Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation

Trevor E. Carlson[*†] tcarlson@elis.ugent.be
Wim Heirman[*†] wheirman@elis.ugent.be
Lieven Eeckhout[*] leeckhou@elis.ugent.be

[*]ELIS Department
Ghent University, Belgium

[†]Intel ExaScience Lab
Leuven, Belgium

ABSTRACT

Two major trends in high-performance computing, namely, larger numbers of cores and the growing size of on-chip cache memory, are creating significant challenges for evaluating the design space of future processor architectures. Fast and scalable simulations are therefore needed to allow for sufficient exploration of large multi-core systems within a limited simulation time budget. By bringing together accurate high-abstraction analytical models with fast parallel simulation, architects can trade off accuracy with simulation speed to allow for longer application runs, covering a larger portion of the hardware design space. Interval simulation provides this balance between detailed cycle-accurate simulation and one-IPC simulation, allowing long-running simulations to be modeled much faster than with detailed cycle-accurate simulation, while still providing the detail necessary to observe core-uncore interactions across the entire system. Validations against real hardware show average absolute errors within 25% for a variety of multi-threaded workloads; more than twice as accurate on average as one-IPC simulation. Further, we demonstrate scalable simulation speed of up to 2.0 MIPS when simulating a 16-core system on an 8-core SMP machine.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Performance, Experimentation, Design

Keywords

Interval simulation, interval model, performance modeling, multi-core processor

1. INTRODUCTION

We observe two major trends in contemporary high-performance processors as a result of the continuous progress in chip technology through Moore's Law. First, processor manufacturers integrate multiple processor cores on a single chip — multi-core processors. Eight to twelve cores per chip are commercially available today (in, for example, Intel's E7-8800 Series, IBM's POWER7 and AMD's Opteron 6000 Series), and projections forecast tens to hundreds of cores per chip in the near future — often referred to as many-core processors. In fact, the Intel Knights Corner Many Integrated Core contains more than 50 cores on a single chip. Second, we observe increasingly larger on-chip caches. Multi-megabyte caches are becoming commonplace, exemplified by the 30MB L3 cache in Intel's Xeon E7-8870.

These two trends pose significant challenges for the tools in the computer architect's toolbox. Current practice employs detailed cycle-accurate simulation during the design cycle. While this has been (and still is) a successful approach for designing individual processor cores as well as multi-core processors with a limited number of cores, cycle-accurate simulation is not a scalable approach for simulating large-scale multi-cores with tens or hundreds of cores, for two key reasons. First, current cycle-accurate simulation infrastructures are typically single-threaded. Given that clock frequency and single-core performance are plateauing while the number of cores increases, the simulation gap between the performance of the target system being simulated versus simulation speed is rapidly increasing. Second, the increasingly larger caches observed in today's processors imply that increasingly larger instruction counts need to be simulated in order to stress the target system in a meaningful way.

These observations impose at least two requirements for architectural simulation in the multi-core and many-core era. First, the simulation infrastructure needs to be parallel: the simulator itself needs to be a parallel application so that it can take advantage of the increasing core counts observed in current and future processor chips. A key problem in parallel simulation is to accurately model timing at high speed [25]. Advancing all the simulated cores in lock-step yields high accuracy; however, it also limits simulation speed. Relaxing timing synchronization among the simulated cores improves simulation speed at the cost of introducing modeling inaccuracies. Second, we need to raise the level of abstraction in architectural simulation. Detailed cycle-accurate simulation is too slow for multi-core systems with large core counts and large caches. Moreover, many practical design studies and research questions do not need cycle accuracy because these studies deal with system-level design issues for which cycle accuracy only gets in the way (i.e., cycle accuracy adds too much detail and is too slow,

SC11, November 12-18, 2011, Seattle, Washington, USA

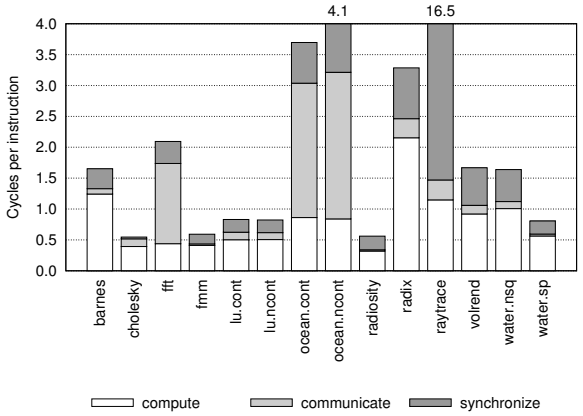

Figure 1: Measured per-thread CPI (average clock ticks per instruction) for a range of SPLASH-2 benchmarks, when running on 16 cores. (Given the homogeneity of these workloads, all threads achieve comparable performance.)

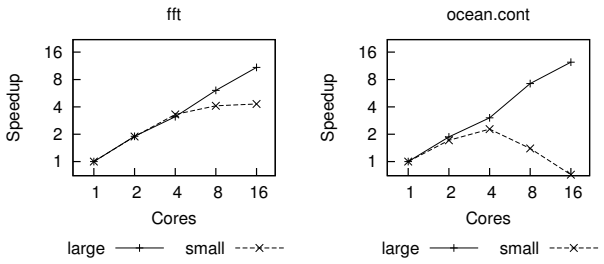

Figure 2: Measured performance of SPLASH-2 on the Intel X7460 using large and small input sets.

especially during the early stages of the design cycle).

This paper deals with exactly this problem. Some of the fundamental questions we want to address are: What is a good level of abstraction for simulating future multi-core systems with large core counts and large caches? Can we determine a level of abstraction that offers both good accuracy and high simulation speed? Clearly, cycle-accurate simulation yields very high accuracy, but unfortunately, it is too slow. At the other end of the spectrum lies the one-IPC model, which assumes that a core's performance equals one Instruction Per Cycle (IPC) apart from memory accesses. While both approaches are popular today, they are inadequate for many research and development projects because they are either too slow or have too little accuracy.

Figure 1 clearly illustrates that a one-IPC core model is not accurate enough. This graph shows CPI (Cycles Per Instruction) stacks that illustrate where time is spent for the SPLASH-2 benchmarks. We observe a wide diversity in the performance of these multi-threaded workloads. For example, the compute CPI component of `radix` is above 2 cycles per instruction, while `radiosity` and `cholesky` perform near the 0.5 CPI mark. Not taking these performance differences into account changes the timing behavior of the application and can result in widely varying accuracy. Additionally, as can be seen in Figure 2, simulated input sizes need to be large enough to effectively stress the memory hierarchy. Studies performed using short simulation runs (using the *small* input set) will reach different conclusions concerning the scalability of applications, and the effect on scaling of proposed hardware modifications, than studies using the more realistic *large* input sets.

