# A Mechanistic Performance Model for Superscalar In-Order Processors

Maximilien Breughe Stijn Eyerman Lieven Eeckhout
ELIS Department, Ghent University, Belgium

## Abstract

*Mechanistic processor performance modeling builds an analytical model from understanding the underlying mechanisms in the processor and provides fundamental insight in program-microarchitecture interactions, as well as microarchitecture structure scaling trends and interactions. Whereas prior work in mechanistic performance modeling focused on superscalar out-of-order processors, this paper presents a mechanistic performance model for superscalar in-order processors. We find mechanistic modeling for in-order processors to be more challenging compared to out-of-order processors because the latter are designed to hide latencies, and hence from a modeling perspective, detailed modeling of instruction execution latencies and dependencies is not required.*

*The proposed mechanistic performance model for superscalar in-order processors models the impact of non-unit instruction execution latencies, inter-instruction dependencies, cache/TLB misses and branch mispredictions, and achieves an average performance prediction error of 2.5% compared to detailed cycle-accurate simulation. We extensively evaluate the model's accuracy and we demonstrate its usefulness through three applications: (i) we compare in-order versus out-of-order performance, (ii) we quantify the impact of compiler optimizations on in-order performance, and (iii) we perform a power/performance design space exploration.*

## 1 Introduction

For studying processor performance, both researchers and designers rely heavily on detailed cycle-accurate simulation. Although detailed simulation provides accurate performance projections of particular design configurations, deriving fundamental insight into the interactions that take place within a processor is more complicated. Understanding trend behavior of microarchitecture structure scaling, 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. This is particularly a concern during the early stages of the design cycle when high-level design decisions need to be made. Detailed cycle-accurate simulation is too time-consuming in the early design stages and, in addition, highly accurate performance estimates are illusory anyway given the knowable level of design detail.

In this paper, we focus on mechanistic analytical performance modeling, which is a better method for gaining insight and guiding high-level design decisions. 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 instruction mix and inter-instruction dependency profiles), machine parameters (such as processor width, number of functional units, 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 which use machine learning techniques and/or statistical methods, e.g., neural networks, regression, etc., to infer a performance model [4, 13, 14, 15, 19, 26]. 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.

Whereas prior work in analytical performance modeling has focused on superscalar out-of-order processors [8, 17], in this paper we propose a mechanistic model for superscalar in-order processors. Counterintuitively perhaps, in-order processor performance is more complicated to model than out-of-order processor performance using a mechanistic model. The reason is that out-of-order processors are designed to hide instruction execution latencies and dependencies which means that these phenomena are not so important from a modeling perspective, i.e., one can assume that latencies and dependencies are largely hidden and hence do not need to be modeled. An in-order processor, on the other hand, cannot hide these phenomena, and instruction execution latencies and dependencies immediately translate in a performance impact. As a result, in-order processors require more extensive modeling.

Interval analysis, which is a mechanistic model for su-

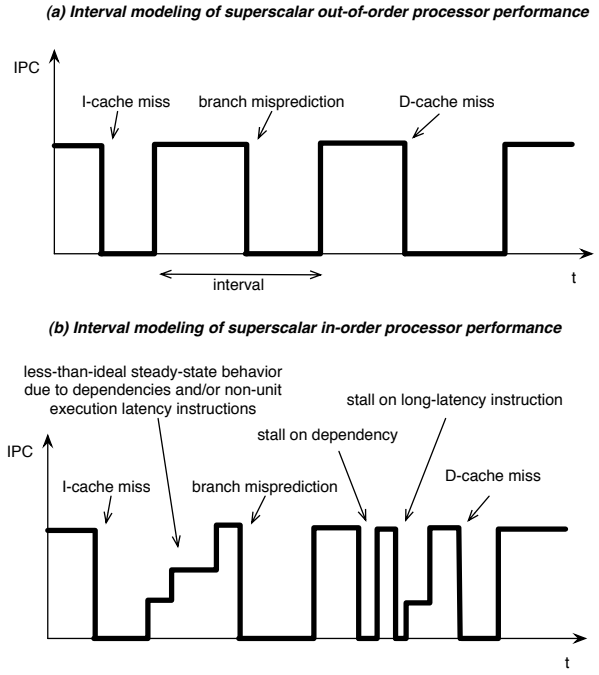


Figure 1: Interval analysis analyzes processor performance on an interval basis determined by disruptive miss events: (a) out-of-order processors and (b) in-order processors.

perscalar out-of-order processors [8, 17], was based on the observation that in the absence of miss events such as cache misses and branch mispredictions, a well-balanced superscalar out-of-order processor can smoothly stream instructions through its pipelines, buffers and functional units. Under ideal conditions the processor sustains a level of performance (instructions per cycle) roughly equal to the width of the processor. However, the smooth streaming of instructions is intermittently disrupted by miss events. The effects of these miss events divide execution time into intervals, and these intervals serve as the entity for analysis and modeling, see Figure 1(a).

