Node Performance and Energy Analysis with the Sniper Multi-Core Simulator

Trevor E. Carlson[1], Wim Heirman[2], Kenzo Van Craeynest[1], Lieven Eeckhout[1]

[1]Ghent University, Belgium
[2]Intel, ExaScience Lab

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. Sniper provides this balance 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. With per-function advanced visualization and coupled power and energy simulations, the Sniper multi-core simulator can provide a fast and accurate way both to understand and optimize software for current and future hardware systems.

1 Sniper Overview

Sniper is a next generation parallel, high-speed and accurate x86 simulator. This multi-core simulator is based on the interval core model and the Graphite simulation infrastructure [1], allowing for fast and accurate simulation and for trading off simulation speed for accuracy to allow a range of flexible simulation options when exploring different homogeneous and heterogeneous multi-core architectures.

The Sniper simulator allows one to perform timing simulations for both multi-program workloads and multi-threaded, shared-memory applications running on 10s to 100+ cores, at a high speed when compared to existing simulators. The main feature of the simulator is its core model which is based on interval simulation, a fast mechanistic core model [2]. Interval simulation raises the level of abstraction in architectural simulation which allows for faster simulator development and evaluation times; it does so by 'jumping' between miss events, called intervals [3]. Sniper has been validated against multi-socket Intel Core 2 systems and provides average performance prediction errors within 25% at a simulation speed of up to several MIPS [4].

This simulator, and the interval core model, is useful for uncore and system-level studies that require more detail than the typical high-level models, but for which cycle-accurate simulators are too slow to allow workloads of meaningful sizes to be simulated. As an added benefit, the interval core model allows the generation of CPI stacks, which show the number of cycles lost due to different characteristics of the system, like the cache hierarchy or branch predictor, and lead to a better understanding of each component's effect on total system performance. This extends the use for Sniper to application characterization and hardware/software co-design. The Sniper simulator is available for download at `http://snipersim.org` and can be used freely for academic research.

2 Speeding Up Simulation

In this section we describe a number of strategies that, when used together, provide the methods where we speed up micro-architectural simulation in Sniper while still maintaining accuracy.

2.1 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 [3]. 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 [2]. The branch predictor and cache models are simulated in detail. The cache 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 simulated 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 [3] for a more elaborate description of the interval simulation paradigm.

2.2 Parallel Simulation

An additional way to speed up architectural simulation in the multi/many-core era is to parallelize the simulation infrastructure in order to take advantage of increasing core counts. This is needed because most of the improvement in processor performance recently has come from increasing the number of cores per processor, and therefore parallelizing the simulator allows us to take advantage of this performance trend.

One of the key issues in parallel simulation is the balance of accuracy versus speed. Cycle-by-cycle simulation advances one cycle at a time, and thus a parallel simulator's threads (which map directly to the application threads) would need to synchronize every cycle to maintain accuracy. The resulting performance of this approach is not very high 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 [5]. 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 [6].

More recently, researchers have been trying to relax multi-threaded simulation even further, beyond the critical latency. If done correctly, the idea would be to gain a simulation speed while accepting a small amount of simulation error. 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. [7] study both unbounded slack and

bounded slack schemes; Miller et al. [1] 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 [1], a relaxed synchronization scheme has been adopted, trading off causality errors in the simulation for the ability to speed up the simulator even further. The simulation infrastructure presents a number of different synchronization strategies. 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 (lax synchronization) does not synchronize at all except when explicitly performed by the application. The random-pairs synchronization method is in the middle between these two extremes and is implemented by random choosing two simulated target cores to synchronize. 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 evaluated these synchronization schemes in terms of accuracy and simulation speed [4] and have determined that lax synchronization can result in a very high simulation error. In Sniper, we therefore use barrier synchronization with a 100 ns quantum to minimize error.

3 Sniper Components

Sniper [4] is the combination of three main components. In Figure 1, we provide an illustration of how these pieces fit together (Note that the simulation infrastructure itself is not shown). The first is the functional execution model which provides the dynamic instruction stream of the running application. It is this stream of instructions that Sniper uses to determine the timing effects of a particular microarchitecture.