The goal of this paper is to explore the middle ground between the two extremes of detailed cycle-accurate simulation versus one-IPC simulation, and to determine a good level of abstraction for simulating future multi-core systems. To this end, we consider the Graphite parallel simulation infrastructure [22], and we implement and evaluate various high-abstraction processor performance models, ranging from a variety of one-IPC models to interval simulation [14], which is a recently proposed high-abstraction simulation approach based on mechanistic analytical modeling. In this process, we validate against real hardware using a set of scientific parallel workloads, and have named this fast and accurate simulator Sniper. We conclude that interval simulation is far more accurate than one-IPC simulation when it comes to predicting overall chip performance. For predicting relative performance differences across processor design points, we find that one-IPC simulation may be fairly accurate for specific design studies with specific workloads under specific conditions. In particular, we find that one-IPC simulation may be accurate for understanding scaling behavior for homogeneous multi-cores running homogeneous workloads. The reason is that all the threads execute the same code and make equal progress, hence, one-IPC simulation accurately models the relative progress among the threads, and more accurate performance models may not be needed. However, for some homogeneous workloads, we find that one-IPC simulation is too simplistic and does not yield accurate performance scaling estimates. Further, for simulating heterogeneous multi-core systems and/or heterogeneous workloads, one-IPC simulation falls short because it does not capture relative performance differences among the threads and cores.

More specifically, this paper makes the following contributions:

1. We evaluate various high-abstraction simulation approaches for multi-core systems in terms of accuracy and speed. We debunk the prevalent one-IPC core simulation model and we demonstrate that interval simulation is more than twice as accurate as one-IPC modeling, while incurring a limited simulation slowdown. We provide several case studies illustrating the limitations of the one-IPC model.

2. In the process of doing so, we validate this parallel and scalable multi-core simulator, named Sniper, against real hardware. Interval simulation, our most advanced high-abstraction simulation approach, is within 25% accuracy compared to hardware, while running at a simulation speed of 2.0 MIPS when simulating a 16-core system on an 8-core SMP machine.

3. We determine when to use which abstraction model, and we explore their relative speed and accuracy in a number of case studies. We find that the added accuracy of the interval model, more than twice as much, provides a very good trade-off between accuracy and simulation performance. Although we found the

one-IPC model to be accurate enough for some performance scalability studies, this is not generally true; hence, caution is needed when using one-IPC modeling as it may lead to misleading or incorrect design decisions.

This paper is organized as follows. We first review high-abstraction processor core performance models and parallel simulation methodologies, presenting their advantages and limitations. Next, we detail the simulator improvements that were critical to increasing the accuracy of multi-core simulation. Our experimental setup is specified next, followed by a description of the results we were able to obtain, an overview of related work and finally the conclusions.

2. PROCESSOR CORE MODELING

As indicated in the introduction, raising the level of abstraction is crucial for architectural simulation to be scalable enough to be able to model multi-core architectures with a large number of processor cores. The key question that arises though is: What is the right level of abstraction for simulating large multi-core systems? And when are these high-abstraction models appropriate to use?

This section discusses higher abstraction processor core models, namely, the one-IPC model (and a number of variants on the one-IPC model) as well as interval simulation, that are more appropriate for simulating multi-core systems with large core counts.

2.1 One-IPC model

A widely used and simple-to-implement level of abstraction is the so-called 'one-IPC' model. Many research studies assume a one-IPC model when studying for example memory hierarchy optimizations, the interconnection network and cache coherency protocols in large-scale multiprocessor and multi-core systems [15, 23, 17]. We make the following assumptions and define a one-IPC model, which we believe is the most sensible definition within the confines of its simplicity. Note that due to the limited description of the one-IPC models in the cited research papers, it is not always clear what exact definition was used, and whether it contains the same optimizations we included in our definition.

The one-IPC model, as it is defined in this paper, assumes in-order single-issue at a rate of one instruction per cycle, hence the name one-IPC or 'one instruction per cycle'. The one-IPC model does not simulate the branch predictor, i.e., branch prediction is assumed to be perfect. However, it simulates the cache hierarchy, including multiple levels of caches. We assume that the processor being modeled can hide L1 data cache hit latencies, i.e., an L1 data cache hit due to a load or a store does not incur any penalty and is modeled to have an execution latency of one cycle. All other cache misses do incur a penalty. In particular, an L1 instruction cache miss incurs a penalty equal to the L2 cache data access latency; an L2 cache miss incurs a penalty equal to the L3 cache data access latency, or main memory access time in the absence of an L3 cache.

2.2 One-IPC models in Graphite

Graphite [22], which forms the basis of the simulator used in this work and which we describe in more detail later, offers three CPU performance models that could be classified as one-IPC models. We will evaluate these one-IPC model variants in the evaluation section of this paper.

The 'magic' model assumes that all instructions take one cycle to execute (i.e., unit cycle execution latency). Further, it is assumed that L1 data cache accesses cannot be hidden by superscalar out-of-order execution, so they incur the L1 data access cost (which is 3 cycles in this study). L1 misses incur a penalty equal to the L2 cache access time, i.e., L2 data cache misses are assumed not to be hidden. This CPU timing model simulates the branch predictor and assumes a fixed 15-cycle penalty on each mispredicted branch.

The 'simple' model is the same as 'magic' except that it assumes a non-unit instruction execution latency, i.e., some instructions such as multiply, divide, and floating-point operations incur a longer (non-unit) execution latency. Similar to 'magic', it assumes all cache access latencies and a fixed branch misprediction penalty.

Finally, the 'iocoom' model stands for 'in-order core, out-of-order memory', and extends upon the 'simple' model by assuming that the timing model does not stall on loads or stores. More specifically, the timing model does not stall on stores, but it waits for loads to complete. Additionally, register dependencies are tracked and instruction issue is assumed to take place when all of the instruction's dependencies have been satisfied.

2.3 Sniper: Interval simulation

Interval simulation is a recently proposed simulation approach for simulating multi-core and multiprocessor systems at a higher level of abstraction compared to current practice of detailed cycle-accurate simulation [14]. Interval simulation leverages a mechanistic analytical model to abstract core performance by driving the timing simulation of an individual core without the detailed tracking of individual instructions through the core's pipeline stages. The foundation of the model is that miss events (branch mispredictions, cache and TLB misses) divide the smooth streaming of instructions through the pipeline into so called intervals [10]. Branch predictor, memory hierarchy, cache coherence and interconnection network simulators determine the miss events; the analytical model derives the timing for each interval. The cooperation between the mechanistic analytical model and the miss event simulators enables the modeling of the tight performance entanglement between co-executing threads on multi-core processors.

