An Architectural Assessment of SPEC CPU Benchmark Relevance

Benjamin C. Lee
Computer Science 261
Division of Engineering and Applied Sciences
Harvard University
Cambridge, Massachusetts, USA
bclee@deas.harvard.edu

Abstract

*SPEC compute intensive benchmarks are often used to evaluate processors in high-performance systems. However, such evaluations are valid only if these benchmarks are representative of more comprehensive real workloads. I present a comparative **architectural analysis** of SPEC CPU benchmarks and the more realistic SPEC Java Server benchmark. This analysis indicates the integer subset of CPU benchmarks are broadly representative of the branching and cache characteristics of server and business workloads.*

*I also leverage the collected data to perform a **statistical regression analysis**, estimating an application's performance as a linear function of its observed architectural characteristics. A linear model derived from simulated SPEC CPU data appears to fit the observed data quite well, with 94 percent of performance variance explained by four predictors that incorporate cache and queue usage.*

The linear model predicts server benchmark performance to within 2 percent of the observed value. Despite the regression model's close fit to the observed data, its predictive ability is constrained by the trace collection facility. It is infeasible to use the performance of one trace suite to predict the performance of another trace suite if the underlying trace collection processes differ across suites.

1. Introduction

Challenges in efficient and comparable performance evaluation led to a standardized set of relevant benchmark suites for high-performance computer systems. One such suite, SPECcpu, is extensively employed in both microarchitecture and systems communities, but is often considered not representative of the workloads executed by real applications. Despite the frequent claims of irrelevance, standardization has led to widespread adoption of the SPECcpu benchmarks. Furthermore, developing alternative benchmarks remains a costly venture. Comparing the architectural characteristics of SPECcpu to those of a more realistic workload, one can assess the relevance of SPECcpu for modern applications.

In particular, I describe the performance for a subset of the compute intensive SPECcpu benchmarks in terms of their architectural characteristics, such as cache activity and branch prediction rates (Section 4). Throughout the discussion, I will make comparisons with the more realistic SPECjbb, a benchmark for evaluating performance of servers running typical Java business applications. Furthermore, I formulate statistical regression models to capture SPECcpu performance as a function of its architectural characteristics and evaluate its predictive ability for other benchmarks. (Section 6).

The following summarizes experimental results from architectural simulation (Section 3):

1. Performance evaluations of SPECcpu must differentiate between integer and floating-point benchmarks as their architectural characteristics are distinctly different. Any statistical aggregation (e.g., mean or median) of the data across these subsets will obscure the system's true performance.
2. The relatively irregular control flow of integer benchmarks is characterized by more branch instructions with relatively poor branch prediction, leading to higher instruction cache misses.
3. The relatively regular control flow of floating-point benchmarks is characterized by greater branch prediction accuracy. These benchmarks incur more frequent data cache misses as their access patterns induce more cache conflicts relative to integer workloads.
4. The architectural characteristics of SPECjbb, the java server benchmark, generally resemble the integer subset of SPECcpu. However, SPECcpu under-represents the instruction cache miss rate and over-represents the data memory traffic of SPECjbb.

5. The performance of SPECcpu benchmarks may be captured with statistical regression models (94% of performance variation is explained by predictors drawn from architectural characteristics). SPECjbb performance is predicted to within 2 percent, but predictive ability is limited for other benchmark traces by differences in trace facilities and instruction sampling methodologies.

2. Related Work

There has been significant work in studying the performance of SPEC benchmarks. These benchmarks have been employed to such an extent that enumerating all the architecture and systems projects that employ them is infeasible. There has been, however, relevant work in studying the SPEC benchmarks for their own sake.

2.1. SPEC Workload Studies

A subset of researchers in the architecture community believe SPEC benchmarks are unnecessarily long for simulation purposes. A short, synthetic benchmark that is statistically equivalent to a longer, real benchmark is one approach to reduce simulation time. Statistical equivalence often refers to equal instruction mix and frequencies. Thus, most studies and characterizations of the SPEC benchmarks aim to identify their smaller equivalent subsets. Eeckhout, *et al.*, have studied statistical simulation in the context of workloads and benchmarks for architectural simulators [1, 2, 3, 4]. Nussbaum, *et al.* has examined similar statistical approaches for simulating superscalar and symmetric multiprocessor systems [5, 6]. Both researchers claim detailed microarchitectural simulations for specific benchmarks are not feasible early in the design process. Instead, benchmarks should be profiled to obtain relevant program characteristics, such as instruction mix and data dependencies between instructions. A smaller synthetic benchmark is then constructed with similar characteristics.

In contrast, I profile benchmarks for microarchitectural characteristics instead of higher level program characteristics. Furthermore, I characterize complete benchmarks for comparison to other complete benchmarks believed to be more representative of real workloads, aiming to evaluate the relevance of the SPEC benchmarks. Researchers studying synthetic benchmarks based on the SPEC suites implicitly make strong assumptions about the relevance of SPEC benchmarks to synthesize new benchmarks that are inherently less relevant at the application level.

Henning provided one of the first performance evaluations for SPECcpu 2000 on the Alpha 21164 and Alpha 21264 chips, comparing integer and floating-point benchmark performance and considering level-3 cache misses [7]. Although evaluating performance in hardware via counters improves accuracy, the level of architectural detail is constrained by counter availability. Simulated evaluation often provides greater detail with only modest costs in accuracy. The simulator I employ has been validated against a hardware implementation.

The techniques I use to compare SPECcpu and SPECjbb are similar to those of KleinOsowski and Lilja [8]. These authors characterize the SPECcpu benchmarks by instruction mix and level-1 cache miss rates as measured in an architectural simulator. They use these characteristics as points at which to compare the architectural behavior of SPEC benchmarks against MinneSPEC, their proposed synthetic approximation to the full SPECcpu suite. Similarly, I employ an architectural simulator, Turandot, to compare the full SPECcpu suite against an alternative benchmark. However, the simulator used in my work provides much more detail regarding misses at multiple levels of cache and branching behavior.

2.2. Statistical Regression Modeling

Although statistical techniques are typically used to generate synthetic benchmarks, there is no prior work, to my knowledge, that applies statistical regression techniques to predict performance from the architectural characteristics of an application. The closest such work is Yi and Lilja's work in developing a statistically rigorous approach for improving simulation methodology by identifying statistically significant architectural design parameters [13, 14, 15]. Given a set of N parameters, they employ Plackett-Burman experimental designs to identify $n < N$ significant parameters that have the greatest effect on a particular metric and thus should vary in a design space exploration [16]. In contrast to ad-hoc approaches to design space studies, Yi and Lilja's work provides computer architects with a statistically significant set of design parameters to consider (*e.g.* cache sizes, pipeline depth).

My approach to and objectives for statistical modeling differ. I apply regression modeling to predict performance as a function of significant architectural characteristics where significance is evaluated by t-tests and p-values.

3. Experimental Methodology

3.1. Microarchitectural Modeling

I employ Turandot, a cycle-based microprocessor simulator [10], to obtain performance measurements and relevant architectural statistics. The simulator is configured to simulate a single-threaded pipeline, resembling the IBM POWER4 architecture (Table 1). The simulated memory hierarchy implements a Harvard architecture (separate in-

Processor Core	
Decode Rate	4 non-branch instructions per cycle
Dispatch Rate	9 instructions per cycle
Reservation Stations	FXU(40),FPU(10),LSU(36),BR(12)
Functional Units	2 FXU, 2 FPU, 2 LSU, 2 BR
Physical Registers	80 GPR, 72 FPR
Branch Predictor	16k 1-bit entry BHT
Memory Hierarchy	
L1 DCache Size	32KB, 2-way, 128B blocks, 1-cy latency
L1 ICache Size	32KB, 1-way, 128B blocks, 1-cy latency
L2 Cache Size	2MB, 4-way, 128B blocks, 9-cy latency
Memory	77-cy latency
Pipeline Dimensions	
Pipeline Depth	19 FO4 delays per stage
Pipeline Width	4-decode

Table 1. Microarchitectural Parameters.

struction and data caches at the first level) with a unified second level cache.

