

Empirical Performance Models for Java Workloads

Rao, Pradeep

Department of Informatics, ISEE, Kyushu University | Institute of Systems and Information Technologies/Kyushu

Murakami, Kazuaki

Department of Informatics, ISEE, Kyushu University | Institute of Systems and Information Technologies/Kyushu

<https://hdl.handle.net/2324/13872>

出版情報 : International Conference on Architecture of Computing Systems. 22, pp.219-232, 2009-03-13

バージョン :

権利関係 :



Empirical Performance Models for Java Workloads

Pradeep Rao and Kazuaki Murakami

Department of Informatics, ISEE, Kyushu University, Japan.
Institute of Systems and Information Technologies/Kyushu, Japan.
{pradeep.rao,murakami}@isit.or.jp

Abstract. Java is widely deployed on a variety of processor architectures. Consequently, an understanding of microarchitecture level Java performance is critical to optimize current systems and to aid design and development of future processor architectures for Java. Although this is facilitated by a rich set of processor performance counters featured on several contemporary processors, complex processor microarchitecture structures and their interactions make it difficult to relate observed events to overall performance. This, coupled with the complexities associated with running Java over a virtual machine, further aggravates the situation. This paper explores and evaluates the effectiveness of empirical modeling for Java workloads. Our models use statistical regression techniques to relate overall Java system performance to various observed microarchitecture events and their interactions. Multivariate adaptive *regression splines* effectively capture non-linear and non-monotonic associations between the response and predictor variables. Our models are interpretable, easy to construct and exhibit high correlation/low errors between predicted and measured performance. Furthermore, empirical models afford additional insights into the characteristics of Java performance and the use of statistical techniques throughout this study allow us to assign confidence levels to our estimates of performance.

1 Introduction

The Java programming paradigm and its associated software engineering benefits have led to its deployment on a variety of computing platforms and the emergence of this class of workload is also recognized by several organizations, software vendors and research groups that have released benchmarks over the past few years to represent this space [1, 2]. The execution characteristics of Java applications have been shown to differ significantly from general purpose workloads [3], thus motivating a closer examination of Java performance components at the microarchitecture level. Performance analyses of Java workloads on a production platform enable optimizations and help identify performance bottlenecks to be addressed during the design and development of future systems for Java.

To facilitate performance analysis at the processor microarchitecture level, most modern processors provide access to a performance monitoring unit (PMU)

to track a wide variety of processor events. However, the complex microarchitecture of contemporary processors obfuscate interpretation of microarchitecture events and their relation to overall performance. The additional complexities associated with running Java over a virtual machine only aggravates the situation. While recent papers [4, 5, 3] study and highlight the influence of low-level processor events on overall Java performance, the approach adopted inherently ignores higher order *interactions* that may exist between parameters that affect performance, thus leading to skewed and possibly incorrect conclusions.

To address these issues, this study evaluates the effectiveness of statistically rigorous empirical models for Java performance analysis. Specifically, we use regression models to relate overall Java system performance to the observed microarchitecture events and their interactions. An *interaction* is said to exist when two events occur at the same time and the performance impact attributed to one event depends on the occurrence of the other event. For example, the performance impact of cache misses would depend on the magnitude of TLB misses since software page-table walk routines tend to pollute the cache. Our approach teases out several such nuances of production environments and quantifies their effect on overall performance. This paper highlights the importance of accounting for higher order interaction effects in performance analyses and shows how this can be facilitated by empirical models. We check for non-linear and non-monotonic influences in our workload and approximate it using models based on *regression splines*. Spline functions are piecewise polynomials used in curve fitting [6] and can be used to approximate a wide variety of functions.

We model Java workloads on a production platform based on the AMD Opteron processor and a recent 64bit Sun JavaSE virtual machine. We also investigate model *transferability* across virtual machines (VM) on our platform, *i.e.*, how accurately models built using a particular VM represent observed behavior using a different VM. Our results indicate that our models are able to faithfully predict performance on an alternate VM (IBM J9 JVM) with an average prediction error of $8 \pm 0.6\%$ at 95% confidence. While we demonstrate the construction and use of empirical models for Java performance analyses on the Opteron processor, the methodology is generic enough to extend to other platforms and workloads.

2 Preliminaries

This section presents the background required to understand the experiments and results presented in this paper. Further details may be found in several texts on statistical methods [6, 7].