The multi-core interval simulator models the timing for the individual cores. The simulator maintains a 'window' of instructions for each simulated core. This window of instructions corresponds to the reorder buffer of a superscalar out-of-order processor, and is used to determine miss events that are overlapped by long-latency load misses. The functional simulator feeds instructions into this window at the window tail. Core-level progress (i.e., timing simulation) is derived by considering the instruction at the window head. In case of an I-cache miss, the core simulated time is increased by the miss latency. In case of a branch misprediction, the branch resolution time plus the front-end pipeline depth is added to the core simulated time, i.e., this is to model the penalty for executing the chain of dependent instructions leading to the mispredicted branch plus the number of cycles needed to refill the front-end pipeline. In case of a long-latency load (i.e., a last-level cache miss or cache coherence miss), we add the miss latency to the core sim-

ulated time, and we scan the window for independent miss events (cache misses and branch mispredictions) that are overlapped by the long-latency load — second-order effects. For a serializing instruction, we add the window drain time to the simulated core time. If none of the above cases applies, we dispatch instructions at the effective dispatch rate, which takes into account inter-instruction dependencies as well as their execution latencies. We refer to [14] for a more elaborate description of the interval simulation paradigm.

We added interval simulation into Graphite and named our version, with the interval model implementation, Sniper[1], a fast and accurate multicore simulator.

2.4 Interval simulation versus one-IPC

There are a number of key differences between interval simulation and one-IPC modeling.

- Interval simulation models superscalar out-of-order execution, whereas one-IPC modeling assumes in-order issue, scalar instruction execution. More specifically, this implies that interval simulation models how non-unit instruction execution latencies due to long-latency instructions such as multiplies, divides and floating-point operations as well as L1 data cache misses, are (partially) hidden by out-of-order execution.

- Interval simulation includes the notion of instruction-level parallelism (ILP) in a program, i.e., it models inter-instruction dependencies and how chains of dependent instructions affect performance. This is reflected in the effective dispatch rate in the absence of miss events, and the branch resolution time, or the number of cycles it takes to execute a chain of dependent instructions leading to the mispredicted branch.

- Interval simulation models overlap effects due to memory accesses, which a one-IPC model does not. In particular, interval simulation models overlapping long-latency load misses, i.e., it models memory-level parallelism (MLP), or independent long-latency load misses going off to memory simultaneously, thereby hiding memory access time.

- Interval simulation also models other second-order effects, or miss events hidden under other miss events. For example, a branch misprediction that is independent of a prior long-latency load miss is completely hidden. A one-IPC model serializes miss events and therefore overestimates their performance impact.

Because interval simulation adds a number of complexities compared to one-IPC modeling, it is slightly more complex to implement, hence, development time takes longer. However, we found the added complexity to be limited: the interval model contains only about 1000 lines of code.

3. PARALLEL SIMULATION

Next to increasing the level of abstraction, another key challenge for architectural simulation in the multi/many-core era is to parallelize the simulation infrastructure in order to take advantage of increasing core counts. One of the key issues in parallel simulation though is the balance of accuracy versus speed. Cycle-by-cycle simulation advances one cycle at a time, and thus the simulator threads simulating the target threads need to synchronize every cycle. Whereas this is a very accurate approach, its performance may be reduced because it requires barrier synchronization between all simulation threads at every simulated cycle. If the number of simulator instructions per simulated cycle is low, parallel cycle-by-cycle simulation is not going to yield substantial simulation speed benefits and scalability will be poor.

There exist a number of approaches to relax the synchronization imposed by cycle-by-cycle simulation [13]. A popular and effective approach is based on barrier synchronization. The entire simulation is divided into quanta, and each quantum comprises multiple simulated cycles. Quanta are separated through barrier synchronization. Simulation threads can advance independently from each other between barriers, and simulated events become visible to all threads at each barrier. The size of a quantum is determined such that it is smaller than the critical latency, or the time it takes to propagate data values between cores. Barrier-based synchronization is a well-researched approach, see for example [25].

More recently, researchers have been trying to relax even further, beyond the critical latency. When taken to the extreme, no synchronization is performed at all, and all simulated cores progress at a rate determined by their relative simulation speed. This will introduce skew, or a cycle count difference between two target cores in the simulation. This in turn can cause causality errors when a core sees the effects of something that — according to its own simulated time — did not yet happen. These causality errors can either be corrected through techniques such as checkpoint/restart, but usually they are just allowed to occur and are accepted as a source of simulator inaccuracy. Chen et al. [5] study both unbounded slack and bounded slack schemes; Miller et al. [22] study similar approaches. Unbounded slack implies that the skew can be as large as the entire simulated execution time. Bounded slack limits the slack to a preset number of cycles, without incurring barrier synchronization.

In the Graphite simulator, a number of different synchronization strategies are available by default. The 'barrier' method provides the most basic synchronization, requiring cores to synchronize after a specific time interval, as in quantum-based synchronization. The most loose synchronization method in Graphite is not to incur synchronization at all, hence it is called 'none' and corresponds to unbounded slack. The 'random-pairs' synchronization method is somewhat in the middle between these two extremes and randomly picks two simulated target cores that it synchronizes, i.e., if the two target cores are out of sync, the simulator stalls the core that runs ahead waiting for the slowest core to catch up. We evaluate these synchronization schemes in terms of accuracy and simulation speed in the evaluation section of this paper. Unless noted otherwise, the multi-threaded synchronization method used in this paper is barrier synchronization with a quantum of 100 cycles.

4. SIMULATOR IMPROVEMENTS

As mentioned before, Graphite [22] is the simulation infrastructure used for building Sniper. During the course of this work, we extended Sniper substantially over the original Graphite simulator. Not only did we integrate the inter-

[1]The simulator is named after a type of bird called a snipe. This bird moves quickly and hunts accurately.

val simulation approach, we also made a number of extensions that improved the overall functionality of the simulator, which we describe in the next few sections. But before doing so, we first detail our choice for Graphite.