The second component is the timing simulation component. The timing models interact with the dynamic instruction stream to determine the rate of forward progress that is being made. The Sniper Multi-Core Simulator brings together the interval core timing model (as outlined in Section 2.1) with a number of detailed microarchitecture models, such as the Pentium M (Dothan) style branch predictor [8], and instruction and data caches, to model timing in the core and memory subsystems. The data cache models support a large number of configuration options, such as private, and shared last-level caches, multiple DRAM controllers (to support NUMA architectures) and hardware prefetchers. Some timing models are simulated at a higher level of abstraction to speed up simulation. The interconnects between the cache components are modeled as queues, with the total latency consisting of two parts: a minimum access latency and a queuing delay that increases as the link's bandwidth is consumed.

The third component in the Sniper multi-core simulator is the event simulation infrastructure that keeps track of events in the system and keeps them in sync with one another. This component is largely based on the Graphite

simulator [1]. For more details on the parallel simulation aspect of Sniper, see Section 2.2.

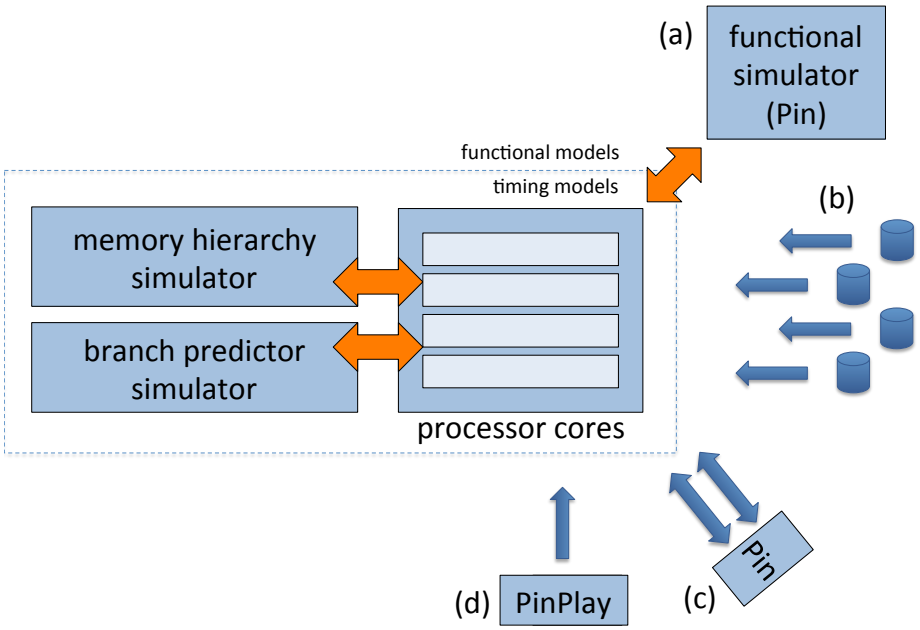

Fig. 1. A high-level diagram of the functional and timing components used in Sniper. There are a number of different options to connect an instruction trace to Sniper such as (a) compiling Sniper as a pintool, (b) using stored SIFT traces in a single- or multi-program configuration, (c) using a number of single- or multi-threaded Pin program with the dynamic bidirectional SIFT protocol, or (d) using PinPlay to replay a number of application pinballs. When using (a), the core and memory timing models are integrated into a single Pin tool. When using (b), (c) or (d), the timing models operate in their own process, while the Pin tool attaches to the application directly. Orange arrows represent communication inside of a process, while blue arrows are bidirectional UNIX pipes that connect the Pin process to the Sniper timing process.

The functional front-end provides a dynamic instruction stream of instructions to the timing models of the simulator. However, Sniper is not a functional-first simulator, but most closely resembles a functional-directed simulator with timing-feedback [9], where the timing models can cause back-pressure on the functional models to control the rate of progress.

