Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics

Sam Van den Steen, Stijn Eyerman, Sander De Pestel, Moncef Mechri,
Trevor E. Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout

Abstract—Optimizing processors for (a) specific application(s) can substantially improve energy-efficiency. With the end of Dennard scaling, and the corresponding reduction in energy-efficiency gains from technology scaling, such approaches may become increasingly important. However, designing application-specific processors requires fast design space exploration tools to optimize for the targeted application(s). Analytical models can be a good fit for such design space exploration as they provide fast performance and power estimates and insight into the interaction between an application's characteristics and the micro-architecture of a processor. Unfortunately, prior analytical models for superscalar out-of-order processors require micro-architecture dependent inputs, such as cache miss rates, branch miss rates and memory-level parallelism. This requires profiling the applications for each cache and branch predictor configuration of interest, which is far more time-consuming than evaluating the analytical performance models. In this work we present a *micro-architecture independent* profiler and associated analytical models that allow us to produce performance *and* power estimates across a large superscalar out-of-order processor design space almost instantaneously. We show that using a micro-architecture independent profile leads to a speedup of 300× compared to detailed simulation for our evaluated design space. Over a large design space, the model has a 9.3 percent average error for performance and a 4.3 percent average error for power, compared to detailed cycle-level simulation. The model is able to accurately determine the optimal processor configuration for different applications under power or performance constraints, and provides insight into performance through cycle stacks.

Index Terms—Modeling, micro-architecture, performance, power

1 Introduction

Over the past few decades, gains in energy-efficiency came primarily from improvements in process technology. Each new process generation provided smaller transistors, which resulted in faster switching speeds *and* also lower power, due to proportional voltage reductions. Taken together, this resulted in an almost constant power density, as Dennard predicted [1]. However, threshold voltage scaling limitations and an increasing fraction of leakage power have ended this trend. Future process generations are not expected to deliver significant energy-efficiency by themselves. This change places the burden of improving energy-efficiency into the hands of the architects, who have to figure out how to use the transistors more efficiently.

One way of improving energy-efficiency is to design application-specific processor cores [2]. Application-specific cores are tailored to (a) specific application(s), by removing or reducing all components that are not used, or under-utilized, by the application(s) (e.g., smaller caches or a narrower pipeline), and/or enlarging and adding components that benefit the application (e.g., accelerators). Embedded processors are a typical use case for application-specific processors, because they execute a limited set of applications and can be tightly optimized. However, general-purpose processors can also benefit from application-specific processor design by only turning on functionality when it benefits the current application(s). This is particularly intriguing with the advent of dark silicon [3], which states that only a limited area of the chip can be powered up at a given time. To make effective use of such a chip, we can build a range of application-specific sub-processors, and only activate the one(s) tailored to the current workload, thereby achieving better efficiency, or we can enable power/clock gating of certain structures or parts of structures based on performance and power predictions.

However, designing optimal application-specific processors is challenging due to a very large design space. The architect has to determine the optimal pipeline depth and width, the sizes of the internal buffers (reorder buffer, load-store queue), the sizes and number of levels of the cache hierarchy, memory bandwidth, etc. These parameters affect performance and power consumption, and they all interact with the the characteristics of the running application(s). This means that the detailed design process needs to be repeated for each application. To enable such designs, we

- *S. Van den Steen, S. De Pestel, and L. Eeckhout are with the Department of Electronics and Information Systems, Ghent University, Belgium. E-mail: {sam.vandensteen, sander.depestel, lieven.eeckhout}@ugent.be.*
- *S. Eyerman is with Intel, Belgium. E-mail: Stijn.Eyerman@elis.UGent.be.*
- *M. Mechri, T.E. Carlson, D. Black-Schaffer, and E. Hagersten are with the Department of Information Technology, Uppsala University, Sweden. E-mail: {moncef.mechri, trevor.carlson, david.black-schaffer, erik.hagersten}@it.uu.se.*

Manuscript received 29 July 2015; revised 2 Mar. 2016; accepted 14 Mar. 2016. Date of publication 27 Mar. 2016; date of current version 14 Nov. 2016. Recommended for acceptance by D. Marculescu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2547387

need a tool that can quickly evaluate large design spaces to find interesting regions that can then possibly be explored with detailed simulation.

Analytical performance models are an excellent fit for pruning large design spaces because they use analytical formulas to produce performance estimates based on application characteristics much faster than cycle-level simulation. Estimating the performance of an application executing on a certain processor configuration involves two steps. First, the application is profiled, and then the model is applied on the profiled data. The latter involves applying mathematical formulas, while the former is the most time-consuming step where large sequences of instructions are profiled and analyzed. Reducing the profiling time is important to keep these models fast. However, prior analytical models for superscalar out-of-order (OoO) processors require a profile that depends on both the application and the micro-architecture (branch miss rates, cache miss rates, and memory-level parallelism (MLP)). This means that the application must be profiled for every configuration of the cache hierarchy and the branch predictor of interest, resulting in multiple time-consuming profile steps. In this paper, we describe in detail and extend an analytical performance and power model for superscalar out-of-order processors based on purely micro-architecture *independent* application profiles, enabling the evaluation of a full design space with only one profiling step.

In particular, we make the following contributions:

- We extend the interval model [4], originally developed for the Alpha architecture, to accurately model x86 out-of-order processor architectures. In particular, we add models for functional unit and issue port contention, memory bandwidth contention, and last-level cache (LLC) hit chaining.
- We incorporate previously proposed micro-architecture independent models to estimate cache miss rates and branch miss rates, and evaluate the impact on the accuracy of the model.
- We propose a new model to estimate memory-level parallelism based on a micro-architecture independent application profile.
- We propose a mechanistic power model to quickly estimate the power and energy consumption of a processor.
- We show how evaluating the model on small windows of instructions instead of the full program trace leads to better overall accuracy as well as accurate phase tracking.
- We implement a fast Pin-based profiler that records the required profile, and we evaluate the impact of different sampling approaches.
- We show that the resulting model is both accurate and fast enough to enable the efficient exploration of large design spaces, including finding Pareto-optimal points.

The resulting model estimates performance with a 9.3 percent average absolute error across a large design space and all SPEC CPU2006 benchmarks; power consumption is estimated within 4.3 percent on average. Using extensive sampling, the profiler runs at 1.9 MIPS, which is at least one order of magnitude faster than detailed simulation, and which needs to be done only once for each application. Evaluating the model takes 21 seconds on average per configuration. Over our entire design space, our model is $300\times$ faster than detailed simulation.

2 Prior Work

Analytical models. Early computer architecture research often used analytical models to evaluate their proposals. However, as designs became more complex, and available computing power increased, researchers switched to detailed simulations. Yet in the last decade this trend has begun to reverse as the community has realized that detailed simulation is becoming prohibitively slow with increasing chip parallelism, and that architectural simulation provides limited insight in spite of its high levels of detail. These trends have led to a resurgence in the use of analytical models as a means to provide fast predictions and insight.

Empirical models. One approach uses empirical models that are automatically built from a training set, e.g., through regression [5] or artificial neural networks [6]. These models are based on the premise that current processor micro-architectures are too complex to model, but that through machine learning one can calibrate a generic model to faithfully reproduce their behavior. Empirical models are easy to build and can be fairly accurate, but they need a non-negligible training set, requiring many slow simulations. Furthermore, they can suffer from overfitting or limited generalization if the training set is not diverse enough. Such models rarely provide much insight into the mechanisms of a processor and why a certain application achieves a given performance on a given configuration.

Mechanistic models. Another approach is mechanistic modeling, which builds on simplifying assumptions and first-order effects, observed by studying the flow of instructions through the processor pipeline. Such mechanistic models are generally less accurate than empirical models, and do not model the whole processor in detail. However, while mechanistic models have limited detail, they do reveal how the program interacts with the micro-architecture through mathematical formulas. This underlying simplicity allows them to provide more insight in the performance bottlenecks of a program or a processor, e.g., by building CPI stacks [7] or by analytically providing sensitivities to various parameters. A first approach to model a superscalar out-of-order processor with mechanistic models was made by Karkhanis and Smith [8], and was further refined by Eyerman et al. [4] into the interval processor model. Chen and Aamodt [9] extended this work by adding prefetching and a more accurate memory model.

Mechanistic models consist of two phases: application profiling and performance estimation. Profiling is usually the most time-consuming step, as the instructions of the application need to be analyzed to obtain the application characteristics required by the mathematical model. One drawback of current mechanistic models is that some characteristics are obtained by simulating parts of the processor (micro-architecturally dependent inputs), such as the cache hierarchy

and the branch predictor. Although these simulations are shorter running than fully simulating the processor, they need to be redone for every configuration of the cache hierarchy and the branch predictor. We tackle this problem in this paper by proposing a single micro-architecture independent profiling step that enables estimating performance and power consumption of a large processor design space.

Hybrid empirical/mechanistic models. An intermediate approach between empirical and mechanistic modeling is to use an intuitive mathematical model, where some of the parameters are determined by fitting results of a training set. This approach tries to combine the accuracy of empirical models with the insights provided by mechanistic models. Examples include the model by Hartstein and Puzak [10] for pipeline depth and width, and the mechanistic-empirical model by Eyerman et al. [11] to build CPI stacks on existing hardware.