Regression analysis is a statistical tool that expresses the relationship between a *response* variable and its *predictors* over a sample space. In its simplest generalized form, the response variable y may be related *linearly* to p independent predictors or *regressor* variables (Z_1, Z_2, \dots, Z_p) as expressed by the model in Eq. 1.

$$y = \beta_0 + \sum_{i=1}^p \beta_i Z_i + \epsilon \quad (1)$$

The parameters $\{\beta_i | 0 \leq i \leq p\}$ are called the *regression coefficients* and represent the expected change in response y per unit change in $\{Z_i | 1 \leq i \leq p\}$ when all the remaining variables $\{Z_j | j \neq i\}$ are held constant. The *residual* or the error due to lack of fit is denoted by ϵ . The model describes a hyperplane in the k -dimensional space of the regressor variables Z_i and β_0 can be interpreted as the intercept of the response surface with the y -axis. Linear regression models are commonly used to estimate parameter significance and also to predict the response variable at arbitrary points in the design space.

Often, the predictors *interact*, *i.e.*, the response of y to a change in Z_i depends on the value of Z_j . In such cases the model in Eq. 1 can be easily extended to model such two-factor interactions as follows:

$$y = \beta_0 + \sum_{i=1}^p \beta_i Z_i + \sum_{i=1}^p \sum_{j=i}^p \beta_{ij} Z_i Z_j + \epsilon \quad (2)$$

Similarly, higher order interactions can be included in the equation above and the complete linear regression model can be represented as a sum of p terms with the generic form:

$$y = \beta_0 + \beta_1 X_{n1} + \beta_2 X_{n2} + \dots \beta_p X_{np} + \epsilon_n \quad (3)$$

where, the subscript n identifies the *trial* and the subscript p denotes the predictor variable X . In matrix terms, the above equation can be written as:

$$\mathbf{y} = \mathbf{X}\beta + \epsilon \quad (4)$$

Hence, given the linear model with p terms, the data set from the design matrix \mathbf{X} and the response vector \mathbf{y} , the *least squares* estimates of the partial regression coefficients $\hat{\beta} = (\beta_0, \beta_1, \dots, \beta_{p-1})$ can be computed using Eq.5.

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} \quad (5)$$

2.1 Multivariate Adaptive Regression Splines (MARS)

Spline functions are piecewise polynomials used in curve fitting [6], *i.e.*, they are polynomials within connected intervals of \mathbf{X} , the endpoints of which are called *knots*. MARS [8] exploit recursive partitioning techniques to derive functions that approximate arbitrarily complex relationships. The domain is recursively partitioned into disjoint subspaces and the response in each region is described using simple functions of predictor variables. The final model is a linear combination of these *basis* functions.

Assume that the predictor X is divided into p regions. The model is represented using the basis functions B_i and the regression coefficients β_i in the form:

$$\hat{y} = \beta_0 + \sum_{i=1}^p \beta_i B_i(X) \quad (6)$$

The MARS approach uses *q-order splines* [8] as basis functions and the normalized coefficients are estimates of the influence of the variables and their interactions on the overall response. Consequently, these models are not only highly accurate, but also interpretable.

2.2 Model Building

We fit a model consisting of appropriate main effects and their second order interactions using both linear regression and MARS. Two or more interacting terms are indicated using an asterisk '*' between them. Since not all interactions are meaningful, based on microarchitectural insights (*e.g.*, l2.i miss*l2.d miss, dl1.miss*l2.i miss *etc.*), we remove these terms from our model.

Model Simplification/Pruning: Overfitting refers to a model that fits the data well, but is unable to generalize beyond that. To avoid overfitting, we reduce model complexity using metrics that estimate the ability of a model to generalize. Our model simplification uses stepwise search based on the *Bayesian Information Criterion* (BIC) which penalizes overfit models. The criterion is defined by the following relation:

$$BIC = \frac{n + p(\log(n) - 1)}{n(n - p)} SSE \quad (7)$$

where, $SSE = \sum_{i=1}^p (y_i - \hat{y}_i)^2$ is the error sum of squares, p is the number of parameters in the model, and n is the number of samples. The model with the minimal BIC tries to find an optimal compromise between model fit and model complexity. MARS models use the modified cross-validation procedure developed in [9] to determine the basis functions to be included in the model.

2.3 Model Diagnostics

The *residuals* $\{\mathbf{r} | \mathbf{r} = \mathbf{y} - \mathbf{X}\hat{\beta}\}$, are the differences between the data and the fitted values. As a consequence of least squares estimation of β , the residuals will be uncorrelated with all predictors (and intercept, if any) and can be used to diagnose problems with the model. The fit of the model can be summarized by the residual standard deviation, $\hat{\sigma} = \sqrt{\sum_{i=1}^n r_i^2 / (n - p)}$ and R^2 in Eq.8, the fraction of variation 'expressed' by the model, and

$$R^2 = 1 - \frac{\hat{\sigma}^2}{s_y^2} \quad (8)$$

where, s_y is the standard deviation of data, and $(n - p)$ are referred to as the *degrees of freedom*.