3.2. Performance and Architectural Metrics

Performance is evaluated in terms of achieved instruction throughput (instructions per cycle). Table 2 lists the architectural characteristics examined for each benchmark. In addition to throughput, I consider instruction and data cache accesses and misses at level one and two. Data cache trailing accesses refer to data cache probes to lines being brought in (*e.g.* trailing-edge effect).

Turandot provides statistics regarding the application's branching characteristics. Branch stalls specify the number of cycles spent fetching from an incorrect path. The branch count is categorized into conditional branch, branch-to-link-register, and branch-to-count-register instructions. The branch prediction logic speculates on the branch condition for the first type and speculates on the branch targets within the registers for the last two types.

The simulator also tracks the cause of microprocessor stalls. The "ibuf" and "inflight" statistics track the number of fetch cycles stalled due to a full instruction buffer and constraints on the number of in-flight instructions, respectively. Similarly, the simulator tracks the memory unit stalls due to a full miss queue from limits on the number of outstanding cache misses and a full cast out queue from cache lines cast out and waiting to be handled by a coherency processor. Store and reorder queue occupancy is also monitored. Lastly, the "resv" and "rename" statistics track dispatch stalls due to full issue queues and rename stalls due to lack of instructions decoded, respectively.

3.3. Benchmarks

The Standard Performance Evaluation Corporation (SPEC) publishes a standard set of benchmarks designed for performance measurements of compute intensive workloads [9]. Standardization implies comparability of these measurements across different systems. Table 3 lists a suite of 21 benchmarks from the SPEC CPU

Instruction Throughput	
instruction count	cycle count
Instruction Cache	
probe count	
I-L1 misses	I-L2 misses
I-TLB1 misses	I-TLB2 misses
Data Cache	
probe count	
D-L1 misses	D-L2 misses
D-TLB1 misses	D-TLB2 misses
trailing accesses	
D-L1 lines cast out	
Branches	
branch count	branch stalls
cond branches	cond mispredict
link branches	link mispredict
counter branches	counter mispredict
Stalls	
ibuf	inflight
dmissq	cast
storeq	reorderq
resv	rename

Table 2. Architectural Characteristics.

Integer		
bzip2	crafty	gap
gcc	gzip	mcf
perlbmk	twolf	vpr
Floating-Point		
ammp	applu	apsi
art	equake	facerec
lucas	mesa	mgrid
sixtrack	swim	wupwise
Java Server Benchmark		
jbb		

Table 3. Benchmarks.

2000 benchmarks (SPECcpu) categorized into integer and floating-point workloads (SPECint and SPECfp, respectively). Each of these workloads were developed in C or Fortran from real applications, such as compression, compilation, and scientific computing.

I compare SPECcpu against a benchmark for evaluating the performance of server-side Java (SPECjbb). This benchmark emulates a three-tier client/server system. Although implementations of the Java Virtual Machine (JVM), Just-In-Time (JIT) compiler, garbage collection, threads, and aspects of the operating system are exercised, business logic representative of current applications (including XML processing and BigDecimal computation) and object manipulation dominate [9]. This benchmark measures CPU, cache, memory hierarchy performance, as well as the scalability of shared memory processors.[1]

I report experimental results from benchmark traces generated with the tracing facility *Aria* [11] using the full reference input set. However, sampling reduced the total trace length to 100 million instructions per benchmark program. The sampled traces were validated against the full traces [12]. I compared the SPECcpu traces with a SPECjbb trace generated in the same manner.

1 SMP scalability is beyond the scope of this paper.

4. Architectural Analysis

I consider the architectural characteristics of SPECcpu and compare them against those of SPECjbb. This comparison is intended to evaluate the relevance of SPECcpu by comparing the degree to which it captures architectural characteristics of real world applications. For example, if SPECjbb were to incur significantly higher cache miss rates relative to SPECcpu, one could further the argument of SPECcpu irrelevance.

I consider four sets of architectural data, including (1) instruction cache, (2) data cache, (3) branch, and (4) throughput characteristics. Intuitively, cache and branch characteristics are primary determinants of processor and application performance. The large memory latencies that may result from poor cache performance will likely overshadow most other processor latencies. Branch characteristics determine strongly impact the amount of useful computation in the processor and the number of pipeline stalls associated with executing incorrectly predicted paths. Thus, cache and branching characteristics will collectively dominate any discussion of performance.

In each category, I compare summary statistics (min, median, mean, max) for both integer and floating-point workload characteristics in SPECcpu against the those in SPECjbb. Green denotes floating-point workloads, blue denotes integer workloads, and red denotes the Java server benchmark in figures throughout this section unless otherwise noted. The full data set is available in the appendices.

4.1. Instruction Cache

Figures 1–2 summarize the instruction cache characteristics normalized to the maximum observed value and Appendix A presents the detailed data. The probe frequency refers to the fraction of total instructions resulting in an instruction cache probe. As shown in Figure 1L, integer benchmarks tend to probe the cache more frequently than floating-point benchmarks (median probe freq: 21 versus 16 percent). These benchmarks not only access the instruction cache more frequently, but Figure 1R and Figure 2L show they also tend to incur higher miss rates at both levels of cache. However, in absolute terms, instruction L1 and L2 cache misses are less than 5.0 and 0.1 percent, respectively. TLB misses are also negligible.

The L1, L2 miss rates indicate the fraction of instruction probes incurring a miss in the L1, L2 cache. Since $L2_{miss}/L1_{miss}$ is the percentage of L1 misses that are also L2 misses, $1-(L2_{miss}/L1_{miss})$ quantifies the percentage of L1 misses resolved in the L2 cache (*i.e.* L2 cache effectiveness). For example, an effectiveness of 95 percent means 95 percent of L1 cache misses are resolved in the L2 cache. Cache effectiveness follows a bimodal distribution for floating-point benchmarks with roughly half of these benchmarks exhibiting less than 15 percent effectiveness and the other half exhibiting greater than 80 percent effectiveness (see Appendix A). In contrast, all but two integer workloads exhibit greater than 90 percent effectiveness. These trends are shown in Figure 2R.

Floating-point scientific computing applications are usually dominated by loops to iterate over and perform computation on the data. Furthermore, these applications exert less pressure on the cache since the loop body may reside in cache from previous loop iterations. Although these are reasonable inferences from the observed data, explicitly verifying this behavior is beyond the scope of this paper.

Although SPECjbb's probe frequency is only 13 percent higher than that of SPECcpu's integer subset, the cache miss rates from SPECcpu are not representative of those from the Java server benchmark. The integer benchmarks underrepresent the L1 miss rate by a factor of 2 and L2 miss rate by a factor of 40 (comparing JBB miss rates against median integer miss rates). Despite this large difference in cache and memory accesses, SPECjbb L2 cache effectiveness to resolve instruction misses is only 20 percent less than the integer median. SPECfp is even less representative. From these measurements, SPECcpu does not appear to fully capture the instruction cache characteristics of SPECjbb.

4.2. Branches

Figures 3–4 summarize the branch characteristics normalized to the maximum observed value and Appendix B presents the detailed data. As shown in Figure 3L, integer benchmarks tend to have more complex control flow with branches comprising a larger percentage of all instructions (median branch rate: 15 versus 5 percent). Figure 3R shows SPECint branches are also less predictable than their floating-point counterparts with only 92 percent accuracy. Collectively, the branch frequency and misprediction rate explain the factor of 3 difference in the number of branch induced stalls (*i.e.* cycles fetching from incorrect path) observed in Appendix B. In contrast with the relatively complex control flow for integer workloads, the regular loops in scientific computing applications likely provide a larger number of easily predicted branches. The branches at the bottom of each loop to check conditions on the induction variable are almost always predicted correctly, especially in loops with many iterations.

