

On the Comparison of Regression Algorithms for Computer Architecture Performance Analysis of Software Applications *

ElMoustapha Ould-Ahmed-Vall, James Woodlee, Charles Yount, Kshitij A. Doshi
Intel Corporation
5000 W Chandler Blvd
Chandler, AZ 85226
couldahm@ece.gatech.edu and {jim.woodlee,chuck.yount,kshitij.a.doshi}@intel.com

Abstract

The ability to provide diagnostic information for workload performance is of great value in the performance tuning process. Not only can it orient the tuning process by identifying key performance issues, it can also be used to estimate the severity of each performance issue and the potential gain from addressing it.

This work investigates the ability of some of the most popular machine learning regression algorithms to provide this diagnostic information. Five regression algorithms are trained using real performance data collected on an Intel® Core™2 Duo processor desktop machine. The algorithms are compared along two axes, prediction quality and usefulness of output, in order to gain key insights into the causes and severity of performance issues.

Although several techniques are found to demonstrate good prediction quality, our study shows that the model-tree-based technique (M5') gives superior interpretability. This class of algorithm produces models that can be used, not only to predict performance, but also to indicate the sources of potential performance improvement and to quantify the potential performance gain. This information can be used to direct performance optimization efforts by prioritizing performance problems.

1 Introduction

Workload performance analysis is used to tune applications to achieve the best possible performance (e.g., shortest execution time) on a given architecture. It can also be used to compare different implementation alternatives during the design and implementation of new applications. Traditionally, performance analysis is conducted by counting the number of occurrences of micro-architectural events, such as cache misses and branch mispredicts, to assess the presence and severity of various performance issues. A fixed penalty (latency cycles) is assigned for each type of event. This methodology ignores the interaction between various events and the ability of modern microprocessors to hide latency using techniques such as out-of-order execution, pre-fetching and speculative execution. This results in micro-architectural performance events having varying penalty depending on the amount of latency that can be hidden, which in turn depends on the characteristics of the workload (e.g., instruction mix and present level of parallelism) and the interaction with other performance events (e.g., the presence or absence of other performance events).

This paper considers two important performance analysis questions:

- The “what” question: This question tries to identify the main performance issues or sources of potential performance improvements. This is important, as it can orient the effort of the performance analyst to optimize for specific performance issues (e.g., reduction of cache misses).

*The authors would like to thank the following people for their help with this work: Seth Abraham, Antonio C. Valles, Garrett T. Drysdale, James C. Abel, Agustin Gonzalez, David A. Levinthal, Stephen P. Smith, Henry Ou, Yong-Fong Lee, Alex A. Lopez-Estrada, Kingsum Chow, Thomas M. Johnson, Michael W. Chynoweth, Annie Foong, Vish Viswanathan

- The “how much” question: This question tries to estimate the potential performance gain (e.g., percentage reduction in execution time) from mitigating a specific performance issue or a set of performance issues. This question is important as there may be several performance issues and one needs to decide which ones are most important and whether it is worth trying to optimize for a specific issue.

In answering the previous two questions, it is important to take into account the potential interactions between different performance events. For example, if two events tend to occur at the same time, it is possible that the actual penalty incurred for one (e.g., Level 1 cache miss) depends on the occurrence or not of the other (e.g., DTLB miss). These interaction effects create non-linearities and can result in distinct performance models for different categories of workloads and even phases of a single workload [18].

We investigate the usefulness of machine learning regression techniques to construct accurate and useful performance models that can address the “what” and “how much” questions. In particular, we focus on models that can be used to diagnose potential performance issues and to estimate the potential gain from addressing one or more specific performance issues. Evaluation is performed on five different regression algorithms to compare the pros and cons of these algorithms and to determine which algorithm or class of algorithms is suitable for performance analysis.

The remainder of the paper is organized as follows. Section 2 discusses some of the related work. Section 3 presents the different regression algorithms used in this study. Section 4 describes in detail our experimental setup used for data collection. Section 5 presents our results. Section 6 concludes the paper.

2 Related Work

Recent years have seen several attempts to build models for performance analysis of processors. Unfortunately, most of these models fail to include many micro-architectural events and design space parameters, which leaves validity unknown when a large set of events and design parameters are present. This is mainly because these models require prior knowledge about significant events and parameters, and the required knowledge is gained from expensive, simulation-based sensitivity analysis. In addition to its prohibitive cost, simulation accuracy is questionable especially in the case of applications whose time varying behaviors are not easily represented in traces. Our work avoids these problems by relying on counts of a broad spectrum of processor events collected during the execution of the entire application, rather than those obtained through simulation. Such counts include, for example, misses in the various code, data and translation caches, branch mispredicts, load-store address overlaps, and many other events that can potentially reduce performance.

In [10], the authors propose a linear formula expressing the cycles-per-instruction (CPI) metric as a function of data and instruction cache misses, branch mispredicts and the ideal steady-state CPI. The performance penalty of cache misses and branch mispredicts is estimated using trace-driven simulation. The work in [20] extends [10] by including the effects of pre-fetching and resource contention in the model and uses a probabilistic approach to limit the required number of trace-driven simulation scenarios. These two approaches do not include other critical potential sources of CPI degradation such as DTLB and ITLB misses, various load blocks and the effects of unbalanced instruction mixes. More importantly, the two models do not account for the inherent interaction effects between various performance events, or for differing behaviors from application to application and often among different phases of the same application [18]. In contrast, this work establishes a classification of workloads or phases of workloads, and builds a model for each class, using measured performance data rather than simulation data.

In [7, 21], analytical models are used to study the effect of pipeline depth on performance for in-order and out-of-order processors. These two works use simulation-based sensitivity analysis to determine important model parameters. In [7], detailed superscalar simulation is used to determine the fractions of stall cycles for different pipeline stages and the degree of superscalar processing that remains viable. In [21], the authors use detailed simulations of a baseline scenario and scenarios with increased processor front-end width to determine the effects of micro-architecture loops (e.g., branch mispredict loops) on the performance. Again, these two models take into account only one aspect of the performance analysis. Our model, on the other hand, considers the processor performance as a whole while including many potential sources of performance degradation.

