Mechanistic-Empirical Processor Performance Modeling for Constructing CPI Stacks on Real Hardware

Stijn Eyerman Kenneth Hoste Lieven Eeckhout
ELIS Department, Ghent University, Belgium

Abstract

Analytical processor performance modeling has received increased interest over the past few years. There are basically two approaches to constructing an analytical model: mechanistic modeling and empirical modeling. Mechanistic modeling builds up an analytical model starting from a basic understanding of the underlying system — white-box approach — whereas empirical modeling constructs an analytical model through statistical inference and machine learning from training data, e.g., regression modeling or neural networks — black-box approach. While an empirical model is typically easier to construct, it provides less insight than a mechanistic model.

This paper bridges the gap between mechanistic and empirical modeling through hybrid mechanistic-empirical modeling (gray-box modeling). Starting from a generic, parameterized performance model that is inspired by mechanistic modeling, regression modeling infers the unknown parameters, alike empirical modeling. Mechanistic-empirical models combine the best of both worlds: they provide insight (like mechanistic models) while being easy to construct (like empirical models).

We build mechanistic-empirical performance models for three commercial processor cores, the Intel Pentium 4, Core 2 and Core i7, using SPEC CPU2000 and CPU2006, and report average prediction errors between 9% and 13%. In addition, we demonstrate that the mechanistic-empirical model is more robust and less subject to overfitting than purely empirical models. A key feature of the proposed mechanistic-empirical model is that it enables constructing CPI stacks on real hardware, which provide insight in commercial processor performance and which offer opportunities for software and hardware optimization and analysis.

1 Introduction

Analytical performance modeling of contemporary computer architectures has received increased interest over the past few years. There are two major approaches to analytical performance modeling, namely mechanistic modeling and empirical modeling. Mechanistic modeling builds a performance model based on first principles, i.e., the performance model is built in a bottom-up fashion starting from a basic understanding of the mechanics of the underlying system [4, 10, 11, 16, 23]. Mechanistic modeling can be viewed as white-box performance modeling and its key feature is to provide fundamental insight, and potentially generate new knowledge. Empirical modeling on the other hand builds a performance model through a black-box approach [7, 8, 9, 12, 13, 14, 15, 18, 19, 24]. Empirical modeling typically leverages statistical inference and machine learning techniques such as regression modeling or neural networks to automatically learn a performance model from training data obtained through simulation. While inferring a performance model is easier through empirical modeling because of the complexity of the underlying system, it provides less insight than mechanistic modeling.

This paper bridges the gap between mechanistic and empirical performance modeling through hybrid mechanistic-empirical modeling or gray-box modeling. Mechanistic-empirical modeling starts from a generic performance formula that is derived from mechanistic performance modeling. This generic formula contains a number of unknown parameters that are subsequently inferred for a particular processor and set of workloads through regression, alike empirical modeling. The key benefit of mechanistic-empirical modeling is that it provides insight (which it inherits from mechanistic modeling) while easing the construction of the performance model (which it inherits from empirical modeling). In particular, mechanistic-empirical modeling allows for constructing CPI stacks on real hardware. Whereas prior work by Eyerman et al. [3] proposed a hardware performance counter architecture for computing CPI stacks on superscalar out-of-order architectures (which would only be available to a user if implemented in a future processor), this paper proposes a method for constructing CPI stacks on existing hardware.

In this paper, we build performance models for three Intel processors from three successive generations (Pen-

tium 4, Core 2 and Core i7) using the SPEC CPU2000 and CPU2006 benchmark suites, and we report average performance prediction errors around 9% to 13%. We demonstrate that mechanistic-empirical modeling is less subject to overfitting than empirical modeling, which makes mechanistic-empirical models more robust. The key feature of mechanistic-empirical performance modeling is its ability to construct CPI (Cycles Per Instruction) stacks on real hardware. Having CPI stacks for the three Intel processors enables a comparative study to gain insight into where the improvements come from across the three generations of Intel processors.

2 Prior work

Before describing our approach to mechanistic-empirical modeling, we first describe prior work in analytical performance modeling. Although we make a distinction between empirical and mechanistic modeling, there is no purely empirical or mechanistic model [17]. A mechanistic model always includes a form of empiricism, for example in the modeling assumptions and approximations. Likewise, an empirical model always includes a mechanistic component, for example in the list of inputs to the model — the list of model inputs is constructed based on some understanding of the underlying system. As a result, the distinction between empirical and mechanistic models is relative, and we base our classification on the predominance of the empirical versus mechanistic component in the model.

2.1 Empirical modeling

Empirical modeling requires little or no prior knowledge about the system being modeled, and seems to be the most widely used modeling technique today. Joseph et al. [8] build linear regression models from simulation data that relate micro-architectural parameters and their mutual interactions to overall processor performance. In their follow-on work, Joseph et al. [9] explore non-linear regression modeling. Similarly, Lee and Brooks [12] propose regression models for both performance and power using splines, and leverage spline-based regression modeling to build multiprocessor performance models [15] and explore the huge design space of adaptive processors [13]. Likewise, Ipek et al. [7] and Dubach et al. [2] build performance models using artificial neural networks. Lee et al. [14] compare spline-based regression modeling against artificial neural networks and conclude that both approaches are equally accurate; regression modeling provides better statistical understanding while neural networks offer greater automation. Vaswani et al. [24] incorporate the interaction between the compiler and the architecture, and build empirical application-specific performance models that capture the effect of compiler optimization flags and micro-architecture parameters. Ould-Ahmed-Vall et al. [18, 19] build tree-based empirical models: the model chooses a path at each node in the tree and finds a linear regression model in the leaves.