The branch prediction logic speculates on the branch condition for conditional branches and speculates on the branch targets for branch-to-link-register and branch-to-count-register instructions. As shown in Figure 4R, conditional branches comprise more than 95 percent of the total. Although integer workloads experience better branch target prediction, branches with such predictions are infre-

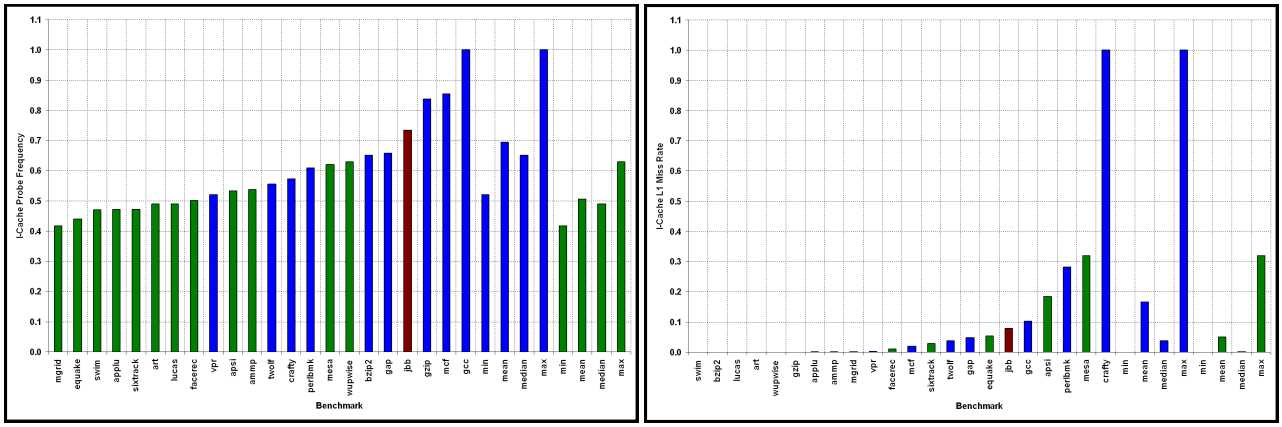

Figure 1. (Left) Number of instruction cache probes per 100 instructions; (Right) Percentage of instruction cache probes missing in L1. Percentages are normalized with respect to the maximum observation.

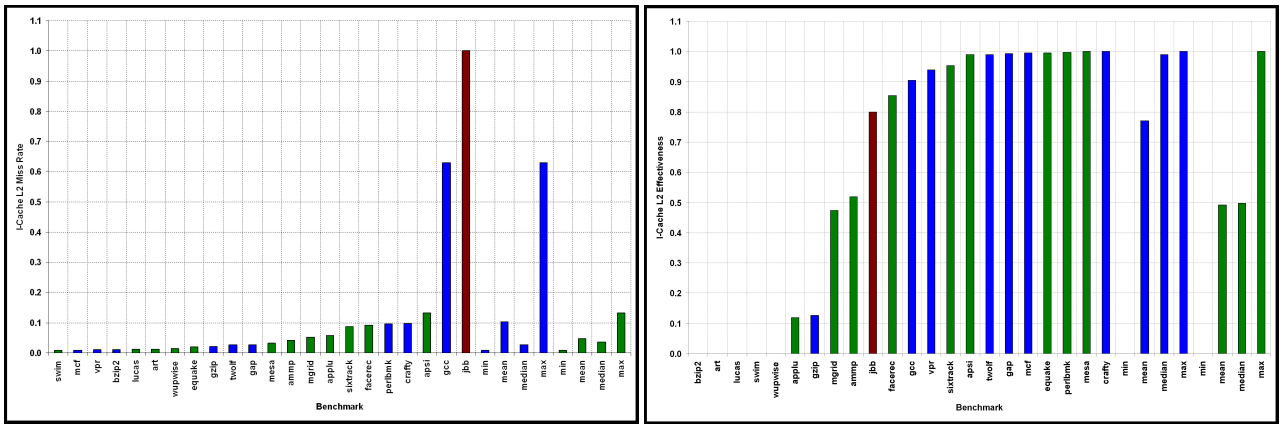

Figure 2. (Left) Percentage of instruction probes missing in both L1 and L2; (Right) Percentage of L1 instruction cache misses resolved in L2 cache. Percentages are normalized with respect to the maximum observation.

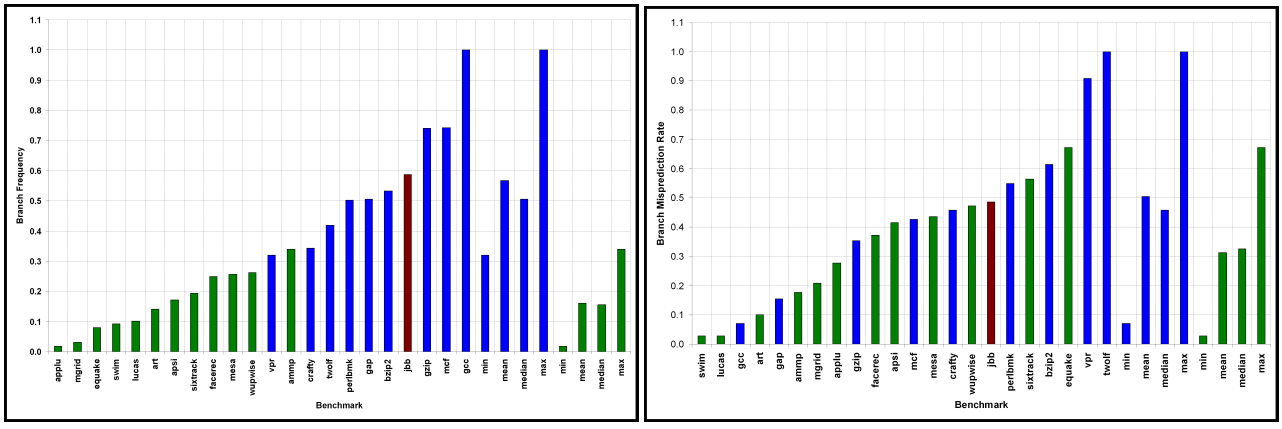

Figure 3. (Left) Number of branches per 100 instructions; (Right) Percentage of branches mispredicted. Values are normalized with respect to maximum observation.

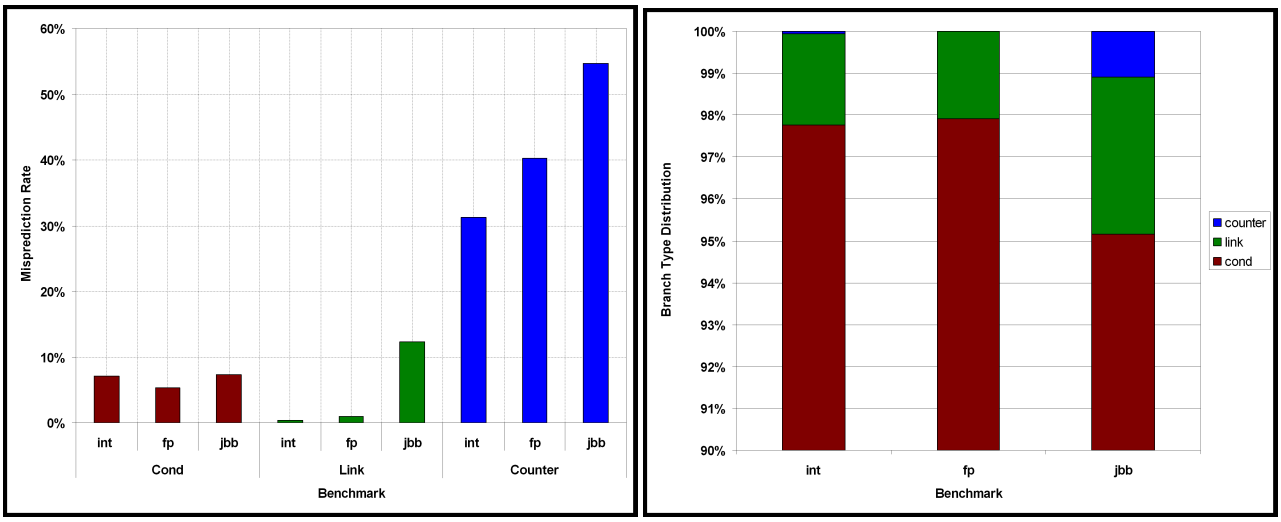

Figure 4. (Left) Percentage of branches mispredicted, categorized by type of branch and normalized with respect to the maximum observation. (Right) Distribution of branch types. Note y-axis starts at 90 percent. Unlike other figures in this section, red denotes conditional branches, green denotes branch-to-link-register instructions, and blue denotes branch-to-count-register instructions.

quent and their poor performance for conditional branches dominate the overall performance impact of branches.