The *goodness of fit* of a reduced model (R^2) with respect to an original model (R_o^2) can be estimated with the F-test using the statistic F in Eq.9.

$$F_{k,n-p-1} = \frac{(R^2 - R_o^2)/k}{(1 - R^2)/(n - p - 1)} \quad (9)$$

where, k is the difference between the degrees of freedom of the original model and that of the reduced model. Given that the statistic follows the F-distribution with $(k, n - p - 1)$ degrees of freedom, the *p-value* is defined as the probability $P(X > |c|)$, where c is a constant. Thus, a small p-value for the F-test leads us to reject the *null-hypothesis*, suggesting that additional predictors in the larger model are statistically significant in predicting the response.

Prediction Accuracy Measuring the accuracy of model predictions is a good way to determine the quality of the models built. Prediction accuracy for a given model is determined using *cross validation* as in other modeling studies [10]. Cross validation involves randomly partitioning the observations into n sets ($n = 5$ here). The models are then built using $n - 1$ sets and the last set is used for testing the prediction accuracy. This process is repeated n times, using a different set each time. Overall accuracy is determined by averaging the prediction metrics across all sets/folds. We use the following metrics to evaluate model prediction.

1. *Prediction error*, expressed as a percentage of the observed performance ($100 * |y_i - \hat{y}_i|/y_i$). While this metric captures the central tendency, we also capture the variation in errors using the confidence interval metric [11] (CIM). CIM is the ratio of the confidence interval for the mean prediction error expressed as a percentage of the mean observed test case performance. This is evaluated using the *paired t-test* at a confidence level of $\alpha = 0.95$.
2. *Correlation metrics* capture the degree of linear association between the model predicted performance and observed performance using statistical tests. Specifically, we use: (a) Pearson's product moment correlation coefficient (r), and (b) Spearman's rank correlation coefficient (ρ), a metric similar to Pearson's correlation, except that the samples are converted into ranks before computing the coefficient. Additionally, Spearman's statistic does not assume that the variables are normally distributed. The coefficients range from +1, indicating a perfect positive linear relationship, to -1, indicating a perfect negative linear relationship between the variables. A score of zero implies that there is no linear relation between the two variables

3 Experimental Methodology

Measurement Methodology The models we build are based on measurements on the 64bit AMD Opteron processor which features a performance monitoring unit (PMU) that can be configured to measure various microarchitecture events using four 48bit counters [12]. The Linux kernel v2.6.25 is modified to support

Event description	abbreviation
General	
Number of unhaltd clock cycles	cycles
Number of retired operations	ops
Number of all data prefetches	d.prefetch
Branch prediction events	
Number of branch instructions retired	br
Number of retired mispredicted branch instructions	br.mispred
Number of return stack overflows	rs.ovfl
Stall events	
Number of instruction fetch unit stalls	if.stall
Number of dispatch stalls (combined)	dispatch.stall
Number of reorder buffer full stall cycles	rob.full
Number of reservation station full stall cycles	rs.full
Number of floating point unit (FPU) full stall cycles	fpu.full
Number of load-store unit full stall cycles	ls.full
Cache, TLB and page miss events	
Number of data cache accesses	dc.access
Number of data cache misses	dc.miss
Number of I-cache fetches	ic.access
Number of I-cache misses	ic.miss
Number of all L2 cache misses	l2.miss
Number of L2 cache misses due to instructions	l2.imiss
Number of L2 cache misses due to data	l2.dmiss
Number of L2 cache misses due to TLB walk	l2.tlmiss
Number DC fills	d.fills
Number of IC fills	i.fills
Number of L1 I-TLB misses but L2 I-TLB hits	l2itlb.hit
Number of L1 I-TLB misses and L2 I-TLB misses	l2itlb.miss
Number of L1 D-TLB misses but L2 D-TLB hits	l2dtlb.hit
Number of L1 D-TLB misses and L2 D-TLB misses	l2dtlb.miss
Number of page misses	pg.miss

Table 1. Processor events measured

access to the processor performance monitoring unit (PMU) using the *perfmon2* [13] interface. The *perfmon2* interface provides additional features which include (1) extending the 48bit physical counters into 64bit virtual counters, (2) aggregating counts across *threads*, *forks* and *execs* and (3) low-overhead PMU sampling support. We choose 27 processor microarchitecture events (Tab.1) to measure and model Java applications. These metrics are similar to those used in earlier Java performance analysis studies referred to in Section 6. These events and their acronyms used in this paper are shown in Tab.1. These events can be roughly classified, based on the aspect of the microarchitecture that they characterize, into general, branch, processor core and cache/memory hierarchy events.

