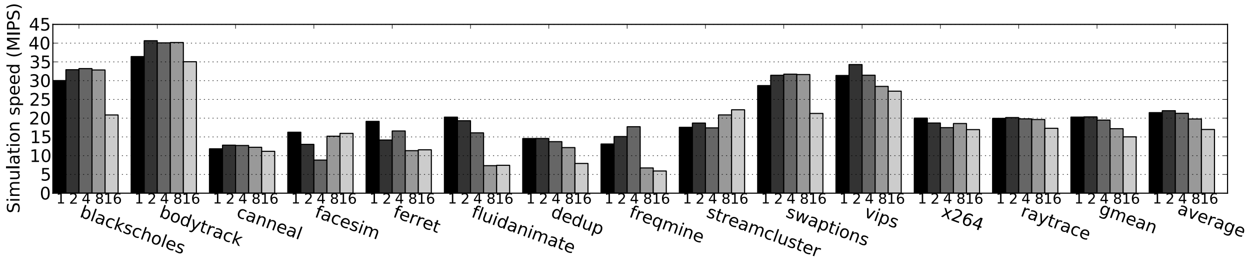


Figure 8: Simulation speed of PriME on the 40-core machine for PARSEC benchmarks. Each benchmark is simulated with multiple thread parameters: 1, 2, 4, 8, 16.

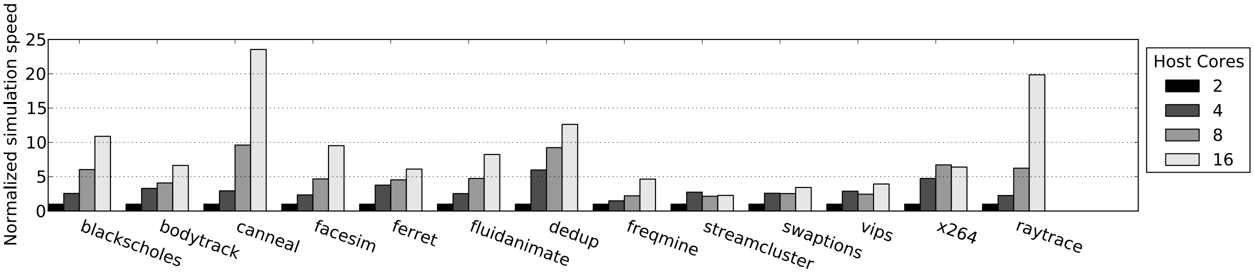


Figure 9: Intra-machine performance speedup of PriME on the 40-core machine as the number of host cores is varied for PARSEC benchmarks. All benchmarks are simulated with a thread parameter of 16, and executed with different number of host cores: 2, 4, 8, 16. Results are relative to simulation with 2 host cores.

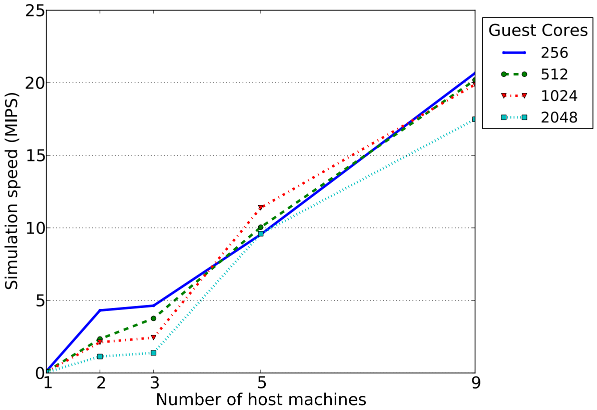


Figure 10: Inter-machine performance speed of PriME on different numbers of host machines with target core counts from 256 to 2048. Each host machine contains 12 cores, so simulations are assigned with 12 cores in the beginning, and up to 108 cores across 9 machines. Workloads are multi-programmed with 8 *blackscholes* applications, each of which contains 32/64/128/256 threads. The uncore process uses 8 threads for all cases.

presents performance speedup of PriME simulating 1024 tiled cores with different numbers of threads in the uncore process. The workload used is the same as the configuration shown in Figure 10. We can see that as the number of threads in the uncore process increases from 1 to 8, simulation performance also improves. Especially when simulating with 9 host machines, using 8 threads almost doubles the performance compared to using 1 thread. For 16 threads, the performance does not continue to scale up because each host machine has only 12 cores so there is not enough hardware parallelism to be taken advantage of.