With branches comprising 17 percent of all instructions and an 8 percent misprediction rate, SPECjbb branch characteristics are comparable to those of SPECint. The observations suggest a positive correlation between instruction cache activity and branch frequency; more branches lead to more activity. SPECjbb's instruction cache probe frequency and branch frequency is 12 and 16 percent higher than those of SPECint. No stronger conclusions may be drawn, however, since the relationship between branches and the instruction cache depend on branching distances, an unobserved statistic. With regard to branches, SPECint seems representative of SPECjbb, a proxy for real workloads.

4.3. Data Cache

Figures 5–6 summarize the data cache characteristics normalized to the maximum observed value and Appendix C presents the detailed data. Integer and floating-point benchmarks have comparable cache access rates with loads/stores comprising approximately 40 percent of all instructions. However, floating-point workloads experience higher miss rates at both levels of cache (median L1: 9 versus 15 percent; median L2: 0.6 percent versus 6.8 percent). Thus, integer workloads could be considered more compute intensive when compared against floating-point workloads. TLB misses from load and store operations occur with negligible frequency.

The complete data set in Appendix C reveals additional information about data cache accesses. Both SPECint and SPECfp include a single benchmark that skew average cache activity. Integer workload `mcf` for combinatorial optimization and floating-point workload `art` for image recognition both experience L1 and L2 miss rates greater than 50 and 25 percent, respectively. As SPECcpu is a compute intensive benchmark, it may be appropriate that only two benchmarks are memory bound.

With probe frequencies and L1 miss rates comparable to SPECint and only 0.5 percent of data memory accesses incurring L2 cache misses, SPECjbb is more compute intensive that SPECcpu. Thus, SPECcpu appears to capture the data cache characteristics of SPECjbb and both benchmarks seem equally suitable for evaluating system research targeted at improving CPU performance.

4.4. Performance

This study emphasizes the architectural characteristics of commonly used benchmarks and their potential relevance. Figure 7 presents the instruction throughput (measured in instructions per cycle). The performance results illustrate the dangers of averaging performance across benchmarks due to outliers (*e.g.* mcf, art, lucas) that skew the mean. SPECjbb performance, however, is well represented by the majority of SPECcpu benchmarks, achieving performance comparable to SPECcpu median throughput.

5. Regression Theory and Practice

Given the significant performance insights provided by the preceding architectural analysis, I evaluate the feasibility of using this data set to statistically predict a benchmark's instruction throughput as a function of its archi-

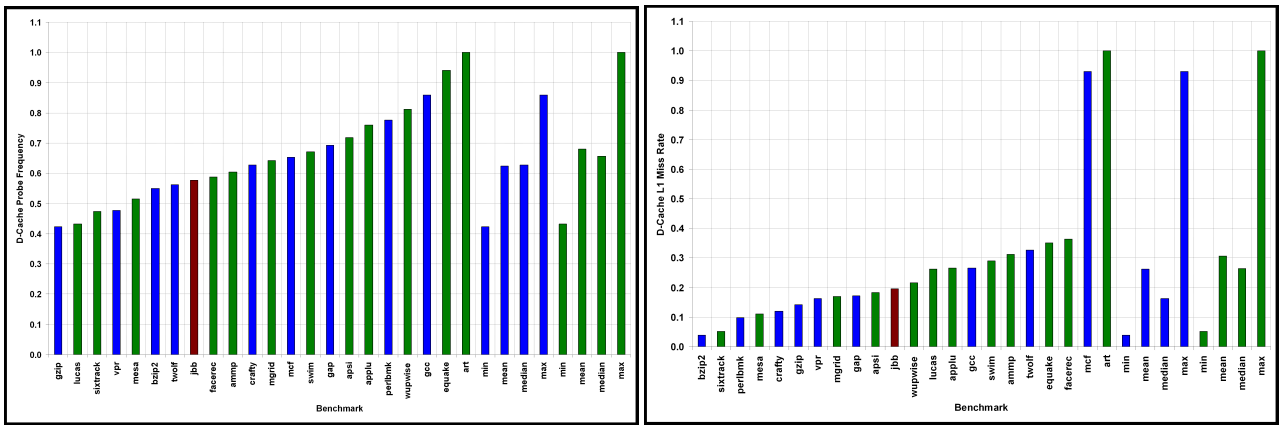

Figure 5. (Left) Number of data cache probes per 100 instructions; (Right) Percentage of data cache probes missing in L1. Percentages are normalized with respect to the maximum observation.

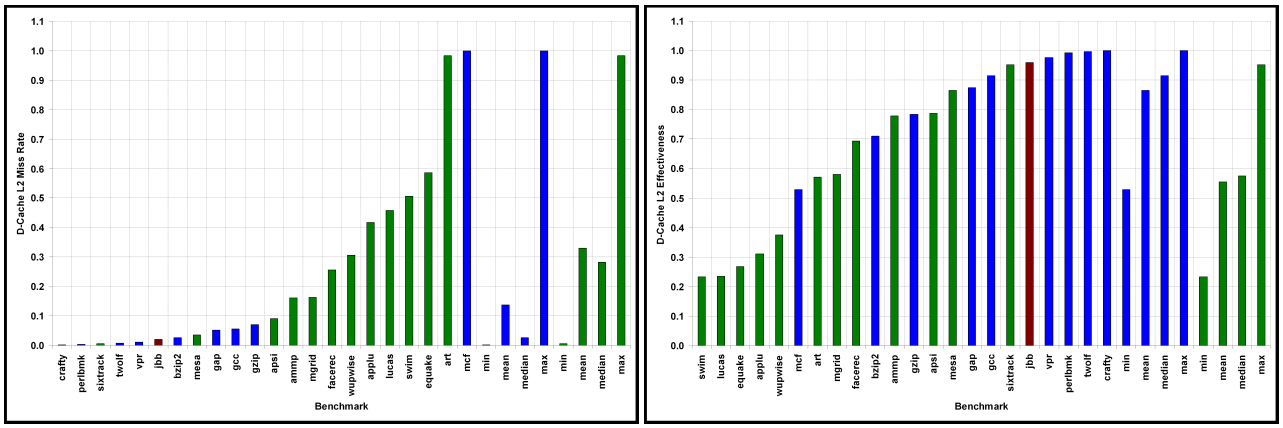

Figure 6. (Left) Percentage of data cache probes missing in both L1 and L2; (Right) Percentage of L1 data cache misses resolved in L2 cache. Percentages are normalized with respect to the maximum observation.

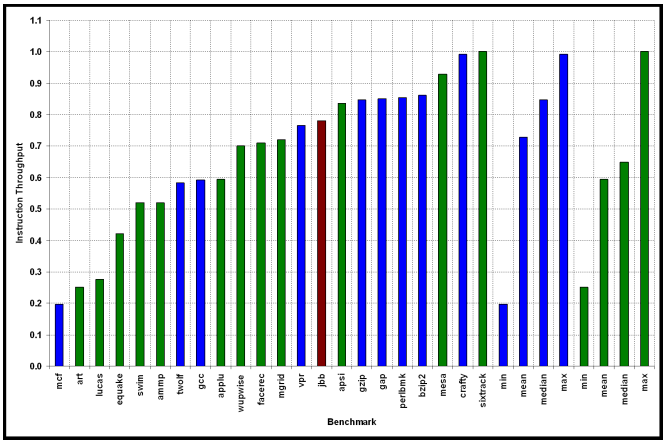

Figure 7. Performance Summary.

tectural characteristics. This section provides a brief background on regression, including the method of least squares to derive coefficients for a linear model, the statistical properties of these coefficients, assessing the fit of a regression model, and prediction with such a model.