Interval simulation. The interval simulation technique [12] combines profiling and modeling into a single step, thereby enabling fine-grained performance estimations. Interval simulation models the timing of individual instructions, such as memory operations, enabling the fast and accurate simulation of a multi-core processor with shared memory hierarchy components. The difference between interval simulation and the interval model is that interval simulation needs to be redone if the configuration of the core changes, whereas the interval model can estimate performance for a large range of configurations. On the other hand, the interval model does not provide timings of individual instructions, complicating the modeling of interference in shared memory components, such as a shared cache.

Power modeling. Several models to estimate the power consumption of a processor core have been developed. Some of them are tightly integrated with a performance simulator [13] or require activity factors that are generated using detailed simulation [14]. Other models estimate power consumption in real time by making use of performance counters [15]. To the best of our knowledge, there exist no power models that estimate the power consumption of a large range of processor core configurations using an analytical model. In this paper, we propose such a model by estimating activity factors based on the outputs of the interval model.

ISA-independent profiling. Shao and Brooks [16] propose to characterize workloads independently of the instruction-set architecture (ISA). They profile memory, branch and opcode behavior of an application using the intermediate representation of an application instead of the binary. Following up on this work, they propose using the ISA-independent profiles for speeding up the design of accelerators [17]. They achieve good speedups with minimal loss in accuracy for both performance and power estimations. Moving to an ISA-independent model to predict performance and power usage by leveraging the proposed techniques in these papers, can be an extension to the models we propose.

3 Micro-Architecture Independent Model

3.1 Overview

To enable fast design space exploration for application-specific out-of-order processors, we propose an analytical performance/power model based on a micro-architecture independent profile. This means that the application needs to be profiled only once, after which performance and power estimates for a large design space can be made, including different sizes of core components, caches and branch predictors, in a very fast way. An overview of this method is shown in Fig. 1. The profiling step on the left needs to be done only once, obtaining an *application-dependent* but *micro-architecture independent* profile. The model itself consists of analytical formulas that take the application profile and micro-architectural parameters, and provide a performance and power estimate for a particular processor configuration. The model can be applied multiple times to obtain results for a large processor design space, without re-profiling the application.

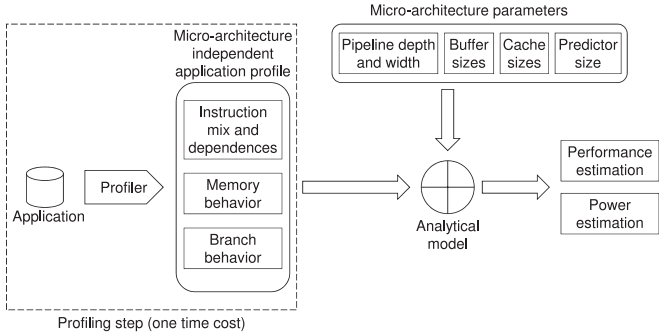

Fig. 1. Overview of the analytical performance/power model using micro-architecture independent application profiling.

The model is based on the interval model [4], a mechanistic performance model for out-of-order superscalar processors. To estimate performance, the original interval model required a few micro-architecture dependent inputs: the number of misses in each level of cache, the number of branch predictor misses, and the amount of memory-level parallelism. These inputs are a complex function of the behavior of the application and the organization of the caches and predictors, and were obtained via simulation. In this paper, we model them in a micro-architecture independent manner, by profiling the application only once and by using cache and branch predictor models to estimate the number of misses, along with extended and newly added models for functional unit contention, MLP, memory bandwidth and chained LLC hits. Furthermore, we add power modeling to evaluate power and energy consumption, to allow the designer to balance performance and power/energy during design space exploration.

3.2 Base Model

The overall model to estimate the number of cycles C uses the number of instructions N, the effective dispatch rate D_{eff}, the number of branch mispredictions m_{bpred}, the branch resolution time c_{res}, the front-end pipeline depth c_{fe}, the number of instruction fetch misses at each level i in the cache hierarchy m_{ILi}, the access latency to each cache level c_{Li}, the size of the ROB (Reorder Buffer) ROB,[1] the number of LLC *load* misses m_{LLC}, memory access time c_{mem}, memory bus transfer and waiting time c_{bus}, the amount of

1. We use ROB (plain letters) to refer to the reorder buffer as a structure, and ROB (italics) as the size of the ROB.

memory-level parallelism MLP, and the LLC hit chain penalty P_{hLLC}:

$$C = \frac{N}{D_{eff}} + m_{bpred}(c_{res} + c_{fe}) + \sum_i m_{ILi} c_{Li+1} + \frac{m_{LLC}(c_{mem} + c_{bus})}{MLP} + P_{hLLC}. \quad (1)$$

This model differs slightly from the model by Eyerman et al. [4] to handle the differences between x86 (the current target ISA of the model) and the Alpha instruction set (the original target ISA). The first important difference is that x86 is a CISC architecture, while Alpha is a RISC architecture. However, the internal processor pipeline in a modern x86 processor also uses a RISC-like instruction set, which is implemented by transforming x86 instructions into micro-operations. Therefore, we first decompose the x86 instructions to obtain the sequence of micro-ops. As a result, the N in Equation (1) is equal to the number of micro-ops, and not the number of instructions. On average, there are 1.2 micro-ops per x86 instruction for the x86 decoder implemented in our profiler.

It is important to note that the model focuses on the dispatch stage of the processor, by calculating the number of cycles it takes to dispatch all instructions. Because every (correct-path) instruction has to go through all pipeline stages, the average dispatch rate equals the average execution and commit rate. In a real processor, the dispatch and commit stage operate independently from each other. However, our model assumes that if commit blocks (e.g., due to a last-level cache miss), dispatch will block a number of cycles later as the ROB fills up [18].

In the first term, the number of micro-ops N is divided by the effective dispatch rate in the absence of miss events (D_{eff}). This equals the minimum number of cycles it takes to execute a program and is called the *base component* or base performance. In the Alpha model, the effective dispatch rate is set to the designed dispatch width D of the processor, assuming the reorder buffer (ROB) is large enough to sustain an IPC of D (balanced design). However, we find that since the x86 architecture offers fewer architectural registers, the dependence paths through register and memory dependences tend to be longer. This causes the effective dispatch rate to be smaller than the dispatch width. Therefore, we define the effective dispatch rate D_{eff}, which is calculated using Little's law, as follows:

$$D_{eff} = \min\left(D, \frac{ROB}{lat \cdot K(ROB)}\right). \quad (2)$$

$K(ROB)$ is the average critical path length, and lat is the average instruction execution latency, including short (L1 and L2) load data cache misses.

3.3 Functional Unit Contention Modeling

Current processors typically have multiple functional units for executing different instructions in parallel, of which several may be connected to a single port (e.g., in the Intel Nehalem processor, there are only six ports for 15 functional units[2]). If multiple instructions are to go to the same port, they need to be issued sequentially. Furthermore, if a non-pipelined functional unit is occupied, no new instructions of that type can be issued, even if the port is available. This has an important impact on performance, which we include in the model for improved accuracy, similar to what Carlson et al. describe [19].

Assuming there are N_p instructions that have to be executed by functional units connected to port p, it will take N_p cycles to forward them to the respective functional units, despite having multiple free functional units. For functional unit contention, we make a distinction between pipelined and non-pipelined functional units. For pipelined units, if there are N_i instructions of type i, and U_i functional units of that type, then it takes at least N_i/U_i cycles to execute. For non-pipelined units with latency lat_j, the minimal execution time equals $\frac{N_j \cdot lat_j}{U_j}$. In other words, the effective dispatch rate may thus be limited by the number of functional units or by the number of ports, hence we rewrite Equation (2) as follows:

$$D_{eff} = \min\left(D, \frac{ROB}{lat \cdot K(ROB)}, \frac{N}{N_p}, \frac{N \cdot U_i}{N_i}, \frac{N \cdot U_j}{N_j \cdot lat_j}\right), \quad (3)$$

in which p ranges over all ports, i ranges over all types of pipelined functional units, and j over all non-pipelined functional units.

3.4 Branch Miss Rate Modeling

The second term in Equation (1) is the branch misprediction component. It is computed by multiplying the number of branch misses with the sum of the branch resolution time c_{res} and the front-end pipeline depth c_{fe} (fetch, decode and rename stages). In the Alpha model, a mispredicted branch is most often the last correct-path instruction to execute [20], so the branch resolution time is approximated by the critical path length. For x86 however, we find that a mispredicted branch is most often not on the critical path, because the critical path mainly consists of chains of memory operations. Therefore, it is more accurate to model the branch resolution time as the average dependence path length (instead of the critical path), multiplied by the average instruction latency.