The fundamental assumption made for modeling superscalar out-of-order processors, namely that the processor can smoothly stream instructions through its pipelines at roughly the designed width, does not hold true for in-order processors. Moreover, for superscalar out-of-order processors, it suffices to model a limited number of miss events only, such as long-latency cache misses (typically last-level cache misses only due to data references), instruction cache misses, TLB misses, and branch mispredictions. In-order processors on the other hand, incur a wider range of miss events and other performance hazards. Beyond the ones mentioned above, in-order processor performance also suffers from pipeline stalls due to inter-instruction dependencies, long-latency instructions, such as multiply and divide operations, and cache misses in first-level cache(s). (An out-of-order processor is designed such that these latencies and inter-instruction dependencies are mostly hidden.) As a result, inter-instruction dependencies and non-unit instruction execution latencies may introduce additional intervals and in addition may lead to a pipeline throughput that is less than the designed width in the absence of miss events, see Figure 1(b). The mechanistic model proposed in this paper models these phenomena using program statistics, such as instruction mix and inter-instruction dependency profiles, that are independent of the underlying machine. Hence, the mechanistic model requires the workload to be profiled only once, which suffices to explore a large part of the superscalar in-order processor design space.

Our experimental results using the M5 simulator and the MiBench benchmarks show the high level of accuracy achieved by the mechanistic model. We report an average absolute prediction error of 2.5%. Further, we demonstrate the usefulness of the model through three case studies. We analyze and compare in-order versus out-of-order processor performance and we pinpoint where the performance differences come from using CPI stacks. Second, we evaluate how compiler optimizations affect in-order performance and we derive interesting conclusions. Third, we use the analytical model to drive a power/performance design space exploration.

We believe 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 has important implications on cooling and total cost of ownership of server and datacenter 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 Intel Atom and ARM Cortex-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 [18] and SeaMicro's Intel Atom based server[1]; recent research projects have also studied in-order processors for internet-sector workloads [1, 21, 27].

## 2 Modeling Context

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

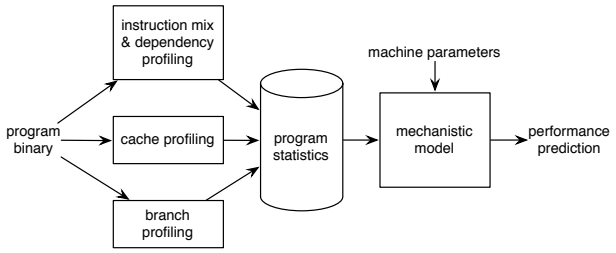


Figure 2: Overview of the mechanistic modeling framework.

### 2.1 General overview

The framework for 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 that are independent of the machine. These statistics relate to the program's instruction mix and inter-instruction dependencies, 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, i.e., statistics that are a function of both the program binary as well as the machine. Example statistics are cache and TLB miss rates, and branch misprediction rates. Although, in theory, collecting these statistics requires separate runs for each cache, TLB and branch predictor configuration of interest, in practice though, most of these statistics can be collected in a single run. In particular, single-pass cache simulation [12, 22] 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, performance can be predicted for any combination of cache hierarchy with any branch predictor and any processor core configuration.

These statistics, along with a number of machine parameters then serve as input to the analytical model, which then estimates superscalar in-order processor performance. The machine parameters include pipeline depth, pipeline width, functional unit latency (multiply, divide, etc.), cache access latencies, and memory access latencies; further, the cache/TLB and branch predictor size and configuration of interest is to be selected.

Because the analytical model basically involves computing a limited number of formulas, a performance prediction is obtained 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.

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

### 2.2 Microarchitecture description

We assume a superscalar in-order processor with five pipeline stages: fetch, decode, execute, memory and write-back. (The model can handle deeper pipelines, as we will describe later.) Fetch and decode are referred to as the front-end stages of the pipeline, whereas execute, memory and write-back are back-end stages. Each stage is $W$-wide, i.e., each stage can hold $W$ 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, i.e., the processor stalls on an instruction that consumes a value that has not been produced yet. These instructions block up the decode stage. A load that results in a cache miss blocks up the memory 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 in the execute stage or a cache miss in the memory stage) block all subsequent instructions.

## 3 Mechanistic Performance Model

### 3.1 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:

$$T = \frac{N}{W} + P_{misses} + P_{LL} + P_{deps}. \quad (1)$$

In this equation, $N$, $W$, $P_{misses}$, $P_{LL}$, $P_{deps}$ stand for the number of dynamically executed instructions, the width of the processor, the penalty due to miss events, the penalty due to non-unit, long-latency instructions (multiply and divide), and the penalty due to inter-instruction dependencies, respectively. Table 1 summarizes the inputs to the model. How we estimate penalty cycles due to misses, long-latency instructions and dependencies from these model inputs will be explained in the following sections.

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, i.e., it takes at least $N/W$ cycles to execute $N$ instructions on a $W$-wide processor. Miss events, non-unit long-latency instructions and inter-instruction dependencies prevent the processor from executing instructions at a rate of $W$ instructions per cycles, which is accounted for by the model by adding penalty cycles. We make a distinction between penalty cycles due to miss events, non-unit latency instructions and inter-instruction dependencies. Before describing how we account for each of these penalties we first explain the general principle employed for penalty accounting.

| Program characteristics | |
|---|---|
| $N$ | no. dynamically executed instructions |
| $N_i$ | no. dynamically executed instructions of type $i$ with $i$ the different types of non-unit, long-latency instructions |
| $misses_i$ | no. misses of type $i$ with $i$ the different types of miss events (L1/L2 cache/TLB misses, branch mispredictions) |
| $deps_{unit}(d)$ | no. instructions dependent on unit-latency instruction at dependency distance $d$ |
| $deps_{LL}(d)$ | no. instructions dependent on long-latency instruction (excl. loads) at dependency distance $d$ |
| $deps_{ld}(d)$ | no. instructions dependent on loads at dependency distance $d$ |
| **Machine characteristics** | |
| $W$ | processor width |
| $D$ | depth of the front-end pipeline |
| $latency_{LL,i}$ | execution latency for non-unit long-latency instruction of type $i$ with $i$ multiply, divide, L1, L2 hit, etc. |
| $MissLatency_i$ | cache miss latency of type $i$ with $i$ L1, L2, etc. |

Table 1: Mechanistic model inputs.

