An Evaluation of High-Level Mechanistic Core Models

TREVOR E. CARLSON, Ghent University
WIM HEIRMAN, Intel, ExaScience Lab
STIJN EYERMAN, Ghent University
IBRAHIM HUR, Intel, ExaScience Lab
LIEVEN EECKHOUT, Ghent University

Large core counts and complex cache hierarchies are increasing the burden placed on commonly used simulation and modeling techniques. Although analytical models provide fast results, they do not apply to complex, many-core shared-memory systems. In contrast, detailed cycle-level simulation can be accurate but also tends to be slow, which limits the number of configurations that can be evaluated. A middle ground is needed that provides for fast simulation of complex many-core processors while still providing accurate results.

In this article, we explore, analyze, and compare the accuracy and simulation speed of high-abstraction core models as a potential solution to slow cycle-level simulation. We describe a number of enhancements to interval simulation to improve its accuracy while maintaining simulation speed. In addition, we introduce the instruction-window centric (IW-centric) core model, a new mechanistic core model that bridges the gap between interval simulation and cycle-accurate simulation by enabling high-speed simulations with higher levels of detail. We also show that using accurate core models like these are important for memory subsystem studies, and that simple, naive models, like a one-IPC core model, can lead to misleading and incorrect results and conclusions in practical design studies. Validation against real hardware shows good accuracy, with an average single-core error of 11.1% and a maximum of 18.8% for the IW-centric model with a 1.5× slowdown compared to interval simulation.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization—Performance of Systems**]: Modeling Techniques; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms: Performance, Experimentation, Design

Additional Key Words and Phrases: Interval simulation, interval model, performance modeling, design space exploration, multicore processor

ACM Reference Format:
Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. ACM Trans. Architec. Code Optim. 11, 3, Article 28 (July 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2629677

This work is supported by Intel and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Additional support is provided by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259295. Experiments were run on computing infrastructure at the Intel HPC Lab, Swindon, UK, and the VSC Flemish Supercomputer Center.
Authors' addresses: T. E. Carlson, S. Eyerman, and L. Eeckhout, Department of Electronics and Information Systems, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; emails: trevor.carlson@elis.ugent.be; lieven.eeckhout@elis.ugent.be; W. Heirman and I. Hur, Intel ExaScience Lab, Kapeldreef 75, B-3001 Leuven, Belgium.

DOI: http://dx.doi.org/10.1145/2629677

1. INTRODUCTION

With increasing numbers of cores being placed on a single processor die, it is becoming more difficult to analyze the performance of next-generation, multicore systems. To speed up investigations of these future platforms, we need to be able to quickly and accurately estimate the performance of workloads running on these microarchitectures. Our toolbox today consists of a number of options. One methodology—analytical modeling [Karkhanis and Smith 2004; Taha and Wills 2008]—uses a fast, high-level approach to understanding application performance. Interval modeling [Eyerman et al. 2009] is an example of an analytical, mechanistic model that can quickly and reliably predict the performance of single-threaded workloads on modern machines. Nevertheless, understanding the performance of multithreaded applications on future multicore processors is a task that goes beyond the ability of current analytical models. Microarchitectural simulation, therefore, is required for accurate performance modeling of modern multithreaded workloads.

Although traditional cycle-level simulation techniques [Emer et al. 2002; Hardavellas et al. 2004; Loh et al. 2009; Binkert et al. 2011; Patel et al. 2011] provide sufficient detail, they can be very slow and result in a large simulation bottleneck where accurate microarchitectural evaluation of a meaningful number of configurations can take a significant amount of time. Several solutions have been proposed to address this issue, including workload sampling [Sherwood et al. 2002; Wunderlich et al. 2003; Ardestani and Renau 2013; Carlson et al. 2013, 2014], and simulation acceleration through software optimization [Sanchez and Kozyrakis 2013] and FPGA hardware [Krasnov et al. 2007; Chiou et al. 2007; Chung et al. 2008; Pellauer et al. 2011].

An orthogonal approach is to design higher-level simulation methodologies that provide reduced simulation times while maintaining accuracy. The one-IPC core model is a simplistic high-level model that may lead to misleading results and conclusions, as we show in this work. Statistical simulation [Oskin et al. 2000; Nussbaum and Smith 2001; Eeckhout et al. 2003, 2004] abstracts parts of the core using statistical methods. Mechanistic performance models, on the other hand, use insight from analytical core models to perform fast and accurate simulation. By starting with higher-level simulation methodologies during early discovery, and then moving to more detailed models as enhancements are refined during the design cycle, one can more easily bridge the gap between fast and relatively accurate simulation with more accurate, detailed simulation that can be significantly slower. Figure 1 provides an illustration of this process.

In this work, we enhance existing high-level core models as well as introduce a new, more accurate core model that allows for fast simulation while maintaining accuracy compared to real hardware. We first provide an overview of interval simulation [Genbrugge et al. 2010], a high-level mechanistic core model that produces fast and accurate simulation results. We present improvements that build on interval simulation as well as extend it in a new direction for higher simulation accuracy. We extend the original interval simulation model to take into account limited execution units and improve its handling of overlapping memory accesses through a more detailed dependency analysis of memory accesses. These modifications improve accuracy for a range of workloads at a minimal increase in simulation time. In addition, we present a new high-level core model that combines the insights from interval modeling with a detailed simulation model of the instruction window, or reorder buffer (ROB). We call this methodology instruction-window centric (IW-centric) simulation. Although the original interval simulation methodology calculates the instruction-level parallelism (ILP) of an application analytically, IW-centric simulation models micro-op dependency and issue timing in detail. This provides additional accuracy with respect to fine-grained

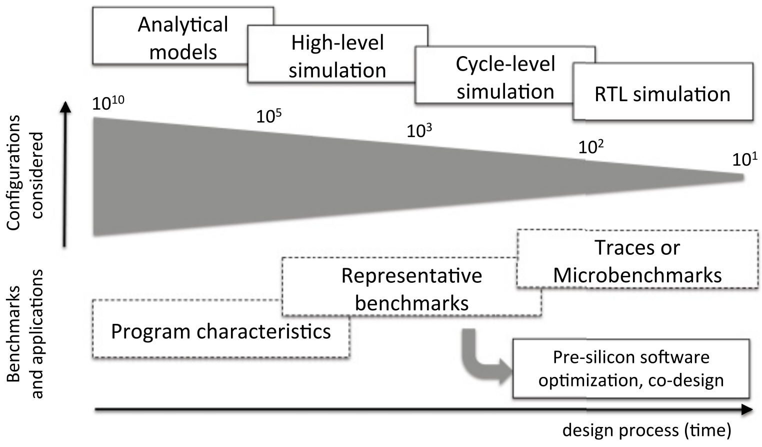

Analytical models
High-level simulation
Cycle-level simulation
RTL simulation
Configurations considered
10^10
10^5
10^3
10^2
10^1
Benchmarks and applications
Traces or Microbenchmarks
Representative benchmarks
Program characteristics
Pre-silicon software optimization, co-design
design process (time)

Fig. 1. A design waterfall example. On the left, analytical models can help us evaluate a large number of designs. Detailed, RTL simulations, on the right, take much longer to simulate but are designed to verify a final implementation. Interval simulation and IW-centric simulation take a middle-ground approach, allowing for the evaluation of a large number of configurations while still providing good accuracy.

events with a small reduction in simulation speed. Through more accurate modeling, the IW-centric model can be used as a step closer to cycle-level simulation. A secondary benefit of the IW-centric core is that it is constructed like a traditional processor core, which allows straightforward modifications without additional high-level modeling. All models are integrated in Sniper [Carlson et al. 2011] and are available for download at http://snipersim.org.

More specifically, we make the following contributions in this work:

—We improve the accuracy of interval simulation by taking into account contention between micro-ops that are ready for execution. Issue contention allows interval simulation to more accurately model modern processors by modeling these structural hazards.
—We present an improvement to dependency analysis tracking in interval simulation by differentiating between instructions or micro-ops that depend on long-latency loads and those that are not for MLP determination.
—We introduce the IW-centric core model, which is a new speed versus accuracy trade-off point between interval simulation and traditional detailed cycle-level core model simulations. Compared to real hardware, IW-centric simulation improves accuracy over interval simulation while maintaining high simulation speed.
—We present a detailed analysis of the trade-offs between the one-IPC, interval simulation and the IW-centric simulation models, and demonstrate that simple, naive models like the one-IPC model can lead to misleading and incorrect conclusions in practical design studies.
—We validate these core models against real hardware and show single-core average errors of 11% for the IW-centric model and 24% for interval simulation.

This article is organized as follows. We first discuss high-level core models that are used to provide a speed versus accuracy trade-off for microprocessor simulation. Next, we present improvements to interval simulation and introduce a new core model—IW-centric simulation—that improves accuracy with respect to hardware. Finally, we provide an evaluation of these core models for both speed and accuracy, and discuss how core model resolution affects microarchitecture design conclusions.