To obtain the number of branch mispredictions without performing branch predictor simulation, we measure branch entropy of an application during profiling, following [21]. Branch entropy quantifies how predictable branches are: an entropy of 0 means that branches are perfectly predictable, while an entropy of 1 indicates random branch behavior. We apply the technique described by De Pestel et al. [21] to create models that estimate branch miss rates from branch entropy. In practice, we profile the outcome of all conditional branches, record history tables, and calculate local, global and tournament branch entropy for different history lengths. For estimating the branch miss rate for a specific branch predictor, we use a linear model

2. Source: http://www.hardwaresecrets.com/inside-intel-nehalem-microarchitecture.

that relates branch entropy to miss rate. To find this linear relationship, we use a set of benchmarks for which we both simulate all types of branch predictors and measure entropy. The linear model for each type of predictor is then constructed using a least-squares fit on the results of the training set. Once the model is constructed, we no longer need to perform branch predictor simulation, but instead we can instantly obtain the branch miss rate for every new application after profiling its branch entropy.

The estimated branch miss rate is used directly in Equation (1) as m_{bpred}, but is also used to calculate the branch resolution time c_{res}. We assume that the average number of instructions between two branch misses equals N/m_{bpred}, and we use the leaky-bucket method [4] to obtain the number of instructions in the ROB when a branch miss occurs. We then compute the average dependence path length on that many instructions in the ROB to obtain an estimate for the branch resolution time.

3.5 Cache Miss Rate Modeling

The next two terms in Equation (1) quantify the impact of instruction and data cache misses. Similar to the Alpha model, the penalty of instruction cache misses is calculated as the number of misses at each level i multiplied by the access time to the next cache level $i+1$. The penalty for long-latency load misses (i.e., LLC load misses) equals the number of load misses in the LLC times the memory access time, divided by the amount of memory-level parallelism, which is the average number of overlapping misses if at least one is outstanding (see Section 3.6). c_{bus} is the number of cycles spent on the memory bus, including waiting time for the memory bus if occupied (see Section 3.7).

In order to estimate cache miss rates using a micro-architecture independent profile, we use the StatStack statistical cache model [22], which provides the miss ratio for LRU caches of arbitrary size.

StatStack uses reuse distances (the number of memory references between two accesses to the same cache line) to estimate cache behavior. Note that reuse distances count total memory references between accesses and not unique references, which makes them far cheaper to collect than stack distances. The reuse distances for an application are used to build a histogram of an application's reuse behavior, which is then transformed into a stack distance distribution. The stack distance distribution gives the number of unique cache lines accessed between two accesses to the same cache line, and, hence the miss ratio for an LRU cache by counting the number of accesses for a stack distance greater than the cache size.

The profile for estimating cache miss rates thus consists of the reuse distance distribution, which is independent of the cache configuration. However, measuring reuse distances for all memory operations would introduce a large overhead and low profiling speed. Therefore, StatStack reduces profiling overhead by collecting only a sample of the reuse distances in an application. Berg and Hagersten [23] show that it is possible to profile an application to get its reuse distance distribution with very low overhead using hardware performance counters. This approach has been further optimized by Sembrant et al. [24]. In our current implementation we use Pin to collect this profile.

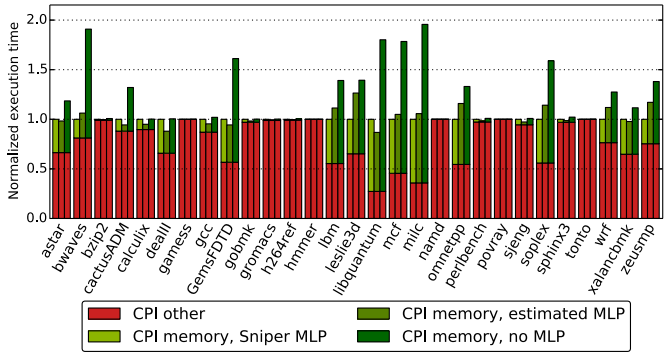

Fig. 2. Normalized execution time with breakdown in memory cycles and other cycles for Sniper simulations (leftmost bar), when using our MLP model (middle bar) and when no MLP is modeled (rightmost bar).

For the interval model, we have extended StatStack to differentiate between load versus store misses. The interval model does not model the performance impact store misses may have, i.e., it is assumed that the processor does not often stall on a store miss. However, store misses contribute to memory bandwidth contention and the power consumption of the cache and core, which we account for in the model. To estimate the miss ratios at each level in the cache hierarchy, we model each cache level independently using StatStack, which implicitly assumes the cache hierarchy to be inclusive. Our evaluation shows that this provides good accuracy across a standard three-level cache hierarchy.

Instruction cache behavior is similarly modeled by the reuse distance distribution computed over the instruction address stream.

3.6 MLP Modeling

Memory-level parallelism is defined as the average number of main memory accesses (LLC misses) that can be processed in parallel, if at least one is outstanding [25]. Main memory typically consists of multiple DRAM banks, that can each process one access at a time. The interval model assumes that the penalty of multiple parallel accesses equals the penalty of a single access, explaining the division of the last term in Equation (1) by the amount of MLP.

MLP has a non-negligible impact on performance, as illustrated in Fig. 2. The leftmost bar shows normalized CPI stacks for detailed simulation using Sniper, with two components: execution time due to DRAM accesses (i.e., the memory component), and all other components, aggregated in 'CPI other'. The next two bars are normalized to the Sniper's simulated execution time, and represent different ways of modeling MLP. The rightmost bar assumes there is no MLP ($MLP = 1$), i.e., all memory accesses are serialized. The middle bar shows the CPI stack with MLP computed using the model as described below. The takeaway is that MLP has a significant impact on overall performance, hence modeling its impact is important. Not modeling MLP (i.e., assuming there is no MLP) leads to an average error of 24.6 percent (96 percent max error). Our MLP model reduces the average error to 6 percent (26 percent max error).

Following interval analysis, the number of parallel memory accesses equals the number of independent LLC misses that occur within the ROB. The amount of MLP is thus dependent on micro-architectural features (the size of the

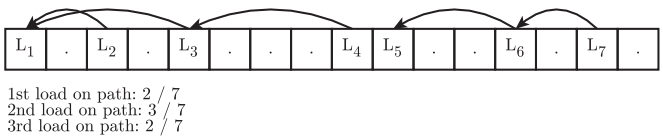

Fig. 3. Example of a load dependence distribution.

ROB, size of the caches, number of MSHRs), as well as application characteristics (which instructions cause misses and how they depend on each other). However, in our micro-architecture independent profile, we only have limited information about these characteristics: we have the LLC miss rate from StatStack, but not the 'location' of the individual misses in the instruction stream, making it hard to estimate MLP; moreover, although we profile dependences between instructions, we do not know the dependences between LLC misses. Modeling MLP accurately is not straightforward and turns out to be one of the largest contributors to the total error of the model, as we will describe later.

LLC misses frequently occur in bursts: when a load misses in the LLC, there is a large probability that loads nearby in the instruction stream will also miss in the LLC. As a result, assuming that LLC misses are uniformly distributed across the application leads to inaccurate MLP estimates. We find that LLC miss bursts are mainly caused by cold misses, i.e., the first time a cache block is accessed. Capacity and conflict misses, i.e., the cache block was in the cache but has been evicted, are more uniformly distributed. The intuition is that throughout its execution, an application will load new data structures on which it will compute. This typically leads to bursts of cold misses. Conflict misses on the other hand, are caused by too many unique accesses to the same set in the cache. This occurs more spread across the application's execution, so there is less burstiness due to conflict misses.

Cold misses can be located using a micro-architecture independent profile by keeping track of the first access to a certain address. Because we have to check for every address if it has been accessed before, keeping track of all addresses leads to a large structure and high lookup times. To reduce this overhead, we assume a limited set of allowed cache block sizes (e.g., 32, 64 and 128 bytes), and we record only cold misses for these cache block sizes. The final profile consists of the distribution of the number of cold misses in an ROB, for different ROB and cache line sizes.

We leverage the following assumptions to estimate MLP:

- m_{LLC}^{cf}, m_{LLC}^{cold} and $m_{LLC}^{cold}(ROB)$ represent the number of capacity/conflict misses, the number of cold misses, and the average number of cold misses per ROB containing at least one cold miss, respectively.
- The load distribution $f(\ell)$ characterizes the dependences between loads. In this distribution, ℓ is the number of loads on the dependence path leading to a load in the ROB including that last load ($\ell = 1$ means that the load is independent of other loads), and $f(\ell)$ is the frequency of loads with ℓ loads on their dependence path. Fig. 3 shows an example for a 16-entry ROB. The oldest instruction is located on the left and the arrows indicate dependences between loads. The ROB contains seven loads; two of those loads appear at the head of a load dependence chain (L_1 and L_5 have $\ell = 1$); there are three loads that appear as the second load on a load dependence chain (L_2, L_3 and L_6 have $\ell = 2$); and there are two loads that appear as the third load on a load dependence chain (L_4 and L_7 have $\ell = 3$). Hence, the corresponding load distribution $f(\ell)$ equals $[2/7; 3/7; 2/7]$.
- M_{LLC} and M_{LLC}^{cf}, which denote the overall LLC miss rate and the capacity/conflict LLC miss rate, respectively. In the model, we use the miss rate as an approximation for the probability for a load to cause a cache miss.
- $\bar{L}(ROB)$ is the average number of loads per ROB, i.e., the fraction of loads in the instruction mix times the ROB size.