### 3.2 Penalty cycle accounting

To compute the number of penalty cycles, we examine the number of instructions that enter the execute stage in the in-order pipeline. If this number equals $W$, then we are in the ideal case, and no penalty cycles are accounted for. If this number is less than $W$, we differ from the ideal case, and penalty cycles should be accounted to the event that caused the reduction in the number of instructions executed.

If no instructions enter the execute stage (e.g., an instruction cache miss ceases the flow of instructions into the pipeline for a number of cycles), the cycle is accounted as a penalty cycle. However, it can be the case that some instructions in the decode stage are able to execute, but the other ones cannot be executed due to a dependency hazard. In this case $x$ instructions are shifted to the execution stage, with $0 < x < W$. This cycle is not a full-penalty cycle, because some instructions execute; on the other hand, this cycle is not ideal either, because less than $W$ instructions are executed. To handle this situation, we use the notion of an *instruction slot*, and we convert instruction slots back to penalty cycles. Each stage has $W$ instruction slots, and each instruction slot represents $1/W$ of the width of that stage. If $W$ instructions execute, $W$ instruction slots are fully occupied and hence we have one ($W \times 1/W = 1$) useful cycle and no penalty cycle. If no instructions executed, we get $0 \times 1/W = 0$ useful cycles and 1 penalty cycle. If $x$ instructions execute, $x$ instruction slots are occupied, hence the fraction of useful cycles equals $x/W$, and thus the number of penalty cycles equals $(W - x)/W$. This means that the fraction of penalty in one cycle is a real number between zero and one (in multiples of $1/W$), reflecting the fraction of unused instruction slots.

In the next few sections, we discuss the different events that incur a penalty (miss events, long-latency instructions and inter-instruction dependencies), and we explain how we compute their respective penalties.

### 3.3 Penalty due to miss events

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

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

This formula computes the weighted sum over the miss events with their respective penalties.

We make a distinction between cache (and TLB) misses versus branch mispredictions when it comes to computing the penalties. We treat instruction and data cache misses (and TLB misses) the same way, since they both block off the execution stage while the miss is being handled by the memory subsystem. In particular, when an instruction cache miss occurs, the instructions in the front-end pipeline can still enter the execution stage, but when the instruction cache miss is resolved, it takes some time for the new instructions to re-fill the front-end pipeline. It is easy to understand that front-end pipeline drain and re-fill offset each other, i.e., the penalty for an instruction cache miss is independent of the front-end pipeline depth. In case of a data cache miss, the memory stage blocks, and no instructions can leave or enter the execution stage while the data cache miss is resolved.

From the above discussion, it follows that the penalty for a cache miss is proportional to its miss latency (i.e., the access time to the next level of cache or main memory). However, when a cache miss occurs, it might be the case that some instructions can still be executed. For example, for an instruction cache miss and a processor with width $W = 4$, it may happen that one, two or three instructions were already fetched before the instruction cache miss occurred. These instructions can execute underneath the cache miss, and are therefore hidden. Assuming that cache misses occur uniformly distributed across a $W$-wide instruction group, the average number of instructions hidden underneath a cache miss equals $\frac{W-1}{2}$. The cache miss penalty should therefore be reduced by $\frac{W-1}{2W}$ cycles. The total penalty for a cache or TLB miss thus equals

$$penalty_{cacheMiss} = MissLatency - \frac{W-1}{2W}. \quad (3)$$

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

$$penalty_{branchMiss} = D + \frac{W-1}{2W}, \tag{4}$$

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 front-end pipeline stages, namely $D$. The second term is the penalty of flushing instructions in the execute 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 a pipeline in which a branch is predicted one cycle after it was fetched, and if it is predicted taken, the instruction(s) in the fetch stage (which were fetched assuming a non-taken branch) need to be flushed. This incurs one penalty cycle per branch that is predicted taken, even if it is correctly predicted. We will refer to this penalty as the taken-branch hit penalty.

### 3.4 Penalty due to long-latency instructions

The penalty due to non-unit, long-latency instructions is computed as follows:

$$P_{LL} = \sum_{i \in \{LLType\}} N_i \times penalty_i. \tag{5}$$

The penalty is computed as a weighted sum over the number of non-unit instructions in the dynamic instruction stream, with the weights being their respective penalty. Non-unit, long-latency instructions include multiply and divide operations that take more than one cycle to execute on a functional unit, as well as L1 cache hits (if the L1 access time takes more than one cycle) and L2 cache hits due to loads.

Because in-order processors execute instructions in program order and because we assume in-order commit to guarantee precise interrupts, a long-latency instruction causes all newer instructions to stall until the long-latency instruction is executed. The penalty of a long-latency instruction thus equals

$$penalty_{LL} = (latency_{LL} - 1) - \frac{W-1}{2W}. \tag{6}$$

The first term subtracts one from the instruction execution latency because one cycle was accounted for already as part of the minimum cycle count $N/W$, see formula 1. The second term accounts for overlap effects of older instructions prior to the long-latency instruction in the execution stage.

