# Mechanistic Analytical Modeling of Superscalar In-Order Processor Performance

MAXIMILIEN B. BREUGHE, STIJN EYERMAN, and LIEVEN EECKHOUT,
Ghent University, Belgium

Superscalar in-order processors form an interesting alternative to out-of-order processors because of their energy efficiency and lower design complexity. However, despite the reduced design complexity, it is nontrivial to get performance estimates or insight in the application–microarchitecture interaction without running slow, detailed cycle-level simulations, because performance highly depends on the order of instructions within the application's dynamic instruction stream, as in-order processors stall on interinstruction dependences and functional unit contention. To limit the number of detailed cycle-level simulations needed during design space exploration, we propose a mechanistic analytical performance model that is built from understanding the internal mechanisms of the processor.

The mechanistic performance model for superscalar in-order processors is shown to be accurate with an average performance prediction error of 3.2% compared to detailed cycle-accurate simulation using gem5. We also validate the model against hardware, using the ARM Cortex-A8 processor and show that it is accurate within 10% on average. We further demonstrate the usefulness of the model through three case studies: (1) design space exploration, identifying the optimum number of functional units for achieving a given performance target; (2) program–machine interactions, providing insight into microarchitecture bottlenecks; and (3) compiler–architecture interactions, visualizing the impact of compiler optimizations on performance.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Superscalar in-order processors, processor design space exploration, functional units, inter-instruction dependences, performance modeling, cycle stacks



This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259295, as well as EU FP7 Adept project number 610490.
Authors' addresses: M. B. Breughe, S. Eyerman, and L. Eeckhout, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: mbreughe@gmail.com, Lieven.Eeckhout@UGent.be.
This article is an extension of "A Mechanistic Performance Model for In-Order Processors," by Maximilien Breughe, Stijn Eyerman, and Lieven Eeckhout, presented at the 2012 International Symposium on Performance Analysis of Systems and Software (ISPASS). The new contributions are:

—We added modeling of an arbitrary number of functional units of any type in contrast to a fixed number in the ISPASS paper (i.e., 4 ALUs and 1 unit for all other types).
—We completely revised the modeling of interinstruction dependences and unified it with the functional unit contention modeling.
—We added modeling of memory-level parallelism, which has a nonnegligible impact on performance for some benchmarks that were not evaluated in the ISPASS paper.
—We validated the model against hardware.
—We added a case study on sizing the number of functional units.
—We reevaluated all other case studies using the new model and revealed new insights about the interaction between dependences and functional unit contention.


DOI: http://dx.doi.org.10.1145/2678277

**ACM Reference Format:**
Maximilien B. Breughe, Stijn Eyerman, and Lieven Eeckhout. 2014. Mechanistic analytical modeling of superscalar in-order processor performance. ACM Trans. Architec. Code Optim. 11, 4, Article 50 (December 2014), 26 pages.
DOI: http://dx.doi.org.10.1145/2678277

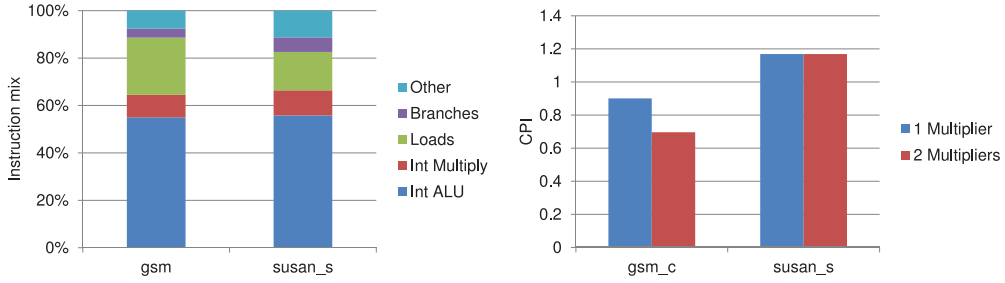
## 1. INTRODUCTION

For studying processor performance, both researchers and designers heavily rely on detailed cycle-accurate simulation. Although detailed simulation provides accurate performance projections for a particular design configuration, deriving fundamental insight into the interactions that take place within a processor is more complicated. Understanding trend behavior of microarchitecture structure scaling and the interactions among microarchitecture structures, as well as how the microarchitecture interacts with its workloads, requires a very large number of simulations. The slow speed of detailed cycle-accurate simulation makes it a poor fit to understand these fundamental microarchitecture–application interactions.

In this article, we focus on mechanistic analytical performance modeling, which is a better method for gaining insight and for reducing the large number of detailed simulations in large design spaces. Mechanistic modeling is derived from the actual mechanisms in the processor. A mechanistic model has the advantage of directly displaying the performance effects of individual mechanisms, expressed in terms of program characteristics such as interinstruction dependence profiles and fine-grained instruction mix; machine parameters such as processor width, number of functional units, and pipeline depth; and program–machine interaction characteristics such as cache miss rates and branch misprediction rates. Mechanistic modeling is in contrast to the more common empirical models that use machine learning techniques and/or statistical methods (e.g., neural networks, regression) to infer a performance model [Dubach et al. 2007; Ipek et al. 2006; Joseph et al. 2006a, 2006b; Lee and Brooks 2006; Ould-Ahmed-Vall et al. 2007; Mariani et al. 2013]. Empirical modeling involves running a large number of detailed cycle-accurate simulations to infer or fit a performance model. In contrast, mechanistic modeling builds a model from the internal structure of the processor and does not require simulation to infer or fit the model.

Detailed simulations or well-trained empirical models can show the performance impacts of different processor designs, but it is time-consuming and challenging at times to reveal the underlying reason why a design improves performance for one application but not another. Figure 1 illustrates this. Figure 1(a) shows that gsm_c and susan_s have a similar fraction of multiply instructions, yet Figure 1(b) shows that they behave differently when we increase the number of multipliers on the baseline microarchitecture from one to two (see Section 6 for a detailed description of the experimental setup). Mechanistic performance modeling provides a level of insight that enables quick and deep understanding of such performance phenomena by breaking up total execution time into different components that account for instruction latencies, dependences, cache misses, branch mispredictions, and so forth. (We refer back to this case study in Section 9.1.)

Whereas prior work in mechanistic performance modeling has focused on superscalar out-of-order processors [Eyerman et al. 2009; Karkhanis and Smith 2004], in this article we propose a mechanistic model for superscalar in-order processors. Compared to out-of-order processors, the performance of superscalar in-order processors is quite sensitive to the order of instructions in the dynamic instruction stream, interinstruction dependences, instruction execution latencies, and the number of functional units available. Therefore, it is impossible to mimic the behavior of an in-order processor with the existing models by constraining out-of-order resources (e.g., limiting the ROB size to the pipeline width).




(a) *Instruction mix*

(b) *Performance results for adding an additional multiply unit*

Fig. 1. Based on the instruction mix for gsm_c and susan_s (left graph), one would expect performance to improve by adding an additional multiply unit. However, simulation results (right graph) show a significant performance increase for gsm_c but not for susan_s.

We believe that this work is timely given that energy and power efficiency are primary design concerns in contemporary computer system design. Whereas the focus is on extending battery lifetime in embedded systems, improving energy and power efficiency also has important implications on cooling and total cost of ownership of server and data center infrastructures. In-order processors are less complex, consume less power, and incur less chip area compared to out-of-order processors, which makes them an attractive design point for specific application domains. In particular, in-order processors are commonly used in the mobile space, ranging from cell phones to tablets and netbooks; example processors are the Intel Atom (except for the recent Silvermont architecture) and ARM Cortex-A7/A8. For server throughput computing, integrating many in-order processor cores on a single chip maximizes total chip throughput within a given power budget. Commercial examples include Sun Niagara [Kongetira et al. 2005] and AMD/SeaMicro's Intel Atom based server[1]; recent research projects have also studied in-order processors for Internet-sector workloads [Andersen et al. 2009; Lim et al. 2008; Reddi et al. 2010].

The overall structure of the article is as follows. We start with a general overview of the framework and a description of the assumed superscalar processor in Section 2. Next, we construct the mechanistic model in Sections 3 through 5. Our experimental setup is explained in Section 6. The evaluation in Section 7 shows that our model reaches an average absolute prediction error of 3.2% compared to detailed cycle-level simulation with gem5. Further, Section 8 shows that our model can be used to predict the performance of the ARM Cortex-A8 with an average absolute prediction error of 10%. In Section 9, we demonstrate the usefulness of the model through three case studies. We first leverage the model to understand program–machine interactions and reveal insight into the example just described in Figure 1. Second, we use the model to minimize the number of functional units while achieving a performance target of 98% compared to using a total of 16 functional units in a four-wide superscalar in-order processor. Third, we evaluate how compiler optimizations affect in-order performance and derive some interesting conclusions. We end by discussing related work in Section 10, by providing ideas for future work in Section 11, and by concluding in Section 12.

## 2. MODELING CONTEXT

Before describing the proposed model in great detail, we first set the context within which we build the model. We present a general overview of the modeling framework, as well as a description of the assumed superscalar in-order processor architecture.

[1]http://www.seamicro.com/.

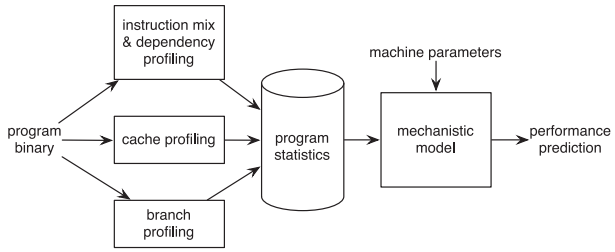



Fig. 2. Overview of the mechanistic modeling framework.

### 2.1. General Overview

The framework of the mechanistic model is illustrated in Figure 2. It requires a profiling run to capture a number of statistics that are specific to the program only and are independent of the machine. These statistics relate to the program's instruction mix and interinstruction dependences, and need to be collected only once for each program binary.