Several statistical techniques have been used to limit the required number of simulation runs for design space exploration needed during the design phase of new processors. In [5, 4], principal component analysis is used to limit design space exploration by identifying key design space parameters and computing their correlations. Plackett and Burman fractional design is used in [24] to establish parameter prioritization for sensitivity analysis. The authors model high and low values of a set of N design parameters using only $2N$ simulations focusing on parameters with high priority.

In [6], the authors define interaction cost to account for the interaction between two different micro-architectural performance events. The authors design new hardware to enable sampling workload execution in sufficient detail to construct representative dependency graphs to be used for the computation of the interaction cost. Our approach also takes into account the interaction between various micro-architectural events. However, we propose the handling of the interaction cost in a statistical manner without the requirement of dedicated new hardware.

3 Methodology: Regression Algorithms

This section describes the different regression approaches used in this paper. Regression consists of fitting a model that relates a dependent variable Y to a set of independent predictors X_1, X_2, \dots, X_k . The functional form of the model can be estimated using training samples from the unknown underlying distribution. In this study, we compare the merits of five different regression algorithms in the context of performance analysis: (1) Multi-linear regression [17], (2) Artificial neural networks [13], (3) Locally weighted linear regression [2], (4) Model trees [22] and (5) Support vector machines [15, 19]. These algorithms are described briefly below.

3.1 Multi-Linear Regression

Linear regression [17] is based on the assumption of a linear relationship between the dependent variable Y and its predictors X_1, X_2, \dots, X_n . Linear regression offers simple and easily interpretable models. However, it can result in inaccurate models that predict poorly in the presence of a nonlinear or non-additive relationship. Due to the complexity of micro-architectural event interaction and varying event performance penalties, however, it is common for a nonlinear relationship to exist. In the linear case, the functional relationship between Y and its predictors is estimated by minimizing the residual sum of squares (RSS). For more details on multi-linear regression, the reader is referred to [17] or any classical statistics text.

3.2 Artificial Neural Networks

Artificial Neural Network (ANN) is a powerful method for generalized nonlinear regression. This class of algorithms is patterned after cooperative processing of information that is found in the biological world's neurons and networks of neurons [13]. A multilayer neural network consists of a number of neurons organized into an input layer, an output layer and a number of hidden layers. Units in the input layer take as input the information to be processed (values of the predictors in our case), while the output layer produces the prediction result. The first hidden layer receives as input the results of the units in the input layer and gives its results as inputs to the units in the next layer.

The training of an ANN establishes the input/output mapping in the form of connections between various units in the network. The training also computes the weights of input connections. The fitted model can be used to predict the values of the dependent variable Y for unseen data points. The ANN approach has two key benefits: (1) it has high prediction accuracy and (2) it does not require any prior knowledge of the form of the functional relationship between the dependent variable and the independent variables. It has the drawback, however, that the black-box nature of ANN thwarts interpretation of results and therefore prevents insight into the sources of performance degradation and the exact performance impact of the different micro-architectural events. In addition, the approach is known to be very sensitive to outliers.

3.3 Locally Weighted Linear Regression

Locally Weighted linear Regression (LWR) [2] is a "lazy" or instance-based learning technique. A new regression equation is fitted every time the model needs to predict on a new instance. This is in contrast with the other methods seen in this section where one regression model is built during the training phase and used with all test instances.

LWR combines linear regression and instance-based learning. Unlike regular linear regression, where one regression is performed on the full unweighted training set, LWR performs a new regression for each instance, weighting training instances based on their distance (e.g., Euclidean distance) from the specific test instance. The main advantage of LWR is its high flexibility, which makes it suitable for the approximation of nonlinear functions. The main disadvantage of this method, like all instance-based learning methods, is that it does not provide much insight into the global structure of the training data. This limits the interpretability of its output.

3.4 Model Trees: M5'

Model trees are a sub-class of regression trees [3], having linear models at the leaf node. In comparison with classical regression trees, model trees deliver better compactness and prediction accuracy. These advantages issue from the ability of model trees to leverage potential linearity at leaf nodes.

The model tree algorithm used in this work is based on M5' [22], an optimized, open-source implementation of the classical M5 [16] algorithm. The input space is recursively partitioned until the data at the leaf nodes constitute relatively homogeneous subsets such that a linear model can explain the remaining variability. This divide-and-conquer approach partitions the training data and provides rules for reaching the models at the leaf nodes. The linear models are then used to quantify, in a statistically rigorous way, the contribution of each attribute (e.g., micro-architectural events here) to the overall predicted value (e.g., performance in this case). A powerful aspect of the prediction model arrived at in this way is that it is interpretable, in contrast with other machine learning approaches, such as neural networks.

3.5 Support Vector Machines

Support Vector Machines (SVM) [15] are a combination of instance-based and numeric modeling learning. The idea behind support vector machines is finding instances, called support vectors, that are at the boundary of the classes and creating linear functions that discriminate them as widely as possible. The biggest advantage of vector machines is that they can use linear, quadratic or higher order models to represent nonlinear boundaries between classes. This is in contrast to basic linear models that only represent linear boundaries. To construct nonlinear boundaries with linear models, support vector machines use nonlinear mapping, where the instance space is transformed, allowing a linear model to represent a nonlinear model in the previous space.

The Sequential Minimal Optimization algorithm (SMO) has been shown to be an effective method for training SVM on classification tasks defined on sparse data sets. SMO differs from most SVM algorithms in that it does not require a quadratic programming solver. The technique used here is a generalization of SMO by Shevade et al [19] to handle regression problems.

The main benefit of using SVMs is that they are robust against overfitting. Like ANNs, a problem with applying this technique to analysis of processor performance is that its black-box nature prevents insight into sources of inefficiencies. In addition, training SVMs is particularly slow. It took the authors more than 10 hours to train a model with performance data. In contrast, training model trees (M5') on the same dataset using the same hardware required less than 10 minutes.

4 Experimental Setup

In this section, we describe the experimental setup we used to collect the necessary training data.