4.1 Simulator choice

There are three main reasons for choosing Graphite as our simulation infrastructure for building Sniper. First, it runs x86 binaries, hence we can run existing workloads without having to deal with porting issues across instruction-set architectures (ISAs). Graphite does so by building upon Pin [19], which is a dynamic binary instrumentation tool. Pin dynamically adds instrumentation code to a running x86 binary to extract instruction pointers, memory addresses, register content, etc. This information is then forwarded to a Pin-tool, Sniper in our case, which estimates timing for the simulated target architecture. Second, a key benefit of Graphite is that it is a parallel simulator by construction. A multi-threaded program running in Graphite leads to a parallel simulator. Graphite thus has the potential to be scalable as more and more cores are being integrated in future multi-core processor chips. Third, Graphite is a user-level simulator, and therefore only simulates user-space code. This is appropriate for our purpose of simulating (primarily) scientific codes which spend most of their time in user-space code; very limited time is spent in system-space code [21].

4.2 Timing model improvements

We started with the Graphite simulator as obtained from GitHub.[2] Graphite-Lite, an optimized mode for single-host execution, was back-ported into this version. From this base we added a number of components that improve the accuracy and functionality of the simulator, which eventually led to our current version of the simulator called Sniper.

The interval core model was added to allow for the simulation of the Intel Xeon X7460 processor core; in fact, we validated Sniper against real hardware, as we will explain in the evaluation section. Instruction latencies were determined through experimentation and other sources [11].

In addition to an improved core model, there have also been numerous enhancements made to the uncore components of Graphite. The most important improvement was the addition of a shared multi-level cache hierarchy supporting write-back first-level caches and an MSI snooping cache coherency protocol. In addition to the cache hierarchy improvements, we modeled the branch predictor for the Dunnington machine as the Pentium-M branch predictor [28]. This model was the most recent branch predictor model publicly available but differs only slightly from the branch predictor in the Dunnington (Penryn) core.

4.3 OS modeling

As mentioned before, Graphite only simulates an application's user-space code. In many cases, this is sufficient, and basic system-call latencies can be modeled as simple costs. In some cases, however, the operating system plays a vital role in determining application performance. One example is how the application and kernel together handle pthread locking. In the uncontended case, pthread locking uses futexes, or fast userspace mutexes [12]. The observation here is that for uncontended locks, entering the kernel would be unnecessary as the application can check for lock availability and acquire the lock using atomic instructions. In practice, futexes provide an efficient way to acquire and release relatively uncontended locks in multithreaded code.

[2]Version dated August 11, 2010 with git commit id `7c43a9f9a9aa9f16347bb1d5350c93d00e0a1fd6`

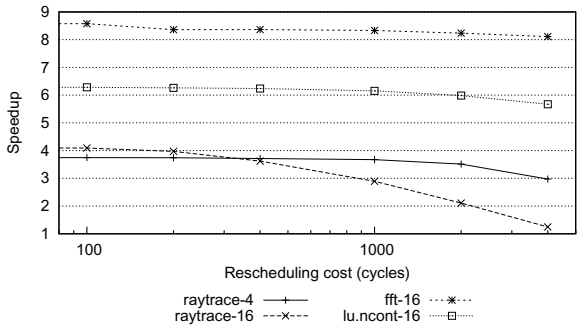

Figure 3: Resulting application runtime from an increasing rescheduling cost. For `fft` (very few synchronization calls), `lu.ncont` (moderate synchronization) and `raytrace` (heavy synchronization), with 4 or 16 threads.

Performance problems can arise, unfortunately, when locks are heavily contended. When a lock cannot be acquired immediately, the `pthread_*` synchronization calls invoke the `futex_wait` and `futex_wake` system calls to put waiting threads to sleep. These system calls again compete for spinlocks inside the kernel. When the pthread synchronization primitives are heavily contended, these kernel spinlocks also become contended which can result in the application spending a significant amount of time inside the kernel. In these (rare) cases, our model, which assumes that kernel calls have a low fixed cost, breaks down.

Modeling kernel spinlock behavior is in itself a research topic of interest [8]. In our current implementation, we employ a fairly simple kernel lock contention model. To this end we introduce the concept of a *rescheduling cost*. This cost advances simulated time each time a thread enters the kernel to go into a wait state, and later needs to be rescheduled once it is woken up. Figure 3 explores the resulting execution time when varying this parameter. For applications with little (`fft`), or even a moderate amount of synchronization (`lu.ncont`), increasing the rescheduling cost does not significantly affect the application's runtime. Yet for `raytrace`, which contains a very high amount of synchronization calls, the rescheduling costs quickly compound. This is because, when one thread incurs this rescheduling cost, it is still holding the lock. This delay therefore multiplies as many other threads are also kept waiting.

Figure 4 shows the run-times measured on `raytrace`. The *hardware* set shows that on real hardware, `raytrace` suffers from severe contention when running on more than four cores. Our initial simulations (center, *baseline*) however predicted near-perfect scaling. After taking a rescheduling cost into account (right, *reschedule-cost*), the run-times are predicted much more accurately. Note that the rescheduling cost is a very rough approximation, and its value is dependent on the number of simulated cores. We used a value of 1000 cycles for simulations with up to four cores, 3000 cycles for 8 cores, and 4000 cycles for 16 cores. Only `raytrace`

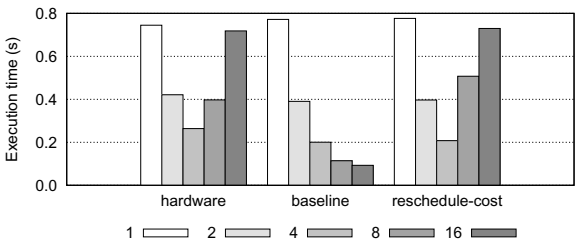

Figure 4: Application runtime for `raytrace` on hardware (left), and simulated before (center) and after (right) adding basic kernel spinlock contention modeling.

Parameter	value
Sockets per system	4
Cores per socket	6
Dispatch width	4 micro-operations
Reorder buffer	96 entries
Branch predictor	Pentium M [28]
Cache line size	64 B
L1-I cache size	32 KB
L1-I associativity	8 way set associative
L1-I latency	3 cycle data, 1 cycle tag access
L1-D cache size	32 KB
L1-D associativity	8 way set associative
L1-D latency	3 cycle data, 1 cycle tag access
L2 cache size	3 MB per 2 cores
L2 associativity	12 way set associative
L2 latency	14 cycle data, 3 cycle tag access
L3 cache size	16 MB per 6 cores
L3 associativity	16 way set associative
L3 latency	96 cycle data, 10 cycle tag access
Coherence protocol	MSI
Main memory	200 ns access time
Memory Bandwidth	4 GB/s

Table 1: Simulated system characteristics for the Intel Xeon X7460.

is affected by this; all other application run-times did not change significantly from the baseline runs with a zero-cycle rescheduling cost.