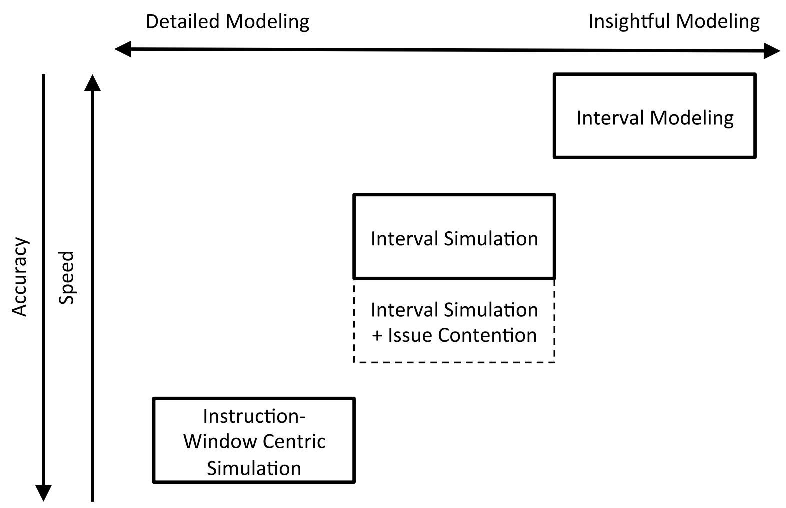


Fig. 2. Interval modeling and simulation technique taxonomy. Interval modeling uses mechanistic performance modeling techniques to provide application performance estimates. Interval simulation and IW-centric simulation build on these insights to provide a more accurate representation of core timing, including timing for multicore simulation.

2. CORE-LEVEL ABSTRACTIONS

There are a variety of high-level core abstractions, from a simple one-IPC model to the IW-centric core model introduced in this work. Figure 2 presents a taxonomy breakdown of the interval model-derived simulation techniques. Next, we provide an overview of one-IPC, interval modeling, interval simulation, and IW-centric simulation. These models are typically integrated in a functional-first simulator, where the functional model executes instructions and generates a dynamic instruction stream, potentially broken up into micro-ops. The core model receives this instruction stream and computes the time required to execute these instructions, querying branch prediction and memory hierarchy simulators to discover miss events.

2.1. One-IPC Models

A one-IPC core model is a cycle-by-cycle simulation technique where the core model executes a single instruction per cycle in the absence of long-latency miss events, like long-latency loads or front-end cache misses and branch mispredictions [Jaleel et al. 2008; Miller et al. 2010]. A one-IPC model attempts to simulate the performance of a typical out-of-order, multi-issue processor, but at a much higher level of abstraction, and therefore can simulate a core's performance at high simulation speed. These models, therefore, provide a faster way to simulate large, multi- and many-core systems that would otherwise not be feasible when using cycle-level core detail.

Unfortunately, there are a number of limitations that occur because of the use of one-IPC core models. One-IPC models do not faithfully model out-of-order MLP [Glew 1998; Chou et al. 2004] effects where multiple outstanding memory accesses are sent to the memory subsystem. Additionally, these models do not evaluate the amount of exposed ILP during the execution of the application itself. Issuing independent off-chip memory requests is one important way to hide the effects of long-latency memory requests that would normally stall an out-of-order core's forward progress. Additionally, the ILP exposed by the core will dictate the request rate as seen by the memory subsystem.

Because of these limitations, there has been an effort to develop both processor modeling techniques (interval modeling) as well as more advanced multicore simulation

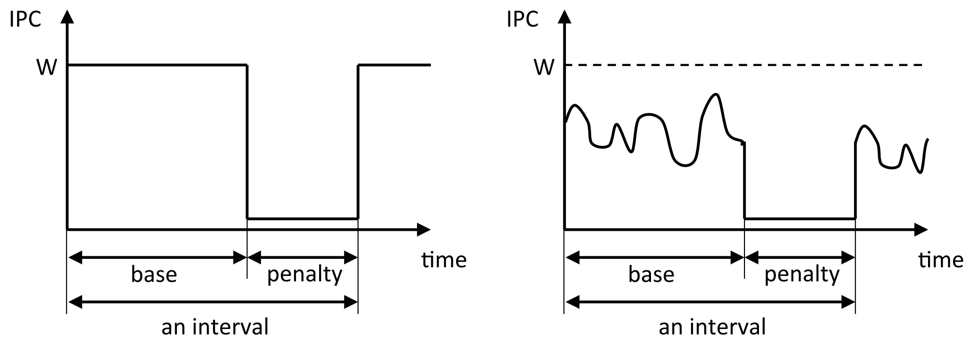


Fig. 3. A comparison between the estimation of IPC with interval modeling (left) and interval simulation (right). The simulation-based models take a dynamic sequence of micro-ops and adjust the core IPC over time.

techniques that use the insight from interval modeling to provide fast yet still accurate simulation.

2.2. Interval Modeling

The interval model [Eyerman et al. 2009] is an analytical core modeling technique designed to estimate the performance of applications using the most important factors of a balanced out-of-order core. The interval model exposes the effects of both ILP and MLP, and has shown good accuracy as compared to simulated program execution.

The interval model models core performance as a collection of intervals, composed of a period of executing instructions followed by a stalling event that causes the normal flow of execution to halt. The performance of the microprocessor core is analyzed at the dispatch stage, in contrast to the issue stage that was the focus of previous work [Karkhanis and Smith 2004]. At each miss event, the interval model estimates the penalty to apply to create an interval. Figure 3 illustrates this process. By combining a number of microarchitectural parameters with application characteristics, such as miss rates and the interval lengths between them, and the average window drain time after a branch misprediction, a performance prediction can be made. Using these parameters, the interval model was shown to have low error, around 7% for a four-issue machine, compared to cycle-level simulation of a single core [Eyerman et al. 2009].

Interval modeling is a good way to understand application performance of single-core, single-threaded applications on out-of-order machines. Nevertheless, interval modeling does not allow for modeling multiprogram and multithreaded workloads running on multi- and many-core processors.

2.3. Interval Simulation

Interval simulation [Genbrugge et al. 2010] is an extension of the interval model to allow for the simulation of multicore processors using the insights of the interval model. The major advantage of interval simulation over interval modeling is the ability to simulate multicore platforms. There is a tight relationship between core performance and the performance of shared resources, as the core performance affects the rate at which requests are sent to the memory hierarchy, which in turn affects core performance. This can be difficult to capture in a stand-alone offline analytical model. Interval simulation bridges this gap. Multiprocessor interval simulation is enabled by combining the insights from interval simulation for core performance with a detailed performance model of the memory hierarchy. An additional advantage of interval simulation is that the instructions (or micro-ops) are held and processed just one time and in program

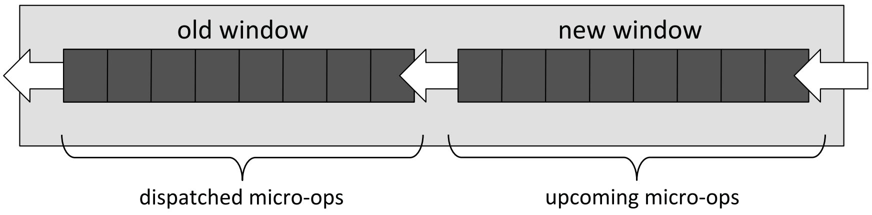


Fig. 4. A diagram of the new and old windows of interval simulation. Recently dispatched micro-ops (old window, left) allow us to predict the performance of upcoming instructions. Upcoming micro-ops (new window, right) are used to determine MLP.

order. This simulation technique allows for speedups of more than 10× when compared to M5 [Binkert et al. 2006], with an average error of 4.6% [Genbrugge et al. 2010].

2.3.1. New and Old Windows. The *new window* and *old window* are structures that allow for the approximation of performance in the interval simulation methodology. Figure 4 provides a representation of these two queues. Each window contains as many micro-ops as would exist in the ROB of an out-of-order processor. In the absence of miss events, such as long-latency loads, the ILP exposed by the ROB of an out-of-order processor will determine the current performance of the core (along with the core's maximum dispatch width). In interval simulation, the application ILP is determined using the old window. The old window contains a list of the most recently dispatched micro-ops, and the ILP exposed by the processor is computed by analyzing the critical path through this collection of micro-ops. MLP and other overlap effects (e.g., a branch miss hidden under long-latency load miss) are extracted from the new window. The new window, representing the upcoming micro-ops, remains full at all times to allow for the identification of MLP.

To determine how much progress can be made each cycle, the old window is analyzed using Little's law [Little 1961]. The instantaneous IPC is calculated as the number of micro-ops in the entire old window divided by the latency of the micro-ops on the critical path [Genbrugge et al. 2010]. By repeating this process for each micro-op (and accumulating the leftover work for future micro-ops), we can accurately estimate the application's ILP during the nonpenalty portion of an interval.

In addition to baseline application ILP, the performance impact of miss events also needs to be taken into account. The interval model distinguishes between front-end and back-end miss events; in the following sections, we will discuss how to account for each type of event.

2.3.2. Front-End Stalls. Front-end cache miss stalls, according to the original interval model, can be approximated by adding a penalty equal to the time it takes to resolve an I-TLB, or I-cache miss. Therefore, if a cache miss occurs, the resulting N cycle penalty is attributed to the level in the memory hierarchy where the hit occurs, and we estimate the penalty of the miss on the core as stalling dispatch for N cycles. This resulting delay occurs because the processor is unable to dispatch additional useful micro-ops while waiting for the next instruction to be fetched and decoded.