4.1 Platform

The data used in this study is collected on an Intel[®] Core[™]2 Duo processor-based desktop platform. The test machine has a speed of 2.4 GHz and 1GB of memory. The memory subsystem consists of a two-level cache. Each core has a 32 KB, level- one instruction cache and a separate level-one data cache of the same size. The two cores share a unified level-two cache of 4 MB. For more details on Core[™]2 Duo processor architecture, the reader is referred to [9, 8]. The data collection platform is running a Microsoft[®] Windows[™]XP 64 bit operating system.

4.2 Data Collection Methodology and Tool

The data was collected using an internally-developed tool. This tool is similar to the Intel VTune Performance Analyzer, but it collects data in counting mode. The counting mode obtains the values of a set of micro-architectural events of interest every time a threshold count is reached for another reference event. In particular, we divide the execution sequence into sections of equal numbers of instructions retired and collect the counts of various micro-architectural events for each section, using instructions-retired as the reference event.

Dividing the execution sequence into fine-grained sections in the above manner is done to capture the phase behavior of the workload [18]. In general, we expect that several phases, each with distinct performance characteristics, are present in the workload. Dividing the execution sequence into multiple sections increases the probability of capturing these phases.

4.3 Micro-Architectural Events

The Core™2 Duo architecture implements processor counters for multiplexed collection of information about several hundred micro-architectural events, that cover different aspects of the processor's behavior. Of these, a significant fraction of events can be excluded from consideration simply because they do not have a performance impact or do not arise except due to error conditions. Of what remains, it is still impractical to collect counts on a majority of events due to the small number of multiplexed counters. To offset these practical difficulties, it was necessary to pre-select a subset of events so a subset of 21 events was identified as candidates likely to be most relevant in the performance analysis. This apparently ad-hoc choice was purely pragmatic and revisable; happily, the prediction accuracy of the model, shown in Section 5, suggests it was on target. The chosen set of events represents the execution time and various performance-related micro-architectural events characterizing the instruction mix, the memory sub-system, the branch prediction accuracy, the data and instruction translation lookaside buffers and other known potential sources of performance degradation.

4.3.1 Execution Time

The execution time is the number of unhalted CPU clock cycles that the workload takes to execute, measured by the event CPU_CLK_UNHALTED.CORE, and considered the primary performance metric in this study. Workload sections consisting of equal numbers of instructions retired have radically different execution times. This event is used to derive the CPI (cycles per instruction) which constitutes our dependent variable.

4.3.2 Instruction Mix

While each section comprises a fixed number of instructions retired, the instruction mix can change from section to section. Different instruction mixes can give rise to different performance issues (e.g., data cache misses can only be caused by memory referencing instructions). In addition, different types of instructions execute on different functional units and stress different resources. For instance, Core™2 Duo architecture can retire up to four instructions per cycle, but these four instructions cannot contain more than one store instruction. This means that a high percentage of store instructions results automatically in a lower average number of instructions retired per cycle (longer execution time) even if there is no other performance issue. For the analysis, the retired instructions are divided into four different groups:

- Load instructions: the number of load instructions, counted using the INST_RETIREDD.LOADS event.
- Store instructions: the number of store instructions, counted using the INST_RETIREDD.STORES event.
- Branch instructions: the number of branching instructions counted using the BR_INST_RETIREDD.ANY event. In this study, it is further divided into correctly predicted and mispredicted branches as will be discussed shortly.
- Other instructions: all other instructions, counted by subtracting the above three counts from the total number of instructions retired. In particular, this category includes both integer and floating point instructions.

4.3.3 Branch Related Events

The distribution of branches between predicted and mispredicted is critical, as each mispredicted branch forces the execution pipeline to be flushed and the fetch engine to be restarted at the correct branch target and costs up to a few dozen cycles.

- The number of mispredicted branch instructions is counted using the event BR_INST_RETIREDD.MISPRED
- Subtracting the above from the total, i.e., (BR_INST_RETIREDD.ANY - BR_INST_RETIREDD.MISPRED) yields the number of correctly predicted branches.

4.3.4 Memory Subsystem Events

Load or Store instructions that miss in caches tend to have a profound impact on performance. We collect data on the number of misses occurring at various caches within the memory subsystem.

- The number of level 1 data cache misses is counted using the event MEM_LOAD_RETIRED.L1D_LINE_MISS. This does not double-count a cache line that is missed while still being brought into L1 cache as a result of a previous cache miss.
- The number of level 1 instruction cache misses. This count is obtained using the event L1I_MISSES.
- The number of level 2 cache misses is counted using the event MEM_LOAD_RETIRED.L2_LINE_MISS. In Core™2 Duo, the L2 cache is shared between the two cores and so this event counts both data and instruction misses in the level two cache.

4.3.5 Translation Lookaside Buffers Events

Data and instruction translation lookaside buffers (DTLB and ITLB) are critical resources for efficient execution across nearly all workloads. Several events are used to monitor the DTLB and ITLB stresses that arise during a specific workload or sections of it.

- The number of load accesses that miss the first level DTLB (L0 DTLB) is counted using the event DTLB_MISSES.L0_MISS_LD.
- The number of load accesses that miss the last level DTLB is counted using the event DTLB_MISSES.MISS_LD.
- The number of non-speculative load accesses that miss the DTLB, a subgroup of the previous event, is counted using the event MEM_LOAD_RETIRED.DTLB_MISS.
- The overall number of DTLB miss events which arise for any reason (i.e., due to loads, stores and hardware initiated memory references, including speculative operations) is counted using the event DTLB_MISSES.ANY.
- The overall number of retired instructions missing the ITLB is counted using the event ITLB.MISS_RETIRED.

4.3.6 Other Events

A number of other events often indicate potential performance issues.

