Measuring and Predicting Computer Software Performance: Tools and Approaches

by

Augusto Oliveira

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

© Augusto Oliveira 2015

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Measurement-based software performance evaluation is essential to computer science and industry alike, yet despite its widespread adoption, the current level of statistical rigor is inadequate, putting published results into question: for example, the majority of publications fail to report *any* dispersion metric at all. To foster widespread adoption of statistically rigorous performance evaluation, the first part of this thesis proposes the use of formal experiment design and non-parametric analysis techniques, and presents a distributed infrastructure that lowers the cost of rigorous experimentation by automating the experiment design and execution process, while minimizing the variability in computer performance response metrics. Then, to address cases where rigorous performance experimentation is infeasible, either due to infrastructure costs or unavailability of target platforms, the second part of this thesis builds on the previously discussed techniques and infrastructure to introduce two performance prediction techniques: one to predict when code changes will cause performance changes during software development, and another to predict performance metrics on unavailable platforms using benchmark-based statistical models.

Acknowledgements

Thanks to my advisor, Prof. Sebastian Fischmeister, for the opportunities, the guidance, and the encouragement. It is still incredible to me that if not for that chance encounter in RTAS'09, in San Francisco, I would not be here working on awesome things in this great group. Really, thanks a lot[1].

Thanks to Amer Diwan at Google, Prof. Matthias Hauswirth at the University of Lugano, and Peter Sweeney at IBM Research, for helping Prof. Fischmeister push me toward more interesting, useful, and challenging work. It feels like I had three "bonus" advisors for much of this journey, and I am very thankful for it.

Thanks to Prof. J. Nelson Amaral, Prof. Lin Tan, Prof. Rodolfo Pellizzoni, and Prof. Tim Brecht for participating in the review committee, and for their invaluable feedback.

Thanks to J. C. Petkovich for building DataMill with me, and patiently enduring my rants about Python, consumer electronics, LaTeX, the electricity in E5, Lordran, and anything else that was the target of my ire any given day. "0.05? Why?", "Because stats!"

Thanks to the Real-Time Embedded Software Group, primarily for the commiseration opportunities. Ahmad, Sam, Akramul, Hany, Wallace, Ramy, everyone: it was great sharing this time with you all.

[1] Passive voice still has its places, though! I mean, places are still had by pa... ugh, you win this one.

Dedication

Aos meus avós.

Aos meus pais.

À minha família.

À Rafaela.

Table of Contents

List of Tables x

List of Figures xii

1 Introduction 1

2 Related Work 14

2.1 Measuring Computer Performance 14

2.1.1 Statistics in Computer Science 14

2.1.2 Hidden Factors in Computer Performance 15

2.1.3 Performance Evaluation Infrastructures 17

2.2 Predicting Computer Performance 19

2.2.1 Managing Performance Testing Overhead 19

2.2.2 Test Selection 20

2.2.3 Computer Performance Prediction 21

I Measuring Computer Software Performance 24

3 Designing and Analyzing Reproducible Experiments 26

3.1 Experimental Setup 27

3.1.1 Latt 27

3.1.2 Background Load Configurations 28
3.1.3 Experimental Platforms 29
3.2 Repeating Prior Experiments 29
3.2.1 Experiment 1: Idle 31
3.2.2 Experiment 2: Compile 31
3.2.3 Experiment 3: Compile with Compression 31
3.2.4 Experiment 4: Encode 32
3.2.5 Experiment 5: Compile with Idleness Data 33
3.2.6 Discussion 33
3.3 Factorial Design 34
3.3.1 End-to-End Times 36
3.3.2 Latency 38
3.4 Quantile Regression 40
3.5 Discussion 44

4 Lowering the Cost of Reproducible Experiments 46

4.1 DataMill: The User Experience 47
4.1.1 Packages 47
4.1.2 Experiment Creation 49
4.1.3 Experiment Results 50
4.2 DataMill: The Infrastructure 50
4.2.1 Master Node 51
4.2.2 Worker Node 53
4.2.3 Factor Variation 54
4.3 Case Studies 55
4.3.1 XZ vs. bzip2: Best Bang for Your Buck? 55
4.3.2 Perlbench: Link Order Effect 59
4.4 Discussion 62

5 Minimizing Performance Variability **64**

5.1 Experimental Setup . . . 64

5.1.1 Data Analysis . . . 65

5.2 Experiment 1: ASLR vs. Environment Padding . . . 67

5.2.1 Results . . . 68

5.3 Experiment 2: ASLR vs. Position-Independent Executables . . . 70

5.3.1 Results . . . 71

5.4 Experiment 3: Warm-ups and Reboots . . . 72

5.4.1 Results . . . 73

5.5 Experiment 1 Revisited . . . 75

5.6 Average Case Variability . . . 76

5.7 Discussion . . . 76

II Predicting Computer Software Performance **78**

6 Predicting Performance Changes During Software Development **80**

6.1 Problem Definition . . . 81

6.2 Anatomy of a Predictor . . . 83

6.3 Case Studies . . . 85

6.4 Perphecy . . . 87

6.4.1 Static and Dynamic Analyses . . . 88

6.4.2 Predictor: Data Processor . . . 88

6.4.3 Predictor: Predictor Engine . . . 91

6.5 Evaluation . . . 95

6.6 Discussion . . . 100

7 Predicting Application Performance on Unavailable Platforms **102**

7.1 Problem Definition 102

7.2 Using Benchmark Performance as Statistical Predictors 106

7.3 Potential Pitfalls 108

7.4 Model Selection 110

7.5 Evaluation 112

7.6 Discussion 118

8 Conclusion and Future Work **120**

A Statement of Contributions **124**

References **126**

List of Tables

1.1 Experiment design. . . . 3
1.2 Example data set for factorial experiment. . . . 4
1.3 ANOVA results for the example experiment. . . . 6
1.4 Illustration of experimental rigor in recent scientific publications. . . . 8

3.1 Factors and their levels. . . . 34
3.2 Factors and their levels for the sequential experiments. . . . 35
3.3 ANOVA results for end-to-end times. . . . 37
3.4 Quantile regression coefficients for the 99th percentile. . . . 43

4.1 Compression rates for each algorithm. . . . 56
4.2 Reduced ANOVA table for XZ execution time on machine F. . . . 58
4.3 Reduced ANOVA table for Perlbench execution time on machines A and K. 62

5.1 Experimental setup. . . . 66
5.2 Reproduction summary. . . . 75

6.1 Software projects used in experiments. . . . 85
6.2 Indicators investigated. . . . 89
6.3 Performance of the algorithm in isolation, and K-fold cross validation. . . . 96
6.4 K-fold cross validation for synthetic training sets. . . . 99

7.1 Benchmarks. . . . 103

7.2 Machines. 105
7.3 Details of worst decile of models. 118

List of Figures

1.1 Graphical representation. 3

3.1 Control flow of the Latt benchmark. 28
3.2 Results of the sequential experiments. 30
3.3 Residual plot for the linear model of end-to-end time response variable, showing no error patterns. 36
3.4 Example of non-normal latency behavior on the Core 2 machine, under the CFS, with no VMStat and the compile load. 38
3.5 Residual plots for the linear model of the latency response variable. 39
3.6 Quantile regression effects for the most relevant factors for the latency response variable. 42
3.7 Effect of autogroup on the *ab* benchmark. 44

4.1 Dhrystone package directory structure. 49
4.2 The DataMill infrastructure. 51
4.3 The worker node disk partitions. 53
4.4 XZ vs. bzip2, compression rate. 57
4.5 Effect of GCC optimization flags on XZ, by host. 59
4.6 Effect of link order on Perlbench execution time, cache misses, and page faults. 61

5.1 ASLR and environment padding effects on Bzip2 on machine 128. 65
5.2 ASLR and environment padding effects by benchmark. 68

5.3 ASLR and environment padding effects by machine. 68

5.4 Worst case ASLR effect on astar on machine 81. 69

5.5 Coefficient of variation for Experiment 1, by machine. 70

5.6 ASLR and `-fPIE` effects. 71

5.7 Coefficient of variation for Experiment 2. 72

5.8 Reboot and warm-up effects. 73

5.9 Coefficient of variation for Experiment 3. 74

6.1 Graph of a period in the development of the HotSpot JVM, with performance data from the DaCapo avrora benchmark. 81

6.2 Abstract model of a performance change predictor. 84

6.3 Indicator performance at various thresholds for HotSpot and Git. 92

6.4 Predictor hit and dismiss rates by size of training dataset. 97

6.5 Predictor hit and dismiss rates by hops between known and unknown commit, for Git. 98

7.1 Individual SPEC benchmark performance by SPEC geometric mean performance. 104

7.2 Models of sjeng execution time. 106

7.3 sjeng by gobmk, small data set. 108

7.4 Example of overfitting. 109

7.5 Clustering of SPEC CPU 2006 benchmark performance. 111

7.6 Predictive model error by training set size. 113

7.7 Model performance, mid-range machines only. 114

7.8 Model performance, mid-range machines only. 115

7.9 Predictive model error by predictor number limit. 117

Chapter 1

Introduction

Computer performance evaluation is a combination of the measurement, interpretation, and communication of a computer system's speed or capacity [59]. In general terms, the process of computer performance evaluation consists of the following steps: selecting a performance metric, designing an experiment, analyzing the resulting data, and, finally, reporting the results [44].

Performance evaluation is essential in computer research and industry alike. Researchers use performance evaluation to measure the speed of hardware and software, and to compare alternative approaches to a problem. In industry, performance evaluation guides the purchase of hardware and software systems, and performance metrics are usually among the main indicators of quality in engineering projects. Google has reported a 20% traffic decrease due to a 500ms delay in page rendering [87], and Amazon has reported a 1% decrease in sales due to a 100ms delay in page rendering [87].

Experimenters can perform performance evaluation through three fundamental techniques: analytical modeling, simulation, and measurement-based statistical inference [44]. In analytical modeling, the computing platforms (i.e., hardware and supporting software, such as the operating system) and the computing workloads are represented through mathematical models, constructed with manufacturer specifications, analysis tools, and expert knowledge. These models tend to abstract away as much detail as possible while still providing a reasonable approximation of the real system performance. In simulation, a functional model of the computing platform is used to execute the real workload, or step through a trace of aspects of the workload. The precision of both of these approaches depends on the quality of the models created. While analytical models and simulation are undoubtedly useful and can be cost-effective, they depend on the construction of a model

of each platform where performance is to be analyzed.

Measurement-based performance evaluation, the remaining approach, is generally the most accurate, since simplifying assumptions about the platform and the workload are unnecessary [59]. A desirable property of this approach is that experimenters can readily execute workloads on a variety of environments, if they aim to argue that their results (e.g., an algorithm optimization) apply in general, whereas in the other approaches this would require modeling of all target platforms individually. The application of this method, however, requires a working prototype of both the platform and the workload. Nevertheless, measurement-based evaluation is one the cornerstones of empirical computer science, and experimental results are widely used to compare hardware and software objects of study in scientific venues.

Due to the complexity of modern computer systems, many performance metrics can no longer be treated as deterministic, even if the result of the computations are. A general-purpose, multi-processor system, with multiple layers of memory – from registers, to multiple caches, to DRAM – and a full-featured operating system will very rarely take the exact same amount of cycles, or incur the exact same number of cache misses, or correctly predict the same number of branches, when executing the same computation twice. Even hardware performance counters for metrics such as the number of retired instructions, which could conceivably be deterministic, show variations between identical runs in modern processors [100]. This type of performance non-determinism is now well known, but generally ignored given its relatively small effect.