5. EXPERIMENTAL SETUP

The hardware that we validate against is a 4-socket Intel Xeon X7460 Dunnington shared-memory machine, see Table 1 for details. Each X7460 processor chip integrates six cores, hence, we effectively have a 24-core SMP machine to validate against. Each core is a 45 nm Penryn microarchitecture, and has private L1 instruction and data caches. Two cores share the L2 cache, hence, there are three L2 caches per chip. The L3 cache is shared among the six cores on the chip. As Graphite did not contain any models of a cache prefetcher, all runs were done with the hardware prefetchers disabled. Although we recognize that most modern processors contain data prefetchers, we currently do not model their effects in our simulator. Nevertheless, there is no fundamental reason why data prefetching cannot be added to the simulator. Intel Speedstep technology was disabled, and we set each processor to the high-performance mode, running all processors at their full speed of 2.66 GHz. Benchmarks were run on the Linux kernel version 2.6.32. Each thread is pinned to its own core.

Benchmark	'small' input size	'large' input size
`barnes`	16384 particles	32768 particles
`cholesky`	tk25.O	tk29.O
`fmm`	16384 particles	32768 particles
`fft`	256K points	4M points
`lu.cont`	512×512 matrix	1024×1024 matrix
`lu.ncont`	512×512 matrix	1024×1024 matrix
`ocean.cont`	258×258 ocean	1026×1026 ocean
`ocean.ncont`	258×258 ocean	1026×1026 ocean
`radiosity`	–room –ae 5000.0 –en 0.050 –bf 0.10	–room
`radix`	256K integers	1M integers
`raytrace`	car –m64	car –m64 –a4
`volrend`	head-scaleddown2	head
`water.nsq`	512 molecules	2197 molecules
`water.sp`	512 molecules	2197 molecules

Table 2: Benchmarks and input sets.

The benchmarks that we use for validation and evaluation are the SPLASH-2 benchmarks [30]. SPLASH-2 is a well-known benchmark suite that represents high-performance, scientific codes. See Table 2 for more details on these benchmarks and the inputs that we have used. The benchmarks were compiled in 64-bit mode with –O3 optimization and with the SSE, SSE2, SSE3 and SSSE3 instruction set extensions enabled. We measure the length of time that each benchmark took to run its parallel section through the use of the Read Time-Stamp Counter (`rdtsc`) instruction. A total of 30 runs on hardware were completed, and the average was used for comparisons against the simulator. All results with error-bars report the confidence interval using a confidence level of 95% over results from 30 hardware runs and 5 simulated runs.

6. RESULTS

We now evaluate interval simulation as well as the one-IPC model in terms of accuracy and speed. We first compare the absolute accuracy of the simulation models and then compare scaling of the benchmarks as predicted by the models and hardware. Additionally, we show a collection of CPI-stacks as provided by interval simulation. Finally, we compare the performance of the two core models with respect to accuracy, and we provide a performance and accuracy trade-off when we assume a number of components provide perfect predictions. Because the interval model is more complex than a one-IPC model, it also runs slower, however, the slowdown is limited as we will detail later in this section.

6.1 Core model accuracy

Reliable and accurate microarchitecture comparisons are one of the most important tools in a computer architect's tool-chest. After varying a number of microarchitecture parameters, such as branch predictor configuration or cache size and hierarchy, the architect then needs to accurately evaluate and trade-off performance with other factors, such as energy usage, chip area, cost and design time. Additionally, the architect needs to be able to understand these factors in order to make the best decisions possible with a limited simulation time budget.

Figure 5 shows accuracy results of interval simulation compared with the one-IPC model given the same memory hierarchy modeled after the Intel X7460 Dunnington machine.

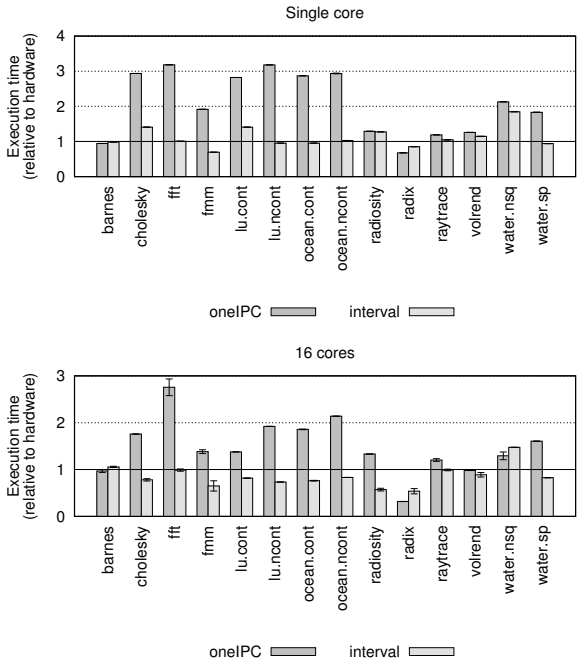

Figure 5: Relative accuracy for the one-IPC and interval models for a single core (top graph) and 16 cores (bottom graph).

We find that the average absolute error is substantially lower for interval simulation than for the one-IPC model in a significant majority of the cases. The average absolute error for the one-IPC model using the large input size of the SPLASH-2 benchmark suite is 114% and 59.3% for single and 16-threaded workloads, respectively. In contrast, the interval model compared to the X7460 machine has an average absolute error of 19.8% for one core, and 23.8% for 16 cores. Clearly, interval simulation is substantially more accurate for predicting overall chip performance than the one-IPC model; in fact, it is more than twice as accurate.

Figure 6 shows a more elaborate evaluation with a variety of one-IPC models for a select number of benchmarks. These graphs show how execution time changes with increasing core counts on real hardware and in simulation. We consider five simulators, the interval simulation approach along with four variants of the one-IPC model. These graphs reinforce our earlier finding, namely, interval simulation is more accurate than one-IPC modeling, and different variants of the one-IPC model do not significantly improve accuracy. Note that performance improves substantially for `fft` as the number of cores increases, whereas for `raytrace` this is not the case. The reason why `raytrace` does not scale is due to heavy lock contention, as mentioned earlier. Our OS modeling improvements to the Graphite simulator, much like the updated memory hierarchy, benefit both the interval and one-IPC models.