- Load block related events: The Core™2 Duo processor uses memory disambiguation [9, 8] to maximize concurrency among loads and stores that don't intersect. In certain cases memory disambiguation fails, leading to different types of load blocks, depending on what causes the failure. LOAD_BLOCK.STA counts the number of load instructions blocked because of a preceding store instruction to an address that is not yet known. LOAD_BLOCK.STD measures the number of load instructions blocked because of a preceding store to the same address when the data to be stored is not yet known. LOAD_BLOCK.OVERLAP_STORE counts the number of load operations blocked because of an actual datum-width overlap with a preceding store, or because of an ambiguous overlap from page aliasing in which the load and a preceding store have the same offset but into different pages. Generally, these load block events can be avoided by increasing the distance between load and store instructions.
- Split events: Accesses that are not aligned to natural type-boundaries of data often cause additional cycles to complete, as the detection of potential conflicts with previous accesses may in general require blocking the current access until previous memory operations have retired. MISALIGN_MEM_REF counts the number of memory reads or writes that cross an eight-byte boundary. L1D_SPLIT.LOADS counts the number of load operations from the level 1 cache that span two cache lines. L1D_SPLIT.STORES counts the number of store operations to level 1 cache that span two cache lines.
- ILD_STALL: This event counts the number of instruction length decoder stall cycles due to a length changing prefix [9, 8]. Normally, instruction decoding takes one cycle; in the presence of a length changing prefix, it requires 6 cycles.

4.4 Data Pre-Processing

The Intel® Core™2 Duo architecture has five performance counters, which means that up to five micro-architectural events can be monitored simultaneously. However, three of these counters are fixed to always monitor the following events: “CPU_CLK_UNHALTED.CORE”, “INST_RETIRED.ANY” and “CPU_CLK_UNHALTED.REF”. As a result, there are only two reconfigurable performance counters, while our study requires data collection on about 20 different performance events. To work around this problem, it was decided to run each workload 11 times to collect the values of the required number of events for each workload section.

While this multiple-run approach is attractive as it allows seemingly simultaneous collection of data on all the necessary events, it has its own limitations. For instance, we observed a certain amount of variability from run to run. This can result from the presence of different operating system processes executing on the machine in addition to our workload. In our study, process affinitization was used to limit this variability. In addition, outliers with large variability were identified and removed from the data set. On the basis of several pilot tests, it was decided to use a 5% cutoff-threshold on variability; that is, workload sections for which the standard deviation for execution time from the 11 runs was higher than 5% of the mean were removed. Multi-run data collection is illustrated in Figure 1. Future work will involve the testing of event multiplexing [12] as an alternative.

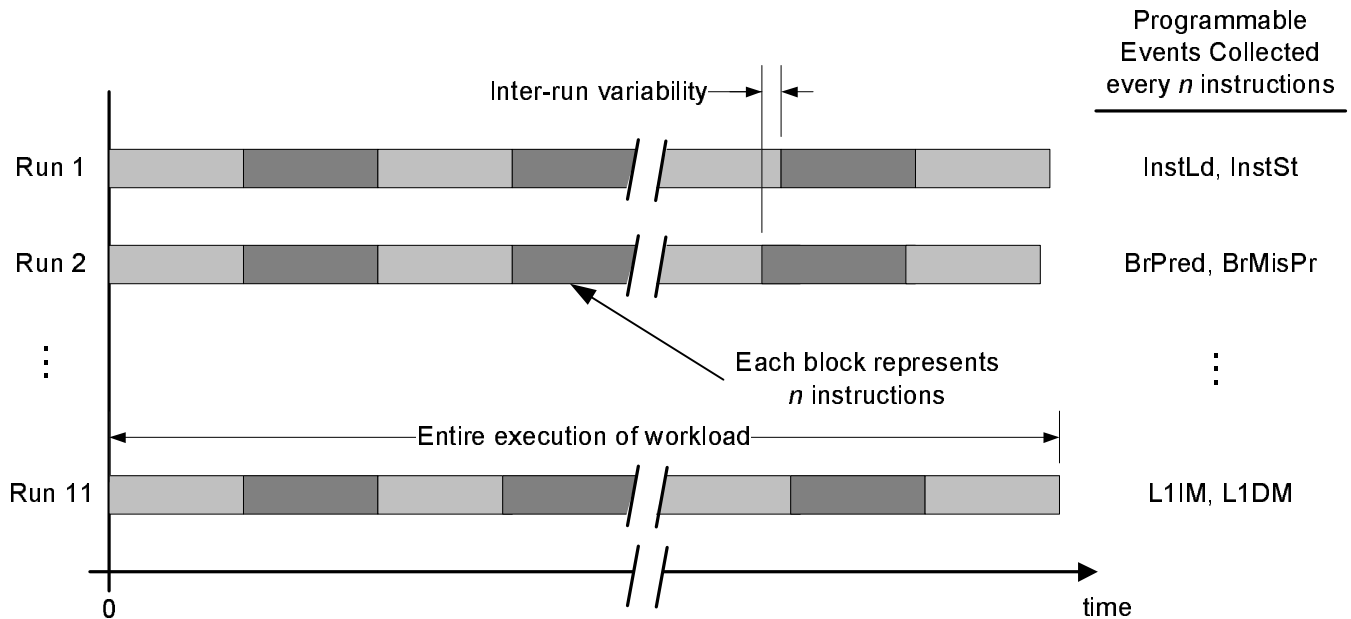


Figure 1. Multi-run data collection approach

The data is normalized by the number of retired instructions. For example, instead of the execution time as the dependent variable, the CPI (cycles per instructions) is used. CPI is computed as the execution time (number of unhalting clockticks) divided by the number of retired instructions. Similarly, the number of level 2 cache misses per retired instruction and the number of branch mispredicts per retired instruction are used instead of the raw counts of the corresponding events.

Table 1 indicates the short names used for various variables (event counts per retired instruction) along with the corresponding events and a short description.