The profiling run also needs to collect a number of mixed program-machine statistics—that is, statistics that are a function of both the program binary as well as the machine configuration. Example statistics are cache and TLB miss rates, as well as branch misprediction rates. Although, in theory, collecting these statistics requires separate runs for each cache, TLB and branch predictor configuration of interest; in practice, however, most of these statistics can be collected in a single run. In particular, single-pass cache simulation [Hill and Smith 1989; Mattson et al. 1970] allows for computing cache miss rates for a range of cache sizes and configurations in a single run. We also collect branch misprediction rates for multiple branch predictors in a single run. Once these statistics are collected, we can predict cache miss rates and branch misprediction rates for any combination of cache hierarchy with any branch predictor and any processor core configuration.

These statistics, along with a number of machine parameters, serve as input to the analytical model, which then estimates superscalar in-order processor performance. The machine parameters include pipeline depth, pipeline width, number of functional units and their types, functional unit latency (multiply, divide, etc.), cache access latencies, and memory access latencies, as well as the configurations and sizes of caches, TLBs, and branch predictors.

Because the analytical model basically involves computing a limited number of formulas, a performance prediction is obtained almost instantaneously. In other words, once the initial profiling is done, the analytical model allows for predicting performance for a very large design space in the order of seconds or minutes at most.

### 2.2. Microarchitecture Description

We assume a superscalar in-order processor with five pipeline stages: fetch (IF), decode (ID), execute (EX), memory (MEM), and write-back (WB). IF and ID are referred to as the front-end stages of the pipeline, whereas EX, MEM, and WB are back-end stages. We consider a five-stage pipeline without loss of generality; we can model deeper pipelines by considering longer front-end pipelines and non–unit-latency instruction execution units, as will become clear later. Each stage has $W$ *slots* (numbered from 0 to $W-1$) to hold a total of $W$ instructions, with $W$ being the width of the processor. We assume forwarding logic such that dependent instructions can execute back-to-back in subsequent cycles. Further, we assume stall-on-use—in other words, the processor stalls on an instruction that consumes a value that has not been produced yet.

These instructions block in the ID stage. Load instructions perform address calculation in the EX stage and perform the cache access in the MEM stage. Finally, we assume in-order commit to enable precise interrupts. This implies that instructions that take more than one cycle to execute (e.g., a multiply instruction or a cache miss) block all subsequent instructions from going to the WB stage. Since each stage can only hold $W$ instructions, this further implies that when a long-latency instruction blocks instructions from passing from the MEM stage to the WB stage, the EX stage will eventually be filled with instructions, and hence no instructions can leave the ID stage.

## 3. OVERALL FORMULA

The overall formula for estimating the total number of execution cycles $T$ of an application on a superscalar in-order processor is as follows:

$$T = \frac{N}{W} + P_{misses} + P_{deps} + P_{FU}. \tag{1}$$

In this equation, $N$ equals the number of dynamically executed instructions, $W$ stands for the width of the processor, $P_{misses}$ is the total penalty due to miss events, $P_{deps}$ is the total penalty due to interinstruction dependences, and $P_{FU}$ stands for the penalty due to functional unit limitations (i.e., structural hazards).

The intuition behind the mechanistic model is that the minimum execution time for an application equals the number of dynamically executed instructions divided by processor width—that is, it takes at least $N/W$ cycles to execute $N$ instructions on a $W$-wide processor in the absence of miss events and stalls. Miss events, interinstruction dependences, and functional unit contention prevent the processor from executing instructions at a rate of $W$ instructions per cycle, which is accounted for by the model by adding penalty cycles.

The next sections discuss each of the terms of the formula. We start with miss event penalties and then discuss instruction dependences and functional unit contention.

## 4. MISS EVENTS

We determine the penalty due to miss events using the following formula:

$$P_{misses} = \sum_{i \in \{missEvents\}} misses_i \times penalty_i. \tag{2}$$

This formula computes the sum over the miss events, weighted with their respective penalties. We make a distinction between cache (and TLB) misses and branch mispredictions when it comes to computing the penalties.

### 4.1. Cache and TLB Misses

When an instruction cache miss occurs, the instructions in the front-end pipeline can still enter the EX stage, but when the instruction cache miss is resolved, it takes some time for the new instructions to refill the front-end pipeline. It is easy to understand that the front-end pipeline drain time and refill time offset each other—that is, the penalty for an instruction cache miss is independent of the front-end pipeline depth. In case of a data cache miss, the MEM stage blocks, and no instructions can leave or enter the EX stage until the data cache miss is resolved.

From the preceding discussion, it follows that the penalty for both an instruction and data cache miss equals its miss latency (i.e., the access time to the next level of cache or main memory). However, some instructions can complete execution in parallel with the miss penalty. In case of an instruction cache miss on a four-wide processor, one,

two, or three instructions could have already been fetched before the instruction cache miss occurred. Similarly, for a data cache miss, depending on the slot number at which the load instruction enters the MEM stage, one (the load instruction enters the MEM stage at slot 1), two (the load instruction enters at slot 2), or three older instructions (the load instruction enters at slot 3) can proceed to the WB stage. These instructions can complete underneath the cache miss and are therefore hidden. Assuming that cache misses are uniformly distributed across a $W$-wide instruction group, the average number of instructions hidden underneath a cache miss equals $\frac{W-1}{2}$. This means that the miss penalty can be reduced by $\frac{W-1}{2W}$ cycles (which is less than one cycle). The total penalty for a cache or TLB miss thus equals

$$penalty_{cacheMiss} = MissLatency - \frac{W-1}{2W}. \tag{3}$$

*Memory-level parallelism.* Memory-level parallelism (MLP) is defined as the number of simultaneously outstanding misses if at least one is outstanding [Chou et al. 2004]. This implies that we only have to account for the first, nonoverlapped memory access latency, as independent memory accesses later in the instruction stream are hidden underneath the first one. Out-of-order processors make use of this property by implementing a reorder buffer and miss status handling registers (MSHRs) to exploit MLP over a large window of instructions. For in-order processors, on the other hand, this window is limited to the width $W$ of the processor, and hence the amount of MLP is very low—that is, MLP can be exploited up until the instruction in the instruction stream that depends on the load miss (stall-on-use). When taking MLP into account, the penalty associated with the cache miss term in Formula (2) gets scaled as in the following Formula (4):

$$P_{cacheMisses} = \frac{cacheMisses_i}{MLP} \times penalty_{cacheMiss}. \tag{4}$$

As described by Van Craeynest et al. [2012], we can calculate the MLP as the number of memory accesses between a load instruction and its first consumer, since this consumer blocks the ID stage. However, since the processor can only hold $W$ instructions per pipeline stage, the load instruction will block any instruction at a distance larger than $(W-1)$ instructions from proceeding to the next stage. This implies that we never need to account for memory accesses outside of a window larger than $W$ instructions, even if the first consumer of the load is further than $(W-1)$ instructions apart. We have implemented a simple profiler that determines the average dependence distance between a load and its first consumer with the interinstruction dependence profile. We combine this with the fine-grained instruction mix profile to count the number of independent load instructions within this distance.

### 4.2. Branch Mispredictions

Branch mispredictions are slightly different from cache misses from a modeling perspective. Upon a branch misprediction, all instructions fetched after the mispredicted branch need to be flushed. In particular, when a branch misprediction is detected in the EX stage, all instructions in the front-end pipeline, as well as the instructions fetched after the branch in the EX stage, need to be flushed. Hence, the penalty of a branch misprediction equals

$$penalty_{branchMiss} = D + \frac{W-1}{2W}, \tag{5}$$

with $D$ the depth of the front-end pipeline. The first term is the number of cycles lost due to flushing the front-end pipeline: there are as many cycles lost as there are

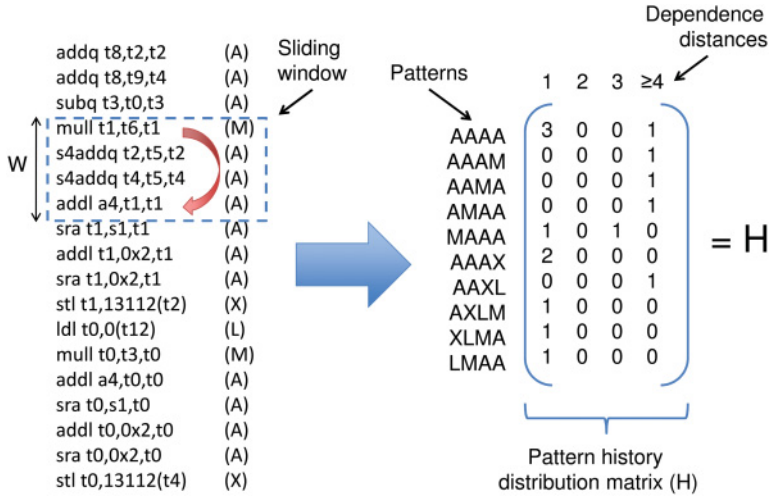



Fig. 3. Pattern distribution matrix generation for part of the dct_chroma routine of the h264 benchmark.

front-end pipeline stages, namely $D$. The second term is the penalty of flushing instructions in the EX stage; this number ranges between 0 and $W - 1$. We again assume a uniform distribution.

Correctly predicted branches may also introduce a performance penalty. In our setup, a branch is predicted one cycle after it was fetched, and if it is predicted taken, the instruction(s) in the IF stage and the instructions in the ID stage that are younger than the branch (which were fetched assuming a nontaken branch) need to be flushed, incurring a pipeline bubble. This incurs $1 + \frac{W-1}{2W}$ penalty cycles per branch that is predicted taken, even if it is correctly predicted. We will refer to this penalty as the *taken-branch hit penalty*.