### 3.5 Penalty due to dependencies

For computing the penalty due to inter-instruction dependencies we consider three groups of dependencies: dependencies on unit-latency producers, dependencies on non-unit latency instructions (excluding loads), and dependencies on load instructions:

$$P_{deps} = P_{dep\ unit} + P_{dep\ LL} + P_{dep\ ld} \tag{7}$$

#### 3.5.1 Dependencies on unit-latency instructions

Unit-latency instructions that depend upon each other and that are in different stages during the execution, do not incur a performance penalty; the result produced by the unit-latency instruction can be communicated to its consumer through the register file or through forwarding logic. A dependency on a unit-latency instruction only incurs a penalty if both instructions reside in the same stage. We compute the penalty due to inter-instruction dependencies on unit-latency instructions using the following formula:

$$\begin{aligned} P_{dep\ unit} = & \sum_{d=1}^{W-1} deps_{unit}(d) \\ & \times Prob[in\ same\ stage](d) \\ & \times penalty(d) \end{aligned} \tag{8}$$

In this equation, $d$ is defined as the dependency distance between the consumer and the producer, and is counted as the number of dynamically executed instructions between the producer and its consumer; subsequent instructions in the dynamic instruction stream that depend upon each other have a dependency distance $d = 1$. The dependency distance $d$ accounts for the shortest dependency distance if a consumer instruction has two producers. This formula sums over $d$ from 1 to $W-1$, and consists of three parts. The first part $deps_{unit}(d)$ is the number of dependent instructions on unit-latency instructions at distance $d$ (see Table 1). The second part estimates the probability for a producer and a consumer to be in the same stage — again, we assume a uniform distribution:

$$Prob[in\ same\ stage](d) = \frac{W-d}{W}. \tag{9}$$

The third part quantifies the penalty in case the consumer and producer are in the same stage and is computed based on the number of lost instruction slots of the consumer and all instructions beyond the consumer. The distance between the producer and the consumer is defined as $d$, hence there are $d$ older instructions than the consumer in the pipeline stage. This means there are $W-d$ instructions that cannot execute due to the dependency. The penalty therefore equals

$$penalty(d) = \frac{W-d}{W} \tag{10}$$

Substituting formulae 9 and 10 into formula 8 yields:

$$P_{dep\ unit} = \sum_{d=1}^{W-1} deps_{unit}(d) \cdot \left(\frac{W-d}{W}\right)^2. \quad (11)$$

#### 3.5.2 Dependencies on long-latency instructions

Long-latency instructions, such as multiply and divide operations — we treat load instructions separately as we will discuss in the next section — block the execute stage for more than one cycle, which means that they will always be the oldest instruction in the execution stage by the end of their execution. Therefore, an instruction that depends on this long-latency instruction and whose dependency distance is less than $W$ will eventually wait in the decode stage. This instruction and the newer instructions cannot proceed to the next stage in the next cycle, and will thus incur a penalty. The number of lost instruction slots is again $W-d$, as in the previous section. The total penalty for instructions that depend on long-latency instructions thus equals

$$P_{dep\ LL} = \sum_{d=1}^{W-1} deps_{LL}(d) \times \frac{W-d}{W}, \quad (12)$$

with $deps_{LL}(d)$ the number of instructions that depend on a long-latency instruction at dependency distance $d$.

#### 3.5.3 Dependencies on load instructions

Dependencies on load instructions are different because loads produce their result in the memory stage, not in the execute stage. This implies that instructions that depend on load instructions not only cause a penalty when they are together in the decode stage, but also when the dependent instruction is in the decode stage and the load is in the execute stage. The maximum dependency distance where there can be a penalty is therefore $2W-1$ instead of $W-1$ as in the previous sections. We therefore make a distinction between two cases: (i) the load and its consumer are in the same stage; and (ii) the load and its consumer are in subsequent stages.

Consider first the case that both the load and its consumer reside in the decode stage. In the next cycle, only the load and its independent instructions will shift to the execution stage. This incurs a penalty of $\frac{W-d}{W}$. The following cycle, the load proceeds to the memory stage and no instructions enter the execution stage. This cycle is therefore a full penalty cycle. If the load hits in the cache (and the cache hit latency is one cycle), the dependent instruction enters the execution stage in the next cycle. If it is a miss (or multiple cycle hit), all instructions have to wait during the miss (or hit) latency; this penalty is already accounted for as a cache miss (or long-latency instruction), see Section 3.3. In summary, the penalty in this case is

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

This case occurs when the load and its consumer are together in the decode stage. The probability for this case to happen equals $(W-d)/W$, as defined in formula 9.