Unfortunately, as the complexity of platforms and workloads increases, more hidden sources of performance variability appear, and ignoring their worst-case effects can lead to wrong conclusions. Many research efforts [69, 70, 38, 49, 34, 29] have demonstrated that seemingly innocuous experimental setup details have surprisingly significant effects on performance, ranging from 3% to as much as 45% in the worst case. These performance changes can be caused by unexpected sources, such as changes to the binary link order, changes to symbol names in the binary, changes to the user name used during experimentation, or switching between similar versions of the Linux kernel.

Researchers and developers can turn to statistical tools to isolate these unwanted sources of variability from the effects they are truly interested in measuring. For simple comparisons between two competing approaches – e.g., bzip2 vs. XZ, two compression algorithms, applied to the same input – statistical tests such as Student's t-test [67] or overlapping confidence intervals for the mean can indicate whether the mean of two populations, as represented by samples from those populations, are equal. These tests do incur the added overhead of repeated measurements (to allow for an estimation of variance) and

specific properties about the distribution of the data, but they can take the variability in the data into account to "filter out" the noise from the information of interest.

	Low (-)	High (+)
Compressor (x_1)	bzip2	XZ
Opt. Flag (x_2)	-O2	-Os
Input Size (x_3)	10MB	100MB

Table 1.1: Experiment design.

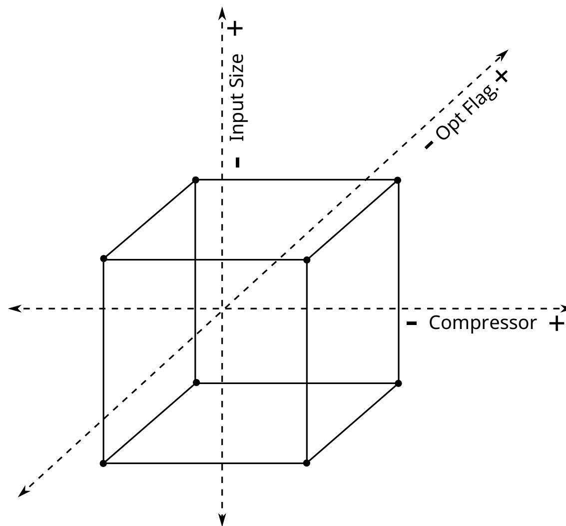


Figure 1.1: Graphical representation.

These simple statistical tools generalize to more complex experiments, where multiple factors are investigated together. For example, consider an extension of the previous example: the evaluation of multiple compression algorithms, but now with multiple GCC compiler optimization levels (-Os, optimizing for size, and -O2, optimizing for speed), and applied to multiple inputs, as listed in Table 1.1. Each of those three factors can vary independently from the others. Figure 1.1 represents this experiment design as a cube in three-dimensional space, each vertex denoting a measurement point: bzip2 at -O2 and 10MB input at the bottom left, XZ at -Os and 100MB input at the top right, and all other combinations at the other vertexes.

The most efficient way to carry out a *factorial experiment design* such as this example is to vary the factors *at the same time*, as opposed to individually and in isolation [67]. To calculate internal variability in the data, developers should repeat performance measurements a number of times (i.e., collect *replicates*) for each combination of factor levels (the cube's vertexes). Table 1.2 shows a synthetic data set to serve as an example; each combination of factors (x_1, x_2 and x_3 for compressor, optimization flag and input size, respectively) contains three replicates, totaling $2 \times 2 \times 2 \times 3 = 24$ individual observations. Their results are shown in seconds, in the y_i columns; note that the experimental conditions

in each row are the same, yet the execution times are different, to represent the variability discussed above.

Compressor (x_1)	Opt. Flag (x_2)	Input Size (x_3)	y_1	y_2	y_3
bzip2 (-)	-O2 (-)	10 (-)	31.139	24.726	25.295
bzip2 (-)	-O2 (-)	100 (+)	45.153	46.096	56.602
bzip2 (-)	-Os (+)	10 (-)	43.440	33.648	31.000
bzip2 (-)	-Os (+)	100 (+)	51.720	54.905	54.666
XZ (+)	-O2 (-)	10 (-)	20.801	21.575	16.429
XZ (+)	-O2 (-)	100 (+)	41.198	47.056	42.211
XZ (+)	-Os (+)	10 (-)	5.932	14.376	15.075
XZ (+)	-Os (+)	100 (+)	30.286	23.144	27.570

Table 1.2: Example data set for factorial experiment.

Data analysis of two-level factorial designs such as this example usually consists of creating a model that relates the response variable to the factors using least-squares linear regression, a statistical method to calculate the best fitting linear function to a given data set, assuming the resulting model's errors are normally and independently distributed (i.e., the occurrence of one error value does not affect the probability of the others), and have constant variance. The resulting model, which estimates the mean for a given combination of factor levels, will be of the form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \ldots + \beta_{12} x_1 x_2 + \ldots + \beta_{ik} x_i x_k + \ldots + \epsilon \quad (1.1)$$

where y is the response variable, which is related to k factors, each with a level x_i and a coefficient β_i, where $i = 1, \ldots, k$. Terms of the form $\beta_{12} x_1 x_2$ allow the modeling of interactions between factors. Higher-level interactions (between three or more factors) can also be included in the model. The ϵ term is a random error component that incorporates other sources of variability (uncontrolled factors and measurement error). This model describes a hyperplane in the k-dimensional space of the k factors, and β_0 is the intercept for the plane, i.e., where the plane crosses the response variable dimension when all $x_i = 0$.

In the case of our compression example from Table 1.1, let x_1 represent the compression algorithm factor. We attribute indicator values to its levels: -1 for low (bzip2) and +1 for high (XZ). The compression algorithm effect β_1 will then be *half* the expected change in the mean response variable when the scheduler factor is modified and all other factors are unchanged. In other words, the low indicator value $x_1 = -1$, will subtract the value of

β_1 from the response variable, the high indicator value $x_1 = +1$, will add it; hence β_1 is half the total expected change in the response value when x_1 changes from -1 to $+1$. A positive β_i signifies positive feedback on the response variable with an increasing value of x_i; a negative β_i signifies a negative feedback under the same conditions. Interaction terms for each combination of factors, e.g., β_{12} for the interaction between compressor and optimization flag, will quantify the synergy between factors: its value will denote if one of the compression algorithms benefits from -Os *to a greater extent* than the other does.

The resulting linear model for the data set in Table 1.2 is:

$$\hat{y} \;=\; 33.502 - 8.031x_1 + 9.882x_3 - 4.719x_1x_2 \tag{1.2}$$

This model suggests that XZ is faster than bzip2 ($\beta_1 = -8.031$), changing the input size from 10MB to 100MB adds approximately 19.76s to execution time on average ($\beta_3 = 9.882$), and that XZ *exclusively* benefits from the -Os optimization flag ($\beta_{12} = -4.719$). The missing coefficients (e.g., β_2) belong to statistically insignificant factors in this case, and were omitted here for simplicity; we discuss them in more depth shortly.

Analysis of variance [67] (ANOVA) can partition the total amount of variability in our data set between an *explained* portion – the variability created by the controlled factors – and the *unexplained* portion – the variability created by measurement error or the uncontrolled factors – making it clear to developers how susceptible to hidden factors their experiment is. The total variability in the data set is called the total sum of squares (SST), the unexplained variability is the error sum of squares (SSE), and the explained variability is the regression sum of squares (SSR). These are formally defined by:

$$SST = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{1.3a}$$

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{1.3b}$$

$$SSR = SST - SSE \tag{1.3c}$$

where n is the number of observations, y_i is the ith observed value, $\bar{y}$ is the mean over all observations, and $\hat{y}_i$ is the model's estimated value for observation y_i. Intuitively, the total sum of squares is the sum of distances between each observation and the overall mean, i.e.,

the total variability in the data. The error sum of squares is the sum of distances between the observed values and the model's estimates for that value, i.e., the error incurred by the model. Finally, the regression sum of squares is the remaining variability after the error is removed, or the *explained* variability. The regression sum of squares can be further divided into each factor, as we will demonstrate shortly.

Factor	Effect (s) Low 95%	Estimate	High 95%	Sum Sq.	p-value
x1	-9.938343	-8.0307	-6.1230735	1547.81	< 0.01
x2	-3.262593	-1.3550	0.5526765	44.06	> 0.01
x3	7.974490	9.8821	11.7897598	2343.75	< 0.01
x1:x2	-6.626593	-4.7190	-2.8113235	534.45	< 0.01
x1:x3	-2.016676	-0.1090	1.7985932	0.29	> 0.01
x2:x3	-3.554760	-1.6471	0.2605098	65.11	> 0.01
x1:x2:x3	-2.430760	-0.5231	1.3845098	6.57	> 0.01
Error				310.95	

Table 1.3: ANOVA results for the example experiment.

Table 1.3 shows the results of analysis of variance for this example experiment. The factor column lists the source factor for the variability. The effect column lists the values of β_i calculated by linear regression, and the boundaries of their 95% confidence interval. The sum of squares column shows the portion of the total sum of squares attributable to each factor (i.e., the higher the sum of squares attributed to each factor, the more variability was *introduced* by that factor). The total regression sum of squares can be calculated by adding together each factors' sum of squares. The p-value column shows the statistical significance of each factor's effect. For example, a significance level below 0.01 means that a factor is significant (i.e., has a non-zero effect) at the 99% confidence level. The statistical significance of an effect does not indicate its magnitude, which is captured by the β_i model coefficients (c.f., the effects column).

The error sum of squares is shown in the last row. Its value, 310.95, is an upper bound on the variability attributable to the sum of all uncontrolled factors. This value is small in comparison to the compressor and input size factors (x_1 and x_3, respectively), but comparable to the compressor-to-optimization-flag interaction ($x_1 : x_2$). Therefore, for this experiment, it may be useful to try and improve measurement techniques to reduce variability in the data set.

This relationship between the explained and unexplained variability can be used to

calculate the fraction of explained variability, or the R^2 statistic of a model:

$$R^2 = 1 - \frac{SSE}{SST} \tag{1.4a}$$

In our example, $R^2 = 0.936$. This means that, out of the total variability observed in the data set (total sum of squares) 93.6% is attributable to one of the factors or a combination thereof, and the remaining 6.4% is attributable to error (error sum of squares). This indicates that there is a strong correlation between the factors investigated and the response variables, and (assuming the assumptions required by linear regression hold) the choice of linear regression was a correct one.

The advantages of using factorial designs and creating statistical performance models are clear: quantification of independent factors *and their interactions*, an upper bound on the unexplained variability in a data set, and optimal number of observations required – minimizing evaluation cost – all while allowing formal statistical methods for hypothesis testing, accounting for any natural variability in the data. Assuming linearity around the experimental conditions, our model of compressor performance (Equation 1.2), can even predict performance for input sizes near the levels explored (e.g., 50MB, 125MB) by varying the value of x_3, giving confidence intervals around each predicted value.