## 5. INTERINSTRUCTION DEPENDENCES AND FUNCTIONAL UNIT CONTENTION

To determine the penalty caused by interinstruction dependences and functional unit contention, we need to keep track of (1) the distance between dependent instructions (the smaller the distance, the more likely the processor will stall to resolve the dependence) and (2) the order in which different instructions execute (subsequent instructions of the same type will put more pressure on the specific functional unit). In many cases, instructions will suffer from both dependences on prior instructions and from contention on functional units. We therefore summarize this information collectively in the pattern history distribution matrix ($H$-matrix), which we will use to calculate the penalty caused by interinstruction dependences and functional unit contention. Before explaining the formula, we will first illustrate how the $H$-matrix is constructed. Each instruction in the dynamic instruction stream can be represented by recording the history of the types of the $W - 1$ previous instructions, which we call a *pattern*, together with the distance to the closest instruction on which it depends. We can use this pattern $i$ and dependence distance $j$ as row and column indices, respectively, to increment a counter in the $H$-matrix for each instruction. As a result, the elements of the $H$-matrix represent counters that indicate the occurrences of pattern $i$ with a dependence distance of $j$.

Figure 3 illustrates this using a small portion of the dynamic instruction flow of the dct_chroma routine of h264 on the left, together with its associated $H$-matrix on the right. For the ease of visualization, each assembly instruction is represented by a

symbol that indicates the type of instruction: A stands for an ALU instruction, M for a multiply/divide, L for a load, and X for all other instructions that are not needed for the penalty calculation (e.g., store instructions).[2] To construct the $H$-matrix, we make use of a sliding window that slides through the whole instruction stream, instruction by instruction. The size of the window equals the maximum processor width $W$ of interest, which we set to 4 here and in all subsequent examples (unless stated otherwise). For each last instruction in the window, we record the distance to the closest instruction on which it depends together with the history of preceding instructions in the order executed from old (left) to most recent (right). Consider the position of the sliding window in Figure 3. The last instruction in the window is an ALU instruction, and the history pattern of instructions is denoted by MAAA. The last instruction has a dependence on the multiply instruction at distance 3 (as indicated by the red arrow). Therefore, we increment the $H$-matrix counter at the row with pattern MAAA and the column that represents dependence distance 3. We can now shift the window one instruction down to increment a counter for pattern AAAA at distance 1. We continue this process for all instructions.

By associating a penalty term to each row and column in the $H$-matrix, we can determine a cost matrix $C$. The $C$-matrix represents the cost (in number of cycles) that a specific history/dependence pattern incurs on the performance of the processor. By multiplying the matrices $C$ and $H$ term-wise and by accumulating them (i.e., taking the Frobenius product), we can calculate the total penalty term for functional unit contention and interinstruction dependences, as shown in Formula (6):

$$P_{deps} + P_{FU} = \sum_{i \in patterns, d=1..2\times W} C_{i,d} \times H_{i,d} = C : H. \tag{6}$$

To determine the individual terms in the $C$-matrix (i.e., the cost associated with a specific history pattern and dependence distance), we need to determine (1) the penalty for the specific history pattern assuming that there are no dependences and (2) the penalty for the specific dependence distance assuming a sufficient number of functional units. In case an instruction waits both for a dependence to resolve and for a functional unit to become available, the largest of those two penalties will be accounted for, as shown in Formula (7):

$$C_{i,d} = \max\left(c_{dep}(i,d), c_{fu}(i)\right). \tag{7}$$

$c_{dep}(i,d)$ represents the penalty of pattern $i$ when the last instruction in the pattern has a dependence distance $d$. $c_{fu}(i)$ is the penalty caused by functional unit contention for pattern $i$—in other words, when multiple instructions from the same instruction type reside in pattern $i$, we need to account a penalty waiting for an appropriate unit to start processing the last instruction of pattern $i$. The terms $c_{dep}(i,d)$ and $c_{fu}(i)$ are the subject of the following subsections.

Although Formula (6) shows how the sum of dependence penalties and functional unit contention penalties can be calculated together, we will show in Section 9.1 how we can determine these penalties separately.

### 5.1. Interinstruction Dependences

*5.1.1. Dependences on Unit-Latency Instructions.* In this section, we derive the penalty of an instruction that depends on the outcome of a close-by (within $W$ instructions)

[2]X stands for don't care instructions, as these instructions do not contribute to the penalty calculation for dependences and functional unit contention. Note that in the remainder of the article, we will mark other instructions irrelevant for the calculation with X.

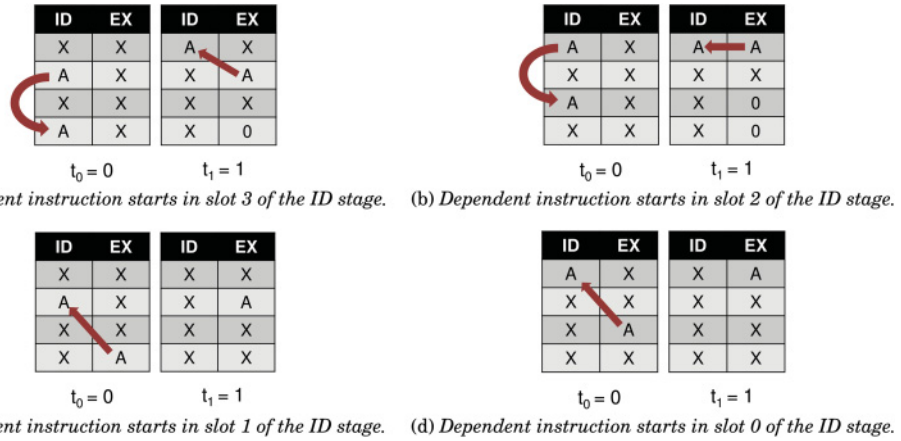



Fig. 4. Four possible instruction flows for an instruction dependent on an ALU instruction at distance $d = 2$.

unit-latency instruction. In our setup, integer ALU instructions are the only unit-latency instructions, so for the remainder of this section, we will refer to them as ALU instructions. Dependences on an ALU instruction resolve the cycle after the ALU instruction gets executed.

To illustrate the penalty calculation, consider the example pattern XAXA, where the first (oldest) ALU instruction produces a data value that is consumed by the next ALU instruction—that is, the dependence distance $d$ equals 2. There are four possible positions at which the instructions can enter the ID stage (see the ID stage at $t_0$ in the four parts of Figure 4). (The oldest instruction in an instruction group is shown at the top of each pipeline stage, and "0" denotes a bubble or an empty slot due to a stall.) In case (a), the dependent instruction enters the ID stage at position 3. When the ALU instruction starts execution at $t_1$, the dependent instruction gets blocked because the result of the ALU instruction is not available yet. Since the dependent instruction is the last instruction in the ID stage at $t_0$, one slot will be unused in the EX stage at $t_1$ (marked with 0); hence, we lose $\frac{1}{4}$ of a cycle. In case (b), the dependent instruction enters the ID stage in slot 2 at $t_0$. As in case (a), it gets blocked from going to the EX stage at $t_1$. In addition, the younger instruction that was at slot 3 in the ID stage of $t_0$ gets blocked from going to the EX stage. This means that we now lose $\frac{2}{4}$ of a cycle. In cases (c) and (d), the producing ALU instruction already started execution at $t_0$. This means that the dependence is resolved by the time the dependent instruction starts execution, and hence no cycles are lost.

Assuming that the four situations have equal probability of occurring (uniform distribution of the instructions in the pipeline), we derive the penalty for an instruction dependent on an ALU instruction at distance 2 as follows:

$$\begin{aligned} c_{dep}(\text{XAXA}, 2) &= (\text{Prob}[\text{pos} = 0] + \text{Prob}[\text{pos} = 1]) \times 0 \\ &\quad + \text{Prob}[\text{pos} = 2] \times \frac{2}{4} + \text{Prob}[\text{pos} = 3] \times \frac{1}{4} \\ &= \frac{1}{4} \times 0 + \frac{1}{4} \times 0 + \frac{1}{4} \times \frac{2}{4} + \frac{1}{4} \times \frac{1}{4} = \frac{3}{16}. \end{aligned} \tag{8}$$

In general, we can calculate the penalty for an instruction dependent on an ALU instruction at distance $d$ (with $d < W$) using the following formula:

$$c_{dep}(i, d < W) = \sum_{j=0}^{W-1} Prob\,[pos = j | pat = i] \times \begin{cases} \frac{W-j}{W} & \text{if} \quad d \leq j \\ 0 & \text{else} \end{cases} \tag{9}$$

$$= \sum_{j=d}^{W-1} \frac{1}{W} \times \frac{W-j}{W} \tag{10}$$

$$= \frac{(W-d)(W-d+1)}{2W^2}. \tag{11}$$

In this equation, $i$ represents the pattern, $W$ the processor width, $d$ the dependence distance to the closest ALU instruction, and $Prob\,[pos = j|pat = i]$ the probability that the newest instruction in the pattern $i$ is at position $j$ in the ID stage. Note that $i$ can be any pattern, with the constraint that the $(d + 1)$-th symbol is an "A"-symbol (i.e., the producer is an ALU instruction). The inequality $j \geq d$ indicates that we only need to account penalty when the producing ALU instruction is in the ID stage at the same cycle. In Formula (10), we make use of the assumption that instructions are uniformly distributed in the pipeline stage. We find this assumption to be accurate for the set of benchmarks used in our setup. However, one could conceive a corner case application that is dominated by instructions with dependence distances of 1 that causes all instructions to be serialized. Hence, most of the instructions of this application will enter the first slot of the ID stage. To model these corner cases, one could estimate the probabilities with a heuristic based on the overall average dependence distance: if the average dependence distance is close to 1, more weight needs to be given to $Prob[pos = 0]$. However, we found this case to be very rare in our setup, and modeling it increases the complexity of the model without noticeably improving its accuracy.