6.2 Application scalability

So far, we focused on absolute accuracy, i.e., we evaluated accuracy for predicting chip performance or how long it takes for a given application to execute on the target hardware. However, in many practical research and development

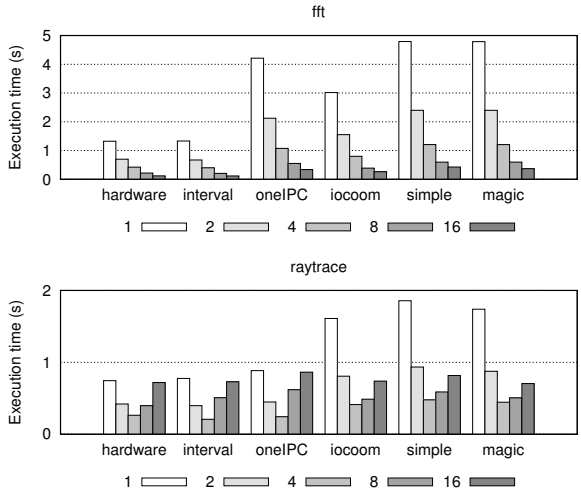

Figure 6: Absolute accuracy across all core models for a select number of benchmarks: `fft` (top graph) and `raytrace` (bottom graph).

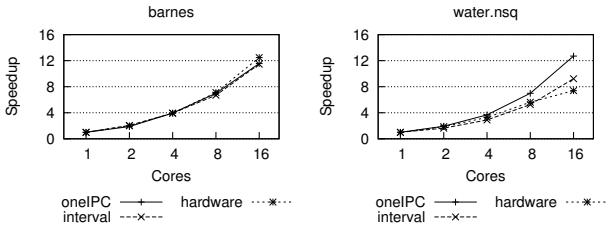

Figure 7: Application scalability for the one-IPC and interval models when scaling the number of cores.

studies, a computer architect is more interested in relative performance trends in order to make design decisions, i.e., a computer architect is interested in whether and by how much one design point outperforms another design point. Similarly, a software developer may be interested in understanding an application's performance scalability rather than its absolute performance. Figure 7 shows such scalability results for a select number of benchmarks. A general observation is that both interval and one-IPC modeling is accurate for most benchmarks (not shown here for space reasons), as exemplified by `barnes` (left graph). However, for a number of benchmarks, see the right graph for `water.nsq`, the interval model accurately predicts the scalability trend, which the one-IPC model is unable to capture. It is particularly encouraging to note that, in spite of the limited absolute accuracy for `water.nsq`, interval simulation is able to accurately predict performance scalability.

6.3 CPI stacks

A unique asset of interval simulation is that it enables building CPI stacks which summarize where time is spent. A CPI stack is a stacked bar showing the different components contributing to overall performance. The base CPI component typically appears at the bottom and represents useful work being done. The other CPI components rep-

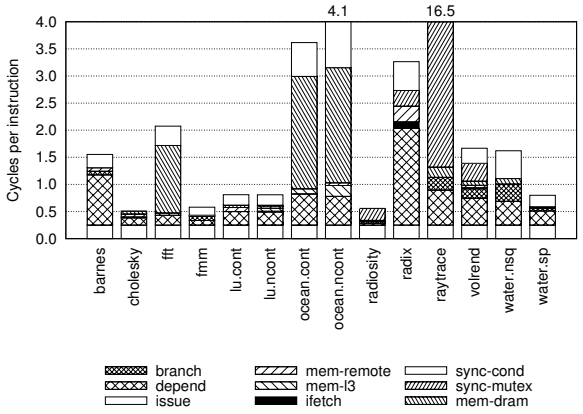

Figure 8: Detailed CPI stacks generated through interval simulation.

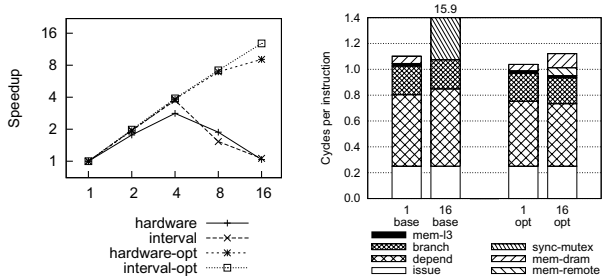

Figure 9: Speedup (left) and CPI stacks (right) for `raytrace`, before and after optimizing its locking implementation.

resent 'lost' cycle opportunities due to instruction interdependencies, and miss events such as branch mispredictions, cache misses, etc., as well as waiting time for contended locks. A CPI stack is particularly useful for gaining insight in application performance. It enables a computer architect and software developer to focus on where to optimize in order to improve overall application performance. Figure 8 shows CPI stacks for all of our benchmarks.

As one example of how a CPI stack can be used, we analyzed the one for `raytrace` and noted that this application spends a huge fraction of its time in synchronization. This prompted us to look at this application's source code to try and optimize it. It turned out that a `pthread_mutex` lock was being used to protect a shared integer value (a counter keeping track of global ray identifiers). In a 16-thread run, each thread increments this value over 20,000 times in under one second of run time. This results in a huge contention of the lock and its associated kernel structures (see also Section 4.3). By replacing the heavy-weight pthread locking with an atomic `lock inc` instruction, we were able to avoid this overhead. Figure 9 shows the parallel speedup (left) and CPI stacks (right) for `raytrace` before and after applying this optimization.

6.4 Heterogeneous workloads

So far, we considered the SPLASH-2 benchmarks, which are all homogeneous, i.e., all threads execute the same code

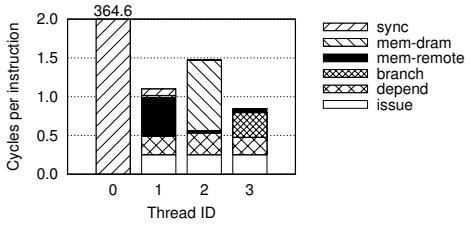

Figure 10: CPI stack for each of the four thread types spawned by `dedup`. (Core 0's very high CPI is because it only spawns and then waits for threads.)

and hence they have roughly the same execution characteristics. This may explain in part why one-IPC modeling is fairly accurate for predicting performance scalability for most of the benchmarks, as discussed in Section 6.2. However, heterogeneous workloads in which different threads execute different codes and hence exhibit different execution characteristics, are unlikely to be accurately modeled through one-IPC modeling. Interval simulation on the other hand is likely to be able to more accurately model relative performance differences among threads in heterogeneous workloads.