2.2 Mechanistic modeling

Michaud et al. [16] build a mechanistic model of the instruction window and issue mechanism in out-of-order processors for gaining insight in the impact of instruction fetch bandwidth on overall performance. Karkhanis and Smith [10, 11] 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 [23] propose a mechanistic model that breaks up the execution into so called macro blocks, separated by miss events. Eyerman et al. [4] propose mechanistic interval modeling, similar to the Karkhanis/Smith and Taha/Wills approaches, but focus on dispatch rather than issue which makes the model more elegant. In addition, the interval model includes an ILP model which eliminates the need for extensive micro-architecture simulations during model construction.

2.3 Hybrid mechanistic-empirical modeling

Hartstein and Puzak [6] propose a hybrid mechanistic-empirical model for studying optimum pipeline depth. The model is parameterized with a number of parameters that are fit through detailed, cycle-accurate micro-architecture simulations. The Hartstein/Puzak model is tied to modeling pipeline depth and is not as generally applicable as the model developed in this paper. In addition, the Hartstein/Puzak model cannot break up the total execution time into CPI components, in contrast to the model presented in this paper.

3 Mechanistic-empirical performance model

The mechanistic-empirical model estimates the total execution time or the number of clock cycles C as follows:

$$C = \frac{N}{D} + m_{L1\,I\$} \cdot c_{L2} + m_{L2\,I\$} \cdot c_{mem} + m_{ITLB} \cdot c_{TLB} + m_{br} \cdot (c_{br} + c_{fe}) + m_{L2\,D\$} \cdot \frac{c_{mem}}{MLP} + m_{DTLB} \cdot \frac{c_{TLB}}{MLP} + c_{stall}. \quad (1)$$

This parameterized model is a derivation from the mechanistic models presented in [4, 10, 11, 23]. (We will discuss specific differences later.) The first term is the base cycle component and equals the minimum number of cycles needed to dispatch[1] N useful micro-operations on a

[1] We refer to dispatch when instructions leave the front-end pipeline and enter the reorder buffer and issue queue(s).

processor of width D. (A CISC processor breaks up CISC instructions into RISC-like micro-operations; for a RISC processor, N would be the number of instructions.) The next three terms represent miss event cycle components due to I-cache misses and I-TLB misses. The L1 I-cache miss term equals the number of L1 I-cache misses ($m_{L1\ I\$}$) times the access time for the L2 cache (c_{L2}). We define the L2 I-cache miss term and the I-TLB miss term similarly. The branch misprediction term is the number of branch mispredictions (m_{br}) multiplied by the penalty per branch misprediction, which equals the branch resolution time (c_{br}) plus the front-end pipeline depth (c_{fe}) [5]. The long-latency load term multiplies the number of last-level cache load misses ($m_{L2\ D\$}$) with the average penalty per memory access; the average memory access latency is the average memory access latency for an individual memory access divided by a memory-level parallelism (MLP) correction factor (i.e., the number of simultaneously outstanding memory accesses if at least one is outstanding). The D-TLB miss term is defined similarly. The last term represents the number of cycles lost due to resource stalls as a result of dispatch stalling on a full reorder buffer or issue queue (due to long chains of dependent instructions).

As mentioned before, the mechanistic-empirical model shows some similarity with the mechanistic models described in [4, 10, 11, 23]. The key difference though is in how we compute the model parameters. In the previously proposed mechanistic models, the model parameters are approximated based on assumptions or require extensive application profiling and/or micro-architecture simulation. Examples are to assume that memory access time is constant (see [4, 10, 11, 23]); or to assume that the processor is balanced and ignore the resource stall component (see [4, 11]). Also, the model may involve extensive simulations to estimate the resource stall component (see [10, 23]); or may involve profiling the application to derive an estimate for the branch resolution time and the number of non-overlapping long-latency load misses for computing the amount of MLP (see [4, 11]). These assumptions and the need for profiling make mechanistic modeling a viable approach for design-stage performance predictions, however, it cannot be readily used for building performance models for real processors, i.e., the assumptions may not hold true, and profiling and simulation tools are required that are validated against real hardware. This is also our primary motivation for the proposed mechanistic-empirical model.

A key feature of mechanistic-empirical modeling is that it allows for constructing a CPI stack. A CPI stack visualizes the individual CPI components stacked on top of each other. The bottom CPI component typically represents time for doing useful work; the other CPI components then represent the fractions of the total execution time lost due to miss events such as I-cache misses, long-latency load misses, branch mispredictions, etc. A CPI stack is easily obtained from the mechanistic-empirical model by dividing the individual terms in Equation 1 with the number of dynamically executed (micro-)operations N.

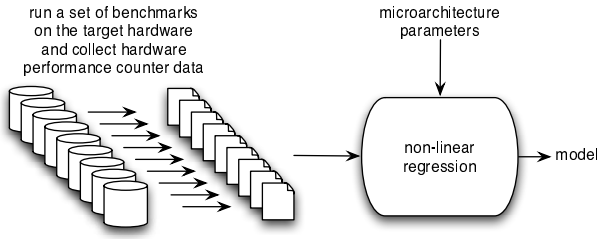


Figure 1. Inferring a performance model through mechanistic-empirical modeling.

3.1 Inferring the model

The mechanistic-empirical model contains two parameter types. The first type is a function of the microarchitecture only, e.g., processor width D, front-end pipeline depth c_{fe}, the L2 cache access time c_{L2}, TLB miss latency c_{TLB}, and main memory access time c_{mem}. These parameters can be obtained from processor specifications, or, in case of cache, TLB and memory access latencies, they can be derived from running microbenchmarks. The second type is a function of the application and the microarchitecture, see all the other parameters in the model. Some of these parameters can be readily obtained using hardware performance counters (if hardware is available) or through simulation (if hardware is not yet available), e.g., the number of I-cache misses, last-level cache load misses and branch mispredictions. The three remaining parameters, the branch resolution time, the MLP correction factor, and the number of cycles lost due to resource stalls, cannot be readily obtained through hardware performance counters, and are not trivially obtained through simulation. These are the regression parameters, and are inferred through non-linear regression.