For branch mispredictions, we can estimate the performance penalty as the sum of two components. The first is the latency from dispatch up to the execution of the branch to detect the wrong prediction, and the second component is the time it takes to refill the front-end pipeline with instructions from the correct path. In interval simulation, we approximate the latency to determine the correct branch target as the critical path in the old window at the time that we dispatch the branch instruction. The front-end refill time is a constant defined by microarchitectural parameters.

2.3.3. Back-End Stalls. Back-end stalls, such as serialization and long-latency loads, are modeled using the insights from the interval model. A long-latency load instruction is one that causes the processor to stall because the core is not able to commit a load that is waiting for the result from the cache hierarchy. When this occurs, the core experiences a penalty that is equivalent to the load's miss latency [Eyerman et al. 2009]. As long-latency off-chip requests stall the forward progress of out-of-order processors, it is beneficial to issue as many of these independent requests in parallel, as it can lead to execution time reductions [Glew 1998; Chou et al. 2004]. MLP is the measure of the amount of simultaneous off-chip requests. To extract MLP in interval simulation, we scan the new window after dispatching each long-latency load to determine if there are upcoming independent loads that could be issued. Any dependent load in the window (e.g., in pointer-chasing applications) would not be issued until the original load has completed. Nevertheless, the out-of-order core can expose additional loads, issuing those loads whose inputs do not depend on the result of a prior long-latency load in the window. By iterating over the new window, we can issue independent loads, exposing application MLP. The processor penalty for multiple independent overlapping long-latency loads exposed as MLP is treated as the penalty of a single long-latency load.

Serialization instructions incur a penalty equal to the processor window drain time. Other second-order effects, such as overlapping I-cache/I-TLB and independent branch mispredictions with long-latency loads, are also taken into account by interval simulation. Additionally, serialization instructions that are encountered while handling MLP halt the detection of additional overlapped long-latency loads and drain the old window.

3. INTERVAL SIMULATION IMPROVEMENTS

Through the mechanistic modeling of modern out-of-order microprocessors, we are able to both better understand how they operate and use those assumptions to improve microprocessor simulation. In the next sections, we introduce several interval simulation enhancements to improve simulation accuracy.

3.1. Functional Unit Contention Modeling

Interval simulation [Genbrugge et al. 2010] assumes that a microprocessor is balanced with respect to its ROB size and dispatch width. This means that the front-end, back-end, and supporting structures, such as execution units, and load and store buffers have been sized to be large enough for a typical application.

Unfortunately, there exist classes of applications that fall outside of this realm. For example, on Intel's Nehalem architecture, there is only a single issue port for 64-bit floating-point multiply instructions. With a microbenchmark tailored to use only 64-bit multiplications, the maximum number of independent multiplies that can be maintained will be just one per cycle and not the four micro-ops that the processor was designed to dispatch. Therefore, when using interval simulation to model the performance of this microbenchmark, the result will not be what is expected. Interval simulation will report that the performance is four times higher (corresponding to the dispatch width of four micro-ops per cycle) than the machine can actually perform. Similar effects can occur with any instruction that can only be issued to a limited number of execution ports.

To be able to account for the discrepancy between dispatch width and issue capabilities, and therefore reduce simulation error, we propose resource contention modeling for interval simulation to improve the resulting accuracy with a negligible reduction in simulation performance.

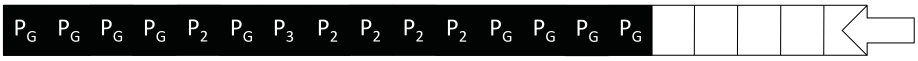


Fig. 5. An example of port-based issue contention in the updated interval simulation model. The window represents the interval simulation's old window for a ROB size of 20. The old window contains 15 micro-ops; the port number from which this micro-op can issue is shown as P_n, where n is the port number and G represents ports 0, 1, and 5. Micro-ops are inserted from the right.

Issue contention is modeled in the interval simulation model by extending the old window analysis in a new direction. Instead of only using Little's law to determine the current IPC of the application, we also take into account the utilization of other resource-sensitive units, such as pipelined and nonpipelined execution units and issue ports. By updating the effective dispatch rate calculation to take into account these additional restrictions, it can be determined how many execution units or ports are being used. Although out-of-order processors are able to schedule events whenever there is an issue slot available, if too many resources are used at a given point in time, the performance of the processor will be reduced accordingly.

The dispatch rate of a processor in the absence of miss events is estimated with interval simulation using Little's law. The dispatch rate of the processor is computed as the number of instructions in the ROB (typically the entire ROB in the absence of miss events) divided by the critical path length:

$$R_{dispatch} = N/L_{criticalpath}.$$

We model this in interval simulation as the number of micro-ops in the old window divided by the critical path length. This formula computes the instantaneous micro-ops/cycle that a processor can handle assuming an infinite dispatch width. Taking into account the dispatch width allows us to model a more realistic architecture. To maintain higher accuracy, the leftover portion of the dispatch rate is collected and used for dispatch at a later time:

$$R_{dispatch} = min(W_{dispatch}, N/L_{criticalpath}).$$

Finally, to take into account resource contention in a processor, we also keep track of the resource utilization of each micro-op. We extend interval simulation by keeping track of the number of issue slots that are used in the old window. The number of resources used in the old window puts a limit to the number of micro-ops that can be issued along the critical path. Each S_n is equal to the minimum number of cycles required to simulate the number of instructions for each contention component. For example, if we would like to model a single issue port for 64-bit pipelined floating-point multiplications, then for an ROB with 32 64-bit multiply operations, the minimum execution time for those operations will be 32 cycles, or 1 cycle per instruction per issue port. We can build a minimum execution time necessary to execute the collection of instructions by taking into account each microarchitectural restriction. The effective critical path is then extended by the number of resources required by the processor at that time:

$$L_{criticalpathcontention} = max(L_{criticalpath}, S_1, S_2, \ldots, S_n)$$
$$R_{dispatch} = min(W_{dispatch}, N/L_{criticalpathcontention}).$$

A detailed example of issue port contention in the interval simulation model is shown in Figure 5. In this example, the ROB window size (W) of the system is 20, and the number of micro-ops in the old window (N) is 15. We assume that the maximum dispatch rate ($W_{dispatch}$) is 4, and we use a value for the critical path ($L_{criticalpath}$) of 4. Additionally, we have several micro-ops that issue to different ports. There are nine generic micro-ops (P_G) that can issue to ports 0, 1, or 5; five micro-ops that issue to port 2 (P_2); and one that issues to port 3 (P_3):

$$\begin{aligned} W &= 20 \\ N &= 15 \\ W_{dispatch} &= 4 \\ L_{criticalpath} &= 4 \\ N_G &= 9 \\ N_2 &= 5 \\ N_3 &= 1. \end{aligned}$$

We first calculate the effective dispatch rate (instantaneous micro-ops per cycle, or μPC) using the original method provided by interval simulation. We then update the calculation taking into account issue contention:

$$\begin{aligned} R_{dispatch} &= min(W_{dispatch}, N/L_{criticalpath}) \\ R_{dispatch} &= min(4, 15/4) \\ R_{dispatch} &= min(4, 3.75) \\ R_{dispatch} &= 3.75 \end{aligned}$$

$$\begin{aligned} L_{criticalpathcontention} &= max(L_{criticalpath}, S_1, S_2, S_3, \dots) \\ L_{criticalpathcontention} &= max(4, ceil(N_G/3), N_1, N_2) \\ L_{criticalpathcontention} &= max(4, 3, 5, 1) \\ L_{criticalpathcontention} &= 5 \\ R_{dispatch} &= min(W_{dispatch}, N/L_{criticalpathcontention}) \\ R_{dispatch} &= min(4, 15/5) \\ R_{dispatch} &= min(4, 3) \\ R_{dispatch} &= 3. \end{aligned}$$

The resulting improvement in accuracy is apparent. In this example, the instantaneous performance with issue contention is 3 μPC, whereas without issue contention, we would see a core performance of 3.75 μPC—a 25% faster result.

In interval simulation [Genbrugge et al. 2010], a very low error of 4.6% compared to the M5 simulator was shown compared for a four-wide Alpha processor. This processor contains a sufficient number of execution units to prevent the issue stage from becoming a bottleneck in general and is therefore a balanced microarchitectural design. In recent microarchitectures, however, trade-offs are made in the number of execution units, which prevent some combinations of micro-ops from being executed in a single clock cycle. When comparing simulation results to real hardware, simulation models must be able to take these microarchitectural imbalances into account if good, absolute accuracy is to be expected.

3.2. Refilling the Window After Front-End Miss Events

In steady state, according to Little's law, the rate at which instructions enter the ROB (dispatch) equals the rate at which they leave the ROB. The interval model computes the latter by analyzing the critical path, which determines the rate of execution and applies Little's law to state that the effective execution rate equals the effective dispatch rate. After a miss event, the old window is flushed to denote the fact that the ROB no longer has independent on which instructions to operate. As instructions are dispatched into the ROB, the parallelism exposed by it increases and the effective execution rate picks up.