Our MLP model is split up into two parts: MLP due to cold misses and MLP due to capacity/conflict misses. The cold-miss MLP is the average number of independent cold misses in the ROB. A load miss that is the ℓth load in a dependence path will be an independent miss if all $\ell - 1$ previous loads on its path are not misses, which has a probability of $(1 - M_{LLC})^{\ell-1}$. From the $m_{LLC}^{cold}(ROB)$ cold misses in the ROB, $m_{LLC}^{cold}(ROB) \cdot f(\ell)$ are the ℓth load on a dependence path, so the number of independent cold misses in the ROB, i.e., the cold-miss MLP, can be estimated as:

$$MLP^{cold} = \sum_{\forall \ell} (1 - M_{LLC})^{\ell-1} \cdot m_{LLC}^{cold}(ROB) \cdot f(\ell). \quad (4)$$

Conflict misses lead to MLP in a similar way. However, we do not know how many loads in the ROB will cause a conflict miss. Therefore, we assume that conflict misses are uniformly distributed, and we estimate the number of conflict misses per ROB as follows: $M_{LLC}^{cf} \cdot \bar{L}(ROB)$. Following the same reasoning as for the cold-miss MLP, we estimate the conflict-miss MLP as follows:

$$MLP^{cf} = \sum_{\forall \ell} (1 - M_{LLC})^{\ell-1} \cdot M_{LLC}^{cf} \cdot \bar{L}(ROB) \cdot f(\ell). \quad (5)$$

Averaging Formulas 4 and 5 based on the relative number of cold and conflict misses gives us an estimation for the overall MLP:

$$MLP = \frac{m_{LLC}^{cf}}{m_{LLC}} \cdot MLP^{cf} + \frac{m_{LLC}^{cold}}{m_{LLC}} \cdot MLP^{cold}. \quad (6)$$

In a processor, there is also an upper limit on the MLP because of the finite number of Miss Status Handling Registers. MSHRs are used to enable parallel pending misses and to check whether a newly dispatched load address is part of one of the cache lines that is currently a pending miss. The number of MSHRs limits how many requests to lower cache levels or DRAM can be processed in parallel. We model this by calculating the number of loads in one ROB that miss at each level of cache, and if this is larger than the number of MSHRs at that level, we scale the MLP down with the appropriate fraction, i.e., the number of MSHRs divided by the number of misses per ROB.

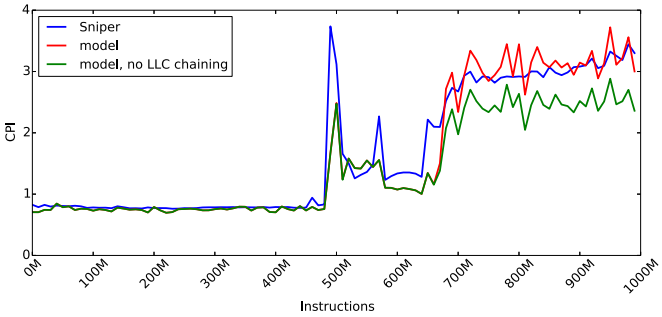

Fig. 4. CPI variation over time for gcc with and without the LLC hit chaining component compared to Sniper.

TABLE 1
Reference Architecture, Based on the Intel Nehalem Architecture

Structure	*Size*
Core frequency	2.66 GHz
Dispatch width	4
ROB	128 entries
Reservation stations	43 entries
L1I and L1D	32 KB, latency = 1 and 4 cycles, respectively
L2	256 KB, latency = 8 cycles
LLC	8 MB, latency = 30 cycles
Branch predictor	pentium-M

3.7 Memory Bandwidth Contention Modeling

We find that for some applications, due to their bursty memory behavior, the available memory bandwidth is often not sufficient, resulting in memory controller congestion and queuing delays. To model this, we assume that the number of concurrent misses equals the MLP on average. Therefore, the first miss has a bus latency equal to the bus transfer time, i.e., the size of a cache block divided by the width of the memory bus. The second concurrent miss has to wait until the first miss releases the bus, so its bus latency equals twice the bus transfer time. And the third miss has a bus latency of three times the bus transfer time, etc. The average bus latency for MLP' concurrent accesses (we define MLP' next) therefore equals:

$$c_{bus}(MLP') = \frac{1}{MLP'} \sum_{i=1}^{MLP'} i \cdot c_{transfer} = \frac{MLP'+1}{2} c_{transfer}. \tag{7}$$

We use linear interpolation to deal with non-integer MLP numbers.

The MLP factor only takes into account loads that miss in the LLC, because store misses usually do not incur a penalty for the core performance (except when they prevent other loads to issue because of a structural constraint, e.g., a full write buffer or an exhaustion of MSHRs). However, they do need to access memory, so they have an impact on memory bandwidth contention. In fact, we find that for benchmarks that have a lot of LLC store misses, memory bandwidth contention is underestimated. We compensate for this by rescaling the MLP to include the store misses (where m_{LLC}^{load} and m_{LLC}^{store} are the number of LLC load and store misses, respectively; note that all previous LLC miss counts only include load misses):

$$MLP' = MLP \cdot \frac{m_{LLC}^{load} + m_{LLC}^{store}}{m_{LLC}^{load}}. \tag{8}$$

This MLP' is used in Equation (7) to calculate the average bus transfer time.

3.8 Chained LLC Hits

The last term in Equation (1) is the penalty of LLC hits, i.e., loads that miss in the L1 and L2 caches, but hit in the LLC.

One of the most important features of a superscalar Out-of-Order processor is its ability to hide instructions with short latency, e.g., floating-point operations or loads that hit in one of the higher (L1 and L2) cache levels. Interval analysis assumes that the latency of an operation can be hidden if that latency is smaller than the time to fill up the ROB, i.e., the ROB size divided by the dispatch width. In our configuration, the only latency that is larger than the ROB fill time is the main memory access time due to a LLC miss. However, the hit latency of the LLC is for most configurations close to this threshold (e.g., 30 cycles LLC hit latency, and an ROB of 128 and dispatch width of 4, which results in 32 cycles fill time). We find that when two or more LLC hits depend on each other, we do notice some penalty. We call this the *chained LLC hit penalty*.

An example of this problem is shown in Fig. 4, which is a visualization of one billion instructions of the gcc benchmark executed on our reference architecture (see Table 1) as simulated by Sniper and calculated by our model with and without modeling chained LLC hits. The first 400 million instructions are executed at a CPI of around 0.8, followed by a few peaks due to many DRAM accesses. The interesting region however starts around 650 M instructions with the average CPI rising to around 3. The reason for this is in part an increase in the number of branch misses, but also, and more importantly, a substantial increase in the number of LLC hits, which leads to a high probability of multiple dependent LLC hits. The LLC-chaining component contributes around 20 percent to the total CPI (see the delta between the 'model' and 'model, no LLC chaining' curves in Fig. 4). Not including the LLC hit chaining term, the estimation error on the total execution time for gcc equals −12.3 percent, while with this component, the error is reduced to −3.6 percent. Note that the underestimation in performance mainly originates from the overestimation of the MLP at around 500 M instructions, rather than from the error on the LLC chain penalty.

Our goal is to estimate the penalty of chained LLC hits without involving additional profiling. To estimate the penalty due to chained LLC hits, we first calculate the average number of LLC load hits in one ROB, $h_{LLC}(ROB)$, as the LLC hit rate (as estimated by StatStack) times the average number of loads in the ROB. Contrary to MLP calculation, where we want to calculate the number of independent LLC misses, we now need to compute the number of LLC hits that are on the same dependence path. All loads on a dependence path will be executed sequentially because of the dependences between them, so all LLC hits on a path will be serialized. To find this number, we reuse the load dependence distribution that is profiled for MLP calculation. All loads that are first on a path (i.e., loads that are

independent of all other loads) initiate a new possible path with dependent LLC hits. So the number of independent loads equals the number of dependence paths with loads on it, denoted $p_{load}(ROB)$. Assuming that LLC hits are uniformly distributed across the dependence paths, the average number of LLC hits on a path (LLC hit chain or LHC) can be estimated as follows:

$$LHC_{avg} = \frac{h_{LLC}(ROB)}{p_{load}(ROB)}. \quad (9)$$

However, the LLC hit chain penalty is not determined by the average chain of LLC hits, but by the longest chain. The longest chain is at least as long as the average chain, and is bounded by the number of LLC hits in the ROB, $h_{LLC}(ROB)$, as well as by the largest number of loads on a dependence path. We cannot deduce the latter from the load dependence distribution, so we approximate it by the average number of loads on a path, $\overline{lop}(ROB)$. The maximum number of LLC hits on a path thus equals:

$$LHC_{max} = \min(h_{LLC}(ROB), \overline{lop}(ROB)). \quad (10)$$

To calculate the expected value of the largest number of LLC hits on a dependence path, we assume that we have at least LHC_{avg}, and that the remaining LLC hits that can possibly belong to this path, i.e., $LHC_{max} - LHC_{avg}$, are distributed uniformly across all $p_{load}(ROB)$ paths. The expected longest chain of LLC hits therefore equals:

$$LHC_{exp} = LHC_{avg} + \frac{LHC_{max} - LHC_{avg}}{p_{load}(ROB)}. \quad (11)$$

The resulting penalty then equals the chain length times the LLC hit latency c_{LLC}:

$$P'_{hLLC}(ROB) = c_{LLC} LHC_{exp}. \quad (12)$$

As explained before, latencies that are smaller than the ROB fill time are hidden by the out-of-order execution. Hence, we have to subtract the cycles it takes to fill the ROB from the penalty calculated in Equation (12), which yields the average penalty for a window of ROB instructions:

$$P_{hLLC}(ROB) = \max\left(0, P'_{hLLC}(ROB) - \frac{ROB}{D_{eff}}\right). \quad (13)$$

The total penalty for the full application thus equals this penalty times the number of windows of ROB instructions across the entire instruction stream:

$$P_{hLLC} = P_{hLLC}(ROB) \cdot \frac{N}{ROB}. \quad (14)$$

3.9 Power Modeling

Designing application-specific processors to improve energy-efficiency requires a power model. To estimate power, we use the McPAT tool [14], which requires the configuration of the processor along with activity factors (i.e., the number of accesses) for each component. The result is an estimation of power and energy consumption.