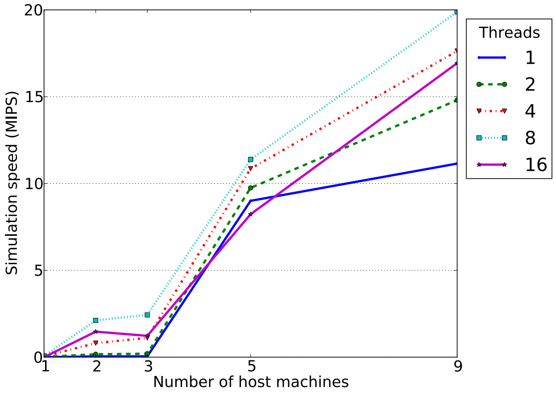


Figure 11: Inter-machine performance speed of PriME across multiple machines as more threads are added from 1 to 16 in the uncore process. The target architecture has 1024 tiled cores with the same workloads described in Figure 10.

4.4. Sensitivity Studies

4.4.1. Core Count Ideally simulation accuracy should not be affected by different host machine configurations. However, since PriME only simulates user-level code and ignores kernel-level execution, it cannot be totally isolated from host machine variations. Figure 12 shows the simulation accuracy variation of PriME on different numbers of host cores from the same experiment in Section 4.3.2. All results are normalized to a host machine with 2 cores. From Figure 12 we can see that simulation variations are within 3% for all applications except *streamcluster*. As we mentioned before in Section 4.2, PriME is not very accurate in simulating applications with a lot of OS involvement such as *streamcluster*, but it exhibits low sensitivity for most other applications.

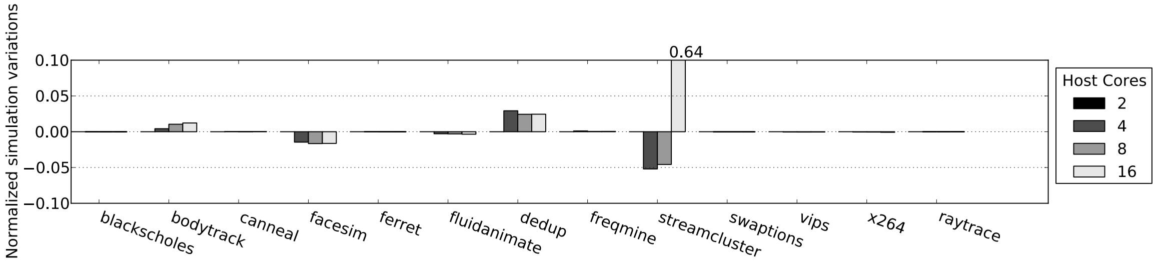


Figure 12: Simulation accuracy variation of PriME as number of host cores are varied. All results are normalized to 2 host cores.

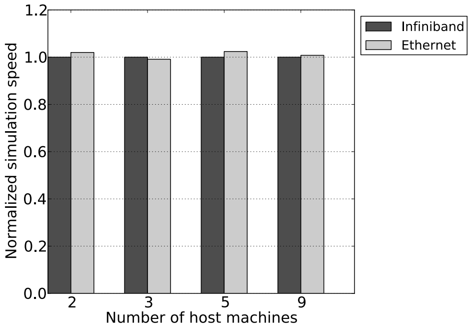


Figure 13: Simulation performance comparison of Infiniband versus Gigabit Ethernet. Results are relative to Infiniband. The Infiniband has 10x lower latency.

4.4.2. Communication Latency Since multi-host PriME requires a significant amount of inter-machine communication, we wanted to know whether its performance is sensitive to how the host machines are connected. We had the opportunity to run the same simulation on the exact same host machines with communication networks with wildly different latency as our cluster has both Infiniband and Gigabit Ethernet. In Figure 13 we compare the performance of PriME with the Infiniband network versus Ethernet. The 128-node computer cluster has quad data rate (QDR) Infiniband whose throughput is around 32Gbps, while the Ethernet only runs at 1Gpbs. Based on some quick MPI tests via those two networks, the Infiniband in our cluster has more than 10x lower latency than Gigabit Ethernet for a single message transmission.

We tested the end-to-end simulation speed of hosts connected by Infiniband and Gigabit Ethernet. We simulated 64 tiled cores and gathered multiple *blackscholes* applications together as multi-programmed workloads. In order to focus solely on communication cost, we allocated exactly one process per host machine. This means the workload is one application for 2 host machines, 2 applications for 3 host machines and so on. Although the Infiniband is much faster than the Ethernet on our cluster, the two networks have similar performance for all cases. This demonstrates that communication cost is well overlapped with computation time in PriME, and PriME's performance does not heavily depend on communication latency.