Now consider the second case in which the consumer of the load is in the decode stage while the load is in the execute stage. In the next cycle, the load moves to the memory stage. The number of instructions that can enter the execute stage depends on the dependency distance $d$. If $d < W$, then all independent instructions have shifted with the load to the memory stage, and no instructions can enter the execute stage (the consumer is now the oldest instruction in the decode stage), which means we have a full penalty cycle. If $W \leq d < 2W$, there are some instructions in the decode stage that are older than the consumer, and they can shift to the execute stage. This number of instructions equals $d-W$, so the penalty in this case equals $\frac{W-(d-W)}{W} = \frac{2W-d}{W}$. In summary, the penalty equals in this case

$$penalty = \begin{array}{cll} 1 & if & d < W \\ \frac{2W-d}{W} & if & W \leq d < 2W \end{array} \quad (14)$$

The probability for this case to happen is defined by the probability that the load and its first dependent instruction are in two consecutive stages; it is computed as follows:

$$Prob = \begin{array}{cll} \frac{d}{W} & if & d < W \\ \frac{2W-d}{W} & if & W \leq d < 2W \end{array} \quad (15)$$

Putting it all together, the penalty due to inter-instruction dependencies on loads equals

$$P_{dep\ ld} = \sum_{d=1}^{W-1} deps_{ld}(d) \left(\frac{W-d}{W}\frac{2W-d}{W} + \frac{d}{W}\right) + \sum_{d=W}^{2W-1} deps_{ld}(d) \left(\frac{2W-d}{W}\right)^2. \quad (16)$$

## 4 Experimental Setup

We use 19 benchmarks from the MiBench benchmark suite [10]. MiBench is a suite of embedded benchmarks from different application domains, including automotive/industrial, consumer, office, network, security, and telecom. We limit ourselves to 19 benchmarks in total in order to limit simulation time during performance model validation while covering the above application domains.We consider the large input for all of these benchmarks.

We use the M5 simulation framework [2]. We derive our profiler from M5's functional simulator, and we validate our model against detailed cycle-accurate simulation with M5. We use McPAT [20] for our power estimates. The inputs for McPAT are various processor configuration parameters, such as pipeline depth, width, cache configuration, memory latency, chip technology (32nm), etc., along

| *Parameter* | *Default* | *Range* |
|---|---|---|
| I-cache | 32KB<br>4 way set-assoc<br>64 byte blocks | 32KB<br>4 way set-assoc<br>64 byte blocks |
| D-cache | 32 KB<br>4 way set-assoc<br>64 byte blocks | 32KB<br>4 way set-assoc<br>64 byte blocks |
| L2-cache | 512KB<br><br>8 way set-assoc<br>10ns latency | 128KB – 256KB –<br>512KB – 1MB<br>8 vs 16 way set-assoc<br>10ns latency |
| pipeline depth | 9 stages<br>1GHz | 5 – 7 – 9 stages<br>600MHz – 800MHz – 1GHz |
| processor width | 4 slots | 1 – 2 – 3 – 4 slots |
| branch predictor | 1KB global history | 1KB global history –<br>3.5KB hybrid 10b local and<br>12b global history |

Table 2: Architecture design space explored along with the default processor configuration being simulated.

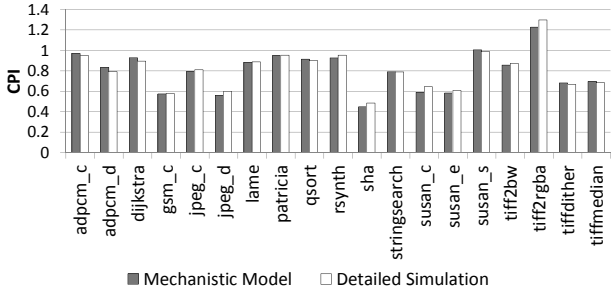


Figure 3: Validation: CPI as predicted by the model against CPI through detailed simulation for MiBench.

with program parameters, such as number of instructions, instruction mix, etc., and finally, program-machine parameters, such as cache misses, branch mispredictions, etc.

The default processor configuration is shown in Table 2: a superscalar in-order processor with private 32KB L1 caches and a unified L2 cache. Further, we also consider a design space in which we vary a number of important microarchitecture parameters, such as pipeline depth and frequency setting (3 configurations), pipeline width (4 configurations), L2 cache size and associativity (8 configurations), as well as branch predictor configuration (2 configurations). This leads to a design space consisting of 192 design points within which we will be evaluating the model's accuracy. Although this is not a very large space compared to real design spaces, it was close to the limit we could explore given our infrastructure because we compare model accuracy against detailed simulation results which are very time-consuming and costly to obtain — which is the motivation for this research in the first place.

## 5 Evaluation

The evaluation of the model is done in a number of steps.

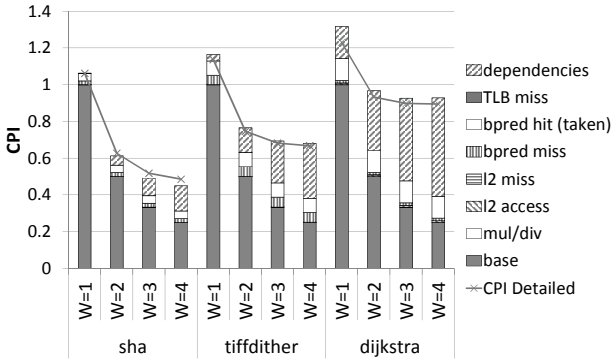


Figure 4: Model accuracy for estimating relative performance as a function of superscalar width.