For applications where miss events are few, or spaced many instructions apart, this assumption holds and core performance can be estimated with good accuracy. However, in some applications, miss events and the associated start-up effects begin to dominate. This becomes a problem when miss events are spaced so closely that the ROB did not have a chance to be refilled—a situation that can cause relatively high errors in the original interval simulation method.

The solution to this problem is to recognize that after front-end miss events, Little's law is not applicable and dispatch can occur at the maximum rate, until the ROB (at this point represented by the contents of the old window) is filled up. We therefore modify the interval simulation model to dispatch at the machine width, rather than the effective dispatch rate computed through analysis of the critical path, as long as the old window is not full.

3.3. Modeling of Overlapped Memory Accesses

One drawback of using the interval model is that all memory accesses are sent to the memory hierarchy in program order, which may not represent the actual ordering seen by the memory hierarchy when out-of-order execution reorders loads with respect to other memory operations. In addition, many higher-abstraction level simulators such as Sniper [Carlson et al. 2011] or the atomic (synchronous) memory access mode in gem5 [Binkert et al. 2011] complete the simulation of each memory access in a single function call, updating all structures (e.g., cache tags) immediately. These implementations approximate memory access timing, which makes it difficult to properly detect overlapping memory accesses in the memory subsystem itself.

We update the interval simulation model to improve modeling of overlapping accesses in two ways. Both changes affect the marking of overlapped accesses and occur when scanning the window of upcoming micro-ops when the model searches for independent loads that can be hidden under an initial long-latency load.

3.3.1. Pending Hits. A first area for improvement arises when two independent loads access the same cache line that initially was not present in the last-level cache. The first access will be a long-latency event, as the data has to be fetched from DRAM. The second access, if made in short succession, will be a pending hit—that is, the cache line in question has already been requested from DRAM, but the request has not yet completed. A synchronous memory hierarchy, however, will return this second access as a hit (simulation of the first access was already completed, leaving the cache line allocated in the L1 cache). A similar problem was also encountered by Chen and Aamodt [2011], who add support for pending hits in an offline analytical performance model. To properly delay the second load, when marking micro-ops in the window looking for overlapping accesses, we check for independent loads that access the same cache line and mark the second load as dependent. This way, keeping the second load's latency as the L1 hit latency, it will complete a few cycles after completion of the first long-latency load.

3.3.2. Dependents of Independent Long-Latency Loads. The second case occurs when a chain of dependent loads, that is in itself independent of the initial long-latency load, contains a long-latency load inside of it. Consider the situation where a long-latency load A stalls the processor. The new window contains two additional loads B and C, where C depends on B but both are independent of A. In the original interval simulation model, both B and C would be marked as independent loads and would hence have been allowed to be fully hidden underneath resolution of the original load A.

However, if load B is itself also of long latency, it will not complete until after A completes. Load C, and all instructions depending on it, could therefore also not be assumed to be hidden under the long-latency event caused by A. To handle this case,

Table I. Microarchitectural Configuration

Component	Configuration
Processor	1 and 2 sockets, 4 cores per socket
Core	2.66GHz, 4-way dispatch, 128-entry ROB
Branch predictor	Dothan [Uzelac and Milenkovic 2009], 8 cycles penalty
L1-I	32KB, 4 way, 4 cycle access time
L1-D	32KB, 8 way, 4 cycle access time
L2 cache	256KB per core, 8 way, 8 cycle
L3 cache	8MB per 4 cores, 16 way, 30 cycle
Main memory	65ns access time, 8GB/s per socket
Interprocessor bus	QPI, 12.8GB/s per direction

we update the interval simulation model such that when checking the new window for overlapping loads, we do not mark loads as overlapped once they depend on a newly found long-latency load.

4. IW-CENTRIC SIMULATION

Interval simulation has been shown to be both a fast and accurate way to simulate the effects of microarchitectural changes on performance [Genbrugge et al. 2010; Carlson et al. 2011]. Nevertheless, there are some limitations that make extending the interval simulation model difficult. For example, extending interval simulation to support functional unit contention (Section 3.1) required the development of a new core modeling methodology to accurately estimate the timing of the microprocessor. Evaluating a system that consists of a variety of core configurations would become much more difficult if new modeling advances were needed for each core type. Therefore, a new core model that provides both higher fidelity with respect to core and cache hierarchy models would benefit the community if it were able to more accurately model a microarchitecture while still providing simulation speedup compared to detailed cycle-level processor models. We therefore introduce IW-centric simulation as this middle ground that provides more detail to allow for future core microarchitectural changes without modeling updates, while continuing to provide faster simulation speeds than traditional cycle-level simulators.

4.1. Overview

IW-centric simulation builds on many of the insights of interval simulation [Genbrugge et al. 2010]. Although the cache hierarchy and branch predictors continue to be simulated in detail (as is done in interval simulation), many structures such as the fetch and decode logic, additional hardware structures for issuing instructions such as the issue queue, and register renaming, as well as the commit stage, are not simulated in detail because of the assumption of a balanced processor microarchitecture. In addition, we assume a fixed penalty from the discovery of a mispredicted branch to the dispatching of new instructions (Table I), and we do not simulate the impact of wrong-path instructions on the cache and branch predictors. The key change required to enable higher accuracy in IW-centric simulation is to model the extraction of ILP by the processor in a more accurate way. Instead of approximating the ILP based on Little's law, where the out-of-order performance is estimated by processing instructions in order, the IW-centric model estimates performance by processing micro-ops out of order, in a way similar to how a real processor would issue them.

The IW-centric model replaces the new and old windows of the interval simulation methodology with a new structure that is sized as large as the ROB. Each dispatched micro-op is contained in this structure and is awaiting the results of the operations on which they depend. As the results are completed, additional micro-ops are issued to each functional unit, potentially out of order. A major difference between the IW-centric model and the interval model is that the complexity of the issue logic increases as the size of a processor's ROB grows. Additionally, the IW-centric model needs to monitor all input dependencies of each micro-op, with the cost of potentially checking micro-ops multiple times during their lifetime, increasing simulation costs over that of the interval simulation model. Because of this added complexity, the simulation time for IW-centric simulation increases; however, at the same time, accuracy is also improved.

4.2. Implementation Details

Many of the assumptions in the IW-centric core model use or extend those that have been established by interval simulation but allow for a more detailed simulation of application ILP. IW-centric modeling continues to be a dispatch-oriented model, where the dispatch stage of the processor is modeled in detail, and all penalties relate to this stage in the pipeline. Front-end events are handled in a way similar to interval simulation. When there is a branch misprediction or an I-cache or I-TLB miss, the processor waits for the front-end refill to complete before dispatching new instructions. Interval simulation estimates the branch penalty as the sum of the branch resolution time and the front-end refill penalty. In IW-centric simulation, the branch resolution time is modeled naturally by issuing the branch at the correct time, and only the front-end refill penalty needs to be added. Instruction decode is not modeled directly but is assumed to be able to keep up with the maximum dispatch rate in the absence of front-end miss events. The maximum number of micro-ops to dispatch per cycle is properly handled and is defined by the core microarchitecture. Loads are executed at issue time, and no special handling needs to take place for long-latency loads. Because the time of the next event can be easily tracked, the core can fast forward time when there are no events to be processed. All overlapping miss events are handled directly through register and memory dependency analysis. Issue port contention is modeled directly in the IW-centric model. Issue port occupancies are monitored and instruction issue is delayed until a free issue slot is available.

CPI stacks [Emer and Clark 1984; Zagha et al. 1996; Emma 1997] are a first-order method for understanding the causes of performance loss in an out-of-order processor without requiring additional simulation runs. Interval simulation allows for the computation of CPI stacks through the insights of interval modeling in a very easy way that is a direct result of the modeling itself. Although IW-centric simulation borrows many insights from interval simulation, the method for CPI stack computation needs to be modified because it is no longer possible to attribute the causes of all miss events (especially back-end events) directly. We therefore calculate CPI stacks in a similar (and more complex) way to a model that is suited to performance analysis on real hardware [Eyerman et al. 2006]. If the microarchitected dispatch width is limiting the forward progress of the core, we attribute the loss of cycles to this CPI stack component. For front-end miss events, such as a branch misprediction, we also attribute the number of cycles that it takes to restart the core directly to the component that caused the delay. This occurs because the out-of-order processor can no longer make forward progress and the reason for the delay is clear. For back-end miss events, the true cause of the delay is not as straightforward to determine. Therefore, we approximate the cause of the stall to the type of instruction present at the head of the window (serialization, load/store, floating-point, etc.). The rationale for this choice is that the most likely cause of the delay is the instruction at the head of the window.

In addition to timing model details, there are other factors that can contribute to the results of the timing simulation. Multicore simulation takes place by first functionally executing the instructions for each core and then by feeding the instructions into each individual core model. This results in a timing model that is slightly behind the natively executed instructions by a maximum of the number of instructions in the ROB of the target microarchitecture. Additionally, because of this direct functional execution, cache and branch predictor pollution, which normally occurs in out-of-order processors because of wrong-path instruction execution, is not present in this model.