For our model, we deduce the activity factors from the analytical performance model, instead of measuring them in simulation. Many of the inputs are already measured by the profiler for the performance model, such as the number of (micro)instructions and the instruction type mix. Other inputs are generated by the performance model, such as the number of misses at each level of cache and in the branch predictor. However, some inputs are not required for the performance model, because it is assumed that they have no impact on performance, but are needed for the power model, because they increase certain activity factors. The most important examples are store misses at all cache levels (we only use LLC store misses in the performance model) and writebacks. Store misses for all cache levels are generated by StatStack, as explained before. The number of writebacks is more difficult to estimate, but we notice that they have a very small impact on total power consumption, so we choose not to model them.

4 Framework Overview

Our framework for estimating performance and power of an application as described in the previous paragraphs consists of two parts. First, we have the profiler, which gathers all statistics of the application, and the second part is the analyzer, which calculates the model formulas.

4.1 Profiler

Our profiler is developed as a Pin-tool [26]. Pin dynamically instruments x86 binaries with custom functions to gather statistics about a program. The profiler collects the following characteristics:

- Micro-operation count and instruction type mix, to calculate the average instruction latency and estimate functional unit contention.
- Average and critical dependency path for multiple ROB sizes, to calculate the effective dispatch rate as well as the branch resolution time, and to model dependences for MLP estimation.
- The distribution of the number of cold misses in an ROB, for different ROB sizes and different cache line sizes.
- The distribution of the number of loads in an ROB, for different ROB sizes; this is used to model MLP and LLC hit chaining.
- The distribution of the number of loads on a dependence path, for multiple ROB sizes, used in the MLP estimation and the LLC hit chaining component.
- Branch entropy for different local and global history sizes.
- Reuse distance profile for data loads and stores, and for instruction addresses, to estimate the number of cache misses at all cache levels.

Our profiler is fully configurable. The configuration parameters can be divided roughly into three categories. The first category includes the parameters for controlling how the profiler takes samples in a binary, which is described in more detail in Section 4.3. A discussion on how this influences the estimation error made by the analyzer is covered in Section 5.5. The second category includes the parameters for controlling the sampling done by StatStack. A detailed discussion on these parameters can be found

in [22]. The last category contains the parameters specifying the range of sizes that certain micro-architecture structures can take. Although our profiler is micro-architecture independent, profiling for a very large design space means gathering statistical profiles that contain data for a lot of different structures and structure sizes (e.g., the average critical dependence path for ROB sizes from 1 to 1,024). It takes more time to generate large profiles and they take up more disk space. The profiler can be accelerated by limiting the range of sizes of micro-architecture structures (and thus the size of the design space), such as the size of the branch predictor or the ROB.

We define our own binary data format for writing the statistical profiles. This reduces the amount of data that is written and also speeds up the profiler by about 10 percent compared to saving the profiles in ASCII files.

As described before, we can optimize the speed of our profiler by limiting the design space for which we can calculate the performance and power. The design space we use in this paper contains 243 different processor architectures and is described in Table 3. If we limit our profiler to only profile for this design space, we achieve a profiling speed of 1.9 MIPS. In comparison, our detailed simulator achieves a simulation speed of 0.13 MIPS or less, and simulation needs to be redone for every configuration, while profiling only needs to be done once. The size of one profile on a pinball of one billion instructions varies from 300 to 500 MB depending on the sampling parameters, the design space profiled and the application.

4.2 Analyzer

The analyzer implements all of the functionality and algorithms as described in Section 3 to estimate performance and power usage of an application. The current version of the analyzer is written in Python, because of the large availability of statistical libraries. However, this comes at the cost of a relatively slow analyzer, because Python is an interpreted language. The average time it takes per benchmark and per processor configuration is approximately 21 seconds.

As will be described in Section 5.1, we use representative regions of one billion instructions per benchmark. Simulating one configuration using Sniper takes around 2.2 hours per benchmark. This means that simulating our design space (243 configurations, 29 benchmarks) takes around 1.75 compute years. Our profiler incurs a one-time cost of 21 minutes on average per benchmark, or 10.1 compute hours to profile all applications. Adding 21 seconds per configuration and benchmark to calculate the model, our framework only needs 2.1 compute days—300× faster than simulation. These evaluation speed numbers are lower than what we presented in [27], because we now evaluate the models in windows (see the next section) and we generate more statistical profiles (e.g., the load dependence distribution). The evaluation can be accelerated by taking large windows and a sparser sampling rate, at the cost of a larger error, see Section 5.5.

4.3 Sampling and Windowing

The most complex part of the profiler is the calculation of the critical dependence path. To reduce the overhead of profiling, we only perform dependence and instruction mix analysis on small samples of instructions, so-called micro-traces. A *micro-trace* is a short (e.g., 1,000) instruction sequence of the full trace, and we profile multiple of these micro-traces spread over the full trace. Between the micro-traces, we fast-forward the profiler until the start of the next micro-trace.

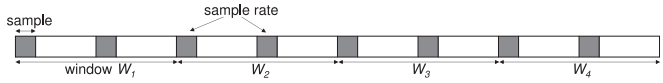

Fig. 5. Dividing the program into windows and samples. The model is applied for every window separately, based on the combined profile of the two samples in the window. Total execution time is then estimated as the sum of the estimated execution times across the four windows.

When applying the model using the profile of the different micro-traces, there are two potential approaches: either we apply the model on every individual micro-trace and add the estimated number of cycles for each micro-trace, or we first combine the profiles of the micro-traces to a single profile, and apply the model on the combined average profile. An intermediate solution would be to group every few micro-traces in a combined profile, apply the model on each group, and we then add the cycle estimates across all groups. We denote a *window* as the part of the application on which we apply the model, a *sample* as the micro-trace that is profiled, and the *sample rate* as the number of samples taken in a window (see Fig. 5). For example, a window size of 10M instructions, a sample size of 1 K instructions and a sample rate of 10 means that we divide program execution into windows of 10 M instructions, and in each window, we take 10 samples of 1 K instructions. We use these 10 samples to estimate the execution time (and power consumption) for each window, and add the estimates for each window in the program to obtain the overall performance and power consumption. We use the same division into windows for StatStack and the branch predictor model to obtain consistent results. The impact of changing window size, sample size and sample rate is evaluated in Section 5.5.

5 Evaluation

In this section, we evaluate the accuracy of the analytical model for a baseline configuration. We show that the micro-architecture independent model does not add much extra error (compared to a model that takes micro-architecture dependent characteristics as input). We also show that our model accurately tracks phase behavior by applying the model across multiple windows, and we investigate the impact on the accuracy of changing the sampling parameters.

5.1 Experimental Setup

We evaluate the model using all 29 SPEC CPU 2006 benchmarks. We validate the model against detailed simulation, for which we select a single representative 1-billion-instruction SimPoint [28] for each benchmark, using the PinPlay technology [29]. Both the model and detailed simulation are applied to these simulation points. For the reference detailed simulations we use the most detailed cycle-level core model in Sniper 6.0, which has been validated against real hardware by Carlson et al. [19]. Power measurements are done using the McPAT script [14] included with Sniper for a 45 nm technology. Table 1 shows the configuration of our baseline architecture, configured after Intel's Nehalem architecture.

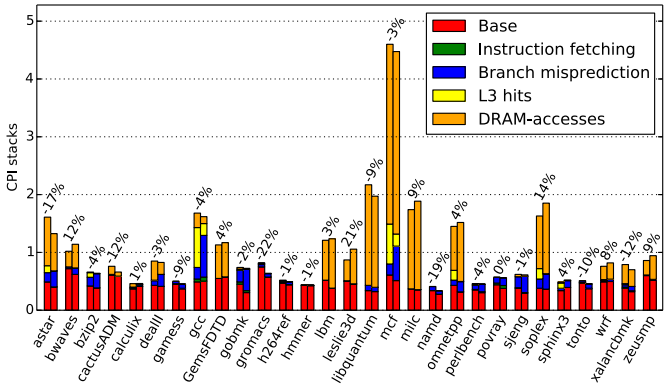

Fig. 6. CPI stacks generated by Sniper (left bar) and by the model (right bar), and the error of the model versus Sniper simulation (top).