The whole process of inferring a performance model through mechanistic-empirical modeling is illustrated in Figure 1. We run a number of benchmarks on the target platform, and we track hardware performance counters during these runs, i.e., we collect a number of events using the hardware performance counters, such as the number of cache misses (I-cache, D-cache, last-level cache), TLB misses, branch mispredictions, and floating-point operations. We then feed that data into the regression along with the known microarchitecture parameters. Regression then fits the unknown regression parameters with the data. The end result is a performance model for the target hard-

ware, with its parameters tweaked for the target hardware and the set of workloads that were used while inferring the performance model.

The following three subsections describe the structure of the three regression parameters: the branch resolution time, MLP, and the resource stall component.

3.2 Branch resolution time

The branch resolution time is defined as the number of cycles between dispatching the mispredicted branch and its resolution. This is the time needed to execute the critical path of dependent instructions leading to the mispredicted branch. According to the interval analysis by Eyerman et al. [5], the branch resolution time depends on: (1) *Interval length*, or the number of instructions since the previous miss event. The longer the interval, the more instructions will be in the instruction window by the time the mispredicted branch enters the instruction window (ROB and issue queue), and thus the longer the dependence path to the branch will be. The length of the interval depends on the number of miss events: the more miss events, the shorter the interval and the shorter the branch resolution time. (2) *Instruction-level parallelism (ILP)*: more ILP implies shorter dependence paths to the branch and thus shorter branch resolution times. The amount of ILP is tied to the average instruction execution latency. The branch resolution time will be larger if there are many long-latency instructions (e.g., floating-point operations as well as L1 D-cache load misses) on the critical dependence path to the branch. We therefore take the number of floating-point operations and the number of L1 D-cache load misses as input to the model.

Given these insights, we impose the following model for the branch resolution time:

$$c_{br} = b_1 \left(\max\left(128, \frac{1}{mp\mu_{br}}\right)\right)^{b_2} \cdot (1 + b_3 \cdot fp) \cdot (1 + b_4 \cdot mp\mu_{DL1})\,. \quad (2)$$

In this formula, $mp\mu_{br}$ stands for the number of branch mispredictions per micro-operation, fp stands for the fraction floating-point operations, and $mp\mu_{DL1}$ stands for the number of L1 D-cache misses per micro-operation that hit in the L2 cache. The b_i parameters are unknown and are to be fitted through non-linear regression. The first factor (b_1) is a constant factor and captures the importance of the branch resolution time in the overall model. The second factor captures the dependence of the branch resolution time on the number of instructions between mispredicted branches, which follows a power law according to [5]: i.e., the more instructions prior to the mispredicted branch since the previous miss event, the more instructions in the instruction window by the time the mispredicted branch enters the window, and thus the longer it will take for the branch to be resolved. We cap this factor (using the $\max$ operator) to prevent the factor to grow indefinitely for workloads that have very few mispredicted branches and a very large number of instructions between two mispredicted branches, i.e., the dependence path to the branch is limited by the size of the instruction window. The third and fourth factor model the dependence of the branch resolution time on the number of floating-point and L1 D-cache misses, respectively. We assume a multiplicative model instead of an additive model for two reasons. First, the different factors are multiplicative in nature, e.g., L1 D-cache misses on a long dependent chain of floating-point operations result in a yet longer dependence chain. Second, an additive model requires more regression parameters than a multiplicative model if one wants to capture interaction effects.

3.3 MLP correction factor

The MLP correction factor is introduced in the mechanistic-empirical model to reflect that memory access time is not constant (in contrast to what mechanistic models typically assume). There are many reasons for a non-constant memory access time, such as memory bank parallelism, open versus closed page accesses, memory controller scheduling, memory bus contention, non-blocking caches sending multiple memory requests to memory simultaneously, etc. The most prominent factor is the amount of memory-level parallelism (MLP), or the number of outstanding memory accesses if at least one is outstanding [1]; e.g., an MLP of 2 means that the penalty per memory access equals half the memory access time. Hence we divide the memory access time c_{mem} by the MLP correction factor in the overall model, see Equation 1.

The amount of MLP depends on the number of long-latency load misses, i.e., the more long-latency load misses, the more likely they will be independent, and thus the more likely they will reside in the reorder buffer simultaneously and they will expose MLP. We model the MLP correction factor's dependence on the number of misses as follows:

$$MLP = b_5 (mp\mu_{DL2})^{b_6} (mp\mu_{DTLB})^{b_7}. \quad (3)$$

There are two independent variables in this formula, namely the number of L2 (or last-level) cache misses and the number of D-TLB misses per micro-operation. We assume a power law relationship; the motivation for doing so goes back to the observation that the amount parallelism that can be extracted from a dynamic instruction stream is a power law relationship with the size of the window of instructions from which parallelism is extracted [16, 21]. Further, we make a distinction between last-level cache misses and D-TLB misses because they incur a different penalty and hence a different impact on the effective MLP.

	Pentium 4	*Core 2*	*Core i7*
microarchitecture	Netburst	Core	Nehalem
family/model/step	15/4/1	6/15/6	6/26/4
chip name	Prescott	Conroe	Bloomfield
frequency	3.4GHz	2.4GHz	2.67GHz
L1 I-cache	12K μops	32KB	32KB
L1 D-cache	16KB	32KB	32KB
L2 cache	1MB	4MB	256KB
L3 cache	—	—	8MB

Table 1. The three Intel processors used for validating the model.

platform	*width*	*depth*	*L2*	*L3*	*mem*	*TLB*
Pentium 4	3	31	31	—	313	70
Core 2	4	14	19	—	169	30
Core i7	4	14	14	30	160	40

Table 2. Micro-architecture parameters: dispatch width, pipeline depth, and cache/TLB miss latencies (in cycles).