4.5 Workloads

The data is collected on a subset of SPEC CPU2006 workloads [1], as limiting the set of workloads was necessary to restrict the size of the data set due to limitations of the WEKA tool [23] used in this study. While the use of a limited set of workloads can affect the generalization of the proposed approach, it is argued that by dividing each workload into many sections we can capture more different performance behaviors. This increases the variance of our data, which provides higher

Table 1. Selected metrics used in this study

Metric	Corresponding event	Description
CPI	CPU_CLK_UNHALTED.CORE	CPU clock cycles per instruction
InstLd	INST_RETIRED.LOADS	Loads per instruction
InstSt	INST_RETIRED.STORES	Stores per instruction
BrMisPr	BR_INST_RETIRED.MISPRED	Mispredicted branches per instruction
BrPred	BR_INST_RETIRED.ANY BR_INST_RETIRED.MISPRED	– Correctly predicted branches per instruction
InstOther	INST_RETIRED.ANY – (INST_RETIRED.LOADS + INST_RETIRED.STORES + BR_INST_RETIRED.ANY)	Non-branch and non-memory instructions per instruction
L1DM	MEM_LOAD_RETIRED.L1D_LINE_MISS	L1 data misses per instruction
L1IM	L1I_MISSES	L1 instruction misses per instruction
L2M	MEM_LOAD_RETIRED.L2_LINE_MISS	L2 misses per instruction
DtlbL0LdM	DTLB_MISSES.L0_MISS_LD	Lowest level DTLB load misses per instruction
DtlbLdM	DTLB_MISSES.MISS_LD	Last level DTLB load misses per instruction
DtlbLdReM	MEM_LOAD_RETIRED.DTLB_MISS	Last level DTLB retired load misses per instruction
Dtlb	DTLB_MISSES.ANY	Last level DTLB misses (including loads) per instruction
ItlbM	ITLB.MISS_RETIRED	ITLB misses per instruction
LdBISta	LOAD_BLOCK.STA	Load block store address events per instruction
LdBIStd	LOAD_BLOCK.STD	Load block store data events per instruction
LdBIOvSt	LOAD_BLOCK.OVERLAP_STORE	Load block overlap store per instruction
MisalRef	MISALIGN_MEM_REF	Misaligned memory references per instruction
L1DspLd	L1D_SPLIT.LOADS	L1 data split loads per instruction
L1DspSt	L1D_SPLIT.STORES	L1 data split stores per instruction
LCP	ILD_STALL	Length changing prefix stalls per instruction

representativeness of the data set. It must be mentioned here that nothing in the proposed approach restricts working only on this specific set of workloads. Future work includes data collection on commercial workloads to train the solution on a richer data set.

A total of several hundred thousand data points is obtained by dividing the workloads into distinct sections. Due to the large data set, a subset of the data is chosen randomly in a way that maintains the statistical distribution of the overall population. This is done using the “stratified remove fold” statistical technique implemented in WEKA. The resulting subset contains 27,448 data points and maintains the same statistical distribution as the original application.

We used the following workloads with varying input files (reference and training input files delivered with SPEC).

- 401.bzip2: A compression benchmark based on Julian Seward’s bzip2 version 1.0.3 with the exception that SPEC bzip2 does not perform any I/O operation except at the reading the input file.
- 403.gcc: A compiler benchmark based on gcc version 3.2. The benchmark modifies the original gcc compiler to perform more inlining and spend more time analyzing the source code input, which induces higher memory usage.
- 429.mfc: This benchmark is derived from a program to optimize single-depot vehicle scheduling in a public transportation system.
- 436.cactusADM: This benchmark is based on the Cactus computational framework used for solving the Einstein evolution equations in the ADM 3+1 formulation.
- 447.dealII: This benchmark is based on the C++ deal.II library used for adaptive finite elements and error estimation.
- 454.calculix: This benchmark is based on a finite element software used for linear and nonlinear three-dimensional structural applications.
- 458.sjeng: This benchmark is based on a program that plays chess and other chess variant games such as drop-chess.

5 Implementation and Results

For the purpose of this study, we use the open-source implementation available in the WEKA software package [23]. WEKA offers a unified framework for comparing the different algorithms described in Section 3. Each regression algorithm is trained on the dataset described in the previous section to obtain a performance model. To avoid any expert knowledge, we decided to use the default parameter for each algorithm. The only exception to this rule is the case of M5', where we report results for two different sets of runs. For the first set, we employ default algorithm parameters and call it the M5Default. To improve interpretability and achieve better compactness, for the second set, we forgo the smoothing technique that is otherwise applied by default and increase the minimum number of instances in leaf nodes and this in turn causes a small reduction in prediction accuracy for the second set. Results for the second set are labeled as M5Modified.

For performance analysis, two important qualities are desired in fitted models: model interpretability and performance predictability. Interpretability refers to the ability to explain the predicted value. For example, it is important to know why the predicted CPI of one class of work is 3, while that of another is 0.5. The understanding that interpretability creates allows one to diagnose performance issues (e.g., the low performance is due to cache misses) and to gauge the possible gain from addressing specific performance issues. Prediction accuracy, measured in multiple ways, provides bounds on modeling errors and is important as a primary measure of confidence in the model. A physical systems exact performance may be readily measured and in such a case, the role of prediction in reaching a performance estimate may be marginal; but, a high prediction accuracy confirms that the dominant characteristics of the physical system are faithfully abstracted by the model.

5.1 Interpretability of the Algorithms

Among the five algorithms, only model trees and multi-linear regression give easily interpretable results. Equation 1 gives the multi-linear regression performance model:

$$\begin{aligned}
 CPI = & 0.64 + 318.05 * LCP + 181.62 * ItlbM - 0.13 * DtlbL0LdM + \\
 & 11.40 * DtlbLdM - 6.14 * DtlbM + 1.74 * DtlbLdReM + 8.77 * L1DM - 4057.39 * L1DspSt + \\
 & 198.25 * L2M + 6.67 * L1IM + 0.95 * LdB1Sta + 1.62 * LdB1OvSt \quad (1) \\
 & -96.66 * LdB1Std + 2.86 * InstSt + 0.19 * InstLd + -832.07 * L1DspLd + \\
 & 91.33 * MisalRef - 1.76 * BrPred + 18.41 * BrMisPr - 0.22 * InstOther
 \end{aligned}$$

This equation can be easily interpreted. It tells us, for example, that a level 2 cache miss (L2M) costs on average about 200 cycles and that a branch mispredicts costs about 18 cycles. However, the equation exposes several model anomalies. In particular, the negative coefficients in front of several micro-architectural events known to impede performance are counterintuitive. It is well known that store and load splits (L1DspSt and L1DspLd), DTLB misses (DtlbM) and load blocks (LdB1Std) degrade performance considerably if they occur, even in moderate frequency. The fact that the model tells otherwise can be the result of the inherent interactions between the different micro-architectural events. These interactions cannot be captured by a linear model. In addition, the approach of one model fits all is not realistic as previous research [18] proved that distinct performance behaviors or phases can exist even within a single workload. This translates to the necessity of using different performance models for different categories of workloads and phases within each workload.