4.5. Message Batching Size

Message batching is a very important mechanism to reduce communication cost in our design, so it is crucial to choose an appropriate batching size to minimize the communication cost. We studied message batching size by running experiments similar to those in Section 4.4.2, but varying the batching size from 100 to 10000 instead. All results are normalized to the simulator performance with a batching size of 100. Figure 14 shows the results from which we observe that PriME has the best performance with the batching size of 1000. With the increase of batching size, the performance improves at first because the total number of messages decreases. As the batching size continues to increase, performance starts to degrade instead because larger communication delay for each memory operation dominates the performance. The batching size of 1000 balances these two trends, hence we use 1000 as the batching size for all other simulations in this paper. We also analyzed the effect of message batching on simulation accuracy and find that the accuracy variation affected by batching size from 100 to 10000 is within 0.1%.

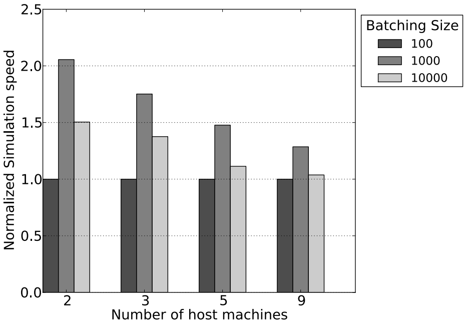


Figure 14: Performance comparison of different batching sizes from 100 to 10000. Results are normalized to 100.

5. Related Work

The work described in this paper is related to several research areas. The existence of new emerging benchmarks drives the design of new architectures, which further drives urgent demands in simulation techniques suitable for future architecture features. As modern processors with increasing core counts continue to develop, there has been an emergence of simulators for parallel architectures, both internally sequential ones and ones which are themselves parallelized. In addition, we have seen the emergence of FPGA-assisted simulation.

In this work, we utilize the PARSEC [3] benchmark suite for evaluation. The PARSEC benchmark suite is a multi-threaded benchmark suite, but does not contain multi-programmed workloads. To approximate both multi-programmed and multi-threaded workloads, we executed multiple concurrent PARSEC applications and timed the worse-case execution time for performance measurement. This is similar to how SPECrate [35] is calculated. We are also interested in trying other emerging parallel, multi-programmed benchmarks such as CloudSuite [13] and the component benchmarks that CloudSuite is built out of [12, 15] in future work.

PriME is designed to explore different architectural considerations of future manycore chips. To that end, PriME's design is influenced by existing high core-count systems such as Intel's MIC architecture [10], Cavium's Octeon [20] architecture, and Tilera's TILE family of processors [30]. In addition, PriME is designed to model future manycore architectures such as Rigel [19], which has been proposed to have over 1024 cores.

In order to achieve high performance, PriME utilizes dynamic binary translation (DBT) for fast functional simulation. PriME leverages Pin [21]. Such fast DBT techniques date back to the Embra [39] core model of SimOS.

There are many sequential processor simulators and emulators including SimOS [33], Simics [22], Rsim [16], SimpleScalar [1], Gem5 [5], PROTEUS [6], and QEMU [2]. Many of these are able to model multicore architectures or even parallel chips, but they are all limited to only leveraging a single processor on a single host machine unlike PriME.

Besides sequential simulators, there are a variety of parallel simulators which contain more similarity to PriME. These include Graphite [24], Sniper [7], CMP$im [17], ZSim [34], MARSS [27], SimFlex [38], GEMS [23], COTSon [25], BigSim [41], FastMP [18], SlackSim [8], Wisconsin Wind Tunnel Simulators (WWT) [31, 26], and those by Penry [29].

Out of the above simulators, Graphite is the closest one to PriME in that they both utilize fast functional execution enabled by Pin [21] and are parallelized both across machines and within a machine, but they still have several significant differences. Graphite can only simulate tiled architectures with private L1 and L2 caches, while PriME provides much more flexible cache hierarchies with an arbitrary number of cache levels, each of which can be either private or shared. Also, PriME is able to execute a richer set of applications through the support of multi-program workloads. In addition, they are designed with different parallelization strategies. Rather then distributing a single guest multi-threaded application across host machines like Graphite, PriME focuses on multi-programmed workloads and only distributes a multi-threaded application within the tight sharing environment of a single shared memory host machine. By structuring PriME in such a manner, we can avoid limiting the total guest memory to a single host machine such as in Graphite. In contrast, the amount of guest memory in PriME can scale with the number of host machines.

Sniper extended Graphite with a superior core model by using interval simulation and by adding multi-program guest support. But unlike Graphite and PriME, Sniper removed the ability to parallelize across multiple host machines. Hence it is also limited to the CPU and memory available in a single machine. CMP$im and ZSim are fast multicore simulators capable of simulating multi-programmed workloads, but are unable to utilize hardware resources beyond a single machine as well.