Despite these clear and numerous advantages, computer research and computer engineering still trail other fields in the adoption of even the simplest statistical tools. This has been recently brought to the attention of the community by several researchers [98, 93, 23, 6], and, in an effort to improve the status quo, some conferences such as PLDI [79] have started rewarding authors that publish experiment code and raw result data. However, the publication of experimental data and artifacts is still optional, and only exercised by a minority of authors.

To further illustrate the state of experimental methodology in computer science, we conducted a survey of the recent published papers in SOSP 2011, ASPLOS 2012, OSDI 2012, ATC 2012, EuroSys 2011, and PLDI 2012. The results are shown in Table 1.4. The columns, in order, show the total number of papers, the fraction that contains empirical performance evaluation, the fraction that describes their hardware platforms and software versions, the fraction that contains a comparison (either with a baseline or a competitor), the fraction of those that contain a formal statistical test for that comparison, the fraction that performs the experiment under different conditions as a sanity check, the fraction that contains a dispersion metric (standard deviations, confidence intervals, empirical CDFs, etc.), the fraction that published the software being evaluated, the fraction that

uses a publicly available workload (established benchmarks, input files, etc.), and, finally, the fraction that published their resulting data.

Conference	No.	Perf. Eval	HW Desc.	SW. Desc.	A/B Test	Formal Test	Sanity Check	Disper	Public SW	Public Load	Public Data
SOSP'11	27	93%	92%	60%	91%	0%	24%	44%	4%	68%	0%
ASPLOS'11	36	61%	100%	55%	54%	0%	5%	31%	4%	82%	4%
OSDI'10	30	93%	71%	61%	54%	0%	0%	32%	7%	39%	0%
ATC'12	40	95%	87%	58%	68%	0%	3%	32%	8%	26%	0%
EuroSys'11	22	86%	95%	74%	84%	12%	42%	47%	21%	58%	0%
PLDI'12	45	64%	90%	79%	93%	0%	37%	10%	24%	72%	0%

Table 1.4: Illustration of experimental rigor in recent scientific publications.

While many papers include an empirical performance evaluation, many papers do not list the versions of the software used for these experiments. Worse, many publications do not contain an empirical comparison to a baseline or a competing approach, listing only absolute performance numbers for reader interpretation. Out of those papers that do include a comparison with a baseline or competitor, only a vanishingly small fraction formally tests that their performance figures are different from the baseline. Although the benefits for using dispersion metrics, as described above, have been pointed out by many researchers [93, 24, 94] in the past, a number of papers that do a performance evaluation do not use *any* measure of dispersion in their performance evaluation.

The lack of rigorous performance evaluation is often blamed on the fact that experimentation is difficult, costly, and will slow innovation [93], but the alternative to statistically rigorous evaluation is the risk of coming to incorrect conclusions. Any publication reporting a performance effect within the range of the hidden factors described above may be simply relying on noise as if it were a true effect, invalidating results and possibly misguiding follow-up research. Table 1.4 also shows that only a small minority of authors make their experiment implementation and results publicly available, which precludes other researchers from reproducing their results or re-analyzing their data, making it even more difficult for rigorous evaluation to take hold.

The current overhead of robust performance evaluation is clearly more than what the community is willing to incur. We believe, therefore, that the solution to these problems must be centered around making it easy for researchers to perform rigorous empirical performance evaluation. The first part of this thesis presents tools and approaches to lower the cost and effort required for statistically rigorous computer performance evaluation.

Chapter 3 demonstrates the danger of ad-hoc computer performance experimentation, and presents a case for the adoption of factorial experiment design and formal statistical analysis through the reproduction of a previous experiment: the comparison between two Linux CPU schedulers, an experiment originally conducted by Linux kernel developers themselves. Our results show that while we were able to repeat prior unstructured experiments for these schedulers, we were unable to reproduce conclusions based on the experimental results. After conducting a series of experiments, we were not only unable to reach a decisive conclusion, but also unable to determine when enough experimentation would had been done. To conduct a statistically rigorous, reproducible experiment, we applied the textbook factorial experimental design and analysis techniques described above, and we found that while factorial design successfully produced the data set required for analysis, linear regression was effective for only some of our metrics, but was inapplicable for others because they failed to satisfy the necessary assumptions of that technique. Finally, we applied a recently developed statistical technique to our experiment: quantile regression [53], which allows researchers to analyze data in greater detail than simple linear regression, without requiring the data to satisfy specific statistical properties. While linear regression focuses on the mean, quantile regression allows the quantification of factor effects on any given quantile (e.g., the median, or the 99th percentile) of the metric of interest. Our factorial experiments with quantile regression led to correct and reproducible results even in the presence of non-normal computer performance data.

Chapter 4 presents DataMill, a distributed infrastructure for computer performance experimentation which aims to allow scientists and researchers to *easily* produce robust, replicable, and reproducible computer performance experiments at low cost. To do so, DataMill executes the experiments on real hardware (with no virtualization layer) and incorporates the results from Chapter 3 on how to set up factorial designs to arrive at reproducible results, and previous research on hidden factors. The infrastructure automatically varies a selection of hardware and software factors, reducing the effort required by the user to set up the experiment while simultaneously increasing the robustness and applicability of the experimental results to a wide range of factors. The user need not know the details of the underlying mechanisms required to vary these factors and may simply take advantage of the DataMill infrastructure. Besides making it easy and low cost to the user, DataMill also aims to alleviate the problem of data availability and the reproduction of experimental setups. Based on our own experience, in addition to not publishing results alongside their publications, few researchers respond to inquiries for experiment setup details and requests for code or raw data. In DataMill, all experiment setup parameters and their experiment files remain stored in the infrastructure. Users can choose to make their experiments public facilitating the replication and reproduction of the their

experimental configurations and results. We believe DataMill has the potential to raise the bar of performance evaluation experiments by making the data publicly and consistently available.

Then, to reduce uncontrolled performance variability on DataMill, Chapter 5 presents an extensive empirical evaluation of hidden memory-layout-based performance effects. This exploration determines (1) how frequently these effects appear, (2) the magnitude of their effect on mean performance, and (3) how much variability they add to performance. The experiment explored five methods of varying the memory layout of an application: POSIX environment size, Linux's address space layout randomization feature, the position-independent executable compiler option, experiment warm-up phases, and reboots between individual runs. By exploring these factors in factorial designs, we quantify not only the average effect (i.e., by how much do they displace mean performance) of several layout-affecting factors and factor interactions, but also their impact on an experiment's variance. Finally, we present our findings as to what combination of factors leads to the most deterministic (i.e., lowest variance) performance results, so that researchers may use this methodology to achieve the most repeatable results.

The approaches and tools described in Chapters 3 through 5 reduce the cost of rigorous, repeatable computer performance experiments, but there are still cases where empirical performance testing is justifiably impractical or even impossible. Part II of this thesis presents performance *prediction* techniques that build on the measurement approaches and tools presented in Part I to reduce or replace direct performance measurement whenever necessary.

The first such case is when the *volume* of performance testing is impractical. For example, if developers of the the HotSpot Java Virtual Machine [72] were to execute each benchmark in the SPECjvm2008 [86] and DaCapo [5] suites *once a day*, it would take over three hours on a modern quad-core 3.60GHz Core i7 CPU. Since there are on average five daily commits for HotSpot, running every benchmark *multiple times* – a requirement for variance estimation and robust statistical analysis, as discussed above – on *every* commit can require multiple days of computation per day of development. Keeping up with this computation demand requires a dedicated cluster of machines, which many software projects cannot spare for this purpose alone. Even if that is an option, each change would require hours of testing before being committed, limiting the throughput of commits for the project. The compromise developers currently employ is periodic testing (e.g., weekly, or before every public release), which also detects performance regressions, but has the following problems. First, it requires further effort on the part of the developers, who need to "bisect" the commit history in order to find the offending change (or changes) in case of regression. Second, it not only delays the detection of regressions, but also increases the

chance that a problem is ingrained in the design of the software and not easily solved once found. Third, if one commit causes a performance regression and another causes a performance improvement between benchmark runs, these performance changes may go entirely unnoticed by the developer, who could otherwise fix the regression from one change while keeping the speedup from the other.

To enable minimal performance regression detection delays without requiring costly performance evaluation infrastructure, Chapter 6 introduces Perphecy. Perphecy (a portmanteau of performance and prophecy), is a general, lightweight, effective, and reliable performance-change prediction strategy. Perphecy is general, because it is programming language, operating system, and processor architecture independent; lightweight, because it does not require long-running analysis to emit a prediction; effective, because it greatly reduces the amount of performance test runs; and, finally, reliable, because it detects the majority of performance-affecting code changes. If developers can predict which commits will cause performance regressions on which benchmarks, they can complement their traditional periodic performance tests with frequent pin-point partial testing, which does not significantly increase performance testing time, yet dramatically reduces performance regression detection delay. By executing *only* benchmarks that are likely to incur performance changes on a subset of commits, they avoid wasting computing time and power, and still detect performance regressions immediately.

The second case where direct performance measurement is infeasible is when the target platform is *unavailable*, either due to cost – developers cannot afford to acquire all candidate targets for empirical comparisons – or manufacturing lead-times – vendors have designed and prototyped the platform, but have not yet made them available on the market. If a timing-accurate emulator of the target platform is available, developers can execute their code on the emulated platform and collect performance metrics. Alternatively, developers can correlate their application's performance to platform properties (e.g., CPU architecture, clock rate, memory quantity and bandwidth), and use the resulting model to make predictions. Finally, developers can use a standard benchmark suite to compare their current platforms with a new target, making the assumption that their application's performance will correlate with the performance of the benchmark suite.

Relying on benchmarks is often the most cost-effective option, given the difficulty of simulating or modeling performance accurately, cheaply, and quickly. Platform manufacturers can publish standard benchmark results, and developers can then predict the performance of their applications using those numbers. The effectiveness of this strategy is evidenced by the number and variety of benchmark suites available, and the widespread use of benchmark scores in promotional material for both hardware and software. However, while a benchmark suite can suggest what the performance difference between two platforms will

be for the *average* application, benchmark suites are not guaranteed to represent any *single* application accurately. For example, the SPEC CPU 2006 documentation explicitly states that "when you are doing vendor or product selection, SPEC does not claim that any standardized benchmark can replace benchmarking your own actual application" [85]. Despite this disclaimer, SPEC does not otherwise indicate how the benchmark suite should be used by developers in comparing platforms. Nonetheless, developers use benchmarks to estimate how their application's performance will transfer between machines, either implicitly assuming their application's performance will scale at a similar rate as that of the benchmark suite, or by constructing a case based on performance-relevant similarities between their application and a benchmark (i.e., instruction mix, memory access patterns, memory hierarchy pressure, etc.). Both of these approaches are error prone, as the benchmark suite may not represent the developer's specific application, and analysis of performance-relevant features is difficult and may overlook important aspects of either the application or the target architecture.

To enable performance prediction when the target platform is unavailable, Chapter 7 introduces performance fingerprinting, a statistical approach to create models of application performance based on standard benchmark performance, using benchmark scores to represent platforms in lieu of their hardware characteristics. Developers create a performance fingerprint of their application by measuring the performance of their application *and* a benchmark suite on DataMill. By combining that performance fingerprint with the performance of the benchmark suite on the target machine (provided by the target machine vendor) developers can accurately predict the application's performance on the target platform, without making any implicit assumptions about the relationship between their application and the benchmark suite.