To illustrate the case for heterogeneous workloads, we consider the `dedup` benchmark from the PARSEC benchmark suite [3]. Figure 10 displays the CPI stacks obtained by the interval model for each of the threads in a four-threaded execution. The first thread is a manager thread, and simply waits for the worker threads to complete. The performance of the three worker threads, according to the CPI stack, is delimited by, respectively, the latency of accesses to other cores' caches, main memory, and branch misprediction. A one-IPC model, which has no concept of overlap between these latencies and useful computation, cannot accurately predict the effect on thread performance when changing any of the system characteristics that affect these latencies. In contrast to a homogeneous application, where all threads are mispredicted in the same way, here the different threads will be mispredicted to different extents. The one-IPC model will therefore fail to have an accurate view on the threads' progress rates relative to each other, and of the load imbalance between cores executing the different thread types.

6.5 Simulator trade-offs

As mentioned earlier in the paper, relaxing synchronization improves parallel simulation speed. However, it comes at the cost of accuracy. Figure 11 illustrates this trade-off for a 16-core `fft` application. It shows the three synchronization mechanisms in Graphite, barrier, random-pairs and none, and plots simulation time, measured in hours on the vertical axis, versus average absolute error on the horizontal axis. No synchronization yields the highest simulation speed, followed by random-pairs and barrier synchronization. For accuracy, this trend is reversed: barrier synchronization is the most accurate approach, while the relaxed synchronization models can lead to significant errors.

In addition, we explore the effect of various architectural options. Figure 11 also shows data points in which the branch predictor or the instruction caches were not modeled. Turning off these components (i.e. assuming perfect

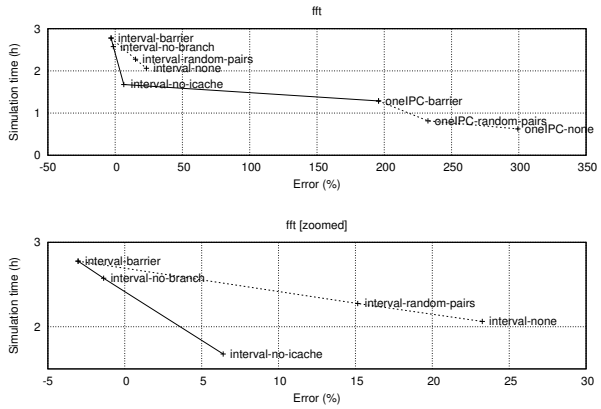

Figure 11: Accuracy vs. speed trade-off graphs comparing both synchronization mechanisms for parallel simulation.

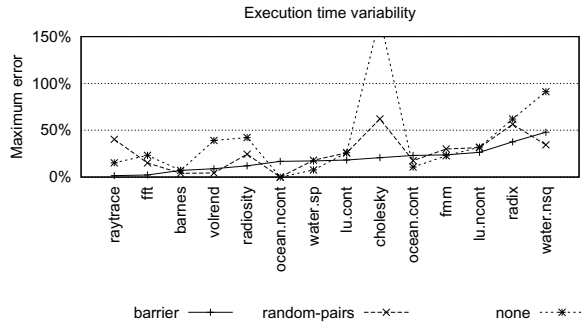

Figure 12: Maximum absolute error by synchronization method in parallel simulation for simulating a 16-core system.

branch prediction or a perfect I-cache, respectively) brings the simulation time down significantly, very near to that of the one-IPC model (which includes neither branch prediction or I-cache models).

6.6 Synchronization variability

It is well-known that multi-threaded workloads incur non-determinism and performance variability [1], i.e., small timing variations can cause executions that start from the same initial state to follow different execution paths. Multiple simulations of the same benchmark can therefore yield different performance predictions, as evidenced by the error bars plotted on Figure 5. Increasing the number of threads generally increases variability. Different applications are susceptible to this phenomenon to different extents, based on the amount of internal synchronization and their programming style — applications employing task queues or work rebalancing often take severely different execution paths in response to only slight variations in thread interleaving. On the other hand, applications that explicitly and frequently synchronize, via pthreads for example, will maintain consistent execution paths.

An interesting observation that we made during our experiments is that performance variability generally is higher for no synchronization and random-pairs synchronization compared to barrier synchronization. This is illustrated in Figure 12 which shows the maximum error observed across five simulation runs. The benchmarks are sorted on the horizontal axis by increasing *max* error for barrier synchronization. The observation from this graph is that, although sufficient for some applications, no synchronization can lead to very high errors for others, as evidenced by the `cholesky` and `water.nsq` benchmarks. Whereas prior work by Miller et al. [22] and Chen et al. [5] conclude that relaxed synchronization is accurate for most performance studies, we conclude that caution is required because it may lead to misleading performance results.

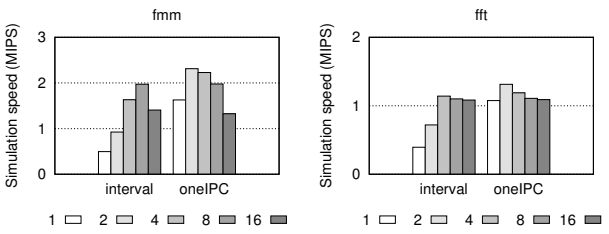

Figure 13: Simulation speed of 1–16 simulated cores on an eight-core host machine.

6.7 Simulation speed and complexity

As mentioned earlier, the interval simulation code base is quite small; consisting of about 1000 lines of code. Compared to a cycle-accurate simulator core, the amount of code necessary to implement this core model is several orders of magnitude less, a significant development savings.

A comparison of the simulation speed between the interval and one-IPC models can be found in Figure 13. Here we plot the aggregate simulation speed, for 1–16 core simulations with no synchronization and the branch predictor and instruction cache models turned off. As can be seen on Figure 11, adding I-cache modeling or using barrier synchronization increases simulation time by about 50% each, which corresponds to a 33% lower MIPS number. These runs were performed on dual socket Intel Xeon L5520 (Nehalem) machines with a total of eight cores per machine. When simulating a single thread, the interval model is about 2–3× slower than the one-IPC model. But when scaling up the number of simulated cores, parallelism can be exploited and aggregate simulation speed goes up significantly — until we have saturated the eight cores of our test machines. Also, the relative computational cost of the core model quickly decreases, as the memory hierarchy becomes more stressed and requires more simulation time. From eight simulated cores onwards, on an eight-core host machine, the simulation becomes communication-bound and the interval core model does not require any more simulation time than the one-IPC model — while still being about twice as accurate.

Note that the simulation speed and scaling reported for the original version of Graphite [22] are higher than the numbers we show for Sniper in Figure 13. The main difference lies in the fact that Graphite, by default, only simulates private cache hierarchies. This greatly reduces the need for communication between simulator threads, which enables good scaling. For this study, however, we model a realistic, modern CMP with large shared caches. This requires additional synchronization, which affects simulation speed.

7. OTHER RELATED WORK

This section discusses other related work not previously covered.