*5.1.2. Dependences on Load Instructions.* Unlike ALU instructions, load instructions do not produce their result in the EX stage but in the MEM stage. This has two consequences for calculating the penalty caused by instructions dependent on a load instruction. First, this means that if a load instruction and its consumer reside in the ID stage in the same cycle, an additional penalty cycle will need to be accounted for on top of the one calculated with Formula (11). Second, even when the load instruction and its dependent instruction reside in consecutive stages (load in the EX stage, dependent instruction in the ID stage), a penalty needs to be accounted for.

When $0 < d < W$, the load instruction can either be in the same stage or in a consecutive stage as the dependent instruction. When $W \leq d < 2W$, the load instruction and the dependent instruction can never reside in the same stage, and hence we will only need to account a penalty when they reside in consecutive stages.

The reasoning for calculating the penalty when $W \leq d < 2W$ is fairly similar as for dependences on ALU instructions where $d < W$. We can find the penalty for dependences on load instructions for a dependence distance $W \leq d < 2W$ by substituting $d$ by $d - W$ in Formula (10):

$$c_{dep}(i, d \geq W) = \sum_{j=d-W}^{W-1} \frac{1}{W} \times \frac{W-j}{W} \tag{12}$$

$$= \frac{(2W-d+1)(2W-d)}{2W^2}. \tag{13}$$

For $0 < d < W$, the penalty can be calculated using Formulas (14) and (15). The first term in Formula (14) reflects that for $d < W$, we always need to account a penalty, since both instructions either reside in the same stage or in consecutive stages. The second term reflects that if both the dependent instruction and the load instruction reside in the ID stage at the same time, we need to account an additional cycle:

$$c_{dep}(i, d < W) = \sum_{j=0}^{W-1} \frac{1}{W} \times \frac{W-j}{W} + \sum_{j=d}^{W-1} \frac{1}{W} \times 1 \tag{14}$$

$$= \frac{3W + 1 - 2d}{2W}. \tag{15}$$

*5.1.3. Dependences on Long-Latency Instructions.* Other long-latency instructions, such as integer multiply instructions, wait in the MEM stage until their result is calculated. Therefore, instructions that depend on long-latency instructions are penalized the same way as instructions that depend on load instructions (see Formulas (15) and (13) for $0 < d < W$ and $W \leq d < 2W$, respectively). Note that the latency of the long-latency instruction itself will be accounted for as a functional unit penalty (see the next section), so for the dependent instruction, we only have to account for the empty issue slots between the long-latency instruction and the dependent instruction.

However, as we will show in Sections 5.2.2 and 5.2.3, long-latency instructions can sometimes be executed in parallel, depending on the number of functional units available. As a result, no penalty is accounted to the instruction executing in parallel with another instruction. However, if there is a dependence between these instructions, the latency will not be hidden. We account for this by adding the latency of the long-latency instruction to the dependence penalty if the closest dependent instruction is of the same type.

### 5.2. Functional Unit Contention

*5.2.1. ALU Contention.* We first derive the penalty for integer ALU instructions due to functional unit contention. We model ALU contention penalties in a way that is analogous to dependence penalties on ALU instructions. For this, we need to define an analogy to the dependence distance:

$d_U(i)$ : distance to the instruction that causes functional unit contention when the processor has $U$ units, for pattern $i$.

For example, for a superscalar processor of width $W = 4$ with two ALUs, the contention distance for pattern XAAA, $d_U$(XAAA), equals 2, because the last instruction in the group can only start execution when the first one (at distance 2) finished its execution.

Replacing the dependence distance $d$ with $d_U(i)$ in Formula (10) allows us to determine the ALU contention penalty:

$$c_{fu}(i) = \text{fr}(i) = \sum_{j=0}^{W-1} Prob\,[pos = j | pat = i] \times \begin{cases} \frac{W-j}{W} & \text{if} \quad d_U(i) \leq j \\ 0 & \text{else} \end{cases}$$

$$= \sum_{j=d_U(i)}^{W-1} Prob\,[pos = j | pat = i] \times \frac{W-j}{W} \tag{16}$$

$$\approx \frac{(W - d_U(i))(W - d_U(i) + 1)}{2W^2}. \tag{17}$$

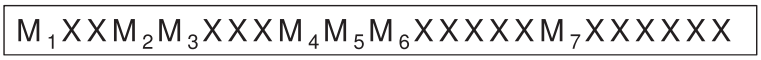



Fig. 5. An example instruction stream with multiply instructions.

These equation, $i$ represents the pattern, $W$ the processor width, $d_U(i)$ the distance to the closest instruction that causes contention when $U$ units are present, as defined before, and $Prob\,[pos = j|pat = i]$ the probability that the newest instruction in this pattern is in position $j$ in the ID stage. The inequality $j \geq d_U(i)$ indicates that we only need to account a penalty when the contending ALU instruction is in the ID stage at the same cycle. Note that we also defined the function fr($i$) for future reference.

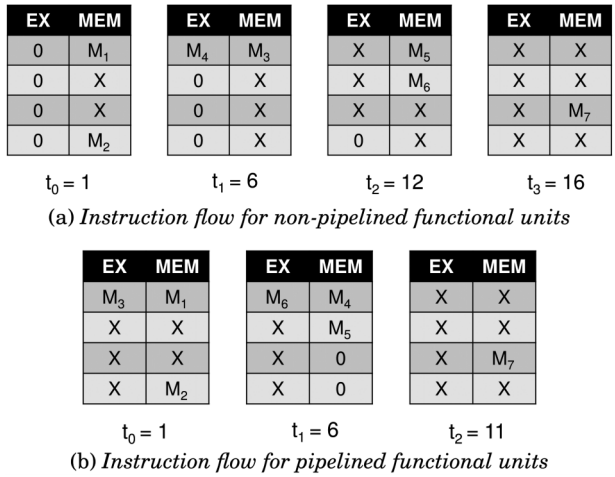
For the approximation in Formula (17), we again use the assumption that instructions are uniformly distributed in a pipeline stage. As with calculating inter-instruction dependencies, this assumption does not always hold true. In case the number of ALU instructions (e.g., three in the pattern AAXA) is larger than the number of ALUs+1 (e.g., $U = 1$) , some positions will make the instruction shift to another position because of contention between older ALU instructions in the pattern. For example, for the pattern AAXA and one ALU, only the first "A" will be executed, meaning that the next ALU instruction will have to stall for one cycle, and the second next ALU instruction will have to stall for (at least) two cycles. We model this effect by considering the pattern distribution matrix and by accounting for the number of stalls per pattern. Calculating these extra stall penalties is done in an automated way, and we find it to be important for the model's accuracy.

*5.2.2. Nonpipelined Long-Latency Functional Units.* We now derive formulas for long-latency instructions for a fixed number of functional units. We first explain how to model nonpipelined units and then discuss pipelined units in Section 5.2.3. Although the formulas are general enough to be applied to all types of functional units, we use integer multiply instructions[3] as an example to derive them. The only parameter that needs to be adjusted for other types of functional units is the latency.

Accounting penalty for a limited number of integer multiply instructions can be split up in two parts: (1) the fraction of cycles that is lost because of multiple M-instructions in the same stage in the same cycle and (2) the additional cycles we need to wait for the previous multiply instruction to finish (because they are not pipelined). The first part can be calculated as before using Formula (16). The second (and largest) part requires knowledge of how many multiply instructions can be issued in parallel.

We start with an example instruction stream that can be found in Figure 5. To ease the discussion, we introduce subscripts to enumerate the multiply instructions. Figure 6(a) shows four snapshots of the execution of this instruction stream, displaying the state of the EX stage and MEM stage, according to detailed simulation for a processor with two multiply units, where the execution latency of a multiply instruction is five cycles. The snapshots are chosen so that they reflect the start of each group of multiply instructions that can issue in parallel. For example, the first two multiply instructions (i.e., $M_1$ and $M_2$) start execution in the EX stage in cycle 0. In cycle 1 (i.e., $t_0 = 1$), they flow to the MEM stage, stay there for four cycles, and leave the MEM stage for the WB stage at cycle 5. This makes the multiply units available to start execution at cycle 5 for instruction $M_3$. At cycle 6, instruction $M_4$ can start execution in parallel with instruction $M_3$. The absolute start cycles are of less importance here. More important is to note that we distinguish four groups of multiply instructions,

[3]In practice, modern integer multiply units are typically pipelined; older processors such as the Alpha 21064 and MIPS R4000 have nonpipelined integer multiply units. We merely use the integer multiply instruction as an example throughout the article to explain the construction of the formulas.




(a) *Instruction flow for non-pipelined functional units*

(b) *Instruction flow for pipelined functional units*

Fig. 6. Instruction flow of the example instruction stream in Figure 5 through a superscalar processor with two multiply units.

Table I. Pattern Distribution for the Instruction Stream in Figure 5 and Penalties Assuming Two Multiply Units

| Pattern | Frequency | Nonpipelined Penalty | Pipelined Penalty |
|---|---|---|---|
| XXXM | 3 | latency − 1 | latency − 1 |
| MXXM | 1 | 0 | 0 |
| XXMM | 2 | 0 | 0 |
| XMMM | 1 | latency − 1 | 0 |

meaning that we penalize the performance by four times the multiply latency for the seven "M"-instructions.

The challenge now is to calculate the penalty while having limited information compared to detailed simulation—that is, using only the pattern distributions of the $H$-matrix (Table I). With Formula (16), we can calculate the fraction of cycles lost because there are more multiply instructions than multiply units available at the same time in the same stage. To determine the parallelism of multiply instructions that can get executed at the same time, we can again use the pattern distribution matrix. We consider patterns that end with an M-symbol and account an appropriate penalty, based on the total number of M-symbols in the pattern.

For the example where we have two multiply units, we do not need to account penalty for patterns with exactly two M-symbols, as the right-most multiply instruction can be issued in parallel with the previous one. This is also true when all four instructions in the pattern are multiply instructions. If, however, the right-most instruction is the only multiply instruction or if it is the third multiply instruction, the full penalty needs to be accounted for. For the example instruction stream of Figure 6, the distribution and penalties of patterns ending in with an M-symbol can be found in the second and third columns of Table I.