This thesis makes the following contributions:

- Chapter 3 [19]:
  - A demonstration of the pitfalls of unstructured computer performance experimentation;
  - A demonstration of the use of factorial design in conducting reproducible computer performance experiments;
  - A demonstration of the shortcomings of linear regression in analyzing computer performance data;
  - A demonstration of the application of quantile regression to successfully analyze non-normal computer performance data.

- Chapter 4 [22]:
  - An infrastructure that:
    1. Automates factorial experimentation;
    2. Provides controlled hardware and software heterogeneity;
    3. Fosters reproducibility and sharing of experiments.
  - A large-scale reproduction of previously published hidden factor effects.
- Chapter 5 [21]:
  - A large-scale experimental evaluation of memory-layout-related performance effects;
  - An experimentation methodology that minimizes performance variability on DataMill.
- Chapter 6:
  - An abstract model of commit-to-commit performance-change predictors;
  - A number of lightweight indicators of performance changes;
  - A heuristic to convert indicator values into usable performance change predictions;
  - An extensive evaluation of predictors that demonstrates their effectiveness on a wide range of production software.
- Chapter 7:
  - The use of standard benchmark scores as predictors in linear models;
  - The use of DataMill and regression-model selection techniques to create predictive models of application performance;
  - An extensive evaluation of predictive models that demonstrates their effectiveness and sensitivity to various parameters.

Chapter 2

Related Work

In this chapter we present previous work in computer performance evaluation and computer performance prediction which are relevant to the research presented in this thesis. We present these related publications in two sections, each relevant to one of the parts of this thesis: Section 2.1 presents the state of the art in empirical computer performance evaluation, while Section 2.2 presents the state of the art in computer performance prediction.

2.1 Measuring Computer Performance

There has recently been growing focus on the application of statistical methods on empirical computer performance evaluation, and the adoption of such methods by the scientific community, in addition to extensive findings into the danger of hidden factors in performance evaluation. Additionally, there are several efforts to design and build computer performance evaluation infrastructures, aiming to lower the cost of robust performance evaluation.

2.1.1 Statistics in Computer Science

Tichy [93] argues that the state of empirical experimentation must improve, despite many arguments to the contrary (e.g., experimentation is too expensive, there is too much noise, technology changes too quickly). Similarly, Denning [23], former president of the ACM,

points out that the state of experimentation in our field lowers its credibility as a science. Blackburn et al. [6] raise the concern that the current standard of experimentation methodology is a particularly significant problem in managed languages, and that it often produces bad results or no result at all.

The Evaluate Collaboratory [13] is a hub of researchers, from academia and industry alike, that are concerned with the state of empirical computer science. The collaboratory maintains a list of relevant publications and evaluation anti-patterns, with the goal of improving the state of experimentation. Vitek and Kalibera [98] report that in PLDI'11, a selective (23.3% acceptance rate) conference where experimental results are commonly published, 39 of the 42 papers that published experimental results did not report a measure of dispersion in their data, a requirement for hypothesis testing.

Georges et al. [28] argue for the use of statistically rigorous analysis methods, however, they only go so far as linear regression, which we demonstrate in Chapter 3 to be insufficient in some cases, and propose the use of quantile regression in its stead. Due to its relatively recent development, quantile regression is also gaining acceptance in other fields. Cade and Noon [8] present the method to ecologists, and Koenker and Hallock [54] demonstrate the method to economists. Similarly, Kalibera and Jones [50] present a random effects model tailored to computer experiments, and also note that current textbook approaches may be insufficient in the field.

In another work, Kalibera and Jones [51] address the problem of balancing accuracy with benchmark execution time. Given a platform and benchmark, their approach can be used to determine the number of repetitions necessary to achieve a given confidence interval. Their study shows that while often around 5 to 10 executions are needed, some less deterministic benchmarks can require dozens (and some extreme benchmarks, like lusearch or xalan from the DaCapo suite [5], may require hundreds) of executions to reach 95% confidence intervals that are tight (i.e., have a half-width within 1% of the mean).

We share these authors' concerns; as we have shown in Chapter 1, this lack of rigor is still widespread, and more work is required before the standard of empirical computer performance evaluation is equivalent to that of other sciences.

2.1.2 Hidden Factors in Computer Performance

Mytkowicz et al [70] show that even presumably innocuous factors, such as link order of a program or the Unix environment size can significantly affect performance measurement results. Their recommendation is to randomly vary those factors, however, this further multiplies the number of required benchmark runs.

Kalibera *et al.* [49] demonstrate that the random initial state of the machine had a significant impact on the results of 6 different benchmarks. Kalibera concluded that benchmarks were likely to be influenced by their random initial state. Four of the benchmarks tested were Fast Fourier Transforms, it is possible that there is a greater incidence of sensitivity to memory layout in computation-heavy benchmarks.

Chen *et al.* [12] recently reported observing small, within 2%, variations in performance from different link-orders and POSIX environment sizes in their study of iterative optimization, a process highly dependent on reliable, reproducible and consistent results.

Curtsinger *et al.* [16] developed a tool for randomizing the memory layout of code, stack and data memory regions of a program at runtime, allowing a researcher to control the effects of memory layout. The tool forces memory-layout effects to approximate a Gaussian distribution, permitting the use of traditional statistical analysis techniques. Curtsinger also reports performance regressions of up to 57% as a result of manipulating link-order.

Harji et al. [38] show that the Linux kernel has had a series of performance-affecting issues, and that papers that present data measured on Linux could contain incorrect results. For example, the authors observe performance regressions of as much as 45% between two subsequent versions of the kernel. Gu et al. [34] show that simply changing the names of symbols in Java code can significantly affect the performance of applications. In some cases, these factors caused as much as 20% variation in cache miss count. Georges et al. [29] show that non-determinism in just-in-time compilation and garbage collection cause significant variability of as much as 6.2% in Java benchmarks. Gil et al. [30] report that simply restarting a virtual machine can cause as much as 3% performance variations in benchmarks.

Kalibera et al. [48] discuss how non-determinism in modern systems complicates the reliable detection of performance regressions. Blackburn et al. [5] describe how software relying on garbage collection may exhibit significantly different performance with different heap sizes and garbage collection policies.

The abundance of these hidden effects and the reported lack of experimental rigor discussed above are a dangerous combination. Even if these worst-case effects are rare, researchers must explicitly quantify unexplained variability in their data in order to avoid publishing incorrect conclusions. However, as we discussed in Chapter 1, a vanishingly small minority of researchers explicitly conduct "sanity checks" in their experiments, meaning either that those researchers ignore the risk of hidden factors, or that the cost of rigorous evaluation is still too high to bear.

2.1.3 Performance Evaluation Infrastructures

Desprez *et al.* [24] surveys numerous large-scale computing installations that follow a similar objective to ours; to lower the cost of reproducible and extensible experiments. The authors provide a survey of experimental methodologies and survey a selection of experimental testbeds. In general, the testbeds surveyed are comprised of homogeneous nodes and do not exhibit significant variation in the hardware used, and therefore do not lead to results that generalize over varied hardware platforms. While the computing resources available are vast and the installations provide support for complex experiments (e.g., distributed systems), the experimental setup process is described as manual and arduous despite exposing high-level interfaces.

Most notable among the surveyed computing installations is OpenCirrus [9]. In addition to virtualized environments they also provide access to the physical machines. The lowest-level service consists of a physical resource set (PRS). A PRS comprises a set of VLAN-isolated compute, storage and network resources. PRSs are dynamically allocated and managed through a PRS service. Through the PRS paradigm different levels of abstractions can be configured that suit research applications reaching from low level systems research (e.g., the evaluation of OS kernel parameters) to complex distributed systems (e.g., several virtual machines that run a distributed middleware). OpenCirrus [9] supports access to low-level hardware features, those features expose little variability compared to the applications we are targeting.

PlanetLab [76, 73] provides planetary-scale data services and is used by the research community to deploy, evaluate and access planetary-scale network services. Planetlab provides “slivers” to users, consisting of distributed networked virtual machines (VMs). The VMs are hosted on physical machines that are maintained in a communal fashion. In order to become a user of PlanetLab, one has to dedicate servers to PlanetLab. PlanetLab exposes the application interface for provisioning the slivers and has facilities to isolate the network of the individual slivers. PlanetLab is used predominantly to evaluate and deploy distributed system [78], including content-distribution networks, name services, location services, file-streaming services, fault-tolerant scalable services, peer-to-peer networks, distributed anomaly detection, distributed research allocation, routing overlays, and resource discovery. Because the experimental environment exposed to the user is a virtualized machine, PlanetLab is not an optimal choice for computer performance experiments that seek to evaluate the impact of varying hardware and software environment factors.

Various other experimentation infrastructures have been proposed with similar properties to PlanetLab. Jaffe *et al.* [43] describe a production platform with similar features to PlanetLab. The project links various large data centers in an effort to provide an experi-

mental platform for distributed systems. Unfortunately, the hardware chosen for the data centers is very homogeneous and does not aid in the exploration of a large factor space.

HTCondor [89] is a distributed job scheduler designed for computation-intensive distributed workloads. HTCondor allows users to prepare packages and to submit them for execution, supporting a wide range of experiments, but, since it is geared toward maximizing a cluster's computing throughput, it is not well suited for clean-room performance evaluation.

Netbed/EmuLab [102] is a testbed that provides a mix of simulation, emulation, and live networks to distributed systems researchers. To use the infrastructure, researchers define experiments in a domain-specific language, which a master node parses and then realizes on the machines. Experimenters receive administrator-level access to machines, which are multiplexed between users through FreeBSD's Jail functionality.

Sharcnet [82] is a Canadian network of high-performance computers that researchers can use to execute long-running experiments. The computing power and memory available on Sharcnet make it a very useful platform for researchers that aim to solve complex, but parallelizable, problems. Researchers are restricted to user-level access to the computing platforms, and multiple jobs are frequently scheduled concurrently, making Sharcnet a poor platform for performance evaluation.

The OpenBenchmarking.org [63] initiative, based on the Phoronix Test Suite [64] is a Linux benchmark results database, where users submit their results of standard tests. Data collection is done through the Phoronix Test Suite, which contains a wide array of benchmarks, and a user's datasets can be published and compared through a web interface.

BEEN [52], a general infrastructure for automated regression benchmarking in a heterogeneous distributed environment. BEEN compiles software and benchmarks, takes care of deployment, runs benchmarks, and collects, evaluates, and visualizes results.

These infrastructures generally serve two experiment categories: high performance, distributed computing, or local performance testing. Neither category serves developers or researchers in need of hardware or software *heterogeneity*, nor do they explicitly explore non-functional factors (such as the "hidden" factors discussed above), both of which are necessary features for generality and reproducibility of results. We believe that to promote widespread adoption of robust statistics, an infrastructure or framework must simplify experiment designs, provide heterogeneity on a larger scale than individual researchers or developers can achieve, and support automated exploration of hidden factors.

2.2 Predicting Computer Performance

There is a smaller amount of related work in the area of computer performance prediction, relative to empirical performance evaluation. Since our ultimate goal with performance prediction is effectively *avoiding* direct performance evaluation, we start by reviewing current developer's efforts to cope with excessive performance testing costs. Then, we briefly review test selection techniques, which relate to our goal in that they predict which functional tests will not fail due to a change in the code, avoiding their execution. Finally, we present the state of the art in direct prediction of computer performance metrics.