5.2 Accuracy and CPI Stacks

Fig. 6 shows the CPI of the benchmarks for the baseline configuration obtained through simulation using Sniper (left bar) and through our model (right bar). The average absolute error across all benchmarks equals 7.6 percent. There are positive and negative errors, which shows that our model is not biased. A maximum error of 22.3 percent is observed for `gromacs`, which is due to severe functional unit contention at very small timescales. We do not model this very accurately because we use samples of at least 1,000 instructions, over which this fine-grained behavior is averaged out.

Fig. 6 breaks up the overall CPI into CPI stack components. CPI stacks visualize the performance bottlenecks of an application, by showing how much performance is impacted by the execution of instructions, cache misses, and branch misses. For the left bar, we use the built-in CPI stack generator of Sniper. The CPI stacks for the model are constructed as follows. The model (Equation (1)) consists of an addition of multiple components that reflect different penalties. We can represent each component separately in a stack, such that the top of the stack equals total cycle count. By dividing the components by the number of instructions, we get the respective CPI stack components.

The CPI stacks generated by Sniper and by our model match well, which suggests that the overall model accuracy is not much embellished by compensating under and overestimations. Note that some of the differences stem from the fact that there is no unambiguous way of defining CPI stacks in an out-of-order processor, because events can occur concurrently. Hence, whether cycles are accounted to one or another event may lead to small differences in how CPI stacks are constructed. The most noticeable example is an L3 cache hit that is part of the dependence path leading to a mispredicted branch: it is accounted to the L3 component in Sniper, because at the time the L3 cache hit occurs, it is impossible for Sniper to detect that it is part of a path to a mispredicted branch. Our model, on the other hand, accounts the miss latency to the branch miss penalty, because we model L3 hits as long-latency instructions that usually do not incur stalls. This is why the branch component for the model CPI stack tends to be larger than for the Sniper CPI stack, and vice versa for the L3 hit component, with `gcc` being the most notable example.

TABLE 2
Average and Maximum Error of Introducing a New Micro-Architecture Independent Component

Component	Arch-dep	I-cache	Branch	LLC-chain	D-cache	MLP + queue
Avg error	6.7%	7.0%	7.0%	7.0%	7.3%	7.6%
Max error	20.7%	20.8%	21.7%	21.7%	22.0%	22.3%

5.3 Error Introduced through Micro-Architecture Independent Modeling

Our main contribution is to make the profile independent of the micro-architecture, such that we require only one profiling step to model a large range of micro-architectures. We do this by proposing models that estimate the number of cache and branch misses, MLP, memory bandwidth, chained LLC hits, and functional unit contention, based on a micro-architecture independent profile. Because each of these models introduces additional inaccuracies, we expect the error of a micro-architecture independent model to be higher than that of a micro-architecture dependent model. In this section, we show how much error each of the components of the micro-architecture independent model introduces.

We start with a model similar to the original interval model [4], where we use cache miss rates, branch miss rates, the amount of MLP, and memory bus queuing time from detailed simulation. The profile only contains the instruction mix and instruction dependency information (critical dependence path, average dependence path, load dependence distribution). We already incorporate the improved functional unit contention modeling and the LLC hit chain modeling. This model (denoted Arch-dep in Table 2) has an error of 6.7 percent on average across all benchmarks for the baseline configuration. Next, we gradually add each of the architecture-independent components, see Table 2. The I-cache column shows the error when using the instruction cache miss rate from StatStack instead of from simulation, which increases the average error with 0.3 percent. Adding the micro-architecture independent branch predictor model does not noticeably increase the average error, which is also the case for using the LLC hit rate from StatStack to model LLC hit chaining. Estimating the D-cache misses (mainly the LLC misses) using StatStack introduces an error increase of 0.3 percent, and a similar increase is incurred by modeling the MLP and memory queuing in an architecture-independent way. Overall, micro-architecture independent modeling only increases the error by 0.9 percent on average, while the maximum prediction error increases by 1.6 percent only.

5.4 Phase Accuracy

Fig. 7 shows the CPI of two benchmarks per 10 million instructions, for both Sniper and the model. Phase graphs for all benchmarks are provided in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2016.2547387. We use average error over the whole execution, and the *phase accuracy coefficient (PAC)* to quantify how well we predict execution time and track phase behavior. We define PAC as the average absolute error of the relative difference between two consecutive intervals of one million instructions:

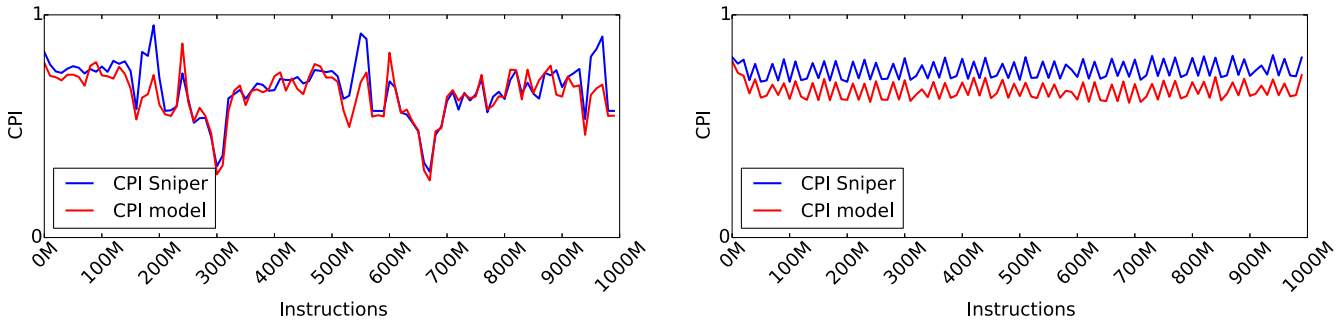

Fig. 7. CPI variation during the execution time as estimated by the model and Sniper with the error on the execution time prediction. The phase accuracy coefficient quantifies the error for predicting phase behavior.

$$PAC = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{C_S(i-1)-C_S(i)}{\overline{C_S}} - \frac{C_M(i-1)-C_M(i)}{\overline{C_M}}\right|. \quad (15)$$

$C_S(i)$ is the number of cycles predicted by Sniper for interval i, while $C_M(i)$ is the number of cycles predicted by the model. $\overline{C_S}$ and $\overline{C_M}$ are the average number of cycles of one interval over the full trace, for Sniper and the model, respectively. These numbers are included in Fig. 7, along with the error for estimating the overall execution time. The model tracks an application's time-varying execution behavior very well for most of the benchmarks, see for example `bzip2`. For some benchmarks, e.g., `cactusADM`, we observe that the PAC is lower than the overall execution time prediction error; in spite of the relative modeling error offset, the model tracks the application's phase behavior fairly well.

5.5 Sampling Parameters

As discussed in Section 4.3, we profile only small parts of the application to obtain the model inputs, in order to speed up the profiling phase. We also apply our model to multiple windows of instructions, which enables tracking phase behavior. To find the optimal sampling and windowing settings, we perform the following exhaustive experiment. We evaluate window sizes of 1 M, 10 M, 100 M and 1B instructions, resulting in 1,000, 100, 10 and 1 window(s), respectively, for the 1B instruction traces. For each of these window sizes, we profile samples (micro-traces) of 1, 5 and 10 K instructions, and multiple sample rates, such that we profile at least 100 K instructions of the 1B instructions trace.

Fig. 8 shows the average error of the performance model for all of the experiments, with on the horizontal axis the total number of profiled instructions (e.g., a window size of 1M instructions, sample size of 1K instructions and sample rate of 1, results in 1M instructions profiled out of 1B). Intuitively, the more instructions are profiled, the slower the profiling step, although the slowdown is not proportional to the number of instructions, because of the fast-forwarding overhead and the overhead for storing the profiles. In terms of the size of the profile, the smaller the window size, the larger the profile, because we need to keep a profile for every individual window.

The lowest error (7.6 percent, the red point at the bottom) is obtained for a 1M instruction window and 1 K micro-trace at a sample rate of one micro-trace per window. (This is the setting used throughout the paper.) If we want to sacrifice some accuracy for faster profiling (8.5 percent error, leftmost blue point), a window size of 10 M instructions with a 1 K micro-trace and sample rate of one micro-trace per window may be a good alternative.

Clearly, decreasing window size improves accuracy. Being able to track short-term phase behavior is important to obtain better accuracy. This can be attributed to the fact that some of our modeling techniques, such as the contention modeling and the modeling of chained LLC hits, rely on characteristics of specific sequences of instructions, which get averaged when the window size or the sample size is too large. For example, the previous version of this model [27] applied the model to a single window of 1B instructions, consisting of 1,000 samples of 1,000 instructions (the second highest purple point at 10^6 instructions in Fig. 8). As a result, we could not accurately model functional unit contention and LLC hit chaining, leading to an error that is 3 percentage points higher.