Traditionally, Sniper was used as a standalone Pin [10] tool. One drawback of these earlier versions of Sniper was that it was restricted to simulating just a single application at a time, preventing Sniper from running multi-program or multiple multi-threaded workloads. To address this issue, we developed the SIFT protocol. SIFT, the Sniper instruction trace format, is an optionally com-

pressed binary format that contains instruction information along with dynamic execution results of the instruction execution (branch taken information, conditionally executed instruction information, and other details). The SIFT protocol can serve as either a file format for single-threaded traces or as a bi-directional communication channel between the functional trace generator and the timing simulator.

As an alternative to stored SIFT traces, Sniper also supports replaying single-threaded PinPlay [11] application snapshots (pinballs) via the SIFT protocol. Pinballs are deterministic execution snapshots where dynamic execution traces are replayed by executing the application directly instead of storing a dynamic trace of the running application. All communication with the operation system (such as file I/O and system calls) and the results of relevant instructions (such as CPUID and RDTSC) are captured and played back to the application to enforce deterministic application replay. One added benefit when using pinballs as an application file trace is the small file sizes that can be produced. The size of the pinballs generated are related to the size of the application binary and input data instead of the total number of dynamic instructions that are executed in the trace. This can significantly reduce the size of the files stored.

Apart from single-threaded traces, Sniper supports multiple single-threaded applications (from any combination of inputs, either stored SIFT traces, pinballs or single-threaded applications), as well as more complex configurations, such as multi-threaded applications and multiple processes that can consist of multiple threads themselves. In addition, we support MPI applications that use shared-memory to communicate, and we do this through virtual-to-physical mapping of the native application addresses. Additionally, applications that use OpenMP inside MPI ranks are also supported.

To support these configurations, especially for multi-threaded applications, a bi-directional SIFT channel is required as the channel enforces timing back-pressure making sure that the functional execution of each thread across all processes are kept in sync relative to one another. The SIFT channels are also used to communicate events such as thread spawning, important system calls (such as OS/pthread synchronization requests) and other out-of-band requests, such as reading or writing to memory.

4 Software Analysis and Visualization

To aid in hardware/software co-optimization we have developed application- and hardware-level software analysis extensions in Sniper to help the application designers better understand the interactions of their software with modern or future hardware designs.

Through an extension of interval analysis [12], the CPI stack for out-of-order cores [13] was developed to provide a new level of insight into the causes of performance loss when running an application in an out-of-order processor. The CPI stack shows the cycles lost due to different events in the system, such as LLC cache misses or branch misprediction penalties. As described in Section 2.1, each

stall event triggers a corresponding penalty from the interval simulation model, which corresponds directly to each of the CPI stack components. The base CPI component measures the obtainable performance according to the amount of ILP that can be extracted from the application by the reorder buffer.

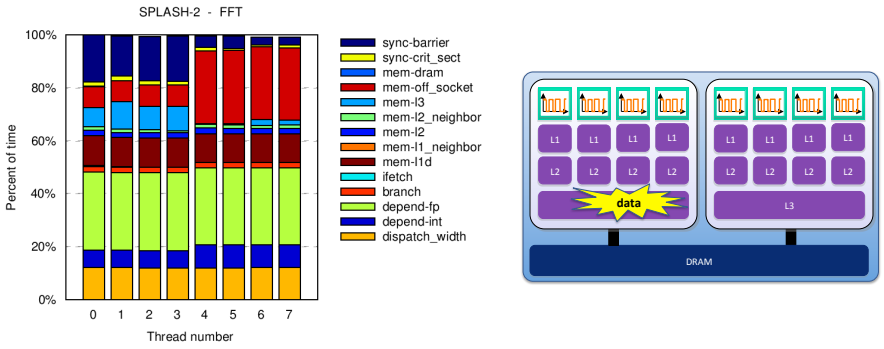

Fig. 2. A CPI stack of the homogeneous multi-threaded application, SPLASH-2, FFT (left) on a homogeneous architecture (right). Because of NUMA effects, where the initial data has been allocated to the L3 of the first processor, threads from another processor need to wait for the data to be transferred from the remote socket.