5. EVALUATION AND METHODOLOGY

5.1. Simulation Infrastructure

We implemented the three core models described earlier, one-IPC, interval simulation, and the IW-centric core model, as well as the enhanced interval model with issue contention, in the Sniper multicore simulator [Carlson et al. 2011]. Sniper is an execution-driven, user-level simulator that uses functional-first simulation with timing feedback based on the Pin dynamic instrumentation framework [Luk et al. 2005] and the Graphite simulation infrastructure [Miller et al. 2010]. It implements parallel simulation to improve simulation speed, keeping threads synchronized using a quantum-based barrier synchronization with a quantum of 100ns. The Wisconsin Wind Tunnel II [Mukherjee et al. 2000] uses a similar approach but guarantees functional and timing correctness by keeping the quantum between barriers small enough to prevent causality violations. Instead of using this more conservative approach, Sniper allows for causality violations in exchange for much greater simulation speed in a similar fashion to SlackSim [Chen et al. 2010].

We use these core models to simulate the execution of a variety of benchmarks. All core models utilize the same branch predictor and cache models, making a direct comparison between them possible. We also compare simulated results to running the same application on real hardware, which allows us to evaluate the accuracy of each core model in addition to its simulation performance.

We model a dual-socket, quad-core per socket configuration that approximates an Intel Nehalem–based server machine. Processor cores are four-wide, have a 128-entry ROB, and run at 2.66GHz. Each core has private L1 instruction and data caches in addition to a unified private L2 cache, and all four cores in a package share a L3 cache and DRAM controller. Two quad-core processors are connected using a coherent QPI connection and make up a single shared-memory machine. Microarchitectural parameters can be found in Table I. When modeling issue contention, we assume that the architecture has a number of issue ports that accept a subset of all micro-ops. Additionally, each issue port can accept a single micro-op for execution per clock cycle. In the Nehalem microarchitecture, there are five modeled issue ports in total. One is dedicated to loads, and a second one can be used only by stores. The other three ports are specialized for branches, floating-point additions, and floating-point multiplications, respectively, and in addition accept all types of integer instructions. The complete mapping of micro-ops to issue ports is configured according to Fog [2013].

The core processor models are configured as either the one-IPC core model, interval simulation core model, or the IW-centric core model. Both the interval and IW-centric models support optionally enabling functional-unit issue contention by enabling the modeling described in Section 3.1 for the interval model or by directly accounting for execution unit occupancy on a cycle-by-cycle basis in the IW-centric model. By comparing simulations with and without this option, we will be able to gauge the increase in accuracy of this additional modeling step. Both the bus and DRAM contention models are configured to use history-list–based contention [Miller et al. 2010]. The simulation models and model-specific parameters are configured as listed in Table II.

Table II. Simulator Configuration Options

Component	Configuration
Core models	IW-Centric, Interval, and One-IPC
Core issue contention models	Enabled, disabled
Bus contention model	History-list
DRAM contention model	History-list
Synchronization method	Time-based barrier
Synchronization interval	100ns
OS emulation	Futex replacement
Reschedule cost (cycles)	1k @ 1 core, 10k @ 2 cores 15k @ 4 cores, 25k @ 8 cores

Table III. Benchmarks and Input Sets

Benchmark	Input Set
barnes	32,768 particles
cholesky	tk29.O
fmm	32,768 particles
fft	4M points
lu.cont	1,024×1,024 matrix
lu.ncont	1,024×1,024 matrix
ocean.cont	1,026×1,026 ocean
ocean.ncont	1,026×1,026 ocean
radiosity	–room
radix	1M integers
raytrace	car –m64 –a4
volrend	head
water.nsq	2,197 molecules
water.sp	2,197 molecules

5.2. Hardware Validation

Hardware validation of Sniper with its respective core models was performed on a dual-socket server based on the Intel Xeon X5550 processor. Benchmark threads are each pinned to their own core using the *pthread_setaffinity_np()* API. In configurations with one, two, or four threads, we allocate threads to the first socket, whereas eight-thread configurations use both sockets. SpeedStep and Turbo Boost are disabled to ensure that the processor cores always run at the intended 2.66GHz frequency. Finally, we disable both hyperthreading (simultaneous multithreading (SMT)) and hardware prefetchers, as these are also not yet validated in Sniper.

5.3. Benchmarks

For validation and evaluation, we use the SPLASH-2 benchmarks [Woo et al. 1995]. SPLASH-2 is a well-known benchmark suite that represents high-performance, scientific codes. Table III provides more details on these benchmarks and the inputs that we have used. The benchmarks were compiled with GCC 4.3.2 in 64-bit mode with –O3 optimization and with the SSE and SSE2 instruction set extensions enabled. On real hardware, we measure the length of time that each benchmark took to run its parallel section (region of interest) 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. In addition, performance counter information was collected using the *perf stat* infrastructure. This allowed us to validate microarchitectural characteristics such as branch misprediction and cache miss rates.

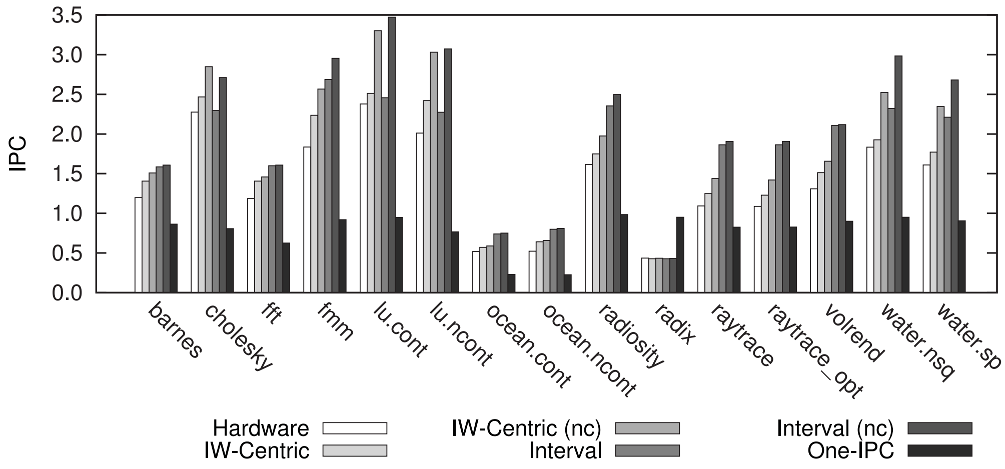


Fig. 6. Single-core IPC on real hardware and simulated using a variety of core models and benchmarks. Models without issue contention enabled are labeled (nc) for no issue contention.

Table IV. Single-Core Average Absolute Execution Time Errors and Average Absolute Differences for Each Simulation Model

	Execution Time		MPKI Average Absolute Difference	
Core Model	*avg. abs. err.*	*max. abs. err.*	*bp*	*L3*
IW-Centric	11.11	18.76	0.17	0.09
IW-Centric (nc)	21.83	33.62	0.17	0.09
Interval	24.29	41.61	0.17	0.09
Interval (nc)	31.76	42.89	0.17	0.09
One-IPC	92.05	182.04	0.19	0.09

6. SIMULATION ACCURACY COMPARISON

In this section, we characterize Sniper's simulation accuracy when compared to our dual-socket Intel Nehalem server. We will vary the core model, comparing one-IPC modeling to interval simulation and the IW-centric core model while keeping the memory hierarchy and branch predictor constant. This comparison allows us to keep all of our infrastructure the same to isolate the differences between the core models.

6.1. Absolute Accuracy Comparison

Starting with single-core results, Figure 6 compares the IPC obtained by real hardware with that predicted by the different core models. The (*nc*) variants of IW-centric and interval simulation disable modeling of issue contention. Unsurprisingly, the one-IPC core model incurs large errors (92% on average with a maximum of 182%; Table IV) and generally underestimates performance on benchmarks where ILP can be exploited to obtain an execution speed of more than one instruction per cycle. On applications with MLP such as *ocean*, the one-IPC model does not allow for multiple simultaneous outstanding memory requests and serializes their latency. Alternatively, performance of the *radix* benchmark suffers because of dependency chains through instructions with nonunit latency; here, the one-IPC model *overestimates* execution speed. In contrast, the more advanced IW-centric and interval simulation models can provide a much more accurate estimation of execution speed (24.3% for interval simulation and just 11.1% for IW-centric on average, with maximum errors of 41.6% and 18.8%, respectively).

Table V. Average and Maximum Absolute Errors Across the Simulation Models

	IW-Centric		IW-Centric (nc)		Interval		Interval (nc)		One-IPC	
Number of Cores	*avg.*	*max.*	*avg.*	*max.*	*avg.*	*max.*	*avg.*	*max.*	*avg.*	*max.*
1	11.11	18.76	21.83	33.62	24.29	41.61	31.76	42.89	92.05	182.04
2	9.68	18.79	20.56	34.64	22.15	41.50	29.52	42.57	90.26	162.78
4	14.77	39.19	22.38	39.96	21.76	40.72	28.32	42.37	78.40	150.06
8	20.79	40.32	27.06	43.89	25.91	41.56	31.11	45.08	54.58	100.26

For some benchmarks, including *lu.cont* and *lu.ncont*, pressure on execution units is quite high, so taking issue-contention modeling into account significantly improves accuracy for these applications. This is especially important for benchmarks with high IPC, which are unrestricted by other hardware components such as memory latency. On average, enabling issue contention modeling improves average accuracy of interval simulation from 31.8% to 24.3%.