5.1. Least Squares

Fitting a straight line to a plot of points (x_i, y_i) produces a linear model for predicting y from x, where $i \in [1, n]$, n is the number of data points, and x, y are known as the predictor and response variables, respectively. To perform this fit, the slope and intercept of the line $y_i = \beta_0 + \beta_1 x_i$ must be found. More generally, predicting y_i from p predictors requires fitting a p-th order polynomial by determining $p + 1$ coefficients in Equation (1). A transformation f may be necessary to convert a non-linear correlation between a predictor and the response into a linear one.

$$y_i = \beta_0 + \sum_{j=1}^{p} \beta_j f_j(x_{ij}) \quad (1)$$

The method of least squares is the most common method for determining coefficients in a curve fitting problem. Least squares determines the best-fitting curve to a set of points by minimizing the sum of squared deviations of the predicted values (given by the curve) from the actual observations. In particular, least squares would find the $p + 1$ coefficients in Equation (1) to minimize Equation (2).

$$S(\beta_0, \ldots, \beta_p) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \quad (2)$$

Equation (2) may be maximized by solving a system of $p + 1$ partial derivatives of S with respect to β_j, $j \in [0, p]$. The solutions to this system, $\hat{\beta}_j$, are estimates of the coefficients in Equation (1). Furthermore, the solutions to this system of linear equations may be expressed in closed form. The statistical properties of these estimates are used to determine the goodness of fit.

5.2. Statistical Properties

The statistical properties of the coefficient estimates are used to evaluate the reliability of the linear model in the presence of deviations from observed values. These deviations may be treated as errors or noise in Equation (3), a revised version of the statistical model in Equation (1). In this model, the errors e_i are independent random variables with zero mean and some constant variance; $E(e_i) = 0$ and $Var(e_i) = \sigma^2$.

$$y_i = \beta_0 + \sum_{j=1}^{p} \beta_j f_j(x_{ij}) + e_i \quad (3)$$

If the errors e_i are independent normal random variables, then the estimated coefficients $\hat{\beta}_j$ are also normally distributed. This normality assumption leads to the relationship in Equation (4), which enables hypothesis tests using the t_{n-p-1} distribution. The estimates $\hat{\beta}_j$ are obtained in closed form by solving a linear system and their standard deviations $s_{\hat{\beta}_j}$ may be obtained analytically from the closed form estimates.

$$\frac{\hat{\beta}_j - \beta_j}{s_{\hat{\beta}_j}} \sim t_{n-p-1} \quad (4)$$

Hypothesis testing employs Equation (4) and the t_{n-p-1} distribution. A commonly tested null hypothesis states the j-th predictor in the model has no predictive value ($H_0 : \beta_j = 0$). This hypothesis is tested by evaluating Equation (4) under the null hypothesis to obtain the *t-test*, $\hat{\beta}_j / s_{\hat{\beta}_j}$. The t-test is typically computed for every coefficient estimate $\hat{\beta}_j$ as a first step toward determining significance of predictor x_j in the model. Note the t-test follows a t distribution with $n - p - 1$ degrees of freedom.

The *p-value* is defined as $2P(X \geq |c|)$ for a random variable X and a constant c. In our analyses, $X \sim t_{n-p-1}$ and $c = \hat{\beta}_j / s_{\hat{\beta}_j}$. The p-value may be interpreted as the probability a value for the t-test greater than or equal to the value actually observed would occur by chance if the null hypothesis were true. If this probability were extremely small, either the null hypothesis holds and an extremely rare event has occurred or the null hypothesis is false. Thus, a small p-value for $\hat{\beta}_j$ casts doubt on the hypothesis $\beta_j = 0$ and suggests the j-th predictor is statistically significant in predicting the response.

5.3. Assessing Fit

Goodness of fit is often assessed by examining residuals, defined in Equation (5) as the differences between observed and predicted values. Residuals are often examined graphically since plotting residuals against each of the predictors (*i.e.* a scatterplot of $(\hat{e}_i, x_{ij})$ for each $j \in [0, p]$) may reveal systematic deviations from the standard statistical model.[2] Ideally, there should be no correlation between the residuals and the predictor.

$$\hat{e}_i = y_i - \hat{\beta}_0 - \sum_{j=0}^{p} \hat{\beta}_j x_{ij} \quad (5)$$

In addition to verifying standard assumptions, the multiple correlation statistic, R^2, is often computed to assess predictor effectiveness for the data used to create the model. Equation (8) computes this statistic by computing regression error (SSE) as a fraction of the total error (SST). From the equation, R^2 will be zero when the error from the regression model is just as large as the error from simply using the mean to predict the response. Thus, R^2 is the percentage of variance in the response variable captured by the predictor variables. Maximizing R^2 while minimizing the number of predictors is desirable. Furthermore, the statistically significant predictors, identified by low p-values in Section 5.2, are most likely to achieve this goal.

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \quad (6)$$

2 "Standard statistical model" refers to normality assumptions in Section 5.2.

$$SST = \sum_{i=1}^{n} \left(y_i - \frac{1}{n} \sum_{i=1}^{n} y_i \right)^2 \quad (7)$$

$$R^2 = 1 - \frac{SSE}{SST} \quad (8)$$

5.4. Prediction

Once a regression model satisfies standard normality assumptions and is found to be a good fit to the observed data, it may be used to predict the response from a new set of predictor values. Given $x_{n+1} = \{x_{n+1,1}, \ldots, x_{n+1,p}\}$, y_{n+1} may be predicted by simply evaluating the model at x_{n+1}.

6. Regression Analysis

After considering the architectural characteristics of SPECcpu and SPECjbb, I proceed to evaluate the potential for statistical performance model. In particular, is it possible to derive models to predict instruction throughput as a function of the architectural characteristics presented in Section 4? Given the large data set collected from the previous study, an exploratory data analysis with regression modeling is a natural starting point for answering this question. After a regression model is formulated from the SPECcpu data, I evaluate its predictive ability for performance of SPECjbb and variants of the original SPECcpu benchmarks obtained by changing the sampling method used to generate the traces.

6.1. Regression Model

I apply the statistical methodology of Section 5 to benchmarks' architectural characteristics and performance evaluation. I develop a regression model in the form of Equation (1). In the simplest case, transformations are unnecessary and $f(x_i) = x_i$. In the context of this work, y is the instruction throughput, x_{ij} is the observed j-th architectural characteristic observed for the i-th benchmark (*e.g.* L1 data cache miss rate for art), and estimates for β_j are obtained by the method of least squares.

A linear regression model is applicable only if performance is linearly correlated with predictors drawn from the architectural data. A case for linear modeling may be presented via scatterplots of instruction throughput versus potential predictors. For example, Figure 8 is such a plot for two possible predictors drawn from data cache characteristics. These plots demonstrate a linear correlation between the value of interest (ipc) and the possible predictors (data probe and L1 cache miss rates). Figure 8 is representative of the trends observed for most architectural characteristics I considered. A linear model is a natural first step; transformations on the predictors may be applied to refine the initial model if necessary.

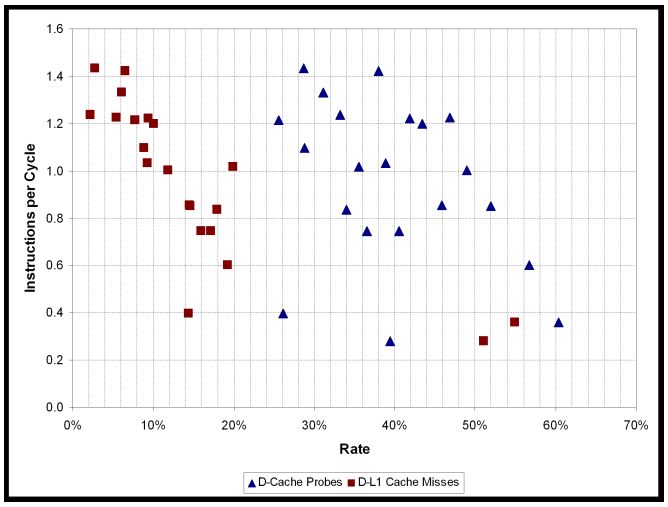

Figure 8. Scatterplots: Plots instruction throughput against data cache probe frequencies and L1 data cache miss rates.