2.2.1 Managing Performance Testing Overhead

Many projects implement their own performance regression testing infrastructures. For example, the Jikes RVM [2] project used to run performance benchmarks every 12 hours, by checking out the most recent version of their VM, building it, and running a suite of benchmarks including DaCapo. They visualized the resulting evolution of benchmark performance on a public web page [18]. The Jikes RVM project did not run performance benchmarks after every commit. On days without commits, this constituted a waste of resources. On days with many commits, it meant a loss of coverage.

Mozilla's Talos performance regression detection system [88] runs performance tests "every time a change is pushed to the Firefox source repository" [7], and detects a performance regression if the performance changed significantly from before to after that push. They provide an online visualization of all the collected performance test results [68]. Talos uses performance tests, not necessarily full-fledged benchmarks. Running such tests can cost much less time than running complete representative workloads, however it also means that developers have to write a specific performance test for every relevant aspect of the application.

The Linux Kernel Performance project [60] runs and reviews performance regression tests on a weekly basis, and for each major kernel release [11]. They reformat the disk, reboot the system, and run a warm-up load before each benchmark run. To minimize measurement variation they use long benchmark running times and multiple repetitions, which drives up the cost. Despite the significant size of their test suite, they find that their tests only cover a portion of performance regressions, and they call for volunteers to contribute additional resources to enable more extensive benchmarking.

End user applications also can collect performance data in the field, and send that data back to the developers, allowing them to determine performance issues in the environments

in which they are actually used [1, 46, 36]. Open source developers like the Mozilla Foundation even provide publicly accessible dashboards showing the aggregate performance data as measured in deployed instances of their applications (e.g., for Firefox [90]). The advantage of such approaches is that performance is measured where it actually matters, i.e., in the users' context. The disadvantage is that performance problems are only detected after they have affected a user.

Continuous integration systems perform fully automated builds and regression tests of a software, whenever a commit occurs. Traditional systems focus on *correctness* tests, which are either run at regular intervals or are triggered by individual commits. Systems like Jenkins [45] provide plug-ins to also support *performance* testing and reporting of performance numbers. Continuous integration is now also offered as cloud-based services, such as drone.io [25], Travis CI [95], or Coveralls [14].

These efforts involve a large financial and time investment in infrastructure, or provide imperfect coverage, while ignoring the robust statistical techniques and hidden factors described above. Often tests are made on one, or a small number of different hardware platforms, and performance regressions are taken to apply to all possible target platforms. While this may often be a correct assumption, it is a dangerous simplification due to the reverse: a performance regression may be undetected on the benchmarking platform, yet manifest in other platforms after deployment. Achieving the experimental rigor we advocate is often prohibitively costly on this scale, so we believe that predictive models *based* on rigorous experiments are the best way to reduce the cost of experimentation with minimal accuracy loss in these cases.

2.2.2 Test Selection

Yoo and Harman [105] present a survey of traditional regression-test selection techniques that focus on correctness tests. These tests have a significantly different goal than performance regression tests, but the same motivations apply. By predicting which functional tests are *not* affected by a code change, they can significantly reduce the cost of testing. Even though performance tests are of a different nature than functional ones – seemingly innocuous changes can cause significant performance regressions – some of the techniques for functional test selection (e.g., choosing tests according to code coverage) can conceivably apply to performance test selection as well.

Huang et al. [42] address performance test selection for cases where it is impractical to run the complete suite of performance tests for every commit. They use a "white-box" approach that requires whole-program static analysis to determine the run-time cost and

the execution frequency of the code affected by a commit. Their analysis requires interprocedural control and data dependence information, which can be difficult to determine with adequate precision. This is especially problematic for modern languages where programs are dominated by heap data and polymorphic calls. The approach they evaluate does not take dynamic information (e.g., traces from previous executions) into consideration, so their run-time cost and execution-frequency measurements depend on the ability of their static analysis to precisely estimate loop bounds and recursion depths. Finally, they do not predict which tests to run, only whether or not to run *all* tests, even if some are not affected by the change at all.

We believe that test selection techniques are ideal for the case where the target platforms are available, yet the volume of testing is infeasible or requires too large an investment. Performance test selection can function as a screening step, detecting which code changes can cause a regression on which tests, and then executing only those change-test pairs on the benchmarking platform. However, we believe that such a solution should be lightweight, so that it can be quickly executed by developers at every code change, precise, so that the majority of the performance regressions are detected while the majority of performance-neutral changes are dismissed, and general, so it can be readily applied to as many programming languages and application domains as possible.

2.2.3 Computer Performance Prediction

Computer performance prediction approaches differ from test selection in that they aim to predict the actual metric of interest (e.g., run time in seconds, throughput in packets per second, etc.), as opposed to which tests will reveal a change in performance. The existing approaches generally combine statistical methods with domain-specific knowledge such as relative performance between machines, or application information such as memory access patterns and instruction mix.

Huang et al. [41] present an approach to model the performance of an application based on its inputs. The approach involves automatically instrumenting code to measure performance-related "features" such as loop counts, branch counts, and variable values with as little execution as possible. The full execution time is then extrapolated from the values of these features using regression, and the authors achieve errors of less than 7%. This approach requires analysis of application code and executing the application on the target platform.

Lee et al. [56] propose using simulated traces of SPECjbb and SPECint on simulated hardware to model the effect of architectural properties such as register count, cache size

and memory latency on performance and power consumption. They reduce the design space from tens of billions of unique processors to a few thousand by using uniformly random sampling. The approach is able to predict performance with a median error of 10.9%. This approach is useful for processor architects, but does not predict application performance of production platforms with full operating systems.

Gustafson and Todi [35] look for correlations between benchmarks, and determine which sets are most capable of correlating to application performance, but do not address prediction errors. Hoste et al. [40] propose actual prediction of machine ranks, based on picking the benchmark that best represents the application based on a series of expert-designed program characteristics such as instruction mix and working set size.

Performance modeling and prediction has seen much attention in high-performance computing venues. Correctly predicting performance for long-running scientific applications is very useful, and the predictable nature of these applications and systems makes modeling their performance easier than the general case.

Similarly, Sharkawi et al. [83] present an approach that relies on standard benchmarks, performance counters, and a genetic programming tool to project performance from a base HPC machine to arbitrary targets. The authors achieve mean errors between 7.2% and 10.5% of the mean performance.

Yang et al. [104] present an approach that uses the partial execution of long-running HPC applications to predict their full execution time. The approach is able to predict performance across platforms by using a relative performance metric, derived from the partial execution of the application on both the target platform and a reference platform, where full execution data is available. The approach achieves predictions within 2% of the target for the same inputs, and the same number of nodes is used. When the number of nodes of the target platform is allowed to vary, however, estimate errors increase to as much as 25.8%.

Lee et al. [57] use statistical modeling and neural networks to predict how the performance of two HPC applications scale as the number of processors and the working set size per processor change. The authors avoid sampling the entire design space (of 200+ million points) by using uniform at random sampling. The authors achieve median error rates of 16.2% or less for 75% of predictions by modeling 600 samples. These last two approaches require executing the application on the target platform, and exploit the steady-state processing that HPC applications reach during their run time to extrapolate how their performance will scale as the machine itself scales.

Snavely et al. [84] present an approach for performance prediction based on application and machine profiles. Since the approach is targeted for HPC applications, the models focus

on modeling memory and interconnect availability (for machine signatures) and utilization (for application profiles). The authors achieve average errors of 23%. This approach is targeted at HPC applications, and requires a detailed model of the target system.

Marin and Mellor-Crummey [62] present a toolkit to construct an architecture-neutral model of the static and dynamic properties of an application, then apply this model to an architecture description to predict execution times. The application models use computer-specific information such as memory access patterns and control-flow graph traversal patterns. The authors achieve an accuracy of approximately 20% depending on the application. This approach requires a detailed model of the both the application and the target system.

These approaches, while appropriate for their domains, have very significant barriers to entry: the creation of precise analytical models, or access to the target platform. They also lack generality, since they target only a specific type of application or platform. More work is needed, therefore, to predict performance for arbitrary applications, where experimenters have no direct access to the target platforms.

Part I

Measuring Computer Software Performance

“He had a suspicion of plausible answers; they were so often wrong.”

— Arthur C. Clarke, *Rendezvous with Rama*

Chapter 3

Designing and Analyzing Reproducible Experiments

In this chapter, we demonstrate the dangers of ignoring robust statistical experiment design and analysis, the usefulness of factorial experiment design, the shortcomings of linear regression when applied to non-normal data (common in empirical computer performance evaluation), and finally, propose the use of quantile regression to replace linear regression in those cases.

Our running example is the comparison between two Linux processor schedulers: the Completely Fair Scheduler (CFS), and the Brain F. Scheduler (BFS). The Linux scheduler is important, because it potentially affects the performance of platforms ranging from cellphones to large data centers. The CFS [65], Linux's official scheduler since 2007, aims to be a unified solution for all target platforms, scaling to a large processor count. The BFS [55] was created in 2009 to minimize scheduling latency, trying to yield better performance in interactive systems.

The release of the BFS led several developers on the Linux Kernel Mailing List to conduct independent experiments to measure the performance of both schedulers [33]. BFS proponents argue that its focus on lowering scheduling latency yields better interactivity and responsiveness on commodity hardware; a claim that experimenters could verify through scheduling latency benchmarks. Furthermore, CPU and mixed-load benchmarks can measure scheduler overhead, predicting applications' throughput under the BFS.

Initial attempts to verify the effectiveness of the BFS were inconclusive, and after some discussion, a kernel developer released the Latt [3] benchmark. Latt measures scheduling latency while generating CPU load; this ensures the scheduler treats the benchmark process

as it would a typical interactive application. Section 3.1.1 describes Latt. Even with the acceptance of Latt by both sides of the discussion, all experimentation conducted was unstructured and unscientific; unsurprisingly, the kernel developers did not come to a consensus, and the BFS and the CFS are both under concurrent development. We use Latt for all experiments, since it adequately measures scheduling latency, and is already accepted and used by kernel developers, precluding the need for re-validating it. Using Latt is also important because of our goal of reproducing the unstructured experiments that the kernel developers conducted.

3.1 Experimental Setup

Through a series of experiments, we attempt to determine how the BFS and the CFS compare in terms of two metrics: latency and throughput.

3.1.1 Latt

Latt is composed of a server thread and a set of one or more client threads. Figure 3.1 shows the control flow of Latt. At the start of execution, the server thread spawns the client threads, and opens a pipe (a stream for interprocess communication found in UNIX-like systems) to each of them. The client threads then block trying to read from the pipe. At random intervals, the server thread then writes a *wakeup request timestamp* to each of the pipes, freeing the client threads. As soon as the clients finish reading that timestamp, they collect a *woken up timestamp*. The difference between the woken up and the wakeup request timestamps is the time during which the client thread was ready to execute but did not receive the processor, i.e., the scheduling latency.

As soon as each client collects the woken up timestamp, they perform a compression task. This is to avoid scheduler heuristics that may prioritize client threads for consistently releasing the processor before using their full share of CPU time. Latt registers the time to complete these compression operations, and depending on the load of the system, these measurements serve as a surrogate for throughput[1]. The user can configure the number of clients and the size of the data to compress. There must be at least one client (so that latency measurements are collected), but the compression size may be zero, meaning that the client threads perform no work, and Latt only collects latency information.