In general, we can derive the following formula to calculate the total cost for executing a long-latency instruction for an arbitrary number of units:

$$c_{fu}(i) = \text{fr}(i) + \begin{cases} \text{latency} - 1 & \text{if } (\#\text{insns}(i) \bmod U) = 1 \\ 0 & \text{else.} \end{cases} \tag{18}$$

In Formula (18), $U$ is the number of units of this type of long-latency instruction, and #insns($i$) represents the number of instructions of this type in pattern $i$. The intuition behind this formula is that the first multiply instruction will appear as an XXXM pattern, for which we account the full penalty. The second multiply instruction will see the first instruction as an older instruction in its pattern (e.g., XMXM), so we do not need to account a penalty if there are two or more multiply units.

However, Formula (18) can lead to underestimations in situations where we have a dense concentration of multiply instructions (e.g., XXXMMMMMMMMM would be accounted only two times the latency instead of four, because all of the MMMM patterns have no latency). This can be solved by modifying the previous formula as follows:

$$c_{fu}(i) = \text{fr}(i) + \begin{cases} \text{latency} - 1 & \text{if } \big(\#\text{insts}(i) \bmod U\big) = 1 \\ \frac{\text{latency}-1}{\min(U, \#\text{insts}(i))} \times Pr[\text{dense}|\text{pattern} = i] & \text{else.} \end{cases} \tag{19}$$

Here, $Pr[\text{dense}|\text{pattern} = i]$ is the probability that pattern $i$ appears in a dense concentration of multiply instructions. We estimate this probability by counting the frequency of occurrence from the pattern distribution matrix.

*5.2.3. Pipelined Long-Latency Functional Units.* We now move to modeling the impact of long-latency instructions that execute on pipelined functional units. Pipelined functional units have the advantage that they are capable of executing a new instruction every cycle, and hence they yield a potential performance improvement over nonpipelined execution, but they also require a more complex design. Because of in-order execution, however, there is a limit on the potential performance improvement: when a long-latency instruction is being executed, it will block all younger instructions from passing the MEM stage (because of in-order commit). This will make the EX stage fill up with instructions, blocking younger instructions from starting execution. So, only instructions that can make it into the EX stage can potentially execute in parallel.

In Figure 6(b), we illustrate what happens with the example of Figure 5 if the multiply units are pipelined. We distinguish three groups of multiply instructions that can be executed in parallel (assuming no dependences between them).

As before, we use the distribution matrix in Table I. To calculate the penalties, we again account for two parts: the part in which we lose a fraction of a cycle because there are more multiply instructions at the same time in the EX stage than multiply units available, and the part where we need to account for the latency of the multiplication itself. For pipelined units, we only need to account for this latency if the current multiply instruction is the only multiply in the pattern. The fourth column in Table I shows the penalties for pipelined units. Using these penalties and the distribution matrix, we can account for the total penalty caused by long-latency instructions in the example instruction stream.

In general, the penalty for long-latency instructions on pipelined functional units can be calculated with Formula (20). We only account a penalty if there is exactly one multiply instruction in the pattern, because all other instructions can be executed in parallel. As is the case with nonpipelined functional units, we also account for a penalty in case there is a high density of multiply instructions:

$$c_{fu}(i) = \text{fr}(i, U) + \begin{cases} \text{latency} - 1 & \text{if } \#\text{insts}(i) = 1 \\ \frac{\text{latency}-1}{\#\text{insts}(i)} \times P[\text{dense}|\text{pattern} = i] & \text{else.} \end{cases} \tag{20}$$

## 6. EXPERIMENTAL SETUP

We use 19 benchmarks from the MiBench benchmark suite [Guthaus et al. 2001], which is a popular suite of embedded benchmarks from different application domains, including automotive/industrial, consumer, office, network, security, and telecom. Next

Table II. Parameter Settings for the Functional Units

| Parameter | Default | Range | Cortex-A8 |
|---|---|---|---|
| Integer ALUs (IA) | 2 | 1 to 4 | 2 |
| Integer Multiply/ Divide Units (IM) | 1 nonpipelined | 1 to 4 nonpipelined/pipelined | 1 pipelined |
| Floating-Point ALUs (FA) | 1 nonpipelined | 1 to 4 nonpipelined/pipelined | 1 nonpipelined |
| Floating-Point Multiply/ Divide Units (FM) | 1 nonpipelined | 1 to 4 nonpipelined/pipelined | merged with FA-unit |

Table III. Default Parameter Settings

| Parameter | Default | Cortex-A8 |
|---|---|---|
| Processor width | 4 | 2 |
| Processor frequency | 1GHz | 1GHz |
| Pipeline depth | 5 stages | 13 stages |
| FP ALU latency | 3 cycles | 10 cycles |
| FP Multiply latency | 15 cycles | 17 cycles |
| FP Divide latency | 15 cycles | 65 cycles |
| FP MAC latency | 15 cycles | 26 cycles |
| FP Sqrt latency | 15 cycles | 60 cycles |
| Integer Multiply latency | 5 cycles | 3 cycles |
| Integer Divide latency | 20 cycles | NA |
| L1 D- and I caches | 128KB 4-way set assoc | 32KB 4-way set assoc |
| L2 cache | 4MB 8-way set assoc | 256KB 8-way set assoc |

to these MiBench benchmarks, we also use 15 benchmarks from SPEC CPU2006. We selected inputs from the KDataSets input database [Chen et al. 2010] so that each MiBench benchmark executes approximately 1 billion instructions. For SPEC CPU2006, we generated representative simulation regions of 1 billion instructions each using SimPoint [Hamerly et al. 2005].

We use the gem5 simulation framework [Binkert et al. 2011]. We derive our profiler from gem5's functional simulator, and we validate our model against detailed cycle-level simulation using gem5. Detailed simulation runs at 92 KIPS on an Intel Xeon Harpertown (L5420) processor. Although our profiler is more than 10 times faster, running at 1.4 MIPS, profiles need be calculated only once for a whole range of processor configurations. With these profiles, we can quickly generate performance estimates by evaluating the preceding analytic formulas. This is done in a couple of seconds for the complete design space.

The parameters that we vary in the next sections are shown in Table II, along with the default settings, which forms a design space of 2,048 configurations. For model validation, we use a subset of 70 randomly selected configurations that span a broad range of the design space of the total 2,048 configurations. Although our model can be applied while varying many more parameters (such as pipeline width, depth, cache sizes, cache associativities, and branch predictor settings), in this article, our primary focus is on the processor core in which we vary the number and configuration of the functional units, as this is the most complicated part to model for superscalar in-order processors. Table III shows the default settings for the other processor parameters. We refer the interested reader to the earlier version of the model for results in which we

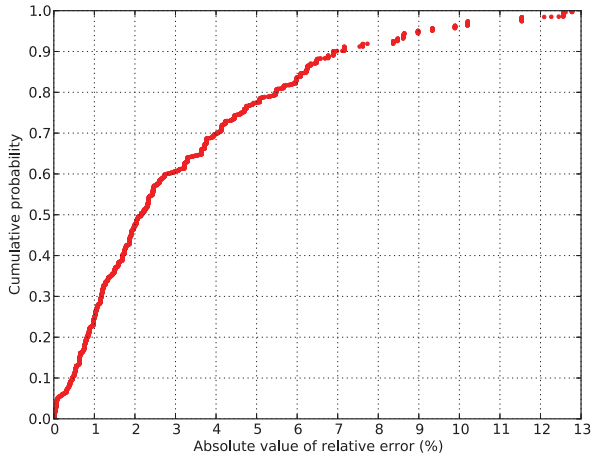



Fig. 7. Cumulative probability distribution of error for 70 configurations on SPEC CPU2006 and MiBench.

vary other processor parameters such as pipeline depth, width, cache configuration, and branch predictor (see Breughe et al. [2012]).

For hardware validation, we use a BeagleBone Black board with the Texas Instruments AM3358 Sitara ARM Cortex-A8 processor, running Debian GNU/Linux 7. The corresponding microarchitectural parameters can be found in the right-most columns of Tables II and III and are based on the technical references from ARM Holdings [2010] and Texas Instruments Incorporated [2014]. Because the Cortex-A8 is an ARM-processor, we extended our profiler to capture profiles for the ARM ISA, in addition to the Alpha ISA, which is used for the validation against detailed simulation. We used Linux's built-in time command to measure user time, and we disable dynamic frequency scaling by setting the processor to a fixed clock frequency of 1GHz in our experiments. To avoid generating new SimPoints for SPEC CPU2006 on ARM, and setting up the framework to execute SimPoints on hardware, our hardware validation is limited to the MiBench suite.

## 7. VALIDATION AGAINST DETAILED SIMULATION

We validate the model against detailed simulation in two steps. First, we evaluate accuracy by simulating a large range of different configurations. We consider 34 benchmarks and 70 configurations, and we compare the CPI values of detailed simulation versus the model. Figure 7 shows a cumulative plot comparing all simulated points with the model. From this figure, we can see that about 90% of all evaluated points have an error of less than 7% in CPI. Overall, the model has an error of 3.2% on average, with a maximum error of 13%.

Second, we evaluate how well the model tracks the relative difference on a number of configurations: starting from the baseline configuration, we increase the number of available functional units. We also study the effect of pipelining functional units. We start by studying the effect of adding floating-point multiply units to our baseline configuration. As can be seen in Figure 8, adding a second multiply unit decreases CPI significantly. (This graph shows results for the floating-point benchmarks only; i.e., the other benchmarks do not see a performance impact from varying the number of floating-point units.) Four multiply units reduces CPI only for a few benchmarks. These trends are tracked accurately by our model. On average, the model has an error of only 2.1% for the configurations in this experiment and a maximum error of 5.5%.