In addition to IPC error, Table IV shows a comparison of simulated branch misprediction and cache miss rates as compared to real hardware. These do not depend on the core model used and are in any case quite low.

6.2. Multicore Scaling Comparison

Moving on to multicore results, Figure 7 plots the speedup obtained for different core counts, both on real hardware and as predicted using the different core models. Most benchmarks, including, for instance, the *barnes* and *fmm* applications, exhibit good scaling on real hardware, as an increase in core count leads to an almost linear increase in performance. This behavior is predicted correctly by all core models.

Notable exceptions are the two variants of *ocean*. This application has a large dataset and is DRAM bandwidth bound; running this benchmark with more than two threads does not provide an additional benefit in performance. This fact is predicted correctly by the IW-centric and interval simulation core models. In contrast, the one-IPC model does not take MLP into account but stalls the core on each DRAM access, underestimating effective DRAM bandwidth pressure and hence overestimating the application's scalability.

Other benchmarks, such as *lu.ncont*, scale well up to four cores, whereas the eight-core version sees only limited gains. This is because beyond four cores, the second processor chip is being used, which incurs intersocket communication over the QPI links. On real hardware, contention on these links limits performance. The *interval* and *IW-centric* core model predict this correctly. However, the one-IPC core model predicts perfect scaling. This is again because the one-IPC model predicts per-core performance too low (by a factor of 2.5×, see Figure 6), which in turn leads to the simulated cores generating a request rate made to the QPI that is too low. The QPI bus is therefore not saturated when being driven by the one-IPC model, which incorrectly leads to the prediction of favorable scaling on the dual-socket run of *lu.ncont*.

This difference in relative (scaling) accuracy can in fact be correlated with each core model's absolute accuracy. Although architects usually claim to require only *relative* simulation accuracy, scaling often largely depends on contention of shared resources such as DRAM and QPI bandwidth, which in turn requires request rates and thus core performance to have a certain level of *absolute* accuracy. Analyzing the summary of absolute accuracy provided in Table V, we can see that the IW-centric core model has very good accuracy (11.1% on average, with a maximum of 18.8%) for single-core results. Error goes up with increasing core count, mostly because of modeling errors in Sniper's memory hierarchy, which dominates performance for four- and eight-core results leading to a 20.8% average error with a maximum of 40% for the *radix* benchmark. This benchmark experiences large numbers of TLB misses, and

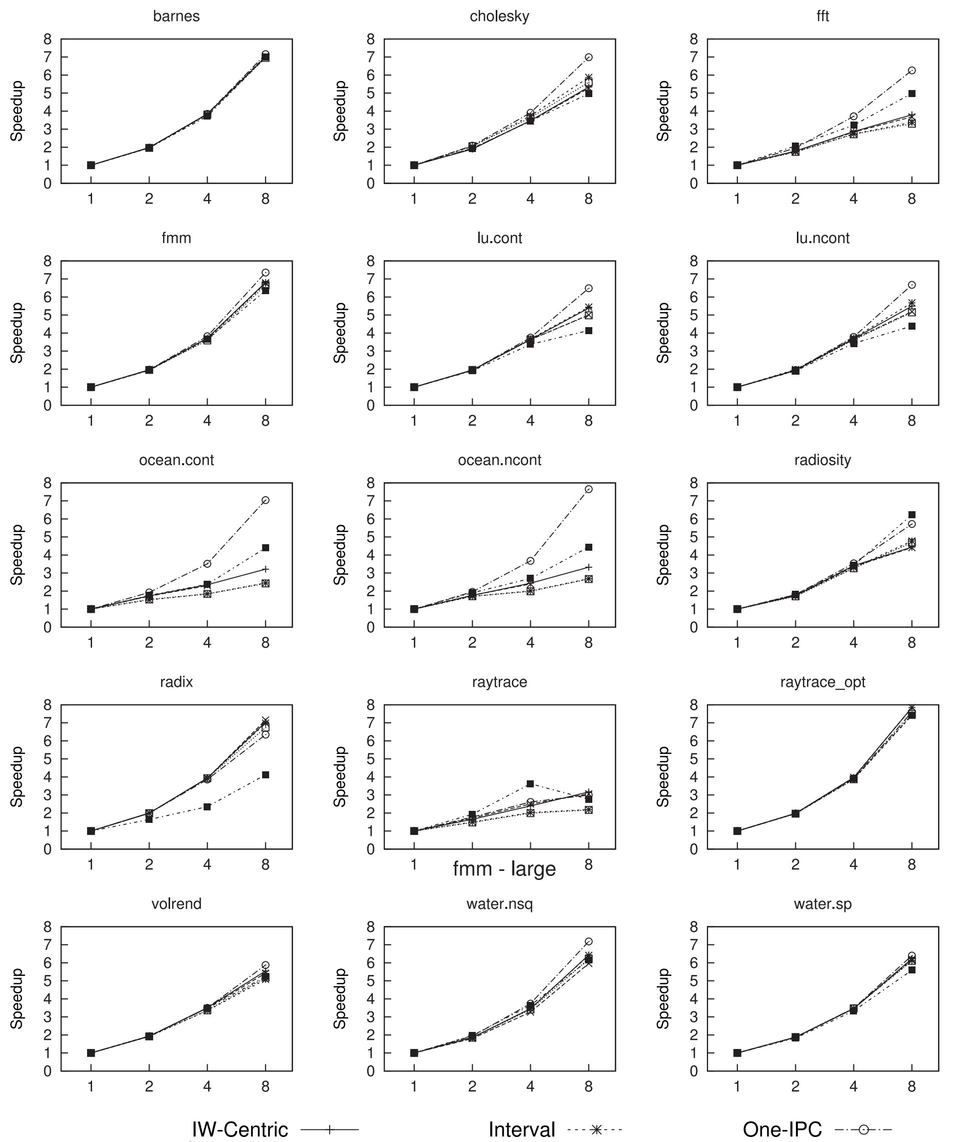


Fig. 7. Relative performance speedup predictions for the SPLASH-2 applications from one to eight cores.

queuing delays caused by the write-back of evicted dirty cache lines push the limits of the level of detail provided in the memory hierarchy. Looking at the other core models, interval simulation starts off at an average 24.3% error for single-core results. With increasing core count, the memory subsystem again starts to dominate results, reducing the contribution of the core model somewhat and leading to an eight-core error of 25.9% with a maximum, again for the *radix* benchmark, of 41.6%. Finally, the one-IPC core model starts off with a 92.0% average error (up to 182.0%) for single-core results, whereas the eight-core error is slightly lower; due to the one-IPC model's inaccurate pressure on the memory subsystem, average error is still high at 54.6%.

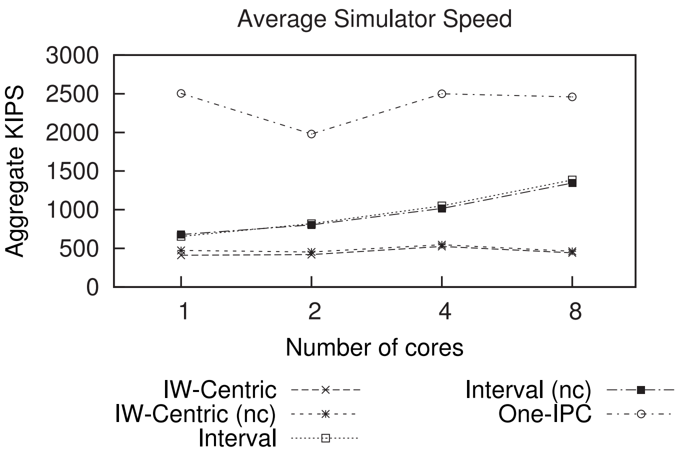


Fig. 8. Average simulator speed, in KIPS, for a variety of simulation models. Models without issue contention enabled are labeled (nc) for no issue contention. Core-level issue contention adds very little simulation overhead but can greatly improve accuracy.

7. SIMULATION SPEED COMPARISON

In Figure 8, we show the average simulation speed (in 1,000 simulated instructions per second of wall time, or KIPS, aggregated over all simulated cores) across a number of different core models while changing the size of the simulated system from single-core to dual-socket quad-core (8 cores in total). All simulations were done on an Intel Xeon E5-2650L (Sandy Bridge)-based system running at 1.80GHz. This machine has 16 cores, so parallelism in the simulator can optimally be exploited, as each simulator thread (running the timing model for one simulated core) can run on its own private host core.

Keeping in mind the accuracy of each core model, we can see a clear trade-off of simulation speed versus accuracy. Concentrating on single-core simulations first, the one-IPC core model runs the fastest at over 2.5 MIPS on average, and up to 5.5 MIPS for the *fmm* benchmark. The interval model's much greater accuracy comes at a cost in simulation speed but still reaches 680 KIPS on average, whereas the more detailed IW-centric core model reaches a single-core simulation speed of around 450 KIPS. An interesting observation is that issue contention, when implemented as described in Section 3.1, does not affect simulation speed much but can significantly improve accuracy on several benchmarks. This can be clearly seen in Figure 9, which plots simulation speed against accuracy (average absolute error to real hardware for all single-threaded workloads): disabling issue contention modeling (the *(nc)* variants) adds around 10% in additional modeling error but does not significantly improve simulation speed. Comparing IW-centric, interval, and one-IPC models provides the architect with a clear choice of core models with different simulation speed over accuracy trade-offs.