	Predictor	Coeff. Estimate	Std. Err.	p-value
β_0	*Intercept*	1.55E+00	1.27E+00	0.308
β_1	iprobe_rate	1.89E+00	5.93E+00	0.771
β_2	iL1miss_rate	6.25E+00	3.52E+00	0.174
β_3	iL2miss_rate	9.19E+01	1.09E+03	0.938
β_4	dprobe_rate	-5.65E-01	1.33E+00	0.699
β_5	dL1miss_rate	-2.11E+00	1.53E+00	0.26
β_6	dL2miss_rate	-5.43E-01	3.31E+00	0.88
β_7	stall_ibuf	-3.45E-09	2.23E-09	0.22
β_8	stall_inflight	-3.23E-08	8.01E-08	0.714
β_9	stall_dmissq	3.45E-09	3.96E-09	0.448
β_{10}	stall_cast	6.66E-08	5.18E-08	0.289
β_{11}	stall_storeq	-9.62E-08	2.74E-07	0.748
β_{12}	stall_reorderq	-7.30E-08	5.16E-07	0.896
β_{13}	stall_resv	-1.62E-09	9.71E-09	0.878
β_{14}	stall_rename	-9.86E-10	6.61E-09	0.891
β_{15}	br_rate	-1.24E+00	7.57E+00	0.88
β_{16}	br_stall	-1.05E-08	5.17E-08	0.852
β_{17}	br_mis_rate	-2.57E-01	3.07E+00	0.939

Table 4. Initial Linear Regression: All architectural characteristics included in model.

Table 4 displays the coefficient estimates for a model employing 17 different architectural characteristics ($p = 17$). These coefficients are estimated by fitting the performance and architectural characteristics of the 21 SPECcpu traces ($n = 21$). In this context, the p-value is the probability that the least squares method computes a non-zero coefficient estimate for β_j listed in the table, given that the true $\beta_j = 0$. In other words, the p-value is the probability of estimating a significant non-zero relationship between y_i and predictor x_{ij} when, in reality, no such relationship exists. Thus, statistically significant predictors would have low p-values. The p-values in Table 4 indicate many of these parameters are not statistically significant.

By ranking predictors according to their p-values, the iL1miss_rate, dL1miss_rate, stall_ibuf, and stall_cast appear

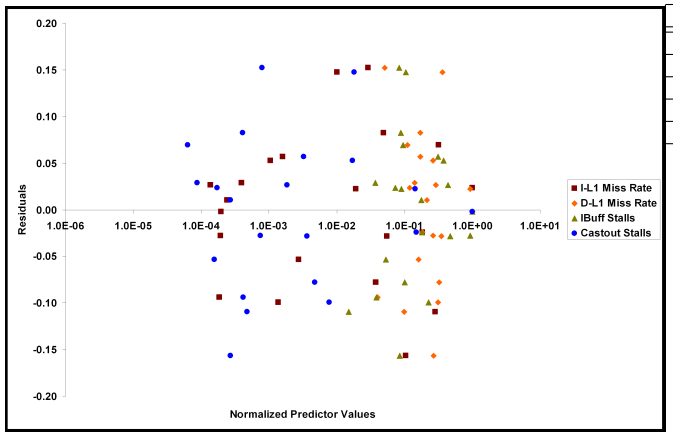

Figure 9. Residual Plots: Plots residuals against the four predictors in the refined model.

	Predictor	Coeff. Estimate	Std. Err.	p-value
β_0	*Intercept*	1.41E+00	4.27E-02	4.01E-16
β_1	il1miss_rate	5.74E+00	2.56E+00	0.0396
β_2	dl1miss_rate	-2.41E+00	2.30E-01	1.41E-08
β_3	stall_ibuf	-2.69E-09	3.69E-10	1.81E-06
β_4	stall_cast	6.58E-08	1.04E-08	1.04E-05

Table 5. Refined Linear Regression: Subset of architectural characteristics included in model.

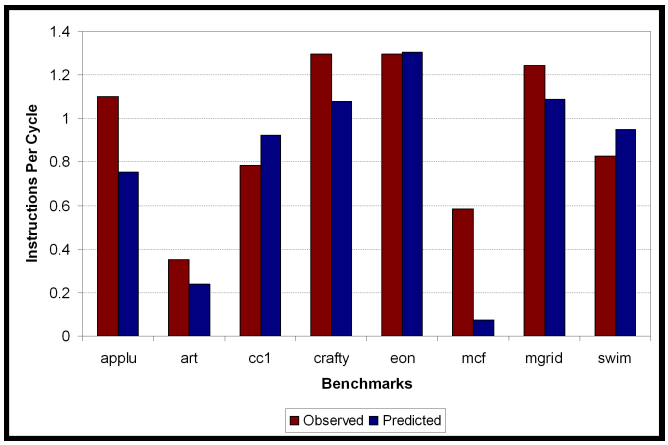

Figure 10. SPECcpu_v2 Prediction: Plots measured simulator data (observed) against predictions from the regression model.

to have some predictive ability. Indeed, a refined linear model of these four parameters alone is a good fit of the observed data (Table 5), confirmed when checking our model assumptions and assessing goodness of fit.

6.2. Goodness of Fit

Goodness of fit is often assessed by examining residuals and computing the multiple correlation statistic R^2. Figure 9 plots the residuals ($\hat{e}_i = y_i - \hat{\beta}_0 - \sum_{j=1}^{p} \hat{\beta}_j x_{ij}$) against each of the four predictors in Table 5. For each predictor, there is no apparent deviation from randomness in the residuals and there is no obvious correlation between the residuals and the predictor's value. These plots validate the assumptions about independent errors in Section 5.2 and strengthen the case for a linear model to capture the relationship between performance and these particular architectural measurements.

The multiple correlation statistic, R^2, measures the percent of the variance in the predicted value (y_i) explained uniquely by the predictors (x_{ij}). In this particular model, $R^2 = 0.94$. Qualitatively, this suggests the four-dimensional "curve" fit to the 21 data points is a relatively good fit. Thus, a simple linear model for predicting instruction throughput of benchmark i is given in Equation (9), when the following predictors are observed for benchmark i:

- x_{i1}: instruction L1 cache miss rate
- x_{i2}: data L1 cache miss rate
- x_{i3}: number of fetch cycles stalled due to a full instruction buffer
- x_{i4}: number of memory cycles stalled due to a full cast out queue

$$y_i = 1.405 + (5.738)x_{i1} - (2.410)x_{i2} - (2.689 \times 10^{-09})x_{i3} + (6.58 \times 10^{-08})x_{i4} \quad (9)$$

6.3. Predictive Ability

To assess the predictive ability of the linear model, evaluate Equation (9) for benchmarks not used in formulating the model. Since the model is based on SPECcpu data, I evaluate the model against SPECjbb and eight benchmarks in another set of SPECcpu traces. From this point forward, the original and alternative traces will be referred to as SPECcpu_v1 and SPECcpu_v2, respectively. *Aria* generated traces for SPECcpu_v1 by splicing multiple significant program segments into a single 100 million instruction trace. In contrast, *Simpoint* generated traces for SPECcpu_v2 by identifying a single continuous program segment of significance [17]. Thus, the architectural characteristics of the two sets of traces should be different, but comparable.

The regression model of Equation (10), evaluated for SPECjbb's observed architectural characteristics, predicted 1.141 instructions per cycle compared to the 1.118 instructions per cycle measured from Turandot. Thus, the architectural characteristics of SPECcpu are good predictors for SPECjbb performance, with only a 2 percent error.

However, the regression model is unable to predict the instruction throughput of SPECcpu_v2. Figure 10 compares observed throughput reported by the simulator against predicted throughput from evaluating Equation (10) with each benchmark's architectural characteristics reported by the simulator. Not only are the absolute predictions inaccurate, the relative predictions are also inaccurate. In other words, ranking these benchmarks first by observed and then by predicted performance would lead to different orderings.

6.4. Experimental Ignorability

Studies in probabilistic inference must distinguish between the methodology for measuring a certain event and the value obtained from the measurement. The *ignorability* of a study's design or data collection mechanism refers to cases in which methodology may be safely ignored. If a data collection mechanism is not ignorable, a naive inference model will result in misleading results. These concepts are often illustrated in the *Monty Hall Problem*.

The Monty Hall Problem, named after the host of a game show, supposes a contestant is given a choice of three doors. A car is behind one door and goats are behind the others. Without loss of generality, suppose the contestant picks door 1 and Monty Hall, who knows what is behind each door, opens door 3 to reveal a goat. He then gives the contestant the choice of staying with door 1 or switching to door 2. Initially, the car is equally likely to be behind each of the doors.

The decision to stay or switch depends on the probability model formulated by the contestant. A model that ignores methodology suggests, given it is not behind door 3, the car is equally likely to be behind door 1 or 2. Thus, there is no reason to switch.