**Default processor.** We first evaluate the model's accuracy for the default processor configuration, see Figure 3. We show predicted CPI by the model against CPI obtained through detailed simulation. The average CPI prediction error equals 3.1% while the highest error is 8.4%. The error mostly comes from second-order effects. The model proposed in this paper essentially is a first-order model and does not model overlap effects. For example, an I-cache miss may be (partially) overlapped by the execution of a long-latency instruction (e.g., multiply), yet the model would account for both penalties. Also, the model does not account for delayed update effects in the branch predictor. We do not account for second-order effects deliberately in order not to complicate the model too much. Moreover, our results show that a first-order model is accurate enough.

**Varying superscalar width.** Figure 4 shows CPI stacks as obtained through the model, as a function of superscalar width. The overall CPI obtained through detailed simulation is also shown as a reference. The three benchmarks were picked based on how they scale with processor width. The sha benchmark benefits the most from superscalar processing, whereas dijkstra benefits the least; tiffdither is somewhere in the middle. This graph in fact demonstrates the amount of insight that mechanistic modeling offers beyond detailed simulation, because it breaks up overall performance into its contributing factors. Clearly, although there is a benefit from going from scalar processing to 2-wide processing for dijkstra, going beyond 2-wide processing does not improve performance much. The reason why is immediately clear from the CPI stacks: although the base component ($N/W$) decreases, its decrease is compensated by more inter-instruction dependencies, which impede the benchmark from benefiting from superscalar processing. This is not the case for sha which seems to suffer less from dependencies; apparently, this benchmark exhibits more ILP.

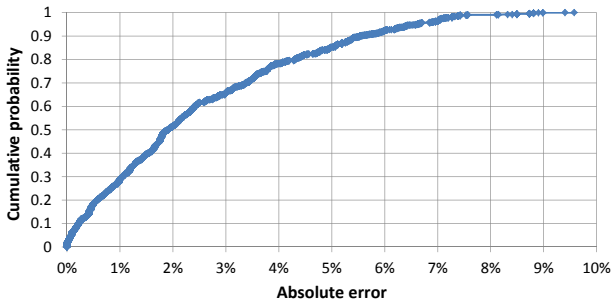


Figure 5: Cumulative distribution of model accuracy across the design space.

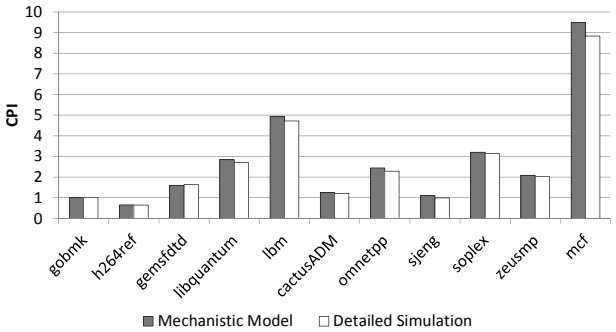


Figure 6: Validation: CPI as predicted by the model against CPI through detailed simulation for SPEC CPU 2006.

**Design space exploration.** Figure 5 reports the cumulative distribution of the model prediction error across the broader design space. Again, this graph confirms our earlier finding, namely the model is accurate compared to detailed simulation: for 90% of the design points, we achieve a prediction error below 6%. The average and maximum prediction errors observed equal 2.5% and 9.6%, respectively. Simulating the entire design space through detailed simulation takes 290 days on a single computer. Exploring the design space using the proposed mechanistic model on the other hand takes only 4.5 hours — a speedup of 3 orders of magnitude. Note that profiling accounts for the largest fraction of the 4.5 hours, and calculating the model only takes a few seconds.

**SPEC CPU2006.** For completeness and to increase our confidence in the modeling, we also evaluated the model's accuracy using a number of SPEC CPU2006 benchmarks which are more memory-intensive than the MiBench applications considered so far. We observe an average error of 4.1% and a maximum error of 10.7%, see Figure 6.

# 6 Applications

Having evaluated the model's accuracy, we now consider three case studies to illustrate the use of the model: (i) we compare and analyze in-order versus out-of-order performance; (ii) we evaluate how compiler optimizations affect in-order performance; and (iii) we use the model to drive a power/performance design space exploration.

## 6.1 In-order versus out-of-order performance

In our first application, we compare in-order versus out-of-order performance using CPI stacks, see Figure 7 [2]; We only show CPI stacks for a selected number of benchmarks to improve readability. The in-order CPI stacks are obtained using the model described in this paper; the out-of-order CPI stacks are obtained using the model described in prior work [8]. In this comparison, we consider four-wide in-order and out-of-order processors. A number of fundamental and insightful observations can be made from this graph.

- Dependencies are largely hidden by out-of-order execution, in contrast to in-order processing. This is apparent for all the benchmarks.
- Non-unit instruction execution latencies due to multiply/divide operations have significant impact on performance on in-order processors for some benchmarks, the most notable example being tiff2bw. Non-unit latencies are mostly hidden by out-of-order execution.
- The cost per mispredicted branch is larger on out-of-order processors than on in-order processors, see for example patricia. The reason is that on an in-order processor, the cost equals the depth of the front-end pipeline, whereas on an out-of-order processor the branch resolution time (the time between the branch being dispatched from the front-end pipeline into the back-end) also contributes to the overall penalty in addition to the front-end pipeline.
- The L2 cache component is smaller on the out-of-order processor compared to the in-order processor, see for example tiff2rgba. The reason is that an out-of-order processor can better exploit memory-level parallelism and issue independent loads and stores to memory simultaneously. An in-order processor on the other hand would stall on the first use of a load miss, preventing subsequent (independent) load misses to go to memory.
- Since I-cache miss penalty is a function of the miss latency only, the penalty is identical on in-order and out-of-order processors.