For multicore simulations, the memory hierarchy and synchronization bottlenecks inside the simulator start to become important—both for modeling shared resources such as LLC and DRAM components, and for keeping local clocks of each simulated core synchronized. Using the interval simulation core model, a speedup of more than 2× can be achieved when simulating an eight-core system, where the simulation model of each core can run on its own host core. When using the one-IPC model, the core model itself is too simple to have any effect on execution time; here, simulation speed is limited by the memory subsystem, which is shared between the simulated cores and therefore can provide only limited parallel speedup. Finally, the IW-centric core

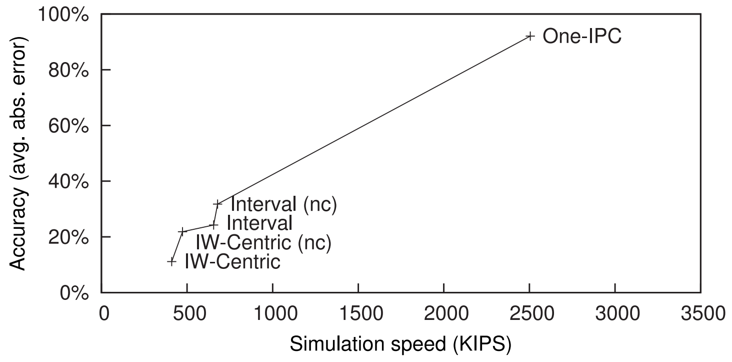


Fig. 9. Simulation speed versus modeling error of all core models for single-core runs.

Table VI. Microarchitectural Configuration for Private and Shared L2 Cache Configurations Used for One-IPC versus Detailed Core Model Comparisons

Component	L2 Configuration	
	private	*shared*
Size	256KB per core	1MB per 4 cores
Associativity	8 way	16 way
Access latency	8 cycles	30 cycles

model shows limited speedup because of bottlenecks in memory allocation. Whereas the one-IPC and interval simulation core models do not dynamically allocate memory, the IW-centric model—as currently implemented in Sniper—relies heavily on dynamic memory allocation, which, in combination with Pin's memory allocator, leads to poor scalability on multiple host cores. Using better allocation techniques such as circular buffers or pool allocation should alleviate this problem.

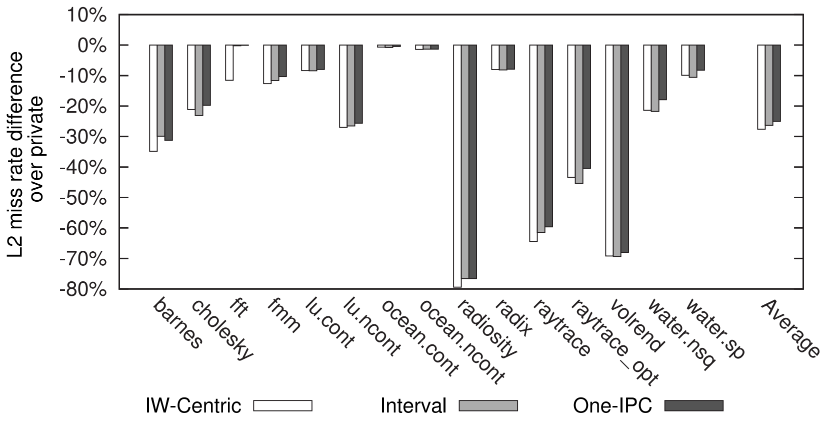
8. CORE MODEL RESOLUTION AFFECTS MICROARCHITECTURE CONCLUSIONS

Although one-IPC core models are often used to reduce simulation time during microarchitectural evaluations, this increase in performance comes with a trade-off. The one-IPC models do not model a number of key core properties that can be crucial to being able to make accurate design decisions when comparing numerous different microarchitectural design choices. The key differences of a one-IPC model and a modern out-of-order core model are the ability to model both application ILP, and therefore the memory request rate appropriately, as well as the MLP, or the amount of memory parallelism of an application.

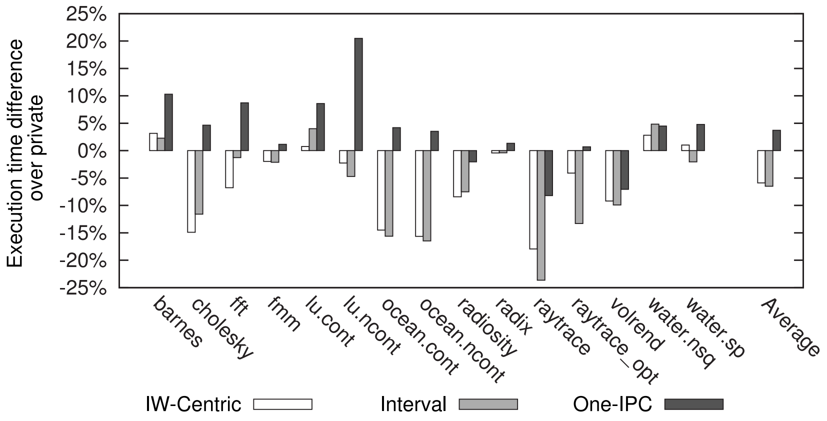
To demonstrate some of the limitations of using one-IPC models to evaluate processor performance, we show that even when modifying only the caches of a system (a memory hierarchy study), we still need the resolution in ILP and MLP to be able to accurately predict microarchitectural performance trends (relative accuracy).

Our experimental configuration is based on the original configuration in Table I but with slight modifications. For our baseline configuration, we assume four cores, with two levels of cache using a private hierarchy, with the L2 of each core at 256KB (Table VI). We compare the configuration with a shared L2 cache across all four cores (Table VI). In this configuration, the total cache capacity remains the same, at 256KB per core, but the latency to access the shared L2 cache goes up to 30 cycles from 8.

Figure 10(a) shows the percentage change in miss rates between the private and shared cache systems, where negative (lower) changes show fewer misses in the shared L2 configuration. We see that across all benchmarks, a shared L2 cache configuration



(a) L2 miss rate



(b) Runtime

Fig. 10. A comparison of L2 miss rates and application execution time of shared versus private caches, as predicted by the one-IPC, interval, and IW-centric core models.

reduces the number of L2 misses significantly when compared to the private L2 cache configuration—up to a 75% reduction in some cases. This is caused by the fact that the shared configuration can avoid the data duplication that is present in the private caches and therefore has a higher effective capacity. More importantly, we see that the cache miss rate changes are stable across all of the core models considered.

When taking application execution time into account, however, the core models no longer agree. Moving from private to shared caches, a reduction in cache miss rate avoids expensive DRAM accesses, but this comes at the cost of a significant increase in latency of (much more common) L2 hits. How this trade-off affects total application execution time will depend on the relative occurrence of both events, and on how much of the corresponding latency can be overlapped by the core. In Figure 10(b), the relative execution time changes are presented between the private L2 and shared L2 cache configurations. A negative value represents a execution time decrease (or improved application performance). According to both the interval and IW-centric core models, most applications prefer the shared L2 cache configuration, which reduces their execution time by more than 5% on average, with a maximum of around 20%. This indicates that most of the latency increase incurred in the common case (i.e., shared L2 hits) is overlapped by the out-of-order core, whereas the reduction in DRAM misses (which are too long to be fully overlapped with useful work) directly corresponds to improved application performance. However, the one-IPC model predicts a very different scenario, with only a few applications showing slight performance improvements. On average, the results show that applications will run *longer* when using a shared L2 cache—a completely different conclusion. This is because the one-IPC model cannot correctly determine how much of the L2 latency can be overlapped and overestimates its impact on application execution time.

In other words, the one-IPC core model concludes that private caches are best, as the increase in L2 hit latency for shared caches cancels out the reduction in latency caused by avoiding DRAM accesses. In contrast, the more detailed models show that shared caches are the better option because the increased L2 hit latency can be hidden almost completely by out-of-order execution. This experiment clearly shows that even when performing experiments that seem to only affect the memory subsystem, certain aspects of the core cannot be ignored but need to be modeled faithfully. As indicated by the results of Figure 10, interval simulation poses the right balance for this type of research, as it is able to make the same conclusions yet eschews modeling the more intricate but less important details of the core that are included in the IW-centric core model.

8.1. Selecting an Appropriate Core Model

The decision of which core type to use largely depends on the type of study being conducted. In this section, we present several example situations to guide this choice while conducting microarchitectural research or development.

One should choose the appropriate core model with the appropriate capabilities for a given experiment. For example, if one is studying heterogeneous multicore machines, such as a big.LITTLE configuration [Greenhalgh 2011], using a one-IPC core model will not provide sufficient fidelity to compare the performance of two different classes of processors. Additionally, if one would like to investigate the impact of the size of the MSHRs of a core (and its MLP), then using the one-IPC core model, which does not model MLP, will not be sufficient for this study. Accordingly, using interval simulation to study detailed front-end pipeline changes might not be the best match, as the front-end pipeline in the interval simulation model is assumed to be able to keep up with the requirements of the maximum dispatch rate of the processor.