Model trees seem to provide the best interpretable performance model. Each leaf node in a model tree represents a distinct workload or sections of workload class. The number in parentheses indicates the percent of the training set that falls into the corresponding leaf. The performance within each class is explained by a linear model. Interaction terms are captured in the tree structure, which can show for instance, that the performance effect of a given density of DTLB misses over an interval differs according to whether or not a significant number of L2 cache misses occur in the same interval. Figure 2 presents the performance analysis model tree obtained from applying M5' to the training set. This tree structure provides key insights to performance analysts. For example, at the root node of the tree, we can immediately see that the model identifies the level 2 cache misses (L2M) as the single event that most strongly affects performance. Among the events used in this study, L2 cache misses are known to be the longest latency event and so this result is highly intuitive. For more details on the interpretation of the model tree and its use for workload performance analysis, the reader is referred to [14].

The three other regression algorithms (LWR, ANN and SMOrg) are black-box techniques. Their outputs do not offer clear insights into the potential sources of performance degradation and, hence, the obtained models cannot be used to guide the performance optimization efforts as can be done with model trees.

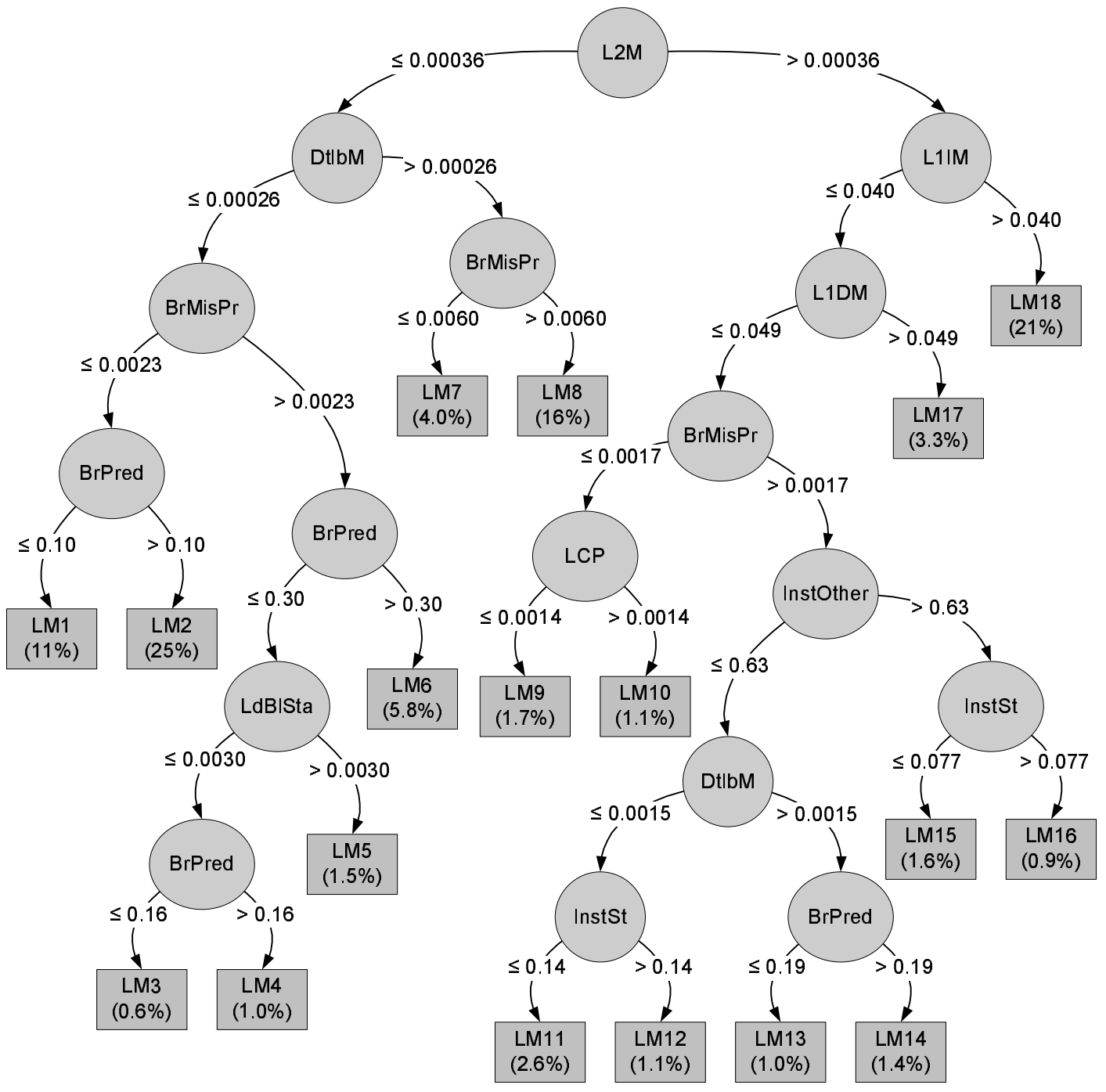


Figure 2. Performance analysis tree

5.2 Prediction Accuracy

To evaluate the prediction accuracy of the different algorithms for the performance data, we used 10-fold cross validation [11]. This technique consists of dividing the overall data in 10 disjoint subsets, or folds. Each algorithm is then trained using 9 of the subsets and evaluated using the tenth subset. The process is repeated 10 times and each time, a different subset is used for testing and the remaining 9 subsets are used to train the model. The algorithm is evaluated by averaging the prediction metrics from the 10 different models. Several prediction metrics can be employed and we use the following common metrics :

- The Correlation Coefficient: This metric is based on the standard correlation coefficient and measures the extent of linear relationship between predicted (P) and actual (A) values. It is a dimensionless index that ranges from -1 to 1 with 1 corresponding to ideal correlation. The correlation coefficient C is given by:

$$C = \frac{Cov(P, A)}{\sigma_p \sigma_a}. \quad (2)$$

where $Cov(A, P)$ is the covariance between the predicted and the actual values, while σ_p and σ_a are their respective standard deviations.