3.4 Resource stalls

Resource stalls occur when dispatch stalls, or when the processor back-end cannot keep up with the front-end in the absence of other miss events. The reason for a dispatch stall could be a full reorder buffer or issue queue, and is caused by lack of ILP, or long chains of dependent long-latency instructions. We model the resource stall component as:

$$c_{stall} = \max\left(0; 1 - \frac{c_{miss}}{N/D + c'_{stall}}\right) c'_{stall}, \qquad (4)$$

with

$$c'_{stall} = b_8(1 + b_9 \cdot fp)(1 + b_{10} \cdot mpi_{DL1}), \qquad (5)$$

and

$$c_{miss} = \sum m_i \cdot c_i. \qquad (6)$$

Equation 5 models the dependence of the resource stall component on the number of long-latency instructions (floating-point and L1 D-cache misses) in the dynamic instruction flow. Equation 4 captures the trend that there are fewer resource stalls if there are more miss events. In other words, if there are fewer instructions between consecutive miss events, the likelihood for the reorder buffer or issue queue to fill up, and eventually dispatch to stall, is smaller. The intuition behind Equation 4 is that the likelihood for a resource stall scales inverse linearly with the fraction of the time handling miss events (c_{miss} is the total estimated time spent handling miss events, see Equation 6, or the sum across all the miss events). The $\max$ operator in Equation 4 ensures that the resource stall component is positive.

4 Experimental setup

We use both the SPEC CPU2000 and CPU2006 benchmark suites for validating the model. We consider all benchmarks and all of their reference inputs (48 and 55 benchmark-input pairs for CPU2000 and CPU2006, respectively); in addition, all benchmarks are run to completion. All benchmarks were compiled statically using the GNU C `gcc` compiler version 4.1.2 and `-O2` optimization flag.

We consider three Intel processor systems, namely Pentium 4, Core 2 and Core i7, see Table 1. The micro-architecture parameters that serve as input for the model are shown in Table 2. The dispatch width and front-end pipeline depth were easy to determine from reading the processor specifications; the Pentium 4 has a very deep pipeline with 31 stages, whereas the Core 2 and Core i7 (which is based on the Pentium M) have a shallower pipeline with 14 stages. The cache miss and TLB miss latencies are not as easily obtained. We therefore use a tool called Calibrator[2] which estimates these latencies by running parameterized micro-benchmarks. It is unclear how accurate this tool is for advanced high-performance micro-architectures, however, this is the best we could do.

The other model parameters are obtained through hardware performance counters for which we use the `perfex` and `perfmon` tools. We collect the following performance counters: number of cycles, number of committed micro-operations, number of committed x86 (macro-)instructions, number of committed branch mispredictions, L1 I-cache misses, L2 misses, L3 misses (only for Core i7), D-TLB misses and I-TLB misses, and the number of floating-point operations.

We use SPSS, a commercial statistical software package, for performing the non-linear regression. The predicted value is the number of cycles per micro-operation (CPI), i.e., the number of cycles as estimated through Equation 1 divided by the number of micro-operations. The optimization criterion is the minimization of the sum of relative squared errors, i.e., $\sum_{i=1}^{n} \frac{(\hat{y_i} - y_i)^2}{y_i}$, which minimizes the average absolute value of the relative error, as suggested by Tofallis [22].

We also compare mechanistic-empirical modeling against purely empirical modeling, namely linear regression and artificial neural networks (ANNs). Both linear regression and ANNs use the exact same input as mechanistic-empirical modeling. The ANN is a multi-layer perceptron with a hidden layer that is connected to the input layer and output layer. Each hidden node is connected to each input, and the output node is connected to each hidden node. A

[2] `http://homepages.cwi.nl/~manegold/Calibrator/`

hidden node computes the $\tanh$ function of the weighted sum of its inputs; the output node computes a weighted sum across the hidden nodes.

5 Evaluation

The model evaluation is done in a number of steps. We evaluate the model's accuracy and robustness, and compare it against purely empirical models, namely linear regression and ANNs. We do not compare against mechanistic modeling because, as mentioned before, this would require building profiling and simulation tools that are validated against real hardware; also, running these profiling and simulation experiments would be (very) time-consuming. This is far from trivial, or at least impractical, which is why we propose hybrid mechanistic-empirical modeling in the first place.

5.1 Model accuracy

For evaluating the model's accuracy, we compare the 'measured CPI' against the 'predicted CPI'. The measured CPI is computed by dividing the number of cycles with the number of committed instructions; both the number of cycles and committed micro-instructions are measured using hardware performance counters while running the benchmark on the processor. The predicted CPI is computed using the mechanistic-empirical model. Figure 2 shows the measured CPI versus the predicted CPI as a scatter plot; there are separate graphs for CPU2000 (top row) and CPU2006 (bottom row), for all three processors, Pentium 4, Core 2 and Core i7. Each benchmark is represented as a point. An accurate model results in all points to be around the bisector. We observe this to be the case for both benchmark suites and all three machines. There are only a few benchmarks that are off the bisector. The average absolute prediction error on Core i7 is 9.7% and 10.5% for CPU2000 and CPU2006, respectively; the max error is 35%; and 90% of all benchmarks have a prediction error below 20%. The highest errors are observed for benchmarks that are somewhat outliers compared to the other benchmarks, e.g., CPU2006's calculix and gromacs have very low branch misprediction rates, very low I-cache and last-level cache miss rates. Outlier benchmarks are more difficult to capture accurately by the model because they are dissimilar from the benchmarks from which the model is inferred.