Measurements are taken at both user and kernel level. Samples are collected at intervals of every 10M retired instructions to capture fine grain behavior [14]. Consequently, all counts are effectively normalized to the number of instructions (per 10M). Program execution is blocked on counter overflows and during sample processing. All experiments employ the following techniques to reduce measurement noise due to extraneous factors: (1) all experiments are performed in single user mode with all unnecessary processes and services turned off, and (2) we disable processor voltage/frequency scaling in the Linux kernel.

Since the Opteron PMU supports only four event counts at a time, multiple runs are required to capture the complete event set listed in Tab.1. Since

Benchmark (BM)	Description	H_{min}
compiler	OpenJDK front end compiler	51MB
compress	Modified Lempel-Ziv compression	29MB
crypto	Encryption/Decryption {aes/rsa/signverify}	2625KB
derby	open source database	250MB
mpegaudio	mp3 decoding	2625KB
serial	serialize/deserialize primitives and objects using JBoss data	69MB
startup	starts each benchmark (except derby) for one operation	2625KB
sunflow	multithreaded global illumination rendering system	2625KB
xml	Application and verification of style sheets to XML documents	15MB

Table 2. SPEC JVM 2008 benchmark applications

the JVMs exploit several dynamic/runtime techniques, we performed the following experiment to determine if there was any significant statistical difference between runs under our experimental conditions: we took three random PMU sampled runs as described above for each benchmark and used a statistical test [7] (paired t-test) to test for the null hypothesis that the difference between the means, for each pair of sampled runs, is zero. Our experiments showed that there was insufficient evidence to reject the null hypothesis that the difference between the means for the two runs is zero, thus asserting the validity of our approach.

Benchmarks and Virtual Machine We use the fixed size (`--lagom`) workload from the recently released SPEC JVM 2008 benchmark [1] suite. The constituent applications and their brief descriptions are indicated in Tab.2.

For our model construction experiments we use the Sun JavaSE runtime environment v1.6.0. The JVM is forced to run in the 64bit server mode All experiments consider a per benchmark heap size as recommended in [5] and the minimum heap size, H_{min} is indicated in Tab.2. We take measurements at three heap sizes – H_{min} , $3H_{min}$ and $6H_{min}$. The minimum heap size is determined empirically to be the least heap size at which the Java application can execute without any `OutOfMemory` errors. However, this is subject to the caveat that the least heap allowed by the SunTM JVM is 2625kB. We also determine model transferability using measurements from an alternate virtual machine. For this purpose we use v1.6.0 of the IBM J9 VM for the AMD x86_64 platform.

4 Statistical Analyses

We first use descriptive and inferential statistics to examine and summarize measurement data. Such analyses also help identify relevant predictors for modeling.

The interrelation between the key events is illustrated in the dendrogram in Fig.1(a). These are arrived at using hierarchical variable clustering based on squared Spearman’s rank correlation coefficients as similarity measures [6]. This statistic is a measure of similarity and a larger ρ^2 indicates higher correlation between the predictors connected on the figure. The horizontal line connecting the predictors marks the value of ρ^2 when extended to the vertical axis.

In cases where overfitting is a concern (defined in Section 2.2), redundant predictors can be eliminated by choosing only one predictor from a pair of highly