However, the methodology for observing data is not ignorable. Monty Hall does not randomly choose a door to reveal, but reveals based on a methodology that never opens the first door chosen by the contestant and never opens the door containing the car. Ignoring this information about how the goat is observed will produce the misleading probability model that suggests no reason to switch.

The contestant picks door 1 which hides a car with probability 1/3. Conversely, one of the other two doors hides a car with probability 2/3. After Hall reveals the goat in door 3, door 2 alone hides the car with probability 2/3. Given these odds, the contestant should switch from door 1 to door 2, since she is twice as likely to win the car after switching.

In a similar fashion, I suspect the regression model's poor predictive ability may be attributed to ignoring differences in the trace collection process. In particular, using a linear model based on SPECcpu_v1 to predict SPECcpu_v2 performance ignores potentially material differences in trace construction. As in the Monty Hall Problem, a better model conditions a prediction, not only on observations (simulated performance and architectural characteristics), but also on the methodology used to generate the observations (splicing distinct instruction segments versus identifying a single contiguous segment).

In an initial attempt to capture tracing methodology in the model, a fifth predictor is added to the original Equation (10), x_{i5}. This binomial predictor takes on a zero if instruction splicing is used and a one if a single contiguous instruction segment is used. Unfortunately, this predictor is not statistically significant and forcing it into the model does not improve predictive ability. Thus, a single binomial predictor appears inadequate in accounting for tracing methodology. Techniques to incorporate non-ignorable mechanisms into a predictive model are likely more subtle and remain an open question.

7. Conclusions and Continuing Work

The SPEC compute intensive benchmarks are relevant and representative of real workloads with regard to architectural characteristics such as cache activity and branching characteristics. A subset of these architectural characteristics are significant predictors of performance in a linear regression model. These regression models are sensitive to the underlying trace collection facilities.

I hope to collect a Berkeley DB trace for the simulator for another comparison to SPECcpu2k. Berkeley DB is another comprehensive benchmark, which exercises the whole system. Resolving current difficulties with Berkeley DB trace collection with present facilities is future work.

Additional traces would also further the evaluation of regression modeling in architectural performance studies. Although the models clearly have some predictive power, the robustness of these models is still unclear. Since the models are sensitive to the trace collection process, it may be possible to incorporate data collection parameters (*e.g.* number of simulation points, contiguous program segments) as predictors in the model for added resilience.

References

[1] L. Eeckhout, H. Vandierendonk, K. DeBosschere. Designing computer architecture research workloads. In *IEEE Computer*, 2003.

[2] L. Eeckhout, S. Nussbaum, J. Smith, K. DeBosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. In *IEEE Micro*, September/October 2003.

[3] L. Eeckhout, K. DeBosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *PACT2001: International Conference on Parallel Architectures and Compilation Techniques*, November 2001.

[4] L. Eeckhout, K. DeBosschere. Early design phase power/performance modeling through statistical simulation. In *ISPASS2001: International Symposium on Performance Analysis of Systems and Software*, November 2001.

[5] S. Nussbaum, J. Smith. Statistical Simulation of Symmetric Multiprocessor Systems. *35th Annual Simulation Symposium*, April 2002.

[6] S. Nussbaum, J. Smith. Modeling Superscalar Processors via Statistical Simulation. *PACT2001: International Conference on Parallel Architectures and Compilation Techniques*, Barcelona, Sept. 2001.

[7] J.L. Henning. SPEC CPU 2000: Measuring cpu performance in the new millenium. In*IEEE Computer*, July 2000.

[8] A.J. KleinOsowski, D.J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. In *Computer Architecture Letters*, June 2002.

[9] Standard Performance Evaluation Corporation. http://www.spec.org

[10] M. Moudgill, J.D. Wellman, J. Moreno. Validation of Turandot, a fast processor model for microarchitecture exploration. *Proceedings of the IEEE International Performance, Computing and Communication Conference*, February 1999.

[11] M. Moudgill, J. Wellman, J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, May/Jun 1999.

[12] V. Iyengar, L.H. Trevillyan, P. Bose. Representative Traces for Processor Models with Infinite Cache. In *Proc. HPCA-2*, Feb 1996.

[13] J. Yi, D. Lilja, D. Hawkins. Improving Computer Architecture Simulation Methodology by Adding Statistical Rigor. In *IEEE Transactions on Computer*, November 2005.

[14] J. Yi, D. Lilja, D. Hawkins. A statistically rigorous approach for improving simulation methodology. In *Proceedings of the International Symposium on High- Performance Computer Architecture.*, February 2003.

[15] J. Yi. Improving processor performance and simulation methodology. *Ph.D. Thesis*, University of Minnesota, December 2003.

[16] R.L. Plackett, J.P. Burman. The design of optimum multifactorial experiments. In *Biometrika 33*, 1945.

[17] G. Hamerly, E. Perelman, J. Lau, B. Calder. SimPoint 3.0: faster and more flexible program analysis. Workshop on Modeling, Benchmarking and Simulation, June 2005.

[18] P.D. Grunwald, J.Y. Halpern. Updating probabilities. Journal of Artificial Intelligence Research, vol. 19, 2003.

A. Instruction Cache Characteristics

	Probe Freq	L1 Misses	L1 Miss Rate	L2 Misses	L2 Miss Rate	L2 Effectiveness	TLB1 Misses	TLB2 Misses
Integer								
bzip2	20.62%	142	0.001%	142	0.001%	0.00%	0	13
crafty	18.17%	726095	3.705%	1123	0.006%	99.85%	0	39
gap	20.86%	37996	0.182%	339	0.002%	99.11%	0	24
gcc	31.69%	122186	0.385%	11808	0.037%	90.34%	4093	674
gzip	26.54%	390	0.001%	341	0.001%	12.56%	0	25
mcf	27.08%	19376	0.071%	138	0.001%	99.29%	1	18
perlbmk	19.31%	213265	1.047%	1147	0.006%	99.46%	19247	128
twolf	17.60%	25937	0.141%	286	0.002%	98.90%	0	33
vpr	16.51%	2175	0.010%	136	0.001%	93.75%	0	11
min	16.51%	142	0.001%	136	0.001%	0.00%	0	11
mean	22.04%	127507	0.616%	1718	0.006%	77.03%	2593	107
median	20.62%	25937	0.141%	339	0.002%	98.90%	0	25
max	31.69%	726095	3.705%	11808	0.037%	99.85%	19247	674
Floating-Point								
ammp	17.05%	877	0.005%	422	0.002%	51.88%	0	47
applu	14.98%	588	0.004%	518	0.003%	11.90%	1	40
apsi	16.89%	116708	0.684%	1326	0.008%	98.86%	0	44
art	15.52%	117	0.001%	117	0.001%	0.00%	0	8
equake	13.95%	28489	0.203%	167	0.001%	99.41%	1	17
facerec	15.86%	3403	0.037%	499	0.005%	85.34%	0	31
lucas	15.53%	116	0.001%	116	0.001%	0.00%	0	12
mesa	19.65%	306431	1.187%	496	0.002%	99.84%	1	48
mgrid	13.26%	796	0.006%	419	0.003%	47.36%	0	31
sixtrack	14.98%	16101	0.107%	765	0.005%	95.25%	1	84
swim	14.93%	76	0.001%	76	0.001%	0.00%	0	5
wupwise	19.96%	178	0.001%	178	0.001%	0.00%	0	9
min	13.26%	76	0.001%	76	0.001%	0.00%	0.00	5.00
mean	16.04%	39490	0.186%	425	0.003%	49.15%	0.33	31.33
median	15.52%	837	0.006%	421	0.002%	49.62%	0.00	31.00
max	19.96%	306431	1.187%	1326	0.008%	99.84%	1.00	84.00
Java Server Benchmark								
jbb	23.26%	69744	0.2936%	14042	0.059%	79.87%	1028	773
w.r.t int	1.128	2.689	2.086	41.422	38.095	0.808	1.000	30.920
w.r.t fp	1.499	83.376	53.163	33.394	27.005	1.610	1.000	24.935

Table 6. Instruction Cache Measurements: Architectural measurements of integer and floating-point SPECcpu2k benchmarks. Measurements of SPECjbb2005 are compared against the median of integer and floating-point measurements.