5.2 Model robustness

The previous section assumes that we build a performance model for a given benchmark suite, say CPU2006, and then evaluate the model's accuracy with that same benchmark suite — no cross-validation. This gives an idea about how accurate the model is for predicting the performance of the given benchmark, but it does not quantify how good the model is at generalizing performance trends outside the benchmark suite. In other words, it is unclear whether the model is subject to overfitting. In order to evaluate the model's robustness, we set up the following experiment. We build two performance models: the first one is inferred using the CPU2000 benchmark suite — this is called the 'CPU2000 model' — and the second one is inferred using the CPU2006 benchmark suite and is called the 'CPU2006 model'. We then evaluate the accuracy for both models on CPU2006. If the model is not subject to overfitting, then the prediction error should be comparable for the CPU2000 and CPU2006 models. This is found to be the case, see Figure 3, which shows the prediction error for the three processors. The prediction errors are sorted, and a point (x, y) says that x% of the benchmarks have a prediction error below y%. These graphs show that the CPU2000 model is only slightly less accurate than the CPU2006 model when evaluated on the CPU2006 benchmarks, which supports our statement that the model is not subject to overfitting and is indeed capable of capturing general performance trends. Only for a handful benchmarks we observe that the CPU2000 model is less accurate. For example, on the Core 2, we observe relatively high errors for milc, namd and soplex when applied to the CPU2000 model. Again, this is due to a lack of similarity between these benchmarks and the CPU2000 benchmark suite, i.e., for milc and soplex, this is due to a high last-level cache and D-TLB miss rate compared to any of the CPU2000 benchmarks; likewise, soplex has a relatively high fraction floating-point operations compared to most of the CPU2000 benchmarks.

5.3 Comparison to empirical modeling

We now compare mechanistic-empirical modeling against purely empirical modeling. The empirical models that we compare against are linear regression and artificial neural networks (ANNs), as detailed before. Figure 4 provides experimental evidence. The top graphs show the average prediction error for all three models in a setup in which we use the same benchmark suite for model construction and evaluation, i.e., no cross-validation. The bottom graphs assumes a cross-validation setup in which we construct a CPU2006 model and evaluate on CPU2000 (bottom left graph), and, vice versa, construct a CPU2000 model and evaluate on CPU2006 (bottom right graph). All three models seem to yield comparable accuracy in the no cross-validation setup, however, the mechanistic-empirical model is a clear winner in the cross-validation setup. The end conclusion is that purely empirical models are subject to overfitting while the mechanistic-empirical model is not.

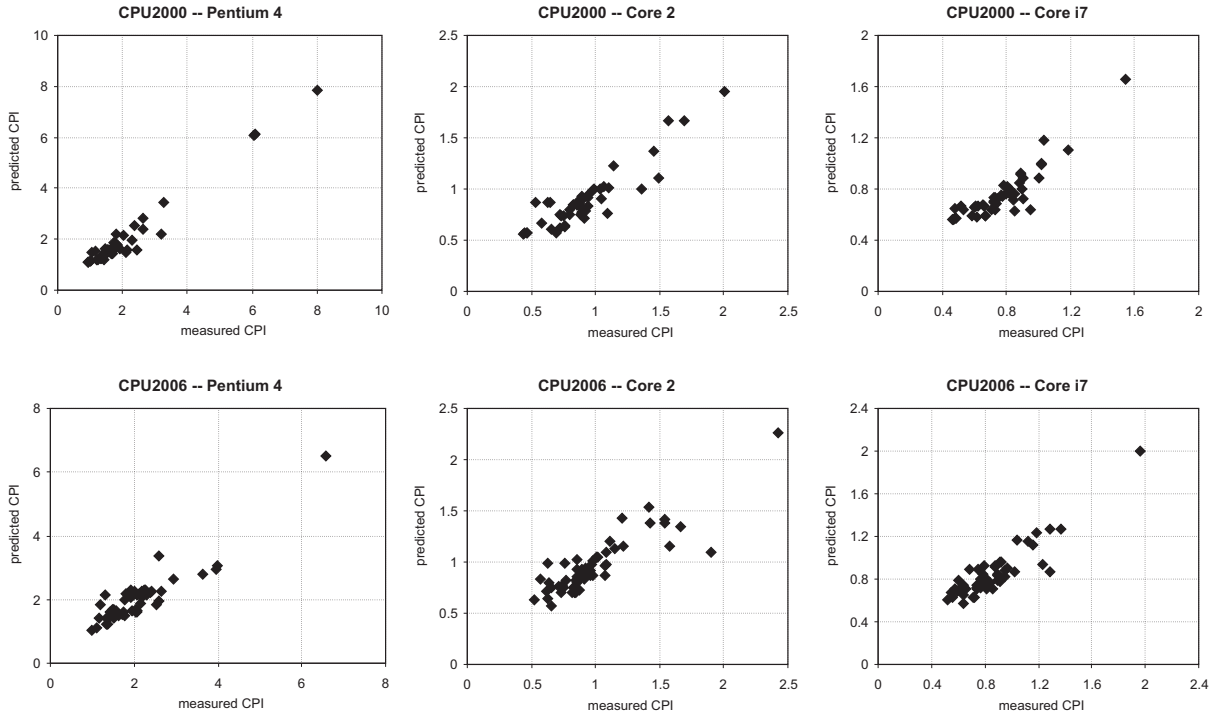


Figure 2. Prediction error scatter plots for CPU2000 (top row) and CPU2006 (bottom row) for Pentium 4, Core 2 and Core i7.