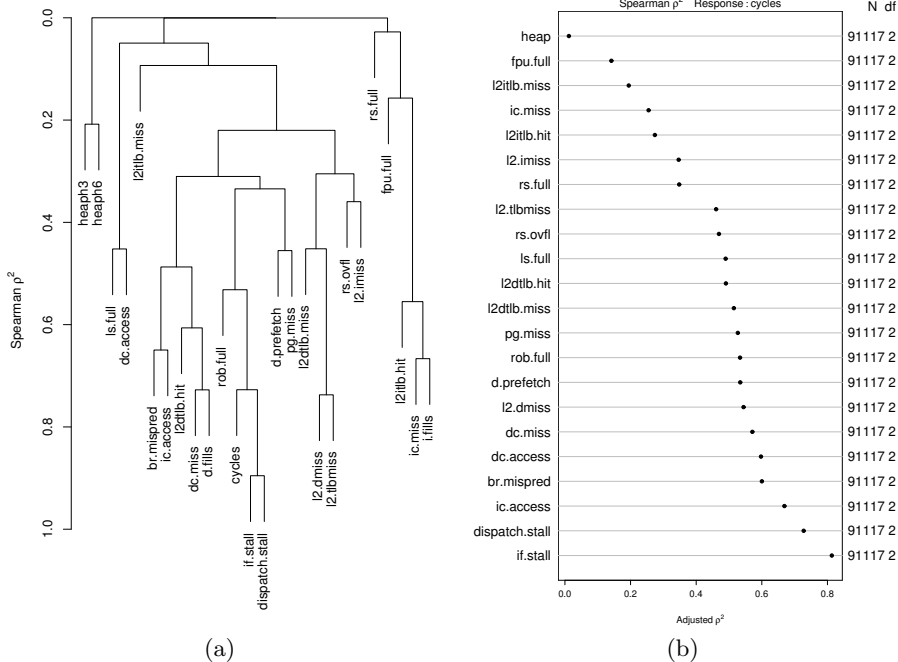


Fig. 1. (a) Predictor clustering and (b) Marginal relationship between predictors and response (cpu cycles).

correlated predictors. This is also effective when the predictors are collinear, since *multicollinearity* [6] influences effect estimation by artificially inflating p-values even though the variable is significant. However, multicollinearity doesn't pose a problem while predicting and the predictions will still be accurate with R^2 (Sec.2.3) capturing how well the model predicts the response y (*cycles*).

Analyzing the clusters in Fig.1(a), we find that *if.stalls* and *dispatch.stalls* are highly correlated to each other and also to overall performance (cpu cycles). We also find that *if.stalls* are indeed a significant component affecting overall performance in the models that we develop in the following section. This suggests that Java application performance would benefit considerably from processor front-end engineering and from latency hiding techniques. A few other highly correlated predictor pairs are intuitive – *e.g.*, *dc.miss* and *d.fills* are expected to be correlated as most D-cache misses would typically be hits in the L2 cache (*d.fills* are equivalent to L2 cache hits due to data and *i.fills* are the equivalent for instructions). Hence, we do not include *i.fills*, *d.fills* and *dispatch.stalls* while modeling.

Fig.1(b) shows the strength of marginal relationships between predictors and the performance response using the non-monotonic (quadratic in rank) generalization of the Spearman’s rank correlation [6]. The lack of fit for predictors with higher ρ^2 will have a larger negative impact on performance predictions. In

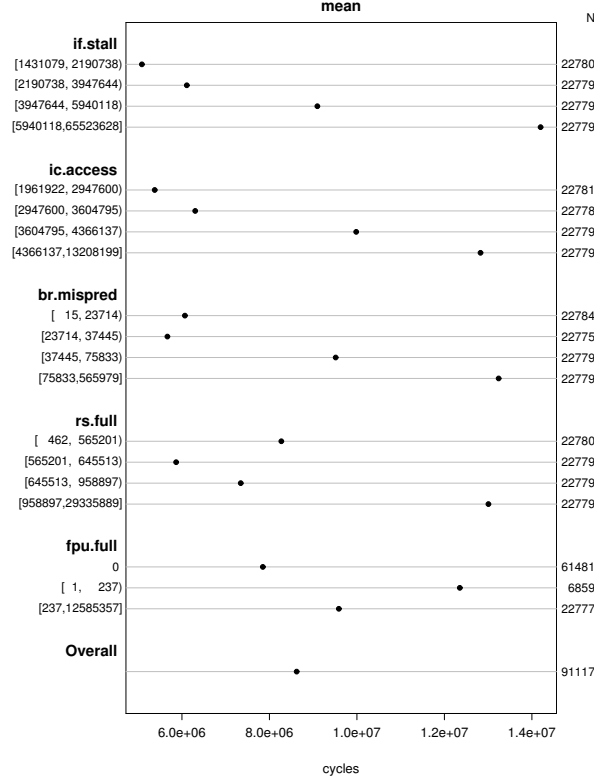


Fig. 2. Summary statistics for selected predictors.

cases where it is difficult to assess how the predictor bears on overall performance based on microarchitectural insights, it is helpful to allocate larger number of *knots* (see Sec.2.1) to predictors that have a higher impact on the response. The figure indicates that for microarchitectural predictors, a lack of fit will be more consequential (in decreasing order of impact) for *if.stall*, *ic.access*, *br.mispred*, *dc.access*, etc.

Fig.2 examines descriptive associations of the predictors by stratifying the mean response by quartiles for each predictor. The left axes indicate the quantiles and the right indicate the average number of measurements in that range. The x-axis corresponds to the mean response (cycles) in each quartile. Such a plot helps identify strong associations, linearity and monotonic and non-monotonic behaviors. For the sake of brevity, we indicate only a few selected predictors and the overall response. The plots indicate that the predictors *if.stalls* and *ic.access* have strong monotonic response while the stall parameters *br.mispred*, *rs.full*, *fpu.full*, *rs.full* all have non-monotonic behaviors. This is also consistent with the non-monotonic behavior reported by [15] for stall parameters for general purpose applications using the IBM Power processor simulator.