[2]The CPI stacks were obtained by reimplementing the model in the SimpleScalar toolset. The use of SimpleScalar, along with the use of another cross compiler, is the reason for the slightly different results for the in-order model compared with the results in the other sections. SimpleScalar expects COFF binaries, unlike M5 which reads the ELF format.

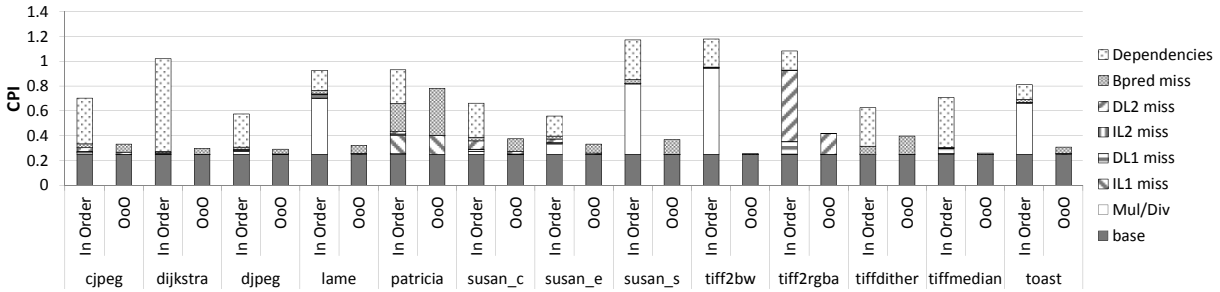


Figure 7: Comparing in-order versus out-of-order performance using CPI stacks obtained through mechanistic modeling.

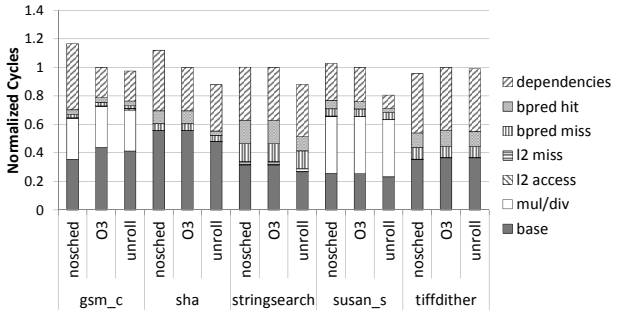


Figure 8: Normalized cycle stacks for five benchmarks across different compiler optimizations.

This case study clearly illustrates the insight that can be obtained from mechanistic analytical modeling, which is much harder to obtain through detailed cycle-accurate simulation.

## 6.2 Compiler optimizations

As a second application we use the model to study how compiler optimizations affect superscalar in-order performance, see Figure 8. We consider `-O3`, `-O3` without instruction scheduling ( `-O3 -fno-schedule-insns`), and `-O3` with loop unrolling turned on (`-O3 -funroll-loops`) for the five benchmarks for which we observed the largest impact due to compiler optimizations. Figure 8 shows normalized cycle stacks, i.e., 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 dependencies. For some benchmarks, e.g., gsm_c, the base component increases slightly through instruction scheduling, the reason being the addition of spill code. However, the cost of spill code is compensated for by the substantial decrease in the impact of inter-instruction dependencies. For one benchmark, tiffdither, the cost of spill code is not compensated for, and the penalty due to inter-instruction dependencies is even worse.

Most of the benchmarks (and all the ones in Figure 8) benefit from loop unrolling. Three components get an important reduction through loop unrolling. First, the number of dynamic instructions decreases because fewer branches and counter increments are needed after loop unrolling. Second, because there are fewer branches, the penalty due to taken branches also decreases. The third and biggest contribution comes from the smaller penalty due to inter-instruction dependencies; clearly, loop unrolling enables the instruction scheduler to better schedule instructions so that fewer inter-instruction dependencies have an impact on in-order performance.

## 6.3 Design space exploration

Processor designers take various metrics into account during the development process. Energy consumption clearly is a key metric when designing embedded processors. We now explore the design space mentioned in Table 2 while considering both performance and energy consumption. We therefore consider energy-delay product (EDP) which is defined as the product of execution time and energy consumption.

We compare EDP obtained through detailed simulation against the mechanistic model; we use McPAT, as mentioned before, for the power modeling. We show EDP graphs for four benchmarks in Figure 9. We find that the model reaches the same optimum processor configuration for 12 of the 19 benchmarks as with detailed simulation; for 6 other benchmarks the model suggests a processor configuration where the difference in EDP compared to the optimal configuration is less than 0.5%. For one benchmark (adpcm_d), the model suggests a configuration that has an EDP difference of less than 5% compared to the optimal configuration, see also Figure 9(a): detailed simulation identifies the optimum width to be 3, whereas our model predicts it to be 2.