SimFlex, GEMS, and COTSon all use sequential functional simulators hooked up to parallel performance models. This arrangement can ultimately lead to a sequential bottleneck in the functional simulation. All of these simulators execute only on a single host machine. BigSim and FastMP focus on distributed memory systems while PriME is designed to simulate a much broader range of architectures such as global cache coherent shared memory systems. The WWT and WWT II simulators utilize direct execution of instructions across multiple processors to perform simulation. Unlike PriME, they require modification of applications in order to use shared memory and do not run multi-programmed workloads. Work by Penry et al. focuses on more detailed simulation than PriME and provides greater accuracy at the expense of performance.

An alternative manner to accelerate simulation is to utilize hardware-assisted simulation or hardware emulators. ProtoFlex [11], FAST [9], RAMP Gold [36], and HASim [28] all use FPGAs to accelerate simulation. ProtoFlex and FAST use FPGAs to accelerate the performance model while RAMP Gold and HASim put both functional and performance models in the FPGA. In contrast to PriME, these techniques leverage FPGAs which cost a significant amount of effort and money. For instance, we run PriME on a shared cluster at our university that is used for many other engineering and science applications, which would not be possible if we used FPGAs to accelerate simulation.

6. Conclusion

In this paper, we present PriME, an MPI-based, x86 architecture, distributed and parallel simulator primarily designed for simulating emerging manycore architectures. PriME is highly parallelized through a combination of shared memory and message passing techniques, allowing PriME to scale beyond a single machine and take full advantage of hardware resources across a cluster of machines. PriME provides a very flexible memory subsystem model as well as models for other uncore components to enable large design space exploration. PriME supports multithreaded and multi-programmed workloads which execute across clusters of multicore servers. Our validation of PriME against real hardware shows an average error of 12% at an average speed of 20 MIPS. PriME also demonstrates good scalability up to 108 host cores across 9 machines in simulating a 2048-core system.

Acknowledgements

We thank our research group for invaluable feedback in the design of PriME and Princeton Research Computing for providing computation support. This work is supported in part by NSF Grant Number CCF-1217553 and in part by DARPA PERFECT under contract #HR0011-13-2-0005.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41. USENIX Association, 2005.

[3] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proceedings of the 2009 International Symposium on Workload Characterization*, October 2009.

[4] C. Bienia. *Benchmarking modern multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[6] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: a high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '92/PERFORMANCE '92, pages 247–248, 1992.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

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

[9] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, D. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (FAST): fast, full-system, cycle-accurate simulators. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 249–261, 2007.

[10] G. Chrysos. Knights Corner, Intel's first many integrated core (MIC) architecture product. In *Hot Chips 24*, 2012.

[11] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi. ProtoFlex: towards scalable, full-system multiprocessor simulations using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):15:1–15:32, June 2009.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing*, pages 143–154, 2010.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[14] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, September 2004.

[15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51, 2010.

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

[17] A. Jaleel, R. Cohn, C.-k. Luk, and B. Jacob. CMP$im: a Pin-based on-the-fly multi-core cache simulator, 2008.

[18] S. Kanaujia, I. E. Papazian, J. Chamberlain, and J. Baxter. FastMP: a multi-core simulation methodology. In *MOBS 2006: Workshop on Modeling, Benchmarking and Simulation*, June 2006.

[19] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 140–151, 2009.

[20] R. Kessler. The Cavium 32 core Octeon II 68xx. In *Hot Chips 23*, 2011.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

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

[23] 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. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005.

[24] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: a distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[25] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, July 2009.

[26] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill, D. Wood, S. Huss-Lederman, and J. Larus. Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator. *Concurrency, IEEE*, 8(4):12–20, 2000.

[27] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: a full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1050–1055, New York, NY, USA, 2011. ACM.

[28] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 406–417, 2011.

[29] D. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. August, and D. Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 29–40, 2006.

[30] C. Ramey. TILE-Gx manycore processor: acceleration interfaces and architecture. In *Hot Chips 23*, 2011.

[31] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, pages 48–60, 1993.

[32] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. Fletcher, O. Khan, N. Zheng, and S. Devadas. HORNET: a cycle-level multicore simulator. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(6):890–903, 2012.

[33] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7(1):78–103, 1997.

[34] D. Sanchez and C. Kozyrakis. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

[35] Standard Performance Evaluation Corporation, 2002. http://www.spec.org/.

[36] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 290–301, 2010.

[37] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar. 2002.

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

[39] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.

[40] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 24–36, June 1995.

[41] G. Zheng, G. Kakulapati, and L. Kale. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 78–, 2004.