5 Model Evaluation

Model building and evaluation routines are written using the *R language* and its associated environment to leverage the rich set of statistical functions and packages available in that environment. To build MARS models, we use routines from the *polspline* package [9].

The Java workload used in this study produce over 1.8M samples (over 1.5GB of data) when sampling at a granularity of 10M instructions. To enable greater flexibility when experimenting with model building, we examine if sampling at lower frequencies or using fewer samples affects the quality of the models produced. We determine this by subsampling, *i.e.*, we take a random sample from the collected samples which constitutes a small percentage of the overall samples. We then build MARS models for each subsample and evaluate the model errors using cross-validation (Sec.2.3). The sample sizes chosen for subsampling are $\{0.5, 1, 2, 4, 8, 10, 15, 20\}$ % of all samples for each benchmark.

We find that the mean prediction errors are $< 1\%$ for all Java applications considered across our chosen subsample sizes. Furthermore, the error confidence intervals are within $\pm 4\%$ at 95% confidence levels. This observation can be used to either reduce the number of samples used for analysis or alternately, the frequency at which the PMU is sampled can be lowered to reduce the measurement interference that arises due to frequent sample collection. In general we observe that (not shown) 400-500 samples per benchmark provide MARS models with prediction errors $< 5\%$. To make multiple model construction tractable we conservatively choose the 5% subsample level for further analyses, unless mentioned otherwise.

Effect of higher order interactions We evaluate the effect of higher order interactions by comparing mean prediction error and its confidence interval for linear regression models with and without interactions between predictors. A model consisting of main effects only is designated as a I-order model, a model with main effects and interactions (Sec.2) between two predictors is designated as a II-order model. Finally, a II-order model with interactions between 3 predictors is designated as a III-order model. The prediction errors are evaluated on a common test unit consisting of 200 random samples for each benchmark and all confidence intervals and statistical tests are specified at 95% ($\alpha = 0.95$) unless mentioned otherwise.

The average R^2 value for the I-order model is 0.67, while that for the II-order model is 0.82. The R^2 measure indicates that the II-order model does a better job (by 22%) of *explaining* the measured variance. An F-test (Sec.2.3), comparing the II-order model and the model with only main effects indicates strong statistical evidence suggesting a lack of fit from some two way interaction term(s) (*i.e.*, null hypothesis stands rejected, Sec.2.3, Eq.9). The significant two way interaction terms can be deduced using the procedure described in Sec.2.2. These tests indicate the importance of accounting for second order effects in performance analysis studies. However, we note that a model with all II-order

Benchmark	MARS models					LR models				
	R^2	error %	CIM %	ρ	r	R^2	error %	CIM %	ρ	r
compiler	0.14	0.78	1.44	0.13	0.37	0.25	0.58	1.33	0.47	0.51
compress	0.56	0.46	0.83	0.56	0.68	0.98	0.35	0.79	0.73	0.75
crypto	0.66	0.32	0.69	0.55	0.76	0.91	0.27	0.64	0.73	0.81
derby	0.66	0.31	0.76	0.56	0.77	0.64	0.27	0.71	0.74	0.82
mpegaudio	0.72	0.27	0.65	0.48	0.81	0.95	0.22	0.61	0.65	0.85
serial	0.76	0.23	0.57	0.53	0.84	0.87	0.23	0.57	0.63	0.86
startup	0.75	0.27	0.70	0.56	0.84	0.64	0.32	0.72	0.66	0.85
sunflow	0.71	0.26	0.67	0.54	0.82	0.48	0.29	0.69	0.63	0.83
xml	0.70	0.28	0.74	0.56	0.82	0.66	0.30	0.75	0.66	0.83

Table 3. Model accuracy

effects blindly included is significantly overfit as they exhibit high prediction errors. Hence, it is imperative to prune the model using the method described in Sec.2.2.

We also perform a F-test between a III-order model and a II-order model (only main effects and two-way interactions). The results indicate the presence of third-order effects that should be included in the models. Though the large number of samples collected at 10M instruction intervals can afford building models with third-order effects, we do not analyze them further here since the prediction errors for the pruned model with only second order interactions are already close to the measurement error.