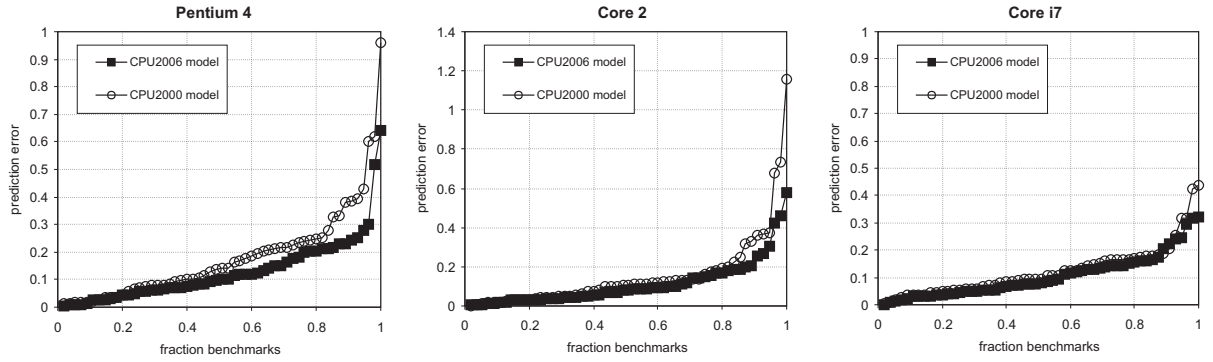


Figure 3. Evaluating model robustness: Comparison of the CPU2000 and CPU2006 models for the CPU2006 benchmark suite.

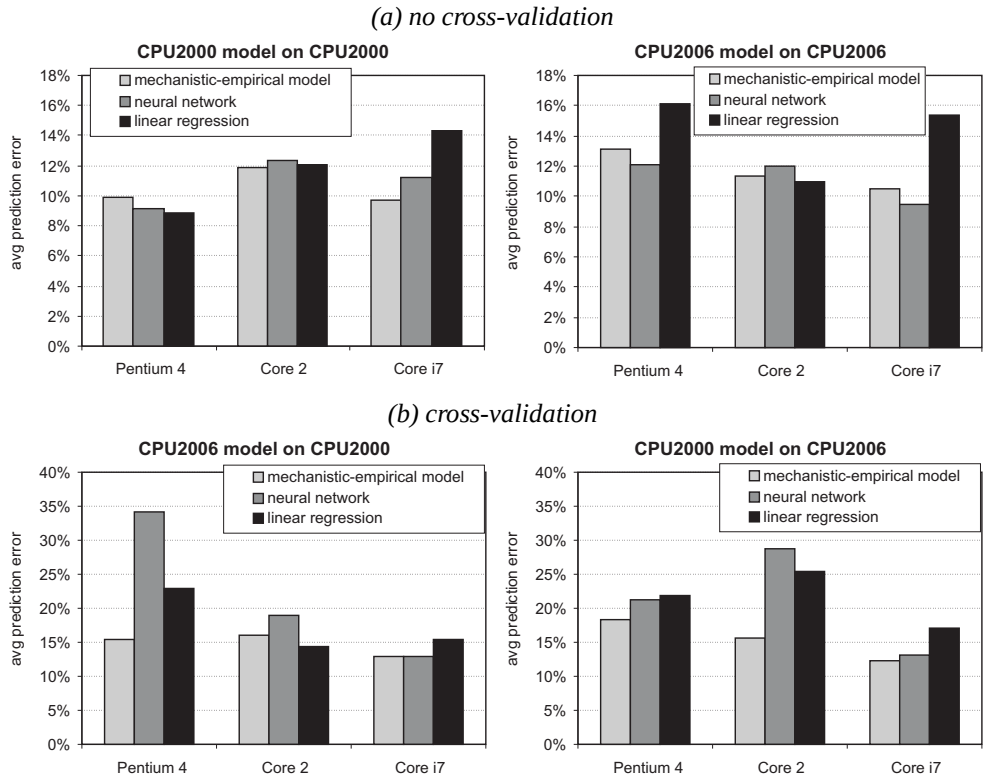


Figure 4. Comparison of mechanistic-empirical modeling against purely empirical modeling (linear regression and ANN).

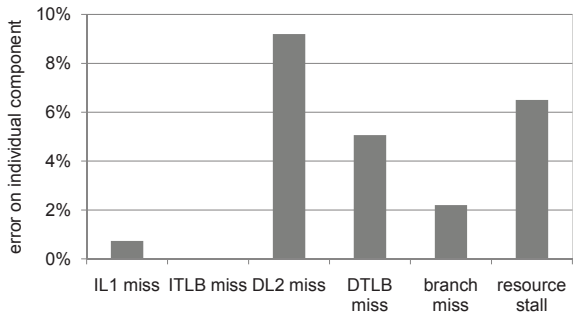


Figure 5. Evaluating the accuracy of the CPI components.