A straight-forward extension to the single-threaded CPI stack is a multi-threaded version, normalized by percent of run time, where the causes of slow-down for each thread can be seen easily. In the example in Figure 2, we present both a homogeneous application along with a homogeneous microarchitecture. Though the use of CPI stacks, we can determine that the longer run time of the first processor is caused by waiting for a barrier to return. The barrier waiting time is about equal to the extra time that the second processor needs to access the memory hierarchy on the other processor. Changing this initial allocation policy, therefore, could reduce the off-socket latency and improve the performance of the application. CPI stacks allows a software developer to directly determine the causes of performance loss and therefore helps the developer to choose where to optimize the software next.

While applications with a few cores are relatively easy to reason about in this form of multi-threaded CPI stack, it becomes more difficult to understand how hundreds or more CPI stacks come together to contribute to overall run time. A more insightful way to view this data is to view how the CPI stacks change over time, not for a single thread, but for all threads in aggregate. In this way, we can better understand how the phase behavior affects the progress of the application for a given time slice.

With the visualization features recently added to version 4.1 of Sniper, this ability to view how the aggregate CPI stacks for the entire system changes over time was added. Examples of these features are shown in Figure 3 and Figure 4 which shows an example CPI stack from an HPC application, and the

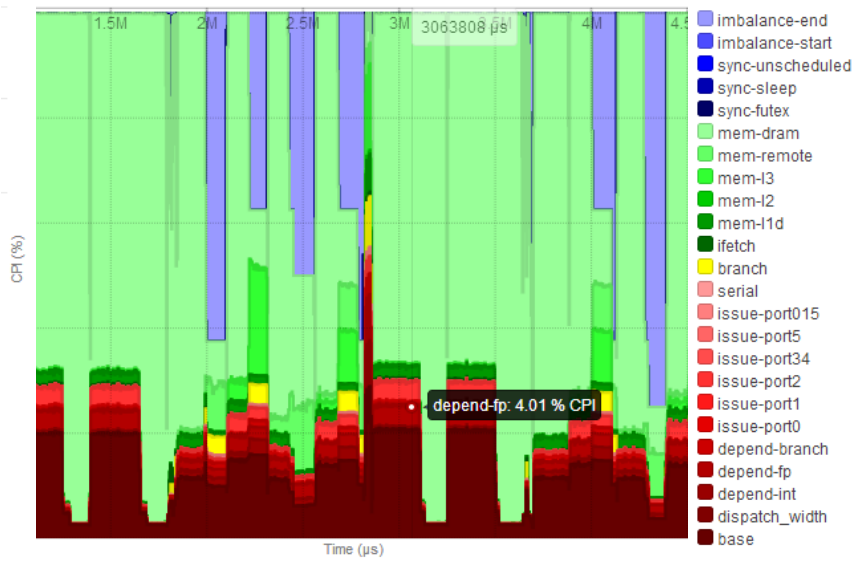

Fig. 3. Sniper visualization output of the CPI stack components of an application in detail. The website view is dynamic (one can zoom into regions of interest) and automatically generated from the Sniper simulation.