Comparing MARS and Linear Regression Models Tab.3 indicates the prediction accuracies for the two regression algorithms considered in this paper, using the metrics described in Sec.2.3. We observe that the MARS model accuracies are comparable to those of II-order LR models suggesting that interaction effects are able to reasonably explain non-linear and non-monotonic behavior of the predictors observed in Sec.4. We note that the stepwise reduction algorithm used a maximum of 100 pruning steps (Sec.2.2). We are currently investigating the effect of increasing the number of steps on the prediction accuracy of the pruned LR model. II-order models that are not simplified result in high prediction errors due to overfitting (Sec.2.2). We also highlight the significant difference in model build times for the two algorithms. The MARS algorithm is, on an average, 7x faster to build than the linear regression algorithm (100 steps) on the same computation platform (based on Opteron 254 processor). Most of the time for the linear regression algorithm is spent in model simplification as described in Sec.2.2.

Interpreting the Model Several techniques [6] are available to interpret the models and carry out sensitivity analyses. We estimate the effect of a predictor using the difference in the predicted response (\hat{y}) at the lower and upper quartiles of X , holding all other X 's at their median values. Fig.3 shows the summary of the predictor effects and the 0.95 confidence intervals for the mean effects. We find that the front-end parameters (*if.stall* and *br.mispred*) are predominant factors that influence Java performance. Further analyses are detailed in [16].

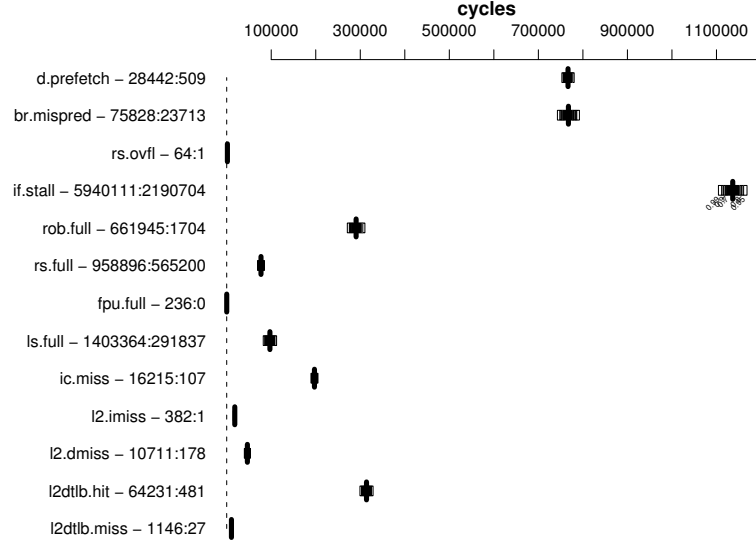


Fig. 3. Summary of predictor effects using interquartile ranges

Benchmark	compiler	compress	crypto	derby	mpegaudio	serial	startup	sunflow	xml
Error %	5.31	5.09	11.01	10.12	9.39	9.11	8.48	7.55	6.99
CIM %	0.53	0.62	0.54	0.63	0.57	0.59	0.61	0.59	0.64

Table 4. MARS model transferability to IBM J9 JVM

Model Transferability The usefulness of the models developed on a particular platform can be enhanced if they can also be used to estimate/predict performance for alternate VMs. We evaluate MARS model transferability for one other production VM on our platform – the 64bit IBM J9 JVM. The prediction errors and the 95% confidence intervals for the errors are indicated in Tab.4. The errors average 8% across all benchmarks with the confidence intervals at $\pm 0.6\%$. Though one cannot generalize this result, it is useful to know that models are transferable across a similar class of virtual machines. This is in conformance with the results presented in [4] that shows the existence of VM clusters using principal component analysis.

6 Related Work

This section places the contributions of this study in perspective with respect to related work. Georges et.al. [17] propose rigorous measurement methodologies for Java performance evaluation, but do not model reasons for observed performance. We adopt the recommendations made in [17] and also extend the statistical toolbox to include empirical models and regression splines for Java performance analysis.

Studies on processor performance modeling [18, 15] are based on software simulation of general purpose workloads using a processor model. Due to the complexity of the workload, these studies use partial traces and reduced input data sets respectively. We perceive several drawbacks while applying the methodology adopted by these papers to model Java performance: (1) contemporary processors are far too complex to be modeled accurately in a simulator and thus, the generated models may not be representative of actual performance, (2) a microarchitecture simulator typically needs to implement several additional features [19] to support the Java virtual machine and to make it feasible to use simulation for model construction, (3) the use of reduced input data sets with Java workloads has been shown [4] to poorly represent Java execution on a real machine. In contrast, our work (a) uses measurements from real machine execution using a recent production JVM, (b) uses large and realistic data inputs and all Java applications are run to completion. Further, to the best of our knowledge, this is the first study that extends rigorous statistical modeling to advance accurate Java workload performance analysis and characterization.