5.4 Validating the CPI components

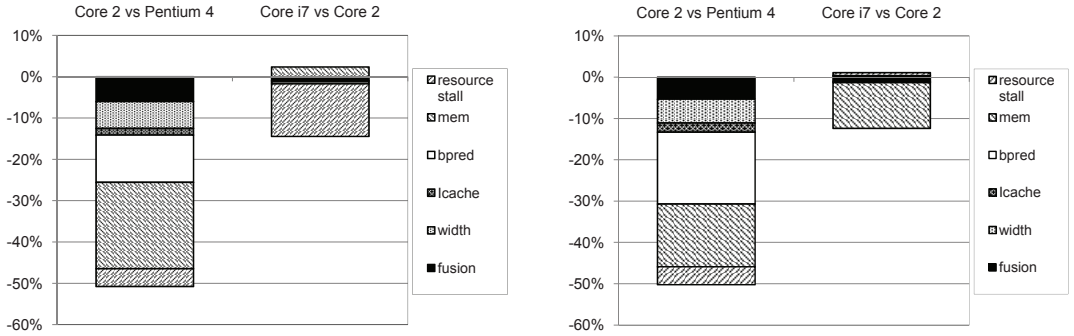
So far, we evaluated the model's accuracy for predicting total execution time. We now focus on the model's ability to accurately predict individual CPI components. Because individual CPI components cannot be obtained from real hardware, we have to resort to simulation. We therefore employ SimpleScalar's out-of-order processor simulator and we compare the estimated CPI components by the mechanistic-empirical model against the method proposed by Eyerman et al. [3]; the latter proposed a novel hardware performance counter architecture for constructing accurate CPI stacks for superscalar out-of-order architectures. Figure 5 shows the accuracy for predicting each of the individual CPI components. The prediction error is fairly low, with the highest error observed for the L2 D-cache component (9.2% error). The reason is that it is impossible to measure MLP on current hardware, and hence our MLP correction factor is rather crude. The second hardest to predict CPI component is the resource stall component. The main difficulty there is to estimate the impact of chains of dependent instructions on resource stalls.

As a summary of the validation, we find mechanistic-empirical modeling to be accurate compared to real hardware, and, in addition, we find it to be accurate for computing CPI components and stacks.

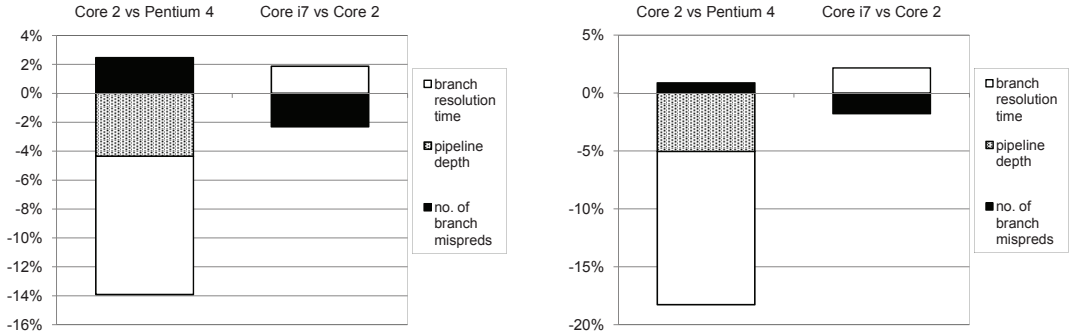
6 CPI Stacks

CPI stacks are extremely useful for a wide variety of applications, ranging from software optimization and performance analysis to workload characterization and var-

(a) Overall CPI-delta stacks



(b) Branch misprediction CPI-delta stacks



(c) Last-level cache CPI-delta stacks

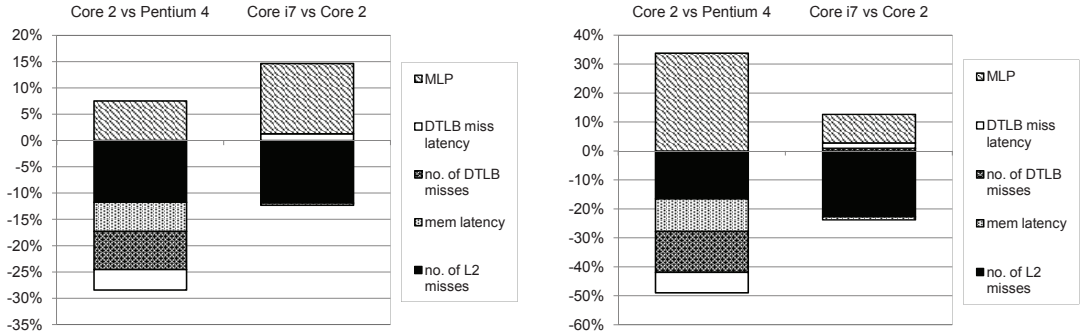


Figure 6. CPI-delta stacks for CPU2000 (left column) and CPU2006 (right column).

ious runtime optimizations. Constructing CPI stacks on out-of-order processors is challenging because of overlap and latency hiding effects [3]. As mentioned before, mechanistic-empirical modeling however enables constructing CPI stacks on existing hardware. Being able to construct CPI stacks, one can derive CPI-delta stacks that provide insight in where the performance differences come from when comparing designs. Basically, a CPI-delta stack visualizes the deltas in the various CPI components when comparing two CPI stacks. In addition, it breaks up each CPI component in a number of factors that collectively determine the CPI component. This is possible given how the mechanistic-empirical model computes these components. For example, the branch misprediction component consists of three major factors, namely the number of mispredicted branches, the branch resolution time and the front-end pipeline depth; CPI-delta stacks visualize each factor.

Different sources of improvement for CPU2000 vs. CPU2006. Figure 6 shows CPI-delta stacks when comparing Core 2 versus Pentium 4, and Core i7 versus Core 2, for both CPU2000 and CPU2006. The top row shows overall CPI-delta stacks. Each CPI-delta component shows the delta due to a particular CPI component; we make a distinction between improvements due to wider dispatch, micro-op fusion, I-cache which includes I-TLB, memory which includes D-cache and D-TLB performance, branch prediction, and finally the resource stall component — the sum of all components equals the total improvement. It shows that the overall performance improvement from Pentium 4 to Core 2 comes from a number of sources, the primary sources being smaller branch misprediction and last-level cache CPI components as well as wider dispatch (base component) and micro-operation fusion, for both CPU2000 and CPU2006. The performance improvement from Core 2 to Core i7 comes from different sources depending on the benchmark suite. For CPU2000, the performance difference mainly comes from a reduced resource stall CPI component; for CPU2006, the performance difference comes primarily from improvements in last-level cache performance. Improvements in the memory hierarchy and micro-architecture between Core 2 and Core i7 (i.e., an additional level of cache and increased size of the last-level cache) have a different effect on CPU2000 versus CPU2006. For CPU2006, which is more memory-intensive, this results in fewer last-level cache misses. For CPU2000, this results in reduced average latency of on-chip cache misses and better latency hiding because of larger-sized ROB and other buffer resources.