Note that taking longer or more samples in a window does not necessarily decrease the error. For example, taking 10 samples of 1 K instructions or one sample of 10 K instructions in a window of 1M instructions causes a 0.5 percent *increase* in error versus taking 1 sample of 1 K instructions. The explanation for this counter-intuitive behavior is that applications may suffer from high functional unit contention during small phases of just a few hundred instructions, and low contention for the rest. Since we take many samples, this extreme behavior is already present in some of the samples. However, if we take larger or more samples in one window, the behavior of these few hundred instructions is averaged out with the other instructions. Because of this averaging effect, we lose the ability to model fine-grained

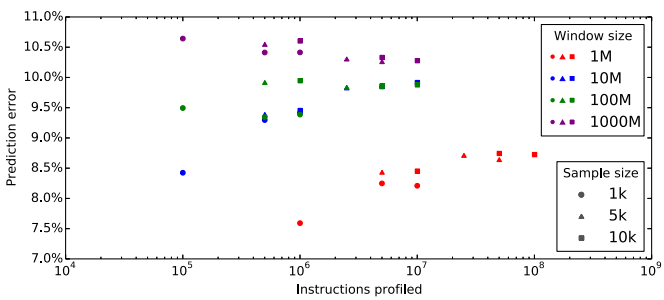

Fig. 8. Average absolute prediction error versus number of instructions profiled. Different colors represent different window sizes; the symbols reflect the sample sizes (see legend).

TABLE 3
Core Configuration Design Space

Parameter	Low-end	Middle	High-end
Dispatch width	2	4	6
ROB entries	32 - 48 - 64	96 - 128 - 160	128 - 192 - 256
Branch predictor	Pentium-M - gshare - global predictor		
L1I cache	32 KB, 4-way, latency 1 cycle		
L1D cache	32 KB, 8-way, latency 4 cycles		
L2 cache	128 KB - 256 KB - 512 KB 8-way, latency 8 cycles		
L3 cache	1 - 2 - 4 MB	4 - 6 - 8 MB	8 - 12 -16 MB 16-way, latency 30 cycles
Memory BW	8 GB/s		
Memory latency	120 cycles		

contention, leading to slightly worse performance predictions. Predicting performance for all samples individually, i.e., having smaller windows, would improve accuracy, but it would also largely slow down the model evaluation time and increase the size of the profile.

6 Model Applications

In the previous section we evaluated the model for a single processor configuration. Because the motivation of this work is fast design space exploration, we now consider a large design space by varying some of the parameters of the processor configuration. We compare the predictions by our model against simulation of the full design space, and we show how well the model performs in practical design space exploration studies.

6.1 Design Space Exploration

Our design space spans 243 different processor configurations, by resizing several structures in the processor core, see Table 3. We evaluate the model for three different branch predictors: Pentium-M [30], a 16 KB gshare and a 16 KB global predictor. We also vary the dispatch width, the size of the L2 and L3 caches, and the size of the ROB. We divide our processor architectures into three different categories, low-end, middle and high-end processors, based on the dispatch width. When we vary the size of the ROB, we proportionally scale the number of reservation station entries.

Fig. 9 shows the cumulative error distribution of the model across all benchmarks and all configurations in the design space (more than 7,000 points), for performance and power estimation, respectively. As a reference, we show the same error distribution for the model presented in our previous ISPASS paper [27]. Clearly, the improved modeling and per-window estimation significantly reduce the overall modeling error. Across the whole design space, the average error on the performance estimation is 9.3 percent, compared to 13.3 percent in the ISPASS paper, a reduction by almost one third. We achieve similar results for the power estimation where the average error is reduced from 7.1 to 4.3 percent. The error on the power consumption is attributed exclusively to the error on the estimation of the dynamic power consumption. In McPAT, static power consumption depends only on the micro-architecture and a fixed temperature setting, hence the static power estimation in our model is exactly the same as in Sniper.

6.2 Pareto-Optimal Designs

A common technique in design space exploration is to find the Pareto-optimal set of configurations, i.e., the set of configurations that have either higher performance or lower power consumption than any other configuration. Or in other words, there exists no other configuration that beats the Pareto-optimal configurations in both performance and power. This section quantifies how well the model is able to construct a Pareto-optimal set of configurations.

Fig. 10 shows the Pareto frontier obtained using Sniper simulations (blue), the Pareto frontier obtained by the model (green), and the actual (simulated) performance and power consumption of the points in the model Pareto frontier (red), for a select number of benchmarks. Pareto frontiers for all benchmarks are given in Appendix B, available in the online supplemental material. The difference between the blue and green curves again shows the error of the model, while the difference between the red points and the blue curve indicates how well we can predict actual Pareto-optimal configurations.

For some benchmarks (e.g., `calculix`), the green and blue curves are close, indicating high accuracy for the model. For other benchmarks (e.g., `gromacs`), we make a systematic error across all micro-architectures. However, this still leads to good relative accuracy when changing the processor configuration: the green curve is a shifted version of the blue curve. Due to the relative accuracy, the designs

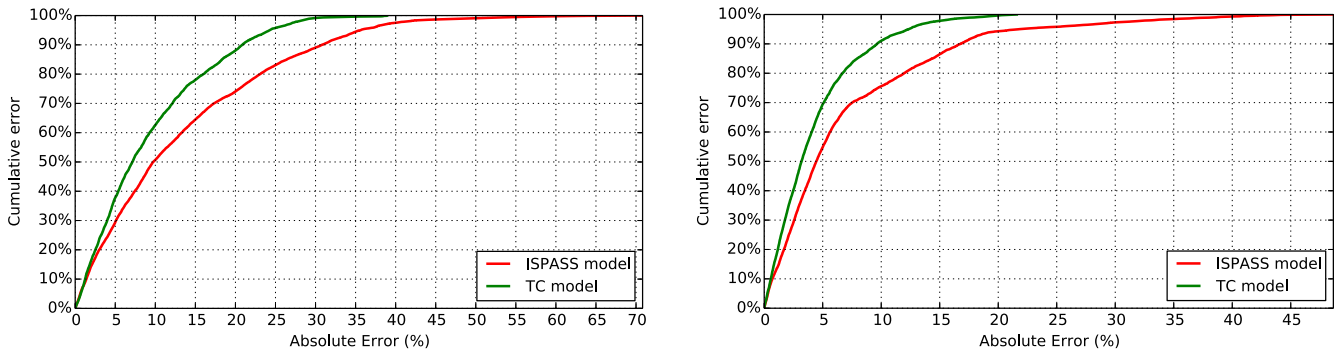

Fig. 9. Cumulative error distribution for (a) performance and (b) power, comparing the current model against the ISPASS model [27].

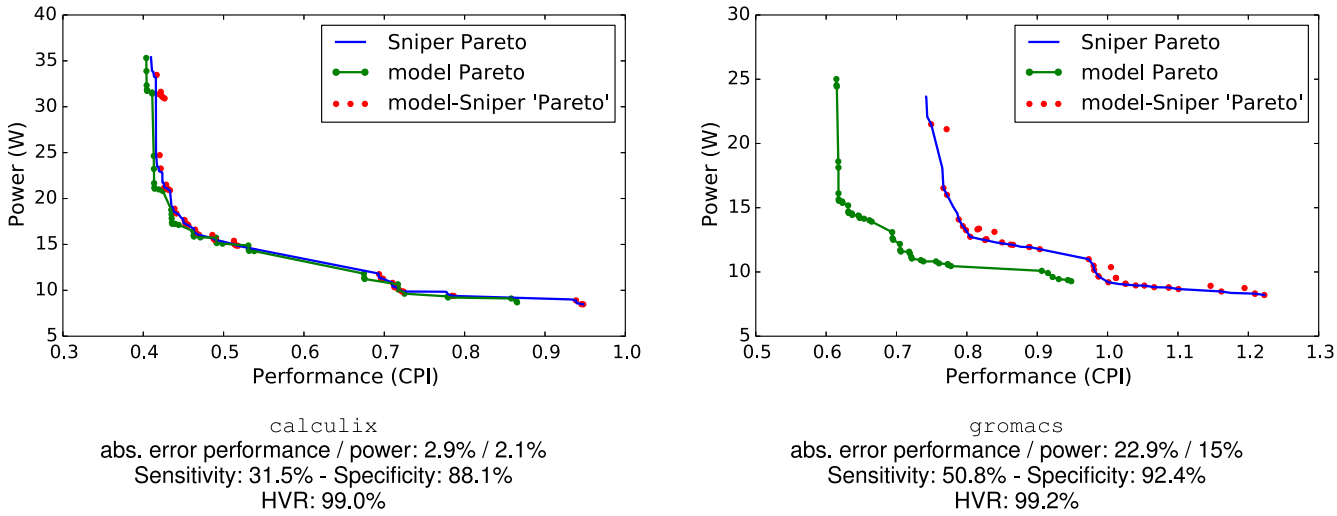

Fig. 10. Pareto frontiers for different benchmarks as calculated by the model (green curve) and simulated by Sniper (blue curve). The red points are the same configurations as the ones on the green curve, but with the simulated performance and power consumption. The error number below the graph is the average error across the entire design space.

on the model Pareto frontier are almost exactly the same as the one on the Sniper Pareto frontier (almost all red points are part of the blue curve).