7.1 Cycle-accurate simulation

Architects in industry and academia heavily rely on detailed cycle-level simulation. In some cases, especially in industry, architects rely on true cycle-accurate simulation. The key benefit of cycle-accurate simulation obviously is accuracy, however, its slow speed is a significant limitation. Industry simulators typically run at a speed of 1 to 10 kHz. Academic simulators, such as M5 [4], GEMS [20] and PTLSim [32] are not truly cycle-accurate compared to real hardware, and therefore they are typically faster, with simulation speeds in the tens to hundreds of KIPS (kilo simulated instructions per second) range. Cycle-accurate simulators face a number of challenges in the multi-core era. First, these simulators are typically single-threaded, hence, simulation performance does not increase with increasing core counts. Second, given its slow speed, simulating processors with large caches becomes increasingly challenging because the slow simulation speed does not allow for simulating huge dynamic instruction counts in a reasonable amount of time.

7.2 Sampled simulation

Increasing simulation speed is not a new research topic. One popular solution is to employ sampling, or simulate only a few simulation points. These simulation points are chosen either randomly [7], periodically [31] or through phase analysis [26]. Ekman and Stenström [9] apply sampling to multiprocessor simulation and make the observation that fewer sampling units need to be taken to estimate overall performance for larger multi-processor systems than for smaller multi-processor systems in case one is interested in aggregate performance only. Barr et al. [2] propose the Memory Timestamp Record (MTR) to store microarchitecture state (cache and directory state) at the beginning of a sampling unit as a checkpoint. Sampled simulation typically assumes detailed cycle-accurate simulation of the simulation points, and simulation speed is achieved by limiting the number of instructions that need to be simulated in detail. Higher abstraction simulation methods use a different, and orthogonal, method for speeding up simulation: they model the processor at a higher level of abstraction. By doing so, higher abstraction models not only speed up simulation, they also reduce simulator complexity and development time.

7.3 FPGA-accelerated simulation

Another approach that has gained interest recently is to accelerate simulation by mapping timing models on FPGAs [6, 29, 24]. The timing models in FPGA-accelerated simulators are typically cycle-accurate, with the speedup coming from the fine-grained parallelism in the FPGA. A key challenge for FPGA-accelerated simulation is to manage simulation development complexity and time because FPGAs require the simulator to be synthesized to hardware. Higher abstraction models on the other hand are easier to develop, and could be used in conjunction with FPGA-accelerated simulation, i.e., the cycle-accurate timing models could be replaced by analytical timing models. This would not only speed up FPGA-based simulation, it would also shorten FPGA-model development time and in addition it would also enable simulating larger computer systems on a single FPGA.

7.4 High-abstraction modeling

Jaleel et al. [16] present the CMP$im simulator for simulating multi-core systems. Like Graphite, CMP$im is built on top of Pin. The initial versions of the simulator assumed a one-IPC model, however, a more recent version, such as the one used in a cache replacement championship[3], models an out-of-order core architecture. It is unclear how detailed the core models are because the simulator internals are not publicly available through source code.

Analytical modeling is a level of abstraction even higher than one-IPC simulation. Sorin et al. [27] present an analytical model using mean value analysis for shared-memory multi-processor systems. Lee et al. [18] present composable multi-core performance models through regression.

8. CONCLUSIONS

Exploration of a variety of system parameters in a short amount of time is critical to determining successful future architecture designs. With the ever growing number of processors per system and cores per socket, there are challenges when trying to simulate these growing system sizes in reasonable amounts of time. Compound the growing number of cores with larger cache sizes, and one can see that longer, accurate simulations are needed to effectively evaluate next generation system designs. But, because of complex core-uncore interactions and multi-core effects due to heterogeneous workloads, realistic models that represent modern processor architectures become even more important. In this work, we present the combination of a highly accurate, yet easy to develop core model, the interval model, with a fast, parallel simulation infrastructure. This combination provides accurate simulation of modern computer systems with high performance, up to 2.0 MIPS.

Even when comparing a one-IPC model that is able to take into account attributes of many superscalar, out-of-order processors, the benefits of the interval model provide a key simulation trade-off point for architects. We have shown a 23.8% average absolute accuracy when simulating a 16-core Intel X7460-based system; more than twice that of our one-IPC model's 59.3% accuracy. By providing a detailed understanding of both the hardware and software, and allowing for a number of accuracy and simulation performance trade-offs, we conclude that interval simulation and Sniper is a useful complement in the architect's toolbox for simulating high-performance multi-core and many-core systems.

9. ACKNOWLEDGEMENTS

We would like to thank the reviewers for their constructive and insightful feedback. We would also like to thank Stephanie Hepner for helping to name the simulator. Trevor Carlson and Wim Heirman are supported by the ExaScience Lab, supported by Intel and the Flemish agency for Innovation by Science and Technology. Additional support is provided by The Research Foundation – Flanders projects G.0255.08 and G.0179.10, UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

[3]JWAC-1 cache replacement championship. `http://www.jilp.org/jwac-1`.

10. REFERENCES

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, Feb. 2003.

[2] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 66–77, Mar. 2005.

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

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

[5] J. Chen, L. K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. Adaptive and speculative slack simulations of CMPs on CMPs. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 523–534. IEEE Computer Society, 2010.

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

[7] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 468–477, Oct. 1996.

[8] Y. Cui, W. Wu, Y. Wang, X. Guo, Y. Chen, and Y. Shi. A discrete event simulation model for understanding kernel lock thrashing on multi-core architectures. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–8, Dec. 2010.

[9] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, Mar. 2005.

[10] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[11] A. Fog. Instruction tables. `http://www.agner.org/optimize/instruction_tables.pdf`, April 2011.

[12] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the 2002 Ottawa Linux Summit*, pages 479–495, 2002.

[13] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[14] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 307–318, Feb. 2010.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. D. an B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 102–113, June 2004.

[16] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA 2008*, pages 28–36, June 2008.

[17] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, pages 208–219, 2008.

[18] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281, Nov. 2008.

[19] 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 (PLDI)*, pages 190–200. ACM, June 2005.

[20] 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. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, Nov. 2005.

[21] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 145–156, Oct. 1994.

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

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 13th International Symposium on High Performance Computer*

Architecture (HPCA), pages 254–265, Feb. 2006.

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

[25] 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 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, Oct. 2002.

[27] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391, June 1998.

[28] V. Uzelac and A. Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 207–217, 2009.

[29] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, Mar. 2007.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[31] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, June 2003.

[32] M. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the 2007 IEEE International Symmposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34. Apr. 2007.