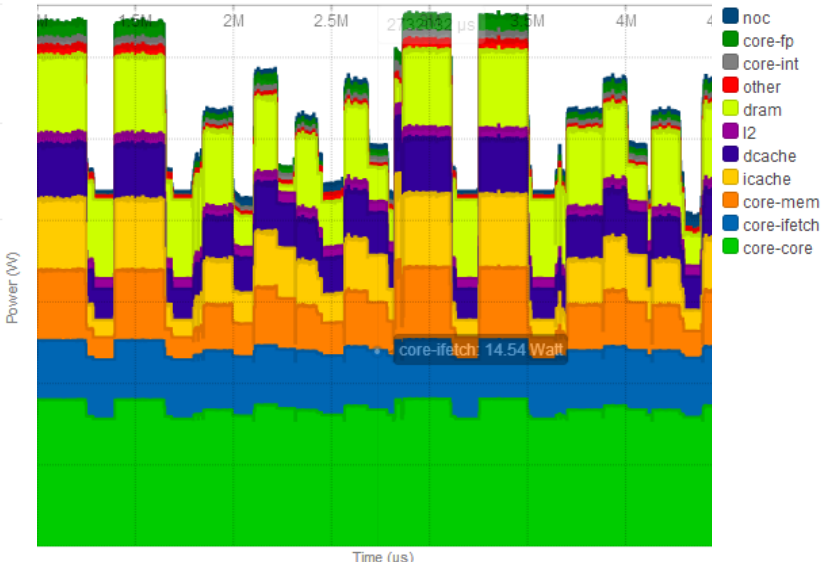

Fig. 4. Sniper visualization output of the power stack components of an application in detail.

corresponding power breakdown over time for each component in the system, respectively. Each simulation can now be augmented with an automatically generated website that allows the user to dynamically change the view of the data at hand: by zooming and providing more or less detail as needed. In addition to CPI stack data, power stacks can also be created using this aggregate view to provide similar insights. Since version 3.2 of Sniper, we have integrated McPAT and demonstrated that accurate power and energy numbers can be obtained with higher-level core models [14].

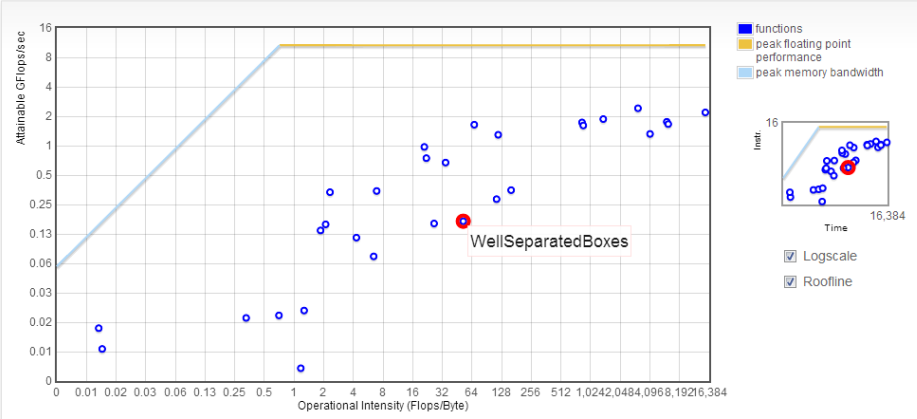

Fig. 5. Sniper visualization output of the Roofline model. Using the theoretical maximum bandwidth and floating-point operations per second, we can compare the performance of each function to determine how close we are to the theoretical maximum performance of that function.

Although CPI stack visualization provides the necessary data to broadly optimize a particular benchmark, knowing where to look inside of a benchmark to find and correct the problem can be a bit more difficult. Most programmers would prefer to understand the breakdown of each CPI component on a per-function basis instead of being broken-down on a per time quantum basis. As a potential solution, we integrated per-function statistics that go beyond what a typical performance analysis tool can provide, even when using performance counters. Examples of data items that can now be exposed are individual CPI stack components on a per-function basis. Additional data that can be more useful in bandwidth-constrained environments include monitoring cache-line use efficiency.

One recently developed application visualization method is called the Roofline model [15]. The Roofline model represents the intersection of two lines, the maximum bandwidth that can be exposed by a given machine, as well as the peak floating-point performance of that machine. By plotting applications on the Roofline model, one can determine how close your application is to the maximum

available bandwidth and computational resources. These two resources tend to be the largest limiting factors for many compute-intensive applications like those found in HPC-style workloads. In Figure 5 we show an example of an automatically generated Roofline model in Sniper. In the Roofline plot, each function of an application under study can be easily evaluated to measure how it performs compared to the maximum performance of a given simulated microarchitecture.