[1] This approach is problematic, and Section 3.5 discusses its shortcomings.

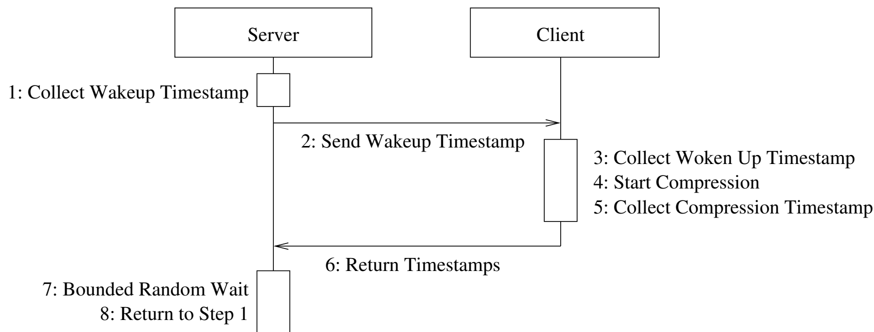


Figure 3.1: Control flow of the Latt benchmark.

Latt thus collects two metrics, scheduling latency (in microseconds) and client throughput (in bytes compressed per microsecond). The lower the latency measurement, the more responsive an interactive system will be. The higher the client throughput, the faster a user would expect a CPU-bound task to complete.

Measuring latency and client throughput in isolation is unrealistic; to solve that, Latt permits defining arbitrary background loads, by taking an arbitrary program as a parameter. Latt collects measurements for as long as that background load executes. For example, if `sleep 30` is passed as a parameter, Latt will collect latency and client throughput data for 30 seconds, resulting in multiple measurements per execution. The background load parameter guarantees that data collection happens only while the load executes (the loads used in our experiments are described in the next section).

By default, Latt outputs only aggregate statistics (such as the mean and standard deviation) of its latency and client throughput measurements. To enable in-depth data analysis, we modified Latt to output each individual latency and client throughput measurement. Our implementation of this feature allocates a large buffer in the beginning of execution and only writes the data to disk once the background load completes, thus minimizing interference on the metrics of interest. Furthermore, in addition to latency and client throughput data, we collected end-to-end execution-time measurements for each execution, to analyze the effect of each scheduler on the background load execution time.

3.1.2 Background Load Configurations

We used three different load configurations: *idle*, *compile*, and *encode*. To differentiate the schedulers under a wide range of conditions, we chose loads that represented the spectrum

from an idle system to one executing a multi-threaded, CPU-bound load. As described earlier, Latt receives each of these loads as a command-line parameter.

The idle load blocks on a timer for a set amount of time, using no CPU time. This load serves to establish a baseline for the metrics on an idle system. The command passed to Latt for this load is `sleep 30`.

The compile load consists of compiling the *xz v5.0.3* [92] open-source compression utility using *GCC 4.6.3*. It is a single threaded load, consisting of a mix of CPU and I/O operations. The command passed to Latt is `./configure; make`.

The encode load is based on *x264 v0.120.x* [96], an open-source H.264 video encoder. It is a multi-threaded, CPU-bound load. The command passed to Latt is `x264 -B 2000 soccer_4cif.y4m -threads 4`. The input file to the encoder is a freely-distributable test sequence [103].

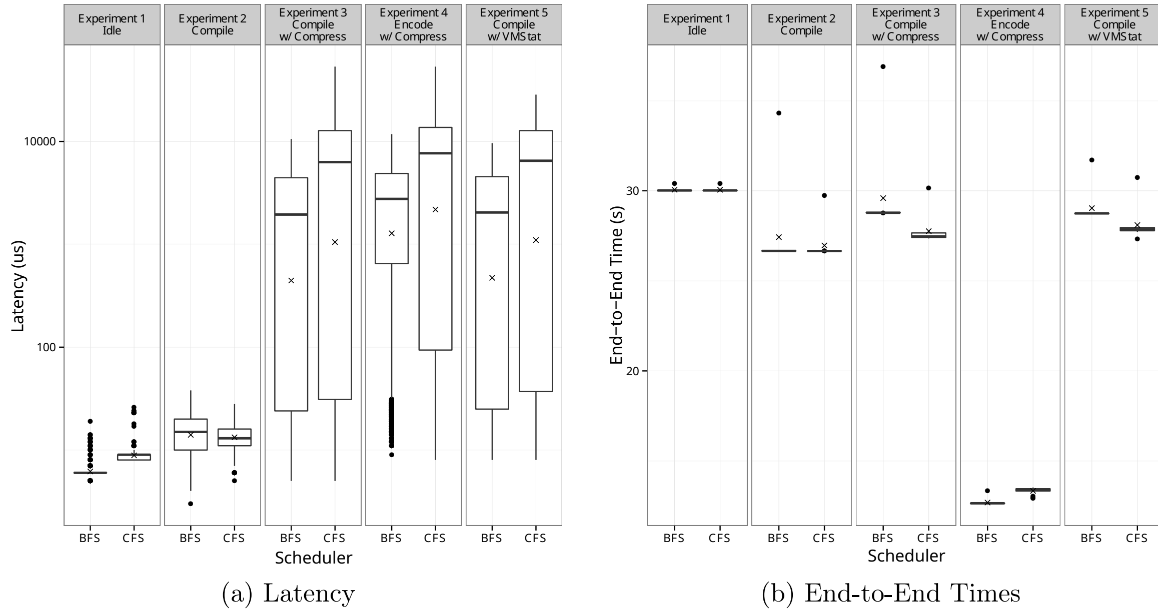
3.1.3 Experimental Platforms

We used two different computers for the experiments. A Core 2 machine, consisting of a 2.4GHz Intel Core 2 Quad Q6600 with 3GB of RAM, and a P4 machine, consisting of a 2.0GHz Intel Pentium 4 with 512MB of RAM. The P4 machine was chosen to measure latency behavior on a slower processor, on which the scheduler choice may prove more significant. Both machines executed version 3.2.7 of the Linux kernel, with BFS version 416. The distribution used was Arch Linux, and only essential software executed during data collection (most notably, no graphical interface was used during benchmarks). Since these machines required remote access, their network connection was active throughout benchmarking. The relatively long running time of the benchmarks and the fact that the machine rejected connections except on the remote terminal port minimize any network interference on the metrics of interest.

3.2 Repeating Prior Experiments

Our first experiments started in an attempt to replicate experiments described in the Linux Kernel Mailing list. The goal of all experiments in this section is to determine how the schedulers compare in terms of latency and end-to-end performance. The initial intuition (according to the developers involved in the discussion) is that the BFS yields better latency, but the CFS yields better end-to-end times. We will now test this intuition with

a sequence of experiments, using information gained along the way to determine the next step.



(a) Latency

(b) End-to-End Times

Figure 3.2: Results of the sequential experiments.

All results presented in this section were collected on the Core 2 machine. For each experiment, the machine boots into the kernel with the appropriate scheduler, executes the benchmark, and offloads the data to a network server. Figures 3.2a and 3.2b show box plots of the latency and end-to-end times under both schedulers for all experiments described in this section. The data set for each experiment consists of ten separate executions of Latt, each of which yields several latency samples and one end-to-end sample. The three horizontal lines in each box represent the upper quartile, the median, and the lower quartile. The lower and upper whiskers extend to the last value within 1.5 times the interquartile range of the lower and upper quartile, respectively. Dots represent outliers that are outside this range. The cross (×) represents the mean value of each distribution. Note that the y-axis on Figure 3.2a is in a logarithmic scale.

3.2.1 Experiment 1: Idle

The first experiment was conducted on an idle system. Latt executed a single client thread, which performed no compression, and we used the idle load. The goal of this experiment was to ascertain if there was a significant performance difference between the schedulers in a system with no load. This experiment also serves to establish that the method used by Latt provides reliable data, that is, the experiment is repeatable.

The plot for Experiment 1 in Figure 3.2a confirms the intuition that the point estimate of mean latency under the BFS is lower than that of the CFS. Nevertheless, even the worst case under CFS is below 30μs (much lower than the human-perceptible 100ms [10]). The standard deviation under both schedulers is very low (under $3\mu s$ for both schedulers), which suggests Latt's method for measuring latency is robust, at least in an idle system.

This experiment confirms the expectations of the developer of the BFS, but the latency behavior measured with this unrealistic load may not be observed in real applications. Therefore, investigating how the schedulers behave under load is still necessary.

3.2.2 Experiment 2: Compile

The goal of this experiment is to evaluate the different schedulers under more realistic conditions than in Experiment 1. This experiment has all the same conditions of Experiment 1, except for the use of the compile load instead of the idle load.

The plot for Experiment 2 in Figure 3.2a shows that, under these conditions, the BFS has a higher mean latency point estimate and higher variance than the CFS. These results contradict the previous experiment *and* the initial claim. However, this could be a scheduling artifact: the lack of work in the Latt client threads may be causing the CFS to prioritize them over all others, since it queues threads by spent processor time. This, in turn, leads to artificially low latency measurements. Adding load to the Latt clients could confirm this hypothesis.

3.2.3 Experiment 3: Compile with Compression

The goal of this experiment is to check whether increasing load in the Latt client threads will affect latency measurements. To accomplish this, we changed two parameters from the last experiment: we increased the number of clients in Latt from 1 to 8, and each client compressed 64kb of data. In a real application, latency-sensitive threads also perform work, which is approximated by the compression functions in Latt.

The plot for Experiment 3 in Figure 3.2a shows that the increase in the number of client threads and the addition of compression to their work cycles causes a reversal in the results from the previous experiment, and now confirm the initial claim once again. In contrast to the previous experiment, now the BFS has better average latency and better worst-case latency than the CFS. The CFS' highest latency measurement approaches 60ms, which is significantly closer to the human perceptible threshold than the BFS' worst observed case. If latencies of 60ms are observed in these relatively powerful desktop systems, the perceptible threshold of 100ms will likely be surpassed on slower processors, such as those found on mobile systems.

While the conditions of this experiment represent a more realistic environment than the previous experiments, the CPU may not have been fully utilized (due to I/O operations in the background load, and the fact that Latt contains sleep cycles). Therefore, the behavior of the two schedulers under full CPU load requires further investigation. Since interactive systems intermittently go through temporary periods of full utilization, making a conclusion at this point could be premature.

3.2.4 Experiment 4: Encode

The goal of this experiment is to compare the schedulers under full CPU load. To accomplish this, we modified the parameters of the last experiment by using the encode background load instead of the compile load. Preliminary experiments showed this multi-threaded background load to fully utilize the CPU on the Core 2 machine.

Figure 3.2b shows end-to-end execution times for all experiments. The BFS has a lower execution time than the CFS for this experiment, contradicting yet again the initial claim and the previous two experiments. Figure 3.2a shows that latency was also lower for the BFS than for the CFS.

With these additional results, there is still insufficient evidence to choose a scheduler based on either metric. To attempt to explain the reversal in end-to-end time performance, we decided to collect idleness data; if the BFS is utilizing the CPU at the same rate as the CFS, then overhead associated with scheduling may be causing the observed discrepancy in end-to-end time. Since the encode load requires fewer scheduling decisions, this overhead would not be as pronounced.