Poorer branch predictor but better branch performance on Core 2 than Pentium 4. The middle row in Figure 6 shows the CPI-delta stacks for the branch misprediction component. A striking result is that the number of branch mispredictions is higher for Core 2 than for Pentium 4. The branch predictor is more accurate in the Pentium 4 than in the Core 2, i.e., for CPU2006, MPKI (number of branch mispredictions per thousand instructions) is 4.1 for Pentium 4 and 5.8 for Core 2. This is compensated for though through a shallower pipeline and shorter branch resolution time for Core 2, resulting in a net performance gain over Pentium 4. Comparing Core i7 against Core 2, we observe the reverse effect: the number of branch mispredictions is reduced, but the branch resolution time grows, resulting in almost no performance impact. The reorder buffer is larger in the Core i7 compared to the Core 2 which may result in a larger branch resolution time, i.e., executing the critical path to the mispredicted branch takes longer, simply because there are more instructions to be executed.

Eliminating hidden memory accesses does not improve performance. The bottom row in Figure 6 shows CPI-delta stacks for the last-level cache component. The interesting result here is that improvements in the memory hierarchy reduce MLP, i.e., reducing the number of last-level cache misses reduces the opportunities for exploiting MLP. In most cases (CPU2006 as well as CPU2000 and Core vs. Pentium 4), the reduction in the number of misses offsets the reduction in MLP which leads to a net performance improvement. However, for CPU2000 and Core i7 vs. Core 2, the reduction in the number of last-level cache misses is completely offset by the decrease in MLP. This means that the misses that are removed through improvements in the memory hierarchy on Core i7, do not affect performance on Core 2 because their penalties are hidden by other outstanding misses on Core 2. This result suggests research in memory hierarchy optimizations that are MLP-aware. (Qureshi et al. [20] already made the case for an MLP-aware cache replacement policy.)

7 Conclusion

Mechanistic-empirical processor performance modeling is a gray-box modeling approach that bridges the gap between mechanistic (white-box) and empirical (black-box) modeling. Starting from a parameterized performance model inspired by mechanistic modeling, mechanistic-empirical modeling infers the unknown parameters through regression, alike empirical modeling. By doing so, it combines the best of both worlds: mechanistic-empirical models provide insight (like mechanistic modeling) and at the same time are easy to construct (like empirical modeling).

We derive mechanistic-empirical models for three commercial Intel processor cores, Pentium 4, Core 2 and Core i7. Unlike prior work in analytical performance modeling which evaluates against simulation models, we evaluate

against real hardware and report average prediction errors around 9% to 13% for the CPU2000 and CPU2006 benchmarks. In addition, we compare mechanistic-empirical modeling against purely empirical modeling in a cross-validation setup (i.e., the evaluation is done on a different benchmark suite than the one used to infer the model), and conclude that empirical modeling is subject to overfitting whereas mechanistic-empirical is not. In other words, mechanistic-empirical modeling can better generalize performance trends. Mechanistic-empirical modeling enables several opportunities for software and hardware optimization and analysis. As an illustrative case study we compare CPI and CPI-delta stacks for the three Intel processors to understand where the performance differences come from and obtain several interesting conclusions. The end conclusion is that mechanistic-empirical modeling provides insight, is easy to construct, is accurate compared to real processor hardware, is more robust than purely empirical modeling, and provides several opportunities for exploitation.

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 European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, and the Intel ExaScience Lab co-funded by the Flanders Agency for Innovation by Science and Technology (IWT).

References

[1] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA*, pages 76–87, June 2004.

[2] 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.

[3] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, pages 175–184, Oct. 2006.

[4] 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.

[5] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *ISPASS*, pages 48–58, Mar. 2006.

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

[7] 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.

[8] 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.

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

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

[11] T. Karkhanis and J. E. Smith. Automated design of application specific superscalar processors: An analytical approach. In *ISCA*, pages 402–411, June 2007.

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

[13] B. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *ASPLOS*, pages 36–47, Mar. 2008.

[14] B. Lee, D. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPOPP*, pages 249–258, Mar. 207.

[15] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, pages 270–281, Nov. 2008.

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

[17] I. Nestorov, M. Rowland, S. T. Hadjitodorov, and I. Petrov. Empirical versus mechanistic modelling: Comparison of an artificial neural network to a mechanistically based model for quantitative structure pharmacokinetic relationships of a homologous series of barbiturates. *The AAPS Journal*, 1(4):5–13, Dec. 1999.

[18] E. Ould-Ahmed-Vall, K. A. Doshi, C. Yount, and J. Woodlee. Characterization of SPEC CPU2006 and SPEC OMP2001: Regression models and their transferability. In *ISPASS*, pages 170–190, Apr. 2008.

[19] 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.

[20] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, pages 167–177, June 2006.

[21] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, Dec. 1972.

[22] C. Tafollis. Least squares percentage regression. *Journal of Modern Applied Statistical Methods*, May 2009.

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

[24] K. Vaswani, M. J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *CGO*, pages 131–143, Mar. 2007.