Since our focus is on model interpretability, we do not consider models based on neural networks and support vector machines proposed in [20, 21] for workload characterization. We also do not consider model tree based approaches for performance analysis as proposed in [10], since model trees partition each predictor space into exactly two disjoint subspaces, while our approach using splines can assign subspaces to predictors based on the amount of data available to fit the function and also based on examining the strength of the predictor on the response. In our approach a highly non-linear predictor can be assigned more subspaces to approximate the function based on its bearing on the response and the impact of this decision is evident in the low prediction errors evaluated at high confidence levels (typically at 95% unless mentioned otherwise).

7 Summary and Outlook

In this paper, we develop empirical models based on performance counter data for Java performance analysis. We show that a small sample of a few events can lead to highly accurate performance predictions. We adopt several guidelines listed in [17] for Java performance analysis and extend the statistical toolbox to include empirical modeling for complex workloads in general and Java applications in particular. We show the importance of accounting for higher order effects during performance evaluation studies, especially for Java applications and we evaluate model transferability across an alternate VM. A variation of MARS [9] can also be used to model multiple responses (e.g., performance and power) and our current work investigates the possibility of joint models for power and performance estimation/prediction.

Acknowledgments. The authors wish to thank ISIT/Kyushu (JET Programme) for supporting this research, Prof. Nandy, Prof. Matthew Jacob (IISc) and the anonymous reviewers for their comments.

References

1. Standard Performance Evaluation Corporation: JVM 2008. <http://www.spec.org/jvm2008/>.
2. Blackburn, S., Garner, R., McKinley, K.: The DaCapo benchmarks: Java benchmarking development and analysis. In: Proc. OOPSLA. (2006) 169–190
3. Georges, A., Eeckhout, L., De Bosschere, K.: Comparing low-level behavior of SPEC CPU and Java workloads. In: Proc. Asia-Pacific Computer Systems Architecture Conference. (2005) 669–679
4. Eeckhout, L., Georges, A., Bosschere, K.D.: How Java programs interact with virtual machines at the microarchitectural level. In: Proc. OOPSLA. (2003) 169–186
5. Blackburn, S.M., Cheng, P., McKinley, K.S.: Myths and realities: The performance impact of garbage collection. In: Proc. SIGMETRICS. (2004) 25–36
6. Harrell, F.: Regression modeling strategies, Springer (2001)
7. Canavos, G.: Applied probability and statistical methods. Little Brown and Company (1984)
8. Friedman, J.: Multivariate adaptive regression splines. *The Annals of Statistics* **19** (1991)
9. Kooperberg, C., O’Connor, M.: Polymars. (2001)
10. Ould-Ahmed-Vall, E., Woodlee, J., Yount, C., Doshi, K.A., Abraham, S.: Using model trees for computer architecture performance analysis of software applications. In: Proc. ISPASS. (2007) 116–125
11. Kodakara, S., Kim, J., Lilja, D., Hawkins, D., Hsu, W., Yew, P.: CIM: A reliable metric for evaluating program phase classifications. *IEEE Computer Architecture Letters* **6**(1) (2007)
12. AMD: BIOS and kernel developer’s guide for AMD Opteron processors. (2006)
13. Perfmon2. <http://perfmon2.sourceforge.net/>.
14. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. *IEEE MICRO* (2003) 84–93
15. Lee, B., Brooks, D.: Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: Proc. ASPLOS. (2006) 185–194
16. Rao, P., Murakami, K.: Empirical performance models for Java workloads. Technical Report, System LSI Laboratory, Kyushu University (2008)
17. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: Proc. OOPSLA. (2007) 57–76
18. Joseph, P., Vaswani, K., Thazhuthaveetil, M.: Construction and use of linear regression models for processor performance analysis. In: Proc. HPCA. (2006) 99–108
19. Rao, P., Murakami, K.: A sampling microarchitecture simulator for Java workloads. In: Proc. Workshop on TIMERS-1 held in conjunction with IEEE ISPASS. (2008)
20. Yoo, R., Lee, H., Chow, K., Lee, H.: Constructing a non-linear model with neural networks for workload characterization. In: Proc. IEEE International Symposium on Workload Characterization. (2006) 150–159
21. Ould-Ahmed-Vall, E., Woodler, J., Yount, C., Doshi, K.A.: On the comparison of regression algorithms for computer architecture performance analysis of software applications. In: Proc. Workshop on SMART co-located with HiPEAC. (2007)