Next to the visual matching of the Pareto frontiers, we also calculate the specificity, sensitivity and hypervolume ratio (HVR) metrics [31]. The first two metrics quantify the fraction of predicted actual Pareto-optimal and non-Pareto-optimal designs, respectively. The third metric shows how well we can predict the range of solutions across the entire frontier. This is important since we are interested in finding the extreme low-power and high-performance Pareto-optimal designs at either end of the Pareto frontier. The first two metrics may reveal good performance, even if we find many designs in a small range, hence the need for a metric that quantifies the range of the frontier.

The average values over the whole design space are 46, 87 and 97 percent for specificity, sensitivity and HVR, respectively. Hence, our model is very good at predicting the actual range of the Pareto frontier (HVR) and also at filtering out the non-Pareto optimal solutions (specificity), but performs less good on detecting *all* Pareto-optimal designs (sensitivity). However, a visual inspection of the Pareto frontiers (see also Appendix B, available in the online supplemental material) shows that either we find only a few designs in a large cluster of Pareto-optimal designs that are very close to each other, which leads to lower sensitivity, but is also acceptable, or we miss some Pareto-optimal designs that are not useful to implement, e.g., configurations that show a large increase in power consumption while improving performance only marginally.

6.3 Comparison to Empirical Modeling

A lot of prior work proposes predicting performance and power using empirical models [5], [6]. These models are constructed from a training set of simulated configurations. Although this training set is much smaller than the full design space, simulating the training set incurs a large overhead compared to our model, which requires only one fast profiling step per application. To quantify this overhead, we construct a polynomial regression model with polynomials up to degree 3 to predict performance and power consumption for our design space. We also evaluated higher degree polynomials, but these led to over-fitting the model. Achieving higher accuracy than our mechanistic model when training one model for the whole design space requires a training set of almost 2,500 simulations, which lines up with the results of Lee and Brooks [5]. Since our profiling step is on average 6× faster than simulation, and taking into account the time needed to calculate the model, this means that building a regression model is more than 100× slower than our model.

Furthermore, empirical models tend to compensate over- and under-estimations for the different parameters, resulting in less accurate predictions of the impact of changing one parameter. Therefore, the Pareto frontiers predicted using an empirical model are much less accurate than for our model, even though the error is similar or lower for the empirical model (not included here because of space constraints). The sensitivity of the Pareto fronts generated by the empirical model is 26 percent on average, which is almost 2× lower than the average sensitivity of the Pareto-fronts found with our model (46 percent). Hence, our mechanistic model is better at predicting the impact of each parameter, and is therefore better suited for design space exploration.

7 Conclusions and Future Work

Designing application-specific processors requires a fast design space exploration tool, as the design process needs to be redone for every application. In this work, we propose an analytical performance and power model that, based on micro-architecture independent application profiles, enables the evaluation of large superscalar processor design spaces using a single application-specific but architecture-independent profile. Performance and power are estimated with an average error of 9.3 and 4.3 percent, respectively, compared to detailed simulation. We show that the model can be used to explore large processor design spaces, such as finding Pareto-optimal configurations.

Our model is limited to estimating performance and power for a single-threaded program on a single core, which can be a starting point for determining (an) optimal core configuration

(s) for designing a multicore processor. One avenue for future work could be to extend the models towards multicore processors running multi-threaded applications. This involves new modeling contributions for shared caches, interconnection networks, and multi-threading (SMT) cores, as well as synchronization and cache coherence.

Acknowledgments

This research was performed while S. Eyerman was at the Department of Electronics and Information Systems, Ghent University, Belgium. This paper is an extension of "Micro-Architecture Independent Analytical Processor Performance and Power Modeling" by the same authors, presented at the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), March 2015 in Philadelphia, PA. The paper is completely revised, and includes the following contributions over the ISPASS paper: (i) Improved modeling of functional unit contention, MLP and chained L3 hits, which leads to lower average prediction errors; (ii) Evaluation of the model over small time windows per benchmark instead of over the full benchmark, leading to improved accuracy and accurate phase behavior estimation; (iii) Analysis of sampling during the profiling phase and its impact on accuracy; (iv) Extended case studies.

References

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Nov. 1974.

[2] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 205–218.

[3] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 365–376.

[4] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith," A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 42–53, 2009.

[5] B. Lee and D. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2006, pp. 185–194.

[6] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2006, pp. 195–206.

[7] P. G. Emma,"Understanding some simple processor-performance limits," *IBM J. Res. Develop.*, vol. 41, no. 3, pp. 215–232, 1997.

[8] T. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 338–349.

[9] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," in *Proc. Int. Symp. Microarchit.*, 2008, pp. 59–70.

[10] A. Hartstein and T. R. Puzak, "The optimal pipeline depth for a microprocessor," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, 2002, pp. 7–13.

[11] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2011, pp. 216–226.

[12] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2010, pp. 307–318.

[13] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 83–94.

[14] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. Int. Symp. Microarchit.*, 2009, pp. 469–480.

[15] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proc. 36th Annu. Int. Symp. Microarchit.*, 2003, pp. 93–104.

[16] Y. S. Shao and D. Brooks, "ISA-independent workload characterization and its implications for specialized architectures," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2013, pp. 245–255.

[17] Y. S. Shao, B. Reagen, G. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 97–108.

[18] T. Karkhanis and J. E. Smith, "A day in the life of a data cache miss," in *Proc. 2nd Annu. Workshop Memory Perform. Issues*, vol. 99, 2002.

[19] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, 2014.

[20] S. Eyerman, J. E. Smith, and L. Eeckhout, "Characterizing the branch misprediction penalty," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2006, pp. 48–58.

[21] S. De Pestel, S. Eyerman, and L. Eeckhout, "Micro-architecture independent branch prediction modeling," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 135–144.

[22] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of lru caches," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 55–65.

[23] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Proc. Int. Conf. Meas. Model. Comput. Syst.*, 2005, pp. 169–180.

[24] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase guided profiling for fast cache modeling," in *Proc. 10th Int. Symp. Code Generation Optim.*, 2012, pp. 175–185.

[25] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 76–87.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200.

[27] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, "Micro-architecure independent analytical processor performance and power modeling," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 32–41.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2002, pp. 45–57.

[29] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proc. 8th Annu. Int. Symp. Code Generation Optimization*, 2010, pp. 2–11.

[30] V. Uzelac and A. Milenkovic, "Experiment flows and micrbenchmarks for reverse engineering of branch predictor structures," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 207–217.

[31] K. Deb, *Multi-Objective Optimization Using Evol. Algorithms*. Hoboken, NJ, USA: Wiley, 2001, vol. 16.

Sam Van den Steen received the MSc degree in computer science engineering from Ghent University in 2013. He is currently working toward the PhD degree at the same university. His research interests include performance and power analysis, modeling and evaluation of (multicore) processors. He is funded by by the Agency for Innovation by Science and Technology (IWT) through a doctoral fellowship.

Stijn Eyerman received the MSc and PhD degrees at Ghent University in 2004 and 2008, respectively. He is currently working as a research scientist for Intel. He has published more than 40 papers at conferences and journals, two of which have been awarded with an IEEE Micro Top Picks selection. His interests include processor performance modeling and scheduling on (heterogeneous) multicore processors.

Sander De Pestel received the MSc degree in computer science engineering from Ghent University in 2013, where he is currently working toward the PhD degree. His research interests include performance analysis and modeling of (multicore) processors as well as compiler optimizations. He is funded by the Agency for Innovation by Science and Technology (IWT) through a doctoral fellowship.

Moncef Mechri received the MSc degree in computer science from Uppsala University in 2013, where he is also currently working toward the PhD degree. His research interests include fast measurement and performance modeling, software optimization, and shared resource contention. He is funded by the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.

Trevor E. Carlson received the BSc and MSc degrees in electrical and computer engineering from Carnegie Mellon University, in 2002 and 2003, respectively and the PhD degree in computer science and engineering from Ghent University in 2014. He is a post-doctoral researcher at Uppsala University. His research interests include highly efficient microarchitectures, performance modeling, and fast and scalable simulation methodologies. He is funded by the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.

David Black-Schaffer received the PhD degree in electrical engineering from Stanford University in 2008. He then worked on the design and implement of the OpenCL standard for heterogeneous computing at Apple, Inc., before joining Uppsala University, where he is currently an associate professor. His research interests include fast measurement, modeling, and simulation of shared resource contention, heterogeneous scheduling, and power-efficient memory systems. His research is funded by the Wallenberg Academy Fellowship program, the Swedish Science Council, the Swedish Foundation for Strategic Research's Future Research Leader program, the EU, and the Uppsala Programming for Multicore Architectures Research Center.

Erik Hagersten received the PhD degree in computer science from the Royal Institute of Technology (KTH) in Stockholm 1992. He was a chaired professor at Uppsala University 1999. Before joining Uppsala, he was the chief architect for high-end server engineering at Sun Microsystems. His research interests include low-power memory hierarchies and efficient modeling techniques. His research is funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC) and the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.

Lieven Eeckhout received the PhD degree in computer science and engineering from Ghent University in 2002. He is a professor at Ghent University, Belgium. His research interests are in the area of computer architecture, with a specific interest in performance analysis, evaluation, and modeling. He is the current editor-in-chief of *IEEE Micro*. His research is funded by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, as well as by the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.