3.2.5 Experiment 5: Compile with Idleness Data

The goal of this experiment is to revisit Experiment 3, with the addition of the idleness data collection. We added calls to the *VMStat* Linux utility to Latt, so that every time Latt collected a latency sample, it used VMStat to collect an idleness value. We instrumented Latt to buffer idleness data in memory, along with latency and client throughput data.

The last plot on Figure 3.2a shows that, while the overall conclusion is the same from Experiment 3 — the BFS has better average and worst-case latency — the worst-case latency for the CFS decreased by nearly 50%, from 53,370μs to 28,530μs. The latency behavior of the BFS, on the other hand, remained the same. This change in behavior was later revealed to be caused by a probe effect specific to the CFS, caused by a scheduling policy introduced in late 2010 (the `CONFIG_SCHED_AUTOGROUP` kernel option). Section 3.5 includes details on this effect.

3.2.6 Discussion

The unstructured experiments demonstrated that several factors affect both the latency and the end-to-end time behavior of both schedulers: the background load, the Latt client load, and the presence of VMStat calls in Latt. Unfortunately, they also led to a series of contradictions, and no clear conclusion as to which scheduler has better latency or end-to-end performance. Worst of all, there is no indication of when these sequential experiments would converge to a conclusion.

There was, however, a significant difference between schedulers on all experiments, and the changes in experimental conditions affected both metrics on both schedulers. We used Kruskal-Wallis tests [67][2] to analyze the data in two ways: first, we verified that the latency and end-to-end time distributions differed between schedulers for all experiments ($P < 0.01$), second, we verified that latency and end-to-end time distributions for each scheduler differed between experiments ($P < 0.01$). Since the schedulers differ, there must be a correct scheduler choice under each operating condition.

An important question to answer is "when have you experimented enough to draw conclusions?" To answer this question, we realized we needed to more systematically explore the factors affecting our metrics. Using traditional design and analysis of experiments, we

[2]The Kruskal-Wallis test is a non-parametric test used to verify if two or more samples belong to the same distribution. Here, it serves approximately the same function as a t-test, without making assumptions about the distribution of the data.

can delimit our experimental space and, if analysis is successful, draw conclusions about the performance of each scheduler.

3.3 Factorial Design

We now employ factorial experiment design and linear regression, the statistical methods introduced in Chapter 1 to try to successfully compare the performance of the two schedulers. Our goal is to isolate the effect of our factors on our response variables: latency, client throughput, and end-to-end times. In this section we will demonstrate how linear regression can be insightful when used correctly, but can also produce misleading results when used incorrectly.

Factor	Low Lvl.	High Lvl.
Scheduler	BFS	CFS
Machine	Core 2	P4
VMStat	off	on
Load	compile	encode

Table 3.1: Factors and their levels.

We used a two-level full factorial design for our experiments. Table 3.1 shows the factors and levels explored in this design. All of the factors listed here measurably affected the worst-case latency in the experiments described in Section 3.2, and the machine factor was added to ensure results were not exclusive to the Core 2 machine used in the previous experiments. We fixed the Latt client number at eight and the compression size at 64kb, since those were the only conditions under which realistic latency measurements were collected. We omitted the sleep load for the same reason. Measuring latency in an idle system would lead to unrealistic results as demonstrated in Section 3.2.1.

For this full factorial design, experiments will consist of executing every combination of factor levels (a total of 16 combinations) five separate, non-consecutive times. This number of runs provided us with enough data for a robust analysis, since all effects were measured with a high significance level; if this were not the case, we would collect more data to improve the statistical power of the test (i.e., reduce the odds of a false negative). To ensure that the measurement error is distributed uniformly across all factor levels, we randomized the order of the execution of the trials.

Table 3.2 shows how the last three experiments from the previous section consist of a six trial subset of the full-factorial design. By simultaneously varying all factors in a series of

Exp.	Scheduler	Machine	VMStat	Load
3	CFS	Core 2	off	compile
	BFS	Core 2	off	compile
4	CFS	Core 2	off	encode
	BFS	Core 2	off	encode
5	CFS	Core 2	on	compile
	BFS	Core 2	on	compile

Table 3.2: Factors and their levels for the sequential experiments.

automated experiments, the full-factorial design will allow us to reach general conclusions without having to sequentially explore the experimental space one factor at a time.

The analysis of the results will produce a model of the form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \quad (3.1)$$

$$\beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{14} x_1 x_4 + \beta_{23} x_2 x_3 + \ldots + \epsilon \quad (3.2)$$

Where x_1 is the scheduler factor, x_2 is the machine factor, x_3 is the VMStat factor, and x_4 is the Load factor. Since these are all discrete factors, we attribute indicator values to each factor's levels: -1 for low and +1 for high. Interaction effect terms of the form β_{ij} are used for all permutations of two factors.

As briefly alluded to in Chapter 1, there are a number of pre-requisites that the underlying distribution of the data must meet for linear regression modeling to be successful, namely that (1) the error is normally distributed, (2) the error is independently distributed, i.e., one observation does not affect the probability of any other, and (3) the error has mean zero and constant variance. If a linear model adequately approximates the data set, it will allow us not only to quantify the impact of the factor on the response variable, but also to quantify unexplained variation. This is particularly useful in the presence of the hidden factors discussed in Chapter 2, since a large aggregated effect by those hidden factors may indicate the need for further experimentation. If these assumptions are broken, the true significance level (i.e., the probability of a false positive) of any statistical tests performed with the model (e.g., testing the statistical significance of a factor) will differ from what the calculations report, generally with a loss of statistical power (i.e., the chance of a false negative increases).

Any factors that affect the response variables but were *not* controlled (i.e., do not appear in Table 3.1) will cause variability that is not attributable to any factor or factor interaction.

In linear regression, this variability will be indistinguishable from any other source of error (such as random measurement error). Experimenters should strive to minimize the amount of error, so that as much variability as possible is adequately explained. The smaller the error, the smaller the effects that will be detectable.

3.3.1 End-to-End Times

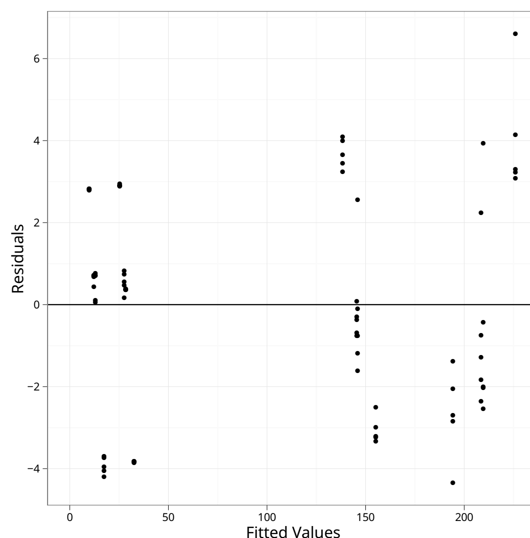


Figure 3.3: Residual plot for the linear model of end-to-end time response variable, showing no error patterns.

We first apply linear regression to the end-to-end time data from our experiment. We start by verifying that the requisite assumptions of linear regression are not violated, by checking whether the individual residuals — the distance, or error, by which each individual observation differs from what the model estimates it should be — are normal, and do not present a pattern or heterogeneous variance. Residual plots verify these properties, by relating the response variable values predicted by the model (plotted on the x-axis) with their distance from the actual, measured values that compose our data set (y-axis). Residual plots should show no discernible pattern when there is no systematic error (e.g., from faulty experimentation or analysis). Figure 3.3 shows that the residuals for this data have no discernible pattern or widely heterogeneous variances. A normal probability plot of the residuals showed that the normality assumption held. Therefore, linear regression produced a model with a good fit, and we can be confident in the conclusions drawn from

it. The more pronounced residual (with a value higher than 6) did not adversely affect the model.[3]

Factor	Effect (s) Low 95%	Effect (s) Estimate	Effect (s) High 95%	Sum Sq.	p-value
machine	77.9	78.6	79.2	493,851	< 0.01
load	11.4	12.0	12.6	11,497	< 0.01
sched	2.5	3.1	3.7	764	< 0.01
vmstat	2.4	3.0	3.6	725	< 0.01
machine:load	19.1	19.7	20.3	31,004	< 0.01
machine:sched	2.4	3.1	3.7	750	< 0.01
machine:vmstat	2.3	3.0	3.6	708	< 0.01
load:sched	1.4	2.0	2.7	333	< 0.01
load:vmstat	1.0	1.7	2.3	219	< 0.01
sched:vmstat	-0.1	0.5	1.1	19	> 0.05
Error				545	

Table 3.3: ANOVA results for end-to-end times.

Table 3.3 shows the ANOVA results for end-to-end times. The error sum of squares of 545 is negligible in comparison to the machine factor (with a sum of squares of 493,851, three orders of magnitude greater). Therefore, for this experiment, in this design space, the effect of controlling for any further factors will be negligible and we can conclude that enough experimentation has been conducted. The R^2 for this experiment is of 99.8%, confirming a good correlation between predictors and response variable.

The effect estimate for the machine factor (78.6s) clearly shows that it is the defining factor for end-to-end times, with a main effect estimate far larger than that of any other main factor. The estimate of the interaction between the machine and load (19.7s) can be interpreted as follows: depending on the machine, the load factor affects end-to-end times at a different rate. In other words, changing the load type incurs a larger increase in end-to-end time on the slower machine than on the faster one. Several other factors and interactions are statistically significant at the 99% confidence level (i.e., have p-values lower than 0.01) but have comparatively negligible effects.

Most notably, these results show that the scheduler factor, despite being statistically significant, had a negligible impact on the execution time of the benchmark. This means

[3] We used Cook's distance, a metric designed to quantify the influence of outliers on a model, to verify this [67].

that (1) if end-to-end time is the user's only concern, then the scheduler choice will be inconsequential and (2) end-to-end time does not differentiate these two schedulers. Thus, end-to-end time is a poor metric to benchmark these two schedulers' performance, unlike it has been discussed in the Linux community.