- Root Mean Squared Error (RMSE): This error measure is used in the determination of confidence intervals. Measured in the same unit as that of the predicted quantity (in this case CPI), it ranges from 0 to infinity with 0 corresponding to the ideal situation. It is computed as:

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (p_i - a_i)^2}{N}} \quad (3)$$

where p_i and a_i are the predicted and actual CPIs for i^{th} test instance and N the number of instances.

- Mean Absolute Error (MAE): This error measure is similar to RMSE, except that it uses absolute error values instead of the squared errors, i.e.,

$$MAE = \frac{\sum_{i=1}^N |p_i - a_i|}{N}. \quad (4)$$

- Root Relative Squared Error (RRSE): The relative squared error is relative to what it would have been if a naive predictor had been used. In particular, this simple predictor is just the mean of the actual values. It takes the total squared error and normalizes it by dividing by the total squared error of the simple predictor. It is given by:

$$RRSE = \sqrt{\frac{\sum_{i=1}^N (p_i - a_i)^2}{\sum_{i=1}^N (\hat{a} - a_i)^2}} \quad (5)$$

where \hat{a} is the mean of the actual CPI values.

- Relative Absolute Error (RAE): This error is computed in a similar way to RSE. It is given by:

$$RAE = \frac{\sum_{i=1}^N |p_i - a_i|}{\sum_{i=1}^N |\hat{a} - a_i|}. \quad (6)$$

The value of RAE ranges from 0% to 100% with 0 being the ideal situation.

Table 2 gives the evaluation results for the different algorithms compared in this study.

The results indicate that all nonlinear regression methods have good prediction accuracy. All except the locally weighted linear regression result in a correlation coefficient exceeding 0.95 and a relative absolute error below 10%. It is also clear that the model tree methods, M5' with the default parameters and the modified version, demonstrate very competitive prediction accuracy. In the case of the modified algorithm, where prediction quality was traded off for more compact tree and improved interpretability, the predictions are still highly accurate.

Table 2. Prediction accuracy for different algorithms

Algorithm	Correlation	RMSE	MAE	RRSE	RAE
LinReg	0.9832	0.1745	0.0844	18.2284	11.342
ANN	0.9955	0.0902	0.0412	9.4254	5.5365
LWR	0.9119	0.3927	0.1931	41.0354	25.9568
M5'Default	0.9962	0.0836	0.0245	8.7301	3.2881
M5'Modified	0.9845	0.1676	0.0582	17.5153	7.8302
SMOreg	0.9758	0.2143	0.0702	22.3865	9.4344

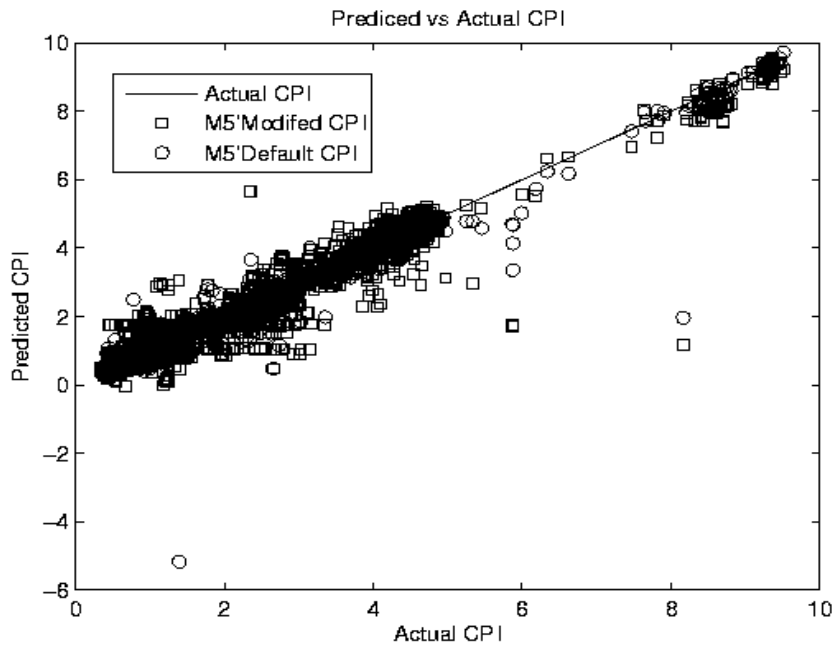


Figure 3. Predicted CPI vs. Actual CPI Using Default and Modified M5'

To illustrate the prediction accuracy of the different approaches, Figures 4 and 5 plot the predicted CPI versus the actual CPI for the cross-validation data. Note that the prediction is performed on data points in the test fold. In other words, the prediction on each data point is performed using a model that was built on training data that does not include the data point. Figure 3 plots the CPI predictions using the default and modified implementations of the M5' algorithm. Clearly, the two algorithms produces CPI predictions that are very close to the actual CPI values. Note, however, that the default implementation of M5' seems to perform poorly when it comes to outlier cases. The figure shows a case where the predicted CPI is negative.

Figure 4 plots the predicted against the actual CPI for the ANN and SMOreg (extension of SVM) algorithms. As noted earlier, artificial neural networks shows exceptionally good prediction accuracy. In the case of SVM-based regression, the predicted CPI shows a negative bias for large CPI. This seems also to be the case for LWR (locally weighted linear regression) and multi-linear regression as shown in Figure 5.