During early-stage research where fast iterations can speed discovery, overly detailed models can stand in the way of quickly converging on a final design. Therefore, while conducting early research, it might be acceptable to start with higher-level simulation techniques, especially if one can use techniques that provide large simulation speed improvements. After the discovery phase has been completed, the move to more detailed models can occur (along with the updated details necessary for the new features) to validate the design (see Figure 1 for an example). This design methodology can also occur before the use of a simulator is needed, with simple proof-of-concept models and formulas. Similarly, as a design moves closer to physical implementation, more detail is required to verify both the accuracy of the result, as well as the ability to construct the new design in a way that meets stated goals. Although there is no one answer to the question of which models to use, we demonstrate that caution needs to be taken when using very high level models like one-IPC core models, especially when trying to compare small performance differences.

9. FUTURE WORK

9.1. Validation of New Components

In the most recent version of Sniper, we have implemented a number of new components, such as in-order and SMT cores, through updates to the IW-centric core model. These new models were straightforward to implement as a part of the IW-centric core, as the software model of the core operates like a traditional processor and does not require additional analytical modeling enhancements. In addition to simulating new types of cores, we have also added a stride prefetcher and DRAM cache to the memory subsystem of Sniper.

Although the components developed for the memory hierarchy of Sniper apply to both core models, developing SMT for interval simulation requires a new analytical model that takes into account the sharing of several components, such as the fetch and decode stages, the ROB, and the functional units. As many modern processors implement SMT, implementing such a model would allow us to better understand the performance characteristics of these cores.

One additional option for follow-up work would be to validate the SMT and memory hierarchy components against modern hardware platforms to better match the characteristics of the hardware and determine the resulting accuracy.

9.2. Comparison to Cycle-Level Simulators

The improved accuracy of the IW-centric core model leads to an important question about high-level simulation methodologies. We find that this higher accuracy can be attributed to the more detailed simulation of the processor core. Taking this one step further, it would be interesting to compare the speed and accuracy of Sniper to that of more detailed cycle-level simulators. For example, detailed industrial simulators tend to run at speeds between 1kHz and 10kHz, whereas cycle-level simulators run at speeds between 10s and 100s of KIPS [Chiou et al. 2007]. Two cycle-level simulators, Flexus [Hardavellas et al. 2004] and gem5 [Binkert et al. 2011], perform at 25 KIPS [Adileh et al. 2012] and 200 KIPS [Beckmann et al. 2011], respectively. Unfortunately, direct comparisons between simulators are not always possible or straightforward. For example, one cycle-level simulator, MARSSx86, achieves an average of 160 KIPS when simulating the SPEC CPU2006 benchmark suite and an average of 200 KIPS when simulating parallel multithreaded workloads [Patel et al. 2011; Ghose et al. 2012]. In addition, they show an average absolute error of 23% when comparing the SPEC2006 benchmark suite to an Intel Xeon E5620 [Ghose et al. 2012]. Sniper simulates $2\times$ to $3\times$ faster and has a similar accuracy, but there are additional variables that need to be taken into account to perform an accurate comparison, such as enabling the same level of detail and simulating the same collection of benchmarks and microarchitecture. More work is therefore necessary before we will be able to compare the accuracy of different simulation options with real hardware.

In addition to comparing simulation results to hardware, high-level models have also been compared to detailed cycle-level simulation models. By integrating interval simulation into gem5 (then M5), Genbrugge et al. [2010] show that when compared to using gem5's detailed core model, interval simulation performs an order of magnitude faster with an average error of 4.6%.

Through the addition of new core models, and other features such as prefetchers, Sniper now allows for the simulation of a larger selection of microarchitectural options. A next step for future work will be to compare both the simulator performance and the simulated workload results to allow microarchitects to better understand the trade-offs being made when choosing between simulation models.

10. CONCLUSION

The microarchitectural trends of modern computer systems continue to push the limit of simulation technology. With larger caches, larger numbers of cores, and increasingly complex memory hierarchies, the complexity and therefore simulation time required to accurately simulate these architectures has increased. There is a desire for faster simulation throughput, and therefore faster core models are one way to move closer to this goal.

To meet these ever-increasing time constraints, there could be a push to move toward using faster, naive core models, such as one-IPC models, to determine the optimal next-generation microarchitectural configurations. We show though that core models like the one-IPC model can lead to misleading and incorrect results and conclusions in practical design studies. These simple models do not properly take into account individual core ILP or MLP, which in turn leads to large discrepancies both from an absolute accuracy and a relative accuracy perspective.

In this work, we provide an overview of fast and accurate high-level core models for use in microarchitectural simulation. We first describe two enhancements to interval simulation, issue contention, and improved dependency analysis tracking. With these enhancements, we demonstrate how more accurate results can be obtained with almost the same simulation performance. We also introduce a new core model, the IW-centric model, that improves accuracy compared to interval simulation while maintaining high simulation speed. Through the use of interval simulation and IW-centric core models, one can speed up microarchitectural simulation while maintaining accuracy of the resulting architectural performance evaluation. Both core models provide good absolute accuracy (11.1% for IW-centric and 24.3% for interval simulation) and provide fast simulation speeds (with IW-centric performing just 1.5× slower than interval simulation).

ACKNOWLEDGMENT

We thank the anonymous referees and associate editor for their valuable feedback and suggestions.

REFERENCES

A. Adileh, C. Kaynak, P. Lotfi-Kamran, and S. Volos. 2012. CloudSuite on Flexus. Retrieved July 22, 2014, from http://parsa.epfl.ch/simflex/doc/CloudSuite-on-Flexus-isca12.pdf.

E. K. Ardestani and J. Renau. 2013. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 448–459.

B. Beckmann, N. Binkert, A. Saidi, J. Hestness, G. Black, K. Sewell, and D. Hower. 2011. The gem5 Simulator. Retrieved July 22, 2014, from http://www.gem5.org/dist/tutorials/isca_pres_2011.pdf.

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. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2, 1–7.

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

T. E. Carlson, W. Heirman, K. V. Craeynest, and L. Eeckhout. 2014. BarrierPoint: Sampled simulation of multi-threaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 52:1–52:12.

T. E. Carlson, W. Heirman, and L. Eeckhout. 2013. Sampled simulation of multi-threaded applications. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

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

X. E. Chen and T. M. Aamodt. 2011. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. *ACM Transactions on Architecture and Code Optimization* 8, 3, 10:1–10:28.

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

Y. Chou, B. Fahs, and S. Abraham. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 76–87.

E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. 2008. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA)*. 77–86.

L. Eeckhout, R. H. Bell Jr, B. Stougie, K. De Bosschere, and L. K. John. 2004. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. 350–361.

L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere. 2003. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro* 23, 5, 26–38.

J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. 2002. Asim: A performance model framework. *Computer* 35, 2, 68–76.

J. S. Emer and D. W. Clark. 1984. A characterization of processor performance in the VAX-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA)*. 301–310.

P. G. Emma. 1997. Understanding some simple processor-performance limits. *IBM Journal of Research and Development* 41, 3, 215–232.

S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. 2006. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 175–184.

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

A. Fog. 2013. Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs. Retrieved July 22, 2014, from http://www.agner.org/optimize/instruction_tables.pdf.

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

K. Ghose, A. Patel, F. Afram, H. Zheng, and J. Tringali. 2012. MARSS: Micro Architectural Systems Simulator. Retrieved July 22, 2014, from http://cloud.github.com/downloads/avadhpatel/marss/Marss_ISCA_2012_tutorial.pdf.

A. Glew. 1998. MLP yes! ILP no! In *Proceedings of the ASPLOS Wild and Crazy Idea Session.*

P. Greenhalgh. 2011. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. ARM white paper.

N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. 2004. SimFlex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Performance Evaluation Review* 31, 4, 31–34.

A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. 2008. CMP$im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA 2008*. 28–36.

T. Karkhanis and J. E. Smith. 2004. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. 338–349.

A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. 2007. RAMP Blue: A message-passing manycore system in FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*. 54–61.

J. D. Little. 1961. A proof for the queuing formula: $L = \lambda W$. *Operations Research* 9, 3, 383–387.

G. Loh, S. Subramaniam, and Y. Xie. 2009. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. 53–64.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

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

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

S. Nussbaum and J. E. Smith. 2001. Modeling superscalar processors via statistical simulation. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 15–24.

M. Oskin, F. Chong, and M. Farrens. 2000. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*. 71–82.

A. Patel, F. Afram, S. Chen, and K. Ghose. 2011. MARSS×86: A full system simulator for ×86 CPUs. In *Proceedings of the Design Automation Conference (DAC)*. 1050–1055.

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

D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. 475–486.

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

T. Taha and D. Wills. 2008. An instruction throughput model of superscalar processors. *IEEE Transactions on Computers* 57, 3, 389–403.

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

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

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

M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. 1996. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC)*. Article No. 16.

Received December 2013; revised March 2014; accepted May 2014