3.3.2 Latency

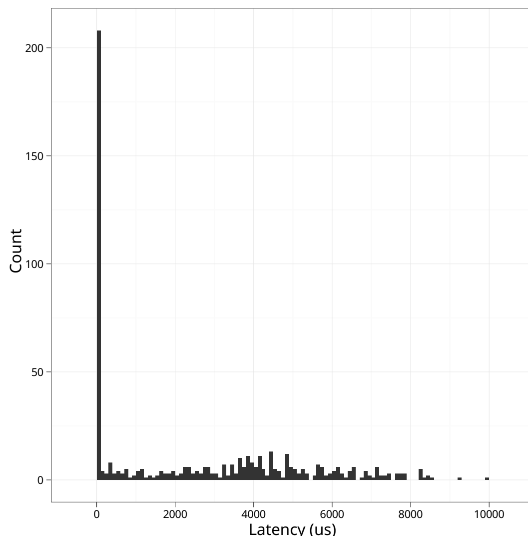
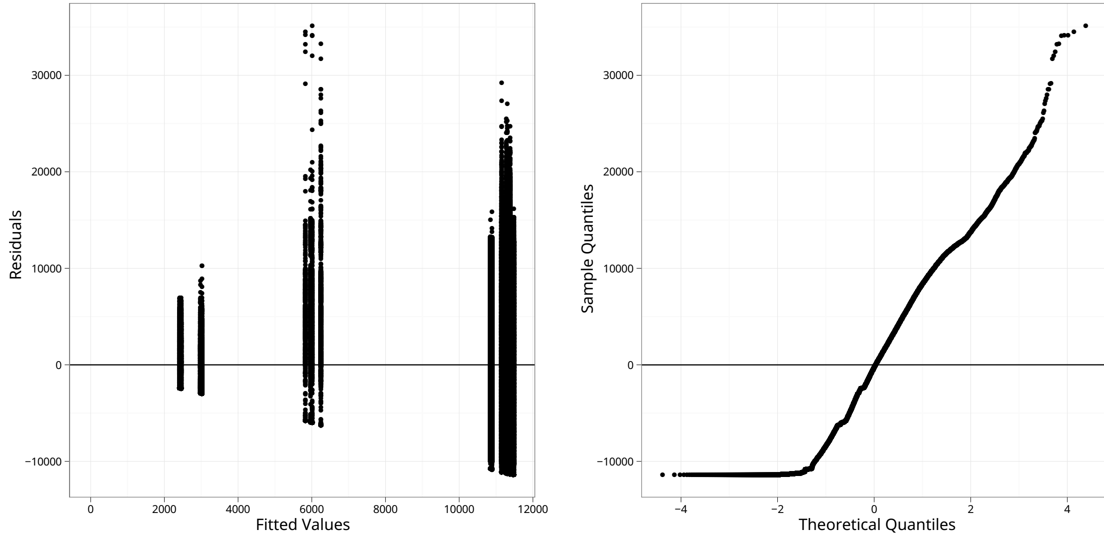


Figure 3.4: Example of non-normal latency behavior on the Core 2 machine, under the CFS, with no VMStat and the compile load.

Although linear regression works well for end-to-end time measurements, our main interest is the worst-case of the latency response variable. We now apply linear regression to latency, and again, must verify if the method applies. Figure 3.4 shows a latency histogram for the following combination of factor levels: CFS scheduler, the Core 2 machine, no VMStat, and the compile load. This distribution is highly non-normal, which, in turn, can cause residuals to also be non-normal; this is the first indication that linear regression is not applicable.

Figure 3.5a shows the residuals plotted against the fitted values. In contrast to Figure 3.3, we see a clear pattern of increasing variability as the latency values themselves increase. While this problem can be mitigated by transforming the data into a form with constant variance and then applying linear regression to the transformed data, these



(a) Residual plot, showing heterogeneous error variance.

(b) Normal residual plot, showing non-normal errors.

Figure 3.5: Residual plots for the linear model of the latency response variable.

residuals are also not normally distributed. Figure 3.5b, a normal probability plot of the residuals, demonstrates this; if the residuals were normal, they would lie alongside the solid, horizontal line — which indicates a perfectly normal distribution — following no discernible pattern. Non-normality of residuals causes errors in the significance of the estimated coefficients, and is difficult to resolve through data manipulation. The application of linear regression on the latency data resulted in a poorly fitting model, with an R^2 of 12.19%.

If we were to ignore these warning signs and attempt to choose a scheduler using this analysis, we would be basing our decision on a problematic model. The effect for the scheduler factor (i.e., the β_i value corresponding to the scheduler choice) given by this poorly fitting model is of $796\mu s$ (± 69 at a 95% confidence level). Consider, then, the plot for Experiments 4 in Figure 3.2a, where the difference in mean latency between the CFS and the BFS is of $5453\mu s$; Experiments 3 and 5 show similar differences. The conditions of Experiments 3 through 5 are in this experimental space, so the results of those experiments should be approximated by this model. However, the model suggests that, on average, a change in mean latency of approximately $1600\mu s$ would be observed by a change of scheduler. That is a severe underestimation, and might lead to incorrect

conclusions.

Most important of all, however, is the fact that the worst-case latency is of special importance for latency-critical applications. This means that using linear regression, a method to model means, would not allow the analysis of the effects on the higher percentiles of the response variable distribution, providing instead an incomplete picture containing only the mean latency value. As we will show in Section 3.4, the scheduler effect discussed before will be even more conservative, if the worst-case is of any interest. Since the non-normal data seen here is common in computer experiments [37, 74, 15], a different analysis method is needed. We now apply quantile regression to analyze this same data set, determining each factors' effects on worst-case behavior, with the added bonus of not requiring normally distributed data.

3.4 Quantile Regression

Quantile regression [53], like linear regression, is a method to model the effect of factors on a response variable. The main difference between the two methods is that, while linear regression models effects on the mean of the response variable, quantile regression can model the effect of factors on *any given quantile*, such as the median or the 99th percentile. By performing multiple quantile regressions on different quantiles of a data set, a researcher can quantify the effect of a factor all along the response variable's probability distribution, and get a more detailed idea of how it is affected by the factors of interest.

Although the probability distribution of a performance metric will depend on many factors (such as processor speed), the effect of factors on this distribution can be more complex than a simple change in its mean. Indeed, factors may affect only *part* of the overall distribution. Consider a hypothetical CPU-bound benchmark. The best case performance of this benchmark occurs when it is entirely in cache at the beginning of measurements; in this case, memory latency is irrelevant to performance. In other executions, it may be partially cached or entirely out of the cache; in these scenarios, memory latency will affect performance to different extents. Applying linear regression in this scenario (investigating memory latency as a factor) would lead to a model of the mean behavior, overlooking the fact that memory latency affects *different quantiles of the response variable at different rates*. Quantile regression, on the other hand, is able to capture these details.

Linear quantile regression models are very similar to the linear models described in Chapter 1. They take the form:

$$Q_y(\tau|X) = \beta_0(\tau) + \beta_1(\tau)x_1 + \beta_2(\tau)x_2 + \ldots + \beta_k(\tau)x_k + \ldots + \beta_{12}(\tau)x_1x_2 + \ldots + \beta_{ik}(\tau)x_ix_k \tag{3.3}$$

where $\tau \in [0, 1]$ is the quantile of interest[4] of the response variable y, X is the combination of factor levels, and $Q_y(\tau|X)$ is the τth conditional quantile of the response variable y (i.e., the τth quantile of y given factor levels X). As with the linear model, the β_i coefficients represent the expected change in the conditional quantile as the corresponding factor levels change.

Quantile regression also differs from linear regression in that it is non-parametric, that is, it does not assume anything about the distribution of the error component. This is because parameter estimates are only affected by the local behavior of the response variable near each quantile of interest. While the application of linear regression on our non-normal latency data yielded an incorrect model because the data failed to meet requirements, we now demonstrate that quantile regression can successfully analyse the same data set.

To quantify the effect of each factor at different points on the latency probability distribution, we apply ten quantile regressions to the dataset: nine from the 10th percentile through the 90th percentile at equal intervals, then one at the 99th percentile. Coefficients were estimated for all factors and interactions up to the four-factor interaction. Figure 3.6 shows all main factors, the scheduler by machine interaction, and the intercept. The intercept, or the $\beta_0(\tau)$ coefficient, does not depend on factors that affect the response variable; it can be thought of as the baseline latency at each of those quantiles before factor effects are applied. The x-axis shows the quantiles of the latency probability distribution, and the y-axis shows the magnitude of effects at each of those quantiles. A black line is plotted through the estimated effects, and the gray band denotes the 95% confidence interval for the effects. For example, consider the machine factor subplot in the figure. This factor has no effect at the 10th percentile (0.1 on the x-axis). Towards the right of the plot, the effect of this factor starts to grow, ultimately reaching 5,594.5μs at the 99th percentile. This means that, all else being equal, a change from the Core 2 machine to the P4 will increase the value of the 99th percentile of the response variable by approximately 11,189μs (as with linear regression in Section 3.3, the net increase is doubled due to the coding used for factor levels). Similarly, a negative effect like the scheduler by machine interaction will cause a quantile to decrease in value.

[4]The τth quantile of a distribution y is the smallest observed value in y such that the probability of obtaining smaller values of y is τ.

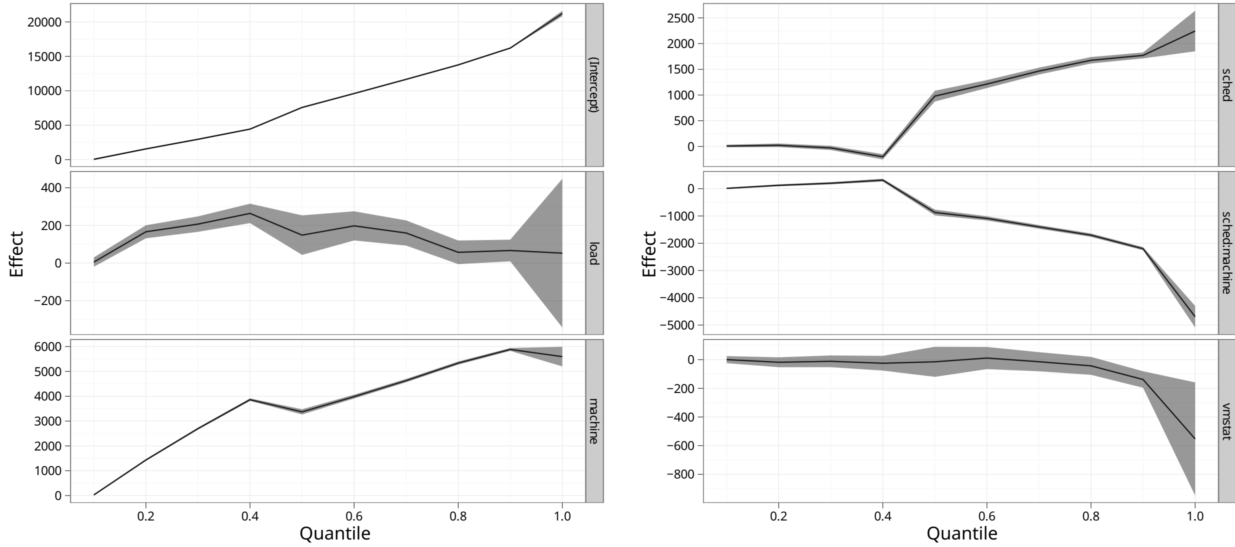


Figure 3.6: Quantile regression effects for the most relevant factors for the latency response variable.

The first observation of note is the significant effect of the scheduler on latency. The scheduler subplot in Figure 3.6 shows that this factor has no effect below the 40th percentile, but then its effect gradually increases with each percentile, eventually reaching 2,244.1μs at the 99th percentile. The difference between the BFS and the CFS near the worst-case latency will therefore be of approximately 4,488.2μs. While this latency may seem to be low, it accounts for *more than 10% of the single worst-case latency observed in all our factorial design trials*, which was of 41,164μs. A 10% improvement in worst-case latency would make a significant interactivity impact in a mobile system that operates near the 100ms threshold.

Table 3.4 provides a list of all effects on the 99th percentile, the highest percentile shown in Figure 3.6. Similarly to Table 3.3, which presented the ANOVA results for the end-to-end time, the effects column lists the expected change in the 99th conditional percentile when each factor level changes and all else remains equal, accompanied by the high and low boundaries of their 95% confidence interval. The statistical significance of each effect is shown in the p-value column. At this extreme percentile, the machine has the largest effect