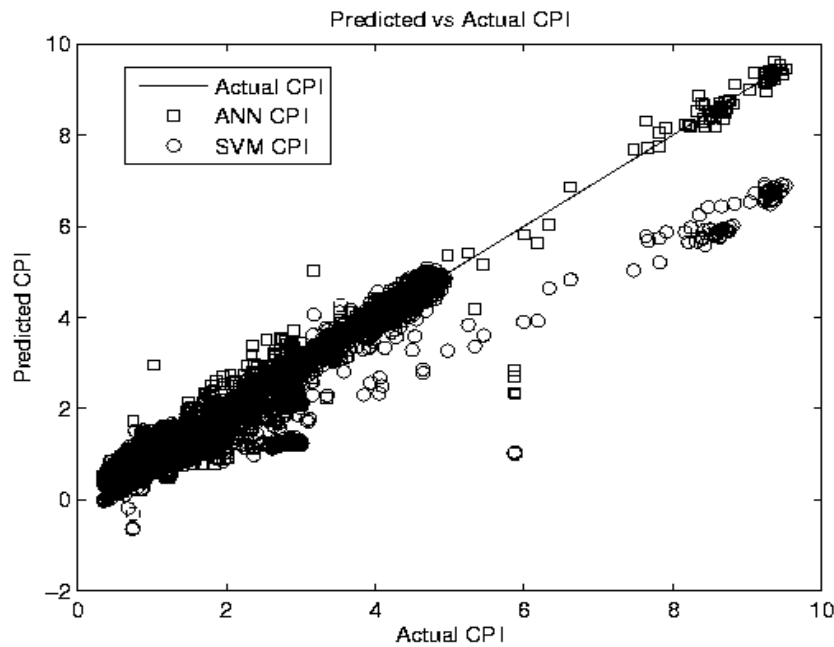


Figure 4. Predicted CPI vs. Actual CPI Using ANN and SVM

It is clear from these figures that the modified implementation of M5' does not sacrifice prediction accuracy for interpretability. It shows very competitive overall prediction and resilience to outlier cases.

6 Summary and Conclusions

In this paper, we presented several regression algorithms and assessed their appropriateness for computer architecture performance analysis for workload tuning. The results show that the M5' algorithm representing model trees demonstrates high prediction accuracy and excellent interpretability. These two properties allow the exploitation of the resulting models for the identification of main performance issues and the quantification of the potential gain from addressing each. The work presented here has many potential applications in the area of micro-architecture and software performance analysis. In particular, the model tree approach is worth investigating for use in processor design space exploration.

References

- [1] Standard performance evaluation corporation. SPEC CPU benchmark suite. <http://www.specbench.org/osg/cpu2006>, 2006.
- [2] C. G. Atkeson, A. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11:11–73, 1997.

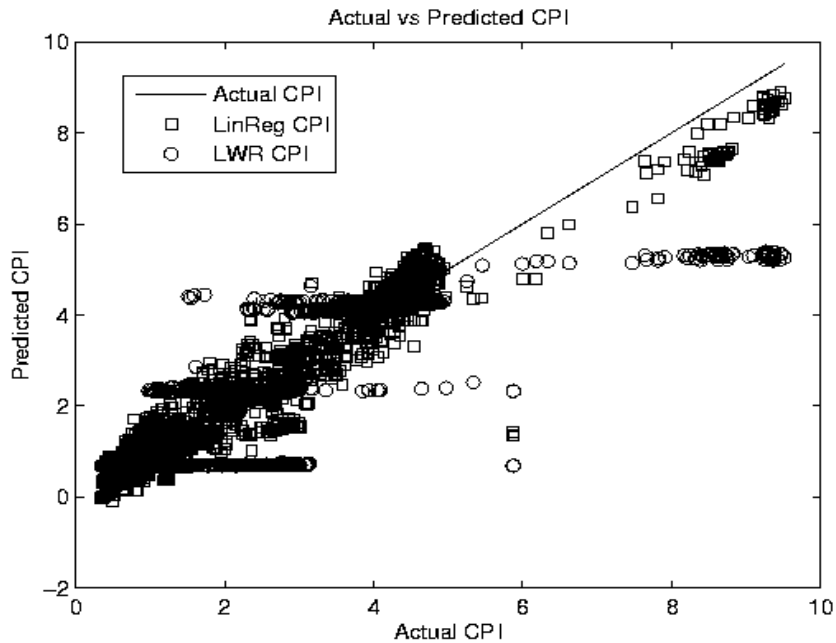


Figure 5. Predicted CPI vs. Actual CPI Using Multi-Linear Regression and Locally Weighted Linear Regression

- [3] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.
- [4] G. Cai, K. Chow, T. Nakanishi, J. Hall, and M. Barany. Multivariate power/performance analysis for high performance mobile microprocessor design. In *Proceedings of Power Driven Microarchitecture Workshop*, 1998.
- [5] K. Chow and J. Ding. Multivariate analysis of Pentium Pro processor. In *Proceedings of Intel Software Developers Conference*, 1997.
- [6] B. Fields, R. Bodick, M. Hill, and C. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization*, 1(3):272–304, 2004.
- [7] A. Hartstein and T. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA'02)*, 2002.
- [8] Intel. IA-32 intel architecture optimization: reference manual. <http://www.intel.com/design/Pentium4/manuals/248966.htm>.
- [9] Intel. Intel 64 and IA-32 architectures optimization reference manual, to appear. http://developer.intel.com/design/-Pentium4/manuals/index_new.htm, 2006.
- [10] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*, 2004.
- [11] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of 14th International Joint Conference on Artificial Intelligence*, 1995.
- [12] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counters. In *Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS05)*, 2005.
- [13] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [14] 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 *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'07)*, to appear, 2007.
- [15] J. Platt. Using sparseness and analytic qp to speed training of support vector machines. In *Proceedings of Advances in Neural Information Processing Systems (NIPS'99)*, 1999.
- [16] R. Quinlan. Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence (AI'92)*, 1992.
- [17] P. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection*. Wiley, 1995.
- [18] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of 30th Annual International Symposium on Computer Architecture (ISCA'03)*, 2003.

- [19] S. Shevade, S. Keerthi, C. Bhattacharyya, and K. Murthy. Hp caliper: A framework for performance analysis tools. *IEEE Concurrency*, 8(4), 2000.
- [20] L. Simonson and L. He. Micro-architecture performance estimation by formula. In *Proceedings of the 5th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'05)*, 2005.
- [21] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the International Symposium on Computer Architecture (ISCA'02)*, 2002.
- [22] Y. Wang and I. Witten. Inducing model trees for continuous classes. In *Proceedings of the 9th European Conf. on Machine Learning, Poster Papers*, 1997.
- [23] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2000.
- [24] J. Yi, D. Lilja, and D. Hawkins. A statistically-rigorous approach for improving simulation methodology. In *Proceedings of 9th IEEE Symposium on High Performance Computer Architecture*, 2003.