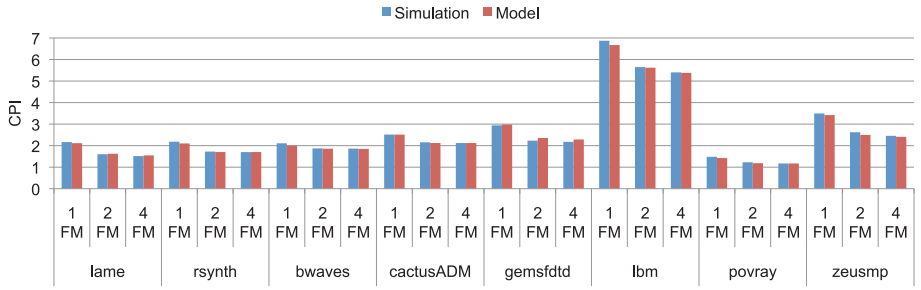



Fig. 8. Model validation while varying the number of floating-point multiply units (FM) for the floating-point benchmarks of MiBench and SPEC CPU2006.

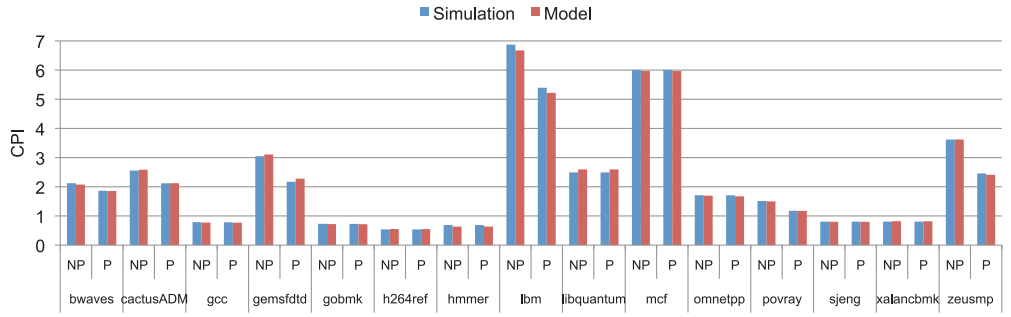



Fig. 9. Evaluation of the model compared to detailed simulation for pipelined (P) and nonpipelined (NP) functional units on SPEC CPU2006.

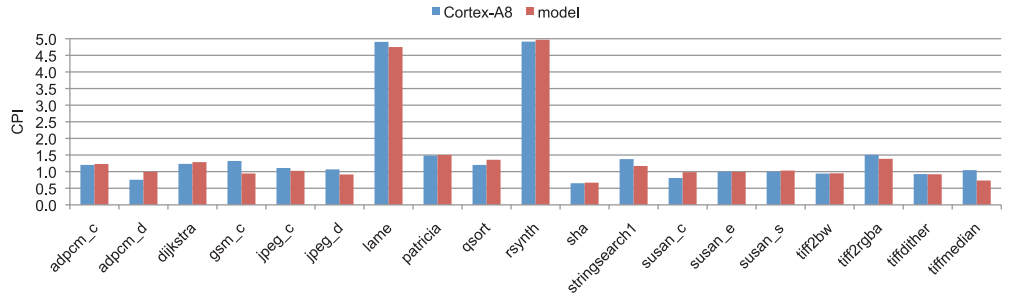



Fig. 10. Evaluation of the model compared to the Cortex-A8 microarchitecture.

We now look at the impact of pipelining all of the units (i.e., integer multiply/divide unit, floating-point ALU, and floating-point multiply/divide unit) of the baseline configuration, as shown in Figure 9. Due to space constraints, we only plot results for SPEC CPU2006; we obtain similar results for the MiBench benchmarks. We observe an average absolute error of 2% and a maximum error of 7.6%.

## 8. HARDWARE VALIDATION

Figure 10 shows CPI values for our set of 19 MiBench benchmarks when executed on the Cortex-A8 processor, along with the prediction of our model. We find that the average absolute prediction error is 10%, with 12 benchmarks showing an error of less than 8%; the maximum error equals 32% for adpcm_d. Overall, the model is fairly accurate, taking into account that we did not make important changes to the model

compared to the ALPHA/gem5 model. The only changes are adjustments to the profiler to be compatible with the ARM ISA and modeling Cortex A8's variable latencies for floating-point instructions.

More specifically, to improve the accuracy of the floating-point benchmarks, we required a more fine-grained breakdown of the floating-point instruction mix (i.e., multiply, division, multiply-accumulate, and square root). We then use this fine-grained instruction mix together with instruction latencies found in the Cortex-A8 technical reference [ARM Holdings 2010] to generate a weighted average of the overall floating-point latency and feed it into our model.

Several adjustments to the model could be made to improve accuracy even further. The gem5 models an fetch buffer with the size of an entire cache line, as observed by Gutierrez et al. [2014], which underestimates the number of instruction cache accesses in gem5. Because our original modeling efforts are targeted at a microprocessor similar to gem5's in-order core, we make the same assumptions on the fetch buffer. For example, gsm_c is an application within the top three benchmarks with most instruction cache misses, indicating that its instruction footprint is non–cache resident, and hence modeling a smaller fetch buffer would correctly predict a higher execution time. Furthermore, we are unaware of the branch prediction latency of the Cortex-A8, whereas we model taken branches with 1 cycle of penalty. In addition, Gutierrez et al. [2014] show that gem5's branch prediction accuracy decreases for low MPKI values. Both of these branch predictor inaccuracies could explain our performance underestimation of adpcm_d. Further, the Cortex-A8 does not allow executing two instructions in the same slot if they have output dependences (WAW dependences), whereas our model abstracts this away. This likely impacts the overall performance overestimation. Finally, we assume a fixed memory latency, whereas DRAM latency tends to depend on the physical memory addresses for which requests are made. As correctly stated by Desikan et al. [2001], this depends on the virtual to physical page mappings of the native system and is quite difficult to replicate. We find that benchmarks spending a lot of time in system calls, such as tiff2rgba and tiffmedian, indeed show performance overestimations.

## 9. CASE STUDIES

### 9.1. Revealing Performance Bottlenecks

In our first case study, we use the model to understand the performance numbers of the example in the introduction of the article (see Figure 1(b)).

Formula (1) derives the number of execution cycles of an application on a target microprocessor as a sum of terms. This property is useful in determining performance bottlenecks. By identifying the largest contributors to the execution cycles, one can find the most promising directions to improve performance. For example, suppose that the penalty due to data cache misses (part of the $P_{misses}$ term) is the largest contributor to the execution cycles; performance could be improved by installing a larger cache or by improving data locality. If, on the other hand, the penalty due to functional units ($P_{FU}$) is relatively large, we can improve performance by adding functional units.

We now build CPI stacks for the benchmarks and configurations of Figure 1(b). Thereto, we divide the terms in Formula (1) by the total number of instructions $N$, and we split up the terms $P_{misses}$ and $P_{FU}$ into smaller terms to get a better level of detail. The "base" component is the first term (i.e., $\frac{N}{W}$), which becomes $\frac{1}{W}$ when divided by $N$. $P_{misses}$ can easily be split up by multiplying the number of miss events of each type with their respective penalty as explained in Section 4.

Since $P_{FU}$ and $P_{deps}$ are modeled in a unified matrix $C$, we have to split up the matrix $C$ into $C_{FU}$ and $C_{deps}$ to determine penalties for functional units and interinstruction

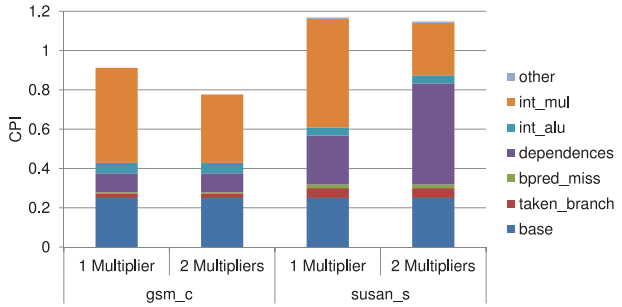



Fig. 11. CPI stacks reveal that interinstruction dependences between multiply instructions are the underlying bottleneck that is preventing performance improvement for susan_s. The "other" component is all other terms in the model that only have a small component.

dependences separately. We do this similarly to how the matrix was constructed in Formula (7):

$$C_{deps}(i,d) = \begin{cases} c_{dep}(i,d) & \text{if} c_{dep}(i,d) > c_{fu}(i) \\ 0 & \text{else} \end{cases}$$

$$C_{FU}(i,d) = \begin{cases} c_{fu}(i,d) & \text{if} c_{dep}(i,d) \leq c_{fu}(i) \\ 0 & \text{else.} \end{cases}$$

The term $P_{FU}$ can be further broken down into $P_{intALU}$, $P_{intMultiply}$, $P_{fpALU}$, and $P_{fpMultiply}$ by accounting for the $C_{FU}(i,d)$ terms only, in which $i$ denotes a pattern associated with that functional unit (i.e., the last instruction in pattern $i$ executes on functional unit type $FU$).

The CPI stacks for the benchmarks and configurations of Figure 1(b) are shown in Figure 11. Although the model has a small error on the performance prediction, it reveals why there is hardly any performance improvement for susan_s. With a single multiply unit, we observe that the penalties due to integer multiplies ($P_{intMultiply}$) are relatively high. When we add a second multiply unit, we see that this term is significantly reduced for gsm_c, which can be explained by the many patterns with more than one multiply instruction. For susan_s, we see that the $P_{intMultiply}$ term is also reduced; however, the term $P_{deps}$ is increased by the same amount that $P_{intMultiply}$ was reduced. The decrease in $P_{intMultiply}$ can again be explained by patterns consisting of multiple multiply instructions that can execute in parallel. However, the increase in $P_{deps}$ means that these multiply instructions depend on each other, which inhibits parallel execution.