B. Branch Characteristics

	Branch Freq	Stalls(mil)	Overall Mispr	Cond	Cond Mispr	Link	Link Mispr	Ctr	Ctr Mispr
Integer									
bzip2	15.63%	13.195	10.17%	1.284E+07	10.45%	3.501E+05	0.05%	0.000E+00	n/a
crafty	10.08%	9.493	7.57%	8.912E+06	7.58%	5.114E+05	0.35%	7.032E+04	58.76%
gap	14.82%	12.224	2.56%	1.054E+07	2.91%	1.598E+06	0.06%	9.041E+04	6.36%
gcc	29.33%	28.989	1.16%	2.867E+07	1.01%	2.541E+05	1.79%	6.680E+04	61.60%
gzip	21.71%	21.178	5.86%	2.105E+07	5.89%	1.294E+05	0.23%	2.720E+02	7.72%
mcf	21.76%	19.902	7.06%	1.979E+07	7.08%	1.017E+05	0.04%	6.420E+03	50.05%
perlbmk	14.71%	13.685	9.09%	1.138E+07	3.32%	1.145E+06	0.43%	1.157E+06	74.43%
twolf	12.31%	11.170	16.55%	1.095E+07	16.50%	2.149E+05	18.90%	7.030E+02	12.52%
vpr	9.38%	10.994	15.02%	1.084E+07	15.22%	1.535E+05	0.80%	4.600E+01	2.17%
min	9.38%	9.493	1.16%	8.912E+06	1.01%	1.017E+05	0.04%	0.000E+00	2.17%
mean	16.64%	15.648	8.34%	1.500E+07	7.77%	4.953E+05	2.52%	1.546E+05	34.20%
median	14.82%	13.195	7.57%	1.138E+07	7.08%	2.541E+05	0.35%	6.420E+03	31.28%
max	29.33%	28.989	16.55%	2.867E+07	16.50%	1.598E+06	18.90%	1.157E+06	74.43%
Integer									
ammp	9.95%	9.525	2.91%	9.399E+06	2.86%	1.066E+05	0.93%	1.929E+04	36.58%
applu	0.54%	0.545	4.60%	5.445E+05	4.58%	3.140E+02	27.39%	4.900E+01	48.98%
apsi	5.01%	4.823	6.87%	4.724E+06	7.00%	9.850E+04	1.05%	0.000E+00	n/a
art	4.13%	4.306	1.67%	4.306E+06	1.67%	1.300E+01	23.08%	0.000E+00	n/a
equake	2.35%	2.226	11.12%	2.138E+06	11.42%	7.927E+04	0.03%	8.854E+03	40.05%
facrec	7.33%	3.749	6.17%	3.650E+06	6.11%	9.057E+04	4.52%	7.917E+03	52.96%
lucas	2.95%	1.976	0.46%	8.766E+05	0.92%	1.099E+06	0.10%	0.000E+00	n/a
mesa	7.52%	8.549	7.21%	7.249E+06	8.48%	1.081E+06	0.07%	2.193E+05	0.55%
mgrid	0.89%	0.898	3.44%	8.969E+05	3.41%	9.320E+02	25.64%	1.540E+02	40.26%
sixtrack	5.67%	5.223	9.32%	5.135E+06	8.73%	4.998E+04	7.81%	3.793E+04	90.41%
swim	2.68%	2.683	0.46%	2.682E+06	0.46%	1.529E+03	0.20%	0.000E+00	n/a
wupwise	7.69%	6.661	7.82%	6.303E+06	8.25%	3.576E+05	0.21%	0.000E+00	n/a
min	0.54%	0.545	0.46%	5.445E+05	0.46%	1.300E+01	0.03%	0.000E+00	0.55%
mean	4.73%	4.264	5.17%	3.992E+06	5.32%	2.472E+05	7.58%	2.446E+04	44.26%
median	4.57%	4.028	5.38%	3.978E+06	5.34%	8.492E+04	0.99%	1.015E+02	40.26%
max	9.95%	9.525	11.12%	9.399E+06	11.42%	1.099E+06	27.39%	2.193E+05	90.41%
Java Server Benchmark									
jbb	17.19%	16.039	8.02%	1.526E+07	7.32%	6.020E+05	12.32%	1.740E+05	54.65%
w.r.t int	1.160	1.216	1.060	1.341	1.034	2.369	35.091	27.103	1.747
w.r.t. fp	3.762	3.982	1.491	3.837	1.371	7.089	12.447	1714.315	1.358

Table 7. Branch Measurements: Architectural measurements of integer and floating-point SPECcpu2k benchmarks. Measurements of SPECjbb2005 are compared against the median of integer and floating-point measurements.

C. Data Cache Characteristics

	Probe Freq	L1 Misses(E+06)	L1 Miss Rate	L2 Misses(E+06)	L2 Miss Rate	L2 Effectiveness	TLB1 Misses(E+03)	TLB2 Misses(E+03)
Integer								
bzip2	33.19%	0.725	2.18%	0.213	0.64%	70.56%	85.292	5.336
crafty	37.95%	2.690	6.57%	0.019	0.05%	99.28%	73.094	0.380
gap	41.88%	3.939	9.39%	0.519	1.24%	86.82%	3.868	4.639
gcc	51.93%	7.617	14.64%	0.690	1.33%	90.94%	245.397	8.756
gzip	25.58%	1.990	7.77%	0.441	1.72%	77.87%	0.129	1.880
mcf	39.48%	20.175	51.03%	9.567	24.20%	52.58%	2654.277	393.713
perlbmk	46.82%	2.655	5.37%	0.040	0.08%	98.50%	75.595	1.718
twolf	33.96%	6.365	17.90%	0.069	0.19%	98.92%	455.044	0.335
vpr	28.80%	3.344	8.90%	0.102	0.27%	96.96%	422.301	0.312
min	25.58%	0.725	2.18%	0.019	0.05%	52.58%	0.129	0.312
mean	37.73%	5.500	14.36%	1.296	3.30%	85.82%	446.111	46.341
median	37.95%	3.344	8.90%	0.213	0.64%	90.94%	85.292	1.880
max	51.93%	20.175	51.03%	9.567	24.20%	99.28%	2654.277	393.713
Floating-Point								
ammp	36.50%	6.301	17.14%	1.435	3.90%	77.22%	263.787	212.993
applu	45.85%	6.702	14.56%	4.641	10.08%	30.75%	53.137	54.997
apsi	43.42%	4.416	10.07%	0.961	2.19%	78.24%	831.712	123.641
art	60.38%	34.586	54.88%	14.997	23.80%	56.64%	794.558	0.738
equake	56.76%	11.004	19.30%	8.077	14.16%	26.60%	588.951	64.344
facerec	35.55%	4.095	19.91%	1.277	6.21%	68.81%	357.810	14.389
lucas	26.10%	3.930	14.43%	3.015	11.07%	23.29%	1062.433	2799.049
mesa	31.13%	2.503	6.12%	0.354	0.86%	85.86%	64.424	4.701
mgrid	38.82%	3.659	9.33%	1.548	3.95%	57.70%	2.080	36.262
sixtrack	28.64%	0.802	2.80%	0.043	0.15%	94.60%	19.902	0.987
swim	40.54%	6.465	15.91%	4.975	12.24%	23.05%	147.654	636.572
wupwise	49.01%	5.793	11.81%	3.639	7.42%	37.19%	4.877	34.145
min	26.10%	0.802	2.80%	0.043	0.15%	23.05%	2.080	0.738
mean	41.06%	7.521	16.77%	3.747	8.00%	54.99%	349.277	331.902
median	39.68%	5.104	14.50%	2.281	6.81%	57.17%	205.721	45.630
max	60.38%	34.586	54.88%	14.997	23.80%	94.60%	1062.433	2799.049
Java Server Benchmark								
jbb	34.89%	3.817	10.71%	0.181	0.51%	95.26%	60.242	1.558
w.r.t. to int	0.919	1.142	1.20	0.847	0.79	1.03	0.706	0.829
w.r.t. to fp	0.879	0.748	0.74	0.079	0.07	1.80	0.293	0.034

Table 8. Data Cache Measurements: Architectural measurements of integer and floating-point SPECcpu2k benchmarks. Measurements of SPECjbb2005 are compared against the median of integer and floating-point measurements.