# 7 Related work

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

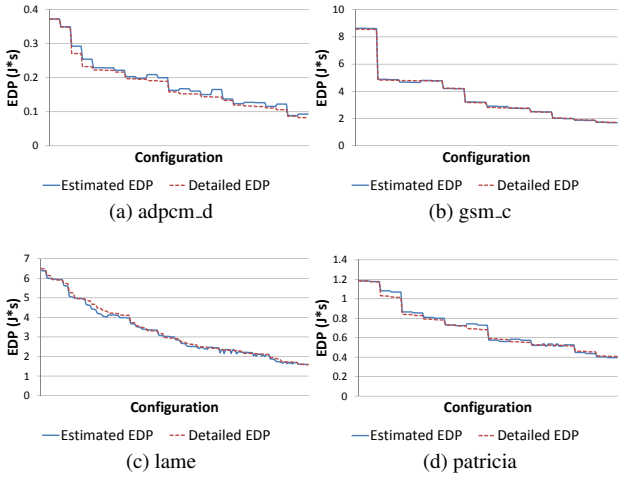


Figure 9: Design space exploration of four benchmarks. Configurations are ordered from high to low EDP values.

related to this paper.

### 7.1 Analytical modeling

There are basically 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. [23] build a mechanistic model of the instruction window and issue mechanism. Karkhanis and Smith [17] extend this simple mechanistic model to build a complete performance model that assumes sustained steady-state issue performance punctuated by miss events. Taha and Wills [28] propose a mechanistic model that breaks up the execution into so-called macro blocks, separated by miss events. Eyerman et al. [8] 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. [3] proposed a mechanistic model for *scalar* in-order processors. This paper presents a mechanistic model for superscalar in-order processors which involves substantial modeling enhancements with respect to long-latency instructions and inter-instruction dependencies, as explained in Section 2.

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 [14]; non-linear regression for performance prediction [15]; spline-based regression for power and performance prediction [19]; neural networks [4, 13]; or model trees [26].

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 [11] 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. [9] propose a more complete mechanistic-empirical model which enables constructing CPI stacks on real out-of-order processors.

### 7.2 Inter-instruction dependency modeling

Dubey et al. [5] present an analytical model for the amount of instruction-level parallelism (ILP) for a given window size of instructions based on the inter-instruction dependency distribution. Kamin et al. [16] approximate the inter-instruction dependency distribution using an exponential distribution. Later, Eeckhout et al. [6] found a power law to be a more accurate approximation.

The inter-instruction dependency distribution is an important program statistic for statistical modeling. Noonburg and Shen [24] 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 [7, 25] generates a synthetic program or trace from a set of statistics.

## 8 Conclusion

Mechanistic analytical modeling of superscalar in-order processor performance is more complicated than for out-of-order processors. The fundamental reason is that out-of-order processors are designed to hide the performance impact of inter-instruction dependencies and non-unit instruction execution latencies, hence these effects do not need consideration in mechanistic modeling. In this paper, we proposed a mechanistic analytical performance model for superscalar in-order processors, that models the impact of non-unit instruction execution latencies, inter-instruction dependencies, cache/TLB misses, and branch mispredictions. The input parameters to the model are a set of program and mixed program/machine statistics, along with a set of machine parameters. Profiling needs to be done only once for a given benchmark to collect the program-specific statistics, and enables predicting performance for a broad range of microarchitectures.

Our experimental results demonstrate the accuracy and speedup of the proposed model: we achieve an average prediction error of 2.5% and three orders of magnitude speedup compared against detailed cycle-accurate simulation. Considering a superscalar microarchitecture design space in which we vary pipeline depth, width, cache size and branch predictor configuration, we demonstrate the model's relative accuracy: for more than 90% of the design points, we achieve a prediction error below 6%. Further, we illustrate the usefulness of the model for providing insight and analyzing in-order versus out-of-order processor performance, the impact of compiler optimizations and for exploring power/performance design spaces.

## Acknowledgements

We thank the reviewers for their constructive and insightful feedback. Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation-Flanders (FWO). Additional support is provided by the FWO projects G.0255.08 and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, the ICT Department of Ghent University, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

## References

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, pages 1–14, Oct. 2009.

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

[3] M. Breughe, Z. Li, Y. Chen, S. Eyerman, O. Temam, C. Wu, and L. Eeckhout. How sensitive is processor customization to the workload's input datasets? In *SASP*, pages 1–7, June 2011.

[4] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. Microarchitecture design space exploration using an architecture-centric approach. In *MICRO*, pages 262–271, Dec. 2007.

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

[6] L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *PACT*, pages 25–34, Sept. 2001.

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

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

[9] S. Eyerman, K. Hoste, and L. Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *ISPASS*, pages 216–226, Apr. 2011.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, Dec. 2001.

[11] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *ISCA*, pages 7–13, May 2002.

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

[13] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, pages 195–206, Oct. 2006.

[14] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *HPCA*, pages 99–108, Feb. 2006.

[15] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO*, pages 161–170, Dec. 2006.

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

[17] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, pages 338–349, June 2004.

[18] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[19] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS*, pages 185–194, Oct. 2006.

[20] 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 *MICRO*, pages 469–480, Dec. 2009.

[21] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ISCA*, pages 315–326, June 2008.

[22] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, June 1970.

[23] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *PACT*, pages 2–10, Oct. 1999.

[24] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *HPCA*, pages 298–309, Feb. 1997.

[25] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor design. In *ISCA*, pages 71–82, June 2000.

[26] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *ISPASS*, pages 116–125, Apr. 2007.

[27] V. J. Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *ISCA*, pages 26–36, June 2010.

[28] T. M. Taha and D. S. Wills. An instruction throughput model of superscalar processors. *IEEE Transactions on Computers*, 57(3):389–403, Mar. 2008.