### 9.2. Minimizing the Number of Functional Units for a Given Performance Target

In our second case study, we use the model to minimize the number of units needed for a specific performance target. We use gem5 to find the performance (expressed as IPC) of our baseline configuration (containing 5 functional units) and the maximum achievable performance when having 4 functional units of each type (i.e., a total of 16 units). The harmonic mean of speedups over the baseline IPC equals 1.087. The maximum speedup is observed for lame (1.52) and zeusmp (1.49).

We now use the model to find optimal configurations per benchmark (i.e., configurations with a minimum number of functional units), where we set the performance target at 98% of the maximum IPC. Detailed simulation of these optimized configurations confirms that these configurations indeed have an IPC of at least 98% of the maximum achievable IPC as predicted by the model, by using a minimum amount of functional units. Figure 12 shows IPC numbers, resulting from detailed simulation, for

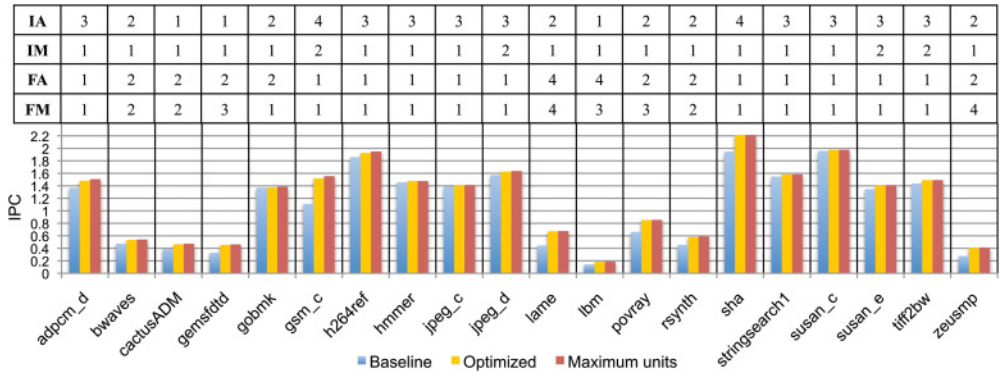



Fig. 12. Baseline performance, the performance of the configuration with four units of each type (Maximum units), and the performance of the configuration picked by the model with a minimum number of functional units within 98% of the optimum (Optimized). The table aligned with the figure details the configuration picked by the model: number of integer ALUs (IA), integer multiply/divide units (IM), floating-point ALUs (FA), and floating-point multiply/divide units (FM).

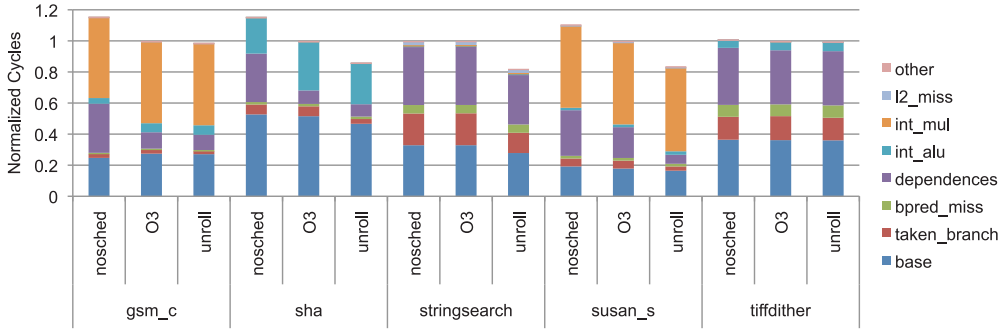



Fig. 13. Normalized cycle stacks for five benchmarks across different compiler optimizations.

a selection of benchmarks, along with the optimal number of units of each type. Because of space constraints, we only show the 20 benchmarks that have the largest delta in IPC between the baseline configuration and the configuration with a total of 16 functional units. (We obtain similar results for the other 14 benchmarks.) The "Optimized" bars represent the IPC results for the configurations found in the table that is aligned with the figure. The bars "Baseline" and "Maximum units" represent our baseline of 5 units, and the configuration with all 16 units, respectively. The harmonic mean of the speedups of the optimized configurations over the baseline configuration equals 1.08 (maximum speedup of 1.51 for lame and 1.47 for zeusmp). We observe that for 29 out of the 34 benchmarks, we only need 7 or fewer functional units to achieve at least 98% of the IPC with 16 functional units. The resulting configurations are nontrivial and time-consuming to find using detailed cycle-accurate simulation, which motivates the use of our fast model to guide design choices.

### 9.3. Compiler Optimizations

In our last case study, we use the model to study how compiler optimizations affect superscalar in-order performance (Figure 13). We consider `-O3`, `-O3` without instruction scheduling (`-O3 -fno-schedule-insns`), and `-O3` with loop unrolling turned on

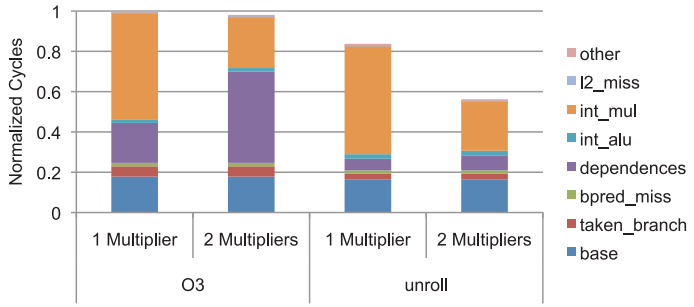



Fig. 14. Normalized cycle stacks for susan_s with and without loop unrolling, and on two different architectures (one and two multiply units).

(`-O3 -funroll-loops`) for the five benchmarks for which we observed the largest impact due to compiler optimizations. Figure 13 shows normalized cycle stacks—that is, a cycle stack is computed by multiplying a CPI stack with the number of dynamically executed instructions; the cycle stacks are then normalized to the execution time with the `-O3` optimization level. For most of the benchmarks, instruction scheduling increases the distance between dependent instructions, resulting in a lower penalty due to dependences. For some benchmarks (e.g., gsm_c), the base component increases slightly through instruction scheduling, meaning that the number of executed instructions increases. The reason for this is the addition of spill code. However, the cost of spill code is compensated for by the substantial decrease in the impact of interinstruction dependences.

Most of the benchmarks (and all of the ones in Figure 13) benefit from loop unrolling. Three components get an important reduction through loop unrolling. First, the number of dynamic instructions decreases because fewer branches and loop iteration counter increments are needed after loop unrolling. Second, because there are fewer branches, the penalty due to taken branches also decreases. Third, for susan_s, we observe the biggest contribution from the smaller penalty due to interinstruction dependences; clearly, loop unrolling enables the instruction scheduler to better schedule instructions so that fewer interinstruction dependences have an impact on in-order performance.

As shown in Section 9.1, dependences prevent a performance improvement when the number of multipliers is increased from one to two for susan_s. In Figure 14, we show normalized cycle stacks for susan_s without loop unrolling (`-O3`) and with loop unrolling enabled (`-unroll`) on the baseline architecture (one multiply unit) and on the baseline architecture with an additional multiply unit (two multiply units). As mentioned earlier, for `-O3` the penalty for multiply instructions transforms into a penalty for interinstruction dependences when a second multiply unit is added. When we enable loop unrolling, however, many of these additional interinstruction dependences can be removed because the instruction scheduler is now able to place independent multiply instructions (that were originally spread across loop iterations) closer together and dependent ones further apart. As a result, we see a considerable performance gain when the number of multiply units is doubled for the loop-unrolled version of susan_s.

## 10. RELATED WORK

We now describe prior work in analytical modeling, statistical modeling, and program characterization that is most related to our work.

### 10.1. Analytical Modeling

Basically, there are three approaches to analytical modeling: mechanistic modeling, empirical modeling, and hybrid mechanistic/empirical modeling.

Mechanistic modeling derives a model from the mechanics of the processor, and prior work focused on mechanistic modeling of out-of-order processor performance for the most part. Michaud et al. [1999] build a mechanistic model of the instruction window and issue mechanism. Karkhanis and Smith [2004] extend this simple mechanistic model to build a complete performance model that assumes sustained steady-state issue performance punctuated by miss events. Chen and Aamodt [2008] improve on this model through more accurate modeling of pending data cache hits, overlaps between computation and memory accesses, and the impact of a limited number of MSHRs. Taha and Wills [2008] propose a mechanistic model that breaks up the execution into so-called macro blocks, separated by miss events. Eyerman et al. [2009] propose the interval model for superscalar out-of-order processors. Whereas all of this prior work focused on out-of-order processors, Breughe et al. [2011] proposed a mechanistic model for *scalar* in-order processors. This article presents a mechanistic model for superscalar in-order processors that involves substantial modeling enhancements with respect to functional unit contention and interinstruction dependences, as explained in Sections 3 through 5.

In contrast to mechanistic modeling, empirical modeling requires little or no prior knowledge about the system being modeled: the basic idea is to learn or infer a performance model using machine learning and/or statistical methods from a large number of detailed cycle-accurate simulations. Empirical modeling seems to be the most widely used analytical modeling technique today and was employed for modeling out-of-order processors only, to the best of our knowledge. Some prior proposals consider linear regression models for analysis purposes [Joseph et al. 2006a], nonlinear regression for performance prediction [Joseph et al. 2006b], spline-based regression for power and performance prediction [Lee and Brooks 2006], neural networks [Dubach et al. 2007; Ipek et al. 2006], or model trees [Ould-Ahmed-Vall et al. 2007].