5 Accuracy and Validation

When viewing the results of a software application simulation run or hardware update to improve performance, it is important to understand how well the underlying tools compare to current, state of the art technologies that they are attempting to model. When developing the Sniper simulator, we wanted to focus on accuracy through white-box modeling of the different components of the system. We therefore used the mechanistic interval simulation technique which allows one to configure the system based on micro-architectural parameters. This provides the benefit of being able to update the system configuration for potential future design points. We then combined this model along with the other essential components of a modern processor to form the Sniper Multi-Core Simulator.

Sniper has been validated against real hardware models. In a recent paper [4] we have demonstrated good accuracy compared to real hardware. On the Splash-2 benchmarks [16], we achieve an average absolute error of 25%, with relative scaling numbers performing even better, tracking the hardware results closely. In addition, we demonstrated the accuracy of Sniper combined with McPAT through validation against real hardware; we reported average power prediction errors of 8.3%, for a set of SPEC OMP2001 benchmarks [14].

6 Conclusion

The increasing complexity of software (with multi-threaded applications and ever-increasing data set sizes) coupled with increasingly complex microarchitectures (with 60+ cores in the Xeon Phi and larger LLC sizes), simulation of new software on next generation hardware is becoming increasingly difficult to perform in a timely manner. Coupling traditional performance simulations with energy and power simulations only exacerbates the problems moving forward.

By providing a detailed understanding of both the hardware and software, and allowing for a number of accuracy and simulation performance trade-offs, we show that Sniper can be a useful complement to traditional performance analysis tools for evaluating and optimizing software for high-performance multi-core and many-core systems.

Acknowledgements

We thank Mathijs Rogiers for his invaluable work on the visualization features of Sniper and the anonymous reviewers for their valuable feedback. 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 ExaScience Lab, Leuven, Belgium; the Intel HPC Lab, Swindon, UK; and the VSC Flemish Supercomputer Center.

References

1. Miller, J.E., Kasture, H., Kurian, G., Gruenwald III, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A.: Graphite: A distributed parallel simulator for multicores. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA). (January 2010) 1–12
2. Eyerman, S., Eeckhout, L., Karkhanis, T., Smith, J.E.: A mechanistic performance model for superscalar out-of-order processors. ACM Transactions on Computer Systems (TOCS) **27**(2) (May 2009) 42–53
3. Genbrugge, D., Eyerman, S., Eeckhout, L.: Interval simulation: Raising the level of abstraction in architectural simulation. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA). (February 2010) 307–318
4. Carlson, T.E., Heirman, W., Eeckhout, L.: 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). (November 2011) 52:1–52:12
5. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33**(10) (October 1990) 30–53
6. Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., Wood, D.A.: The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. (May 1993) 48–60
7. Chen, J., Dabbiru, L.K., Wong, D., Annavaram, M., Dubois, M.: Adaptive and speculative slack simulations of CMPs on CMPs. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). (December 2010) 523–534
8. Uzelac, V., Milenkovic, A.: 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). (April 2009) 207–217
9. Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., Ortega, D.: COTSon: infrastructure for full system simulation. ACM SIGOPS Operating Systems Review **43**(1) (January 2009) 52–61
10. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). (June 2005) 190–200
11. Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J.: PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings

of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO). (April 2010) 2–11
12. Eyerman, S., Smith, J., Eeckhout, L.: Characterizing the branch misprediction penalty. In: Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). (April 2006) 48–58
13. Eyerman, S., Eeckhout, L., Karkhanis, T., Smith, J.: A top-down approach to architecting cpi component performance counters. Micro, IEEE **27**(1) (2007) 84–93
14. Heirman, W., Sarkar, S., Carlson, T.E., Hur, I., Eeckhout, L.: Power-aware multi-core simulation for early design stage hardware/software co-optimization. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT). (September 2012) 3–12
15. Williams, S., Waterman, A., Patterson, D.A.: Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM **52**(4) (April 2009) 65–76
16. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proceedings of the 22th International Symposium on Computer Architecture (ISCA). (June 1995) 24–36