Hybrid mechanistic-empirical modeling targets the middle ground between mechanistic and empirical modeling: starting from a generic performance formula derived from understanding the underlying mechanisms, unknown parameters are derived by fitting the performance model against detailed simulations. For example, Hartstein and Puzak [2002] propose a hybrid mechanistic-empirical model for studying optimum pipeline depth; the model is tied to modeling pipeline depth only and is not generally applicable. Eyerman et al. [2011] propose a more complete mechanistic-empirical model that enables constructing CPI stacks on real out-of-order processors.

### 10.2. Interinstruction Dependence Modeling and Functional Unit Contention

Dubey et al. [1994] present an analytical model for the amount of instruction-level parallelism (ILP) for a given window size of instructions based on the interinstruction dependence distribution. Kamin et al. [1994] approximate the interinstruction dependence distribution using an exponential distribution. Later, Eeckhout and De Bosschere [2001] found a power law to be a more accurate approximation.

The interinstruction dependence distribution is an important program statistic for statistical modeling. Noonburg and Shen [1997] present a framework that models the execution of a program on a particular architecture as a Markov chain, in which the state space is determined by the microarchitecture and in which the transition probabilities are determined by the program. Statistical simulation [Eeckhout et al. 2003; Oskin et al. 2000] generates a synthetic program or trace from a set of statistics.

Most of the previous work on functional unit contention has focused on out-of-order processors, such as Taha and Wills [2008]. Other work, such as Noonburg and Shen [1994], Lee [2010], and Zhu et al. [2005] has models that can be applied for in-order processors as well. Noonburg and Shen [1994] make a distinction between program parallelism and machine parallelism, and both are combined to determine the overall processor performance. Zhu et al. [2005] employ a multiple-class multiple-resource queuing system to model a variable number of functional units to arrive at a hybrid mechanistic-probabilistic approach to estimate processor performance. Although previous work takes the instruction mix into account for determining performance for a different number of functional units, they lose in accuracy because this instruction mix is a global instruction mix. However, this work uses fine-grained instruction mixes and requires only one profiling run (no parameter fitting is needed) to determine performance for all possible numbers of functional units.

## 11. FUTURE WORK

The proposed mechanistic model focuses on single-core processors only. It would be interesting to extend this work toward multicore processors with superscalar in-order cores. The challenge with this extension is the need to account for additional cache misses and longer memory access penalties because of resource sharing. To be able to use the model in a multicore environment, parameters corresponding to cache misses and memory access times would need to be estimated before evaluating Formula (2). Existing work by Van Craeynest and Eeckhout [2011] and Chen and Aamodt [2009] provides mechanisms to calculate these estimates and are orthogonal to our work.

Another interesting extension would be toward multithreaded processors. Multithreaded processors are built to hide stall latencies by executing multiple applications on the same processor. This means that next to resource contention, we would also need to model the overlapping of stalls from the different application threads. Chen and Aamodt [2009] model multithreaded processors for scalar (single-issue) in-order cores. They implement a Markov chain that transitions between states, where each state indicates how many threads are stalled. The inputs to the model are IPC results of the individual applications along with probabilities for stall events. Eyerman and Eeckhout [2010] use similar inputs (i.e., single-thread statistics) to calculate overall system throughput for simultaneous multithreading (SMT) out-of-order processors. The challenge when considering multithreaded *superscalar, in-order* processors is that our detailed instruction profile (the $H$-matrix) would depend on the runtime context, resulting in a complex chicken-and-egg problem—that is, per-thread progress determines interthread interleaving and interference, and vice versa, interthread interference affects per-thread progress. One possible solution might be, instead of computing a single profile for each application, to compute multiple profiles at intervals of a fixed number of instructions for each application individually. We could then build a single combined profile for the possible co-executions in the multithreaded instruction stream, in a manner similar to the co-phase matrix work [Van Biesbrouck et al. 2004]. An alternative solution might be to solve the chicken-and-egg problem through an iterative approach that computes the impact of per-thread progress on multithreaded resource sharing and vice versa, in a manner similar to the work of Van Craeynest and Eeckhout [2011] and Eklöv et al. [2011].

## 12. CONCLUSION

In this article, we propose a performance model for superscalar in-order processors that uses analytical formulas derived from understanding the internal mechanics of the microarchitecture. The formulas reflect the impact of functional unit contention and interinstruction dependences on superscalar processor performance. By combining a

detailed instruction mix and dependence distance profiles of a program's dynamic instruction stream with a number of program-machine characteristics (e.g., cache miss rates, MLP, and branch misprediction rates), we demonstrate that our model has an error of only 3.2% on average compared to detailed simulation with gem5. Hardware validation against the ARM Cortex-A8 demonstrates an average absolute error of 10%. The evaluation speed of the model is close to instantaneous, as it only involves solving a number of analytical formulas. Furthermore, the microarchitectural independent profiling step, which is needed to provide the model input, is a one-time cost and is at least 10 times faster than a single detailed simulation run.

We use the model both as an exploration tool and one to gain insight into an application's execution behavior, as well as to visualize microarchitectural bottlenecks. We demonstrate how the model can find an optimal set of functional units to achieve a given performance target. Finally, we demonstrate the model's usefulness to identify microarchitectural bottlenecks. Instead of analyzing results of many detailed simulations, the model can visualize how an application interacts with a microarchitecture and hence provides insights on how performance can or cannot be improved. By applying this visualization technique on differently optimized binaries of the same application, the model provides insight into how compiler optimizations affect the program–microarchitecture interactions.

## ACKNOWLEDGMENTS

We thank the associate editor and the anonymous reviewers for their valuable feedback. This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, as well as the EU FP7 Adept project no. 610490.

## REFERENCES

D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the International ACM Symposium on Operating Systems Principles (SOSP)*. 1–14.

ARM Holdings. 2010. *Cortex-A8 Technical: Reference Manual* (3p2 ed.). ARM Holdings, Cambridge, NJ.

N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *Computer Architecture News* 39, 2, 1–7.

M. Breughe, S. Eyerman, and L. Eeckhout. 2012. A mechanistic performance model for superscalar in-order processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 14–24.

M. Breughe, Z. Li, Y. Chen, S. Eyerman, O. Temam, C. Wu, and L. Eeckhout. 2011. How sensitive is processor customization to the workload's input datasets? In *Proceedings of the IEEE International Symposium on Application-Specific Processors (SASP)*. 1–7.

X. E. Chen and T. M. Aamodt. 2008. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 59–70.

X. E. Chen and T. M. Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *Proceedings of the IEEE 15th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Los Alamitos, CA, 329–340.

Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. 2010. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 448–459.

Y. Chou, B. Fahs, and S. Abraham. 2004. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. 76–87.

R. Desikan, D. Burger, and S. W. Keckler. 2001. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*. 266–277.

C. Dubach, T. M. Jones, and M. F. P. O'Boyle. 2007. Microarchitecture design space exploration using an architecture-centric approach. In *Proceedings of the IEEE/ACM Annual International Symposium on Microarchitecture (MICRO)*. 262–271.

P. K. Dubey, G. B. Adams III, and M. J. Flynn. 1994. Instruction window size trade-offs and characterization of program parallelism. *IEEE Transactions on Computers* 43, 4, 431–442.

L. Eeckhout and K. De Bosschere. 2001. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 25–34.

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

D. Eklöv, D. Black-Schaffer, and E. Hagersten. 2011. Fast modeling of cache contention in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*. 147–158.

S. Eyerman and L. Eeckhout. 2010. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–102.

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

S. Eyerman, K. Hoste, and L. Eeckhout. 2011. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 216–226.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC)*.

A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. 2014. Sources of error in full-system simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 13–22.

G. Hamerly, E. Perelman, J. Lau, and B. Calder. 2005. SimPoint 3.0: Faster and more flexible program analysis. *Journal of Instruction-Level Parallelism* 7, 1–28.

A. Hartstein and T. R. Puzak. 2002. The optimal pipeline depth for a microprocessor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*. 7–13.

M. D. Hill and A. J. Smith. 1989. Evaluating associativity in CPU caches. *IEEE Transactions on Computers* 38, 12, 1612–1630.

E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. 2006. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 195–206.

P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. 2006a. Construction and use of linear regression models for processor performance analysis. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*. 99–108.

P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. 2006b. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 161–170.

R. A. Kamin III, G. B. Adams III, and P. K. Dubey. 1994. Dynamic trace analysis for analytic modeling of superscalar performance. *Performance Evaluation* 19, 2–3, 259–276.

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

P. Kongetira, K. Aingaran, and K. Olukotun. 2005. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro* 25, 2, 21–29.

B. Lee and D. Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 185–194.

J. Lee. 2010. A superscalar processor model for limited functional units using instruction dependencies. In *Proceedings of the ISCA 25th International Conference on Computers and Their Applications (CATA)*. 294–299.

K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. 2008. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 315–326.

G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano. 2013. Design-space exploration and runtime resource management for multicores. *ACM Transactions on Embedded Computing Systems* 13, 2, Article 20.

R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2, 78–117.

P. Michaud, A. Seznec, and S. Jourdan. 1999. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2–10.

D. B. Noonburg and J. P. Shen. 1994. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*. 52–62.

D. B. Noonburg and J. P. Shen. 1997. A framework for statistical modeling of superscalar processor performance. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture (HPCA)*. 298–309.

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

E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. 2007. Using model trees for computer architecture performance analysis of software applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 116–125.

V. J. Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. 2010. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 26–36.

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

Texas Instruments Incorporated. 2014. *AM335x Sitara Processors: Technical Reference Manual*. Texas Instruments Incorporated, Dallas, TX.

M. Van Biesbrouck, T. Sherwood, and B. Calder. 2004. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 45–56.

K. Van Craeynest and L. Eeckhout. 2011. The multi-program performance model: Debunking current practice in multi-core simulation. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Los Alamitos, CA, 26–37.

K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*. 213–224.

Y. Zhu, W. Wong, and Ş. Andrei. 2005. An integrated performance and power model for superscalar processor designs. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference (ASP-DAC)*. 948–951.

Received April 2014; revised August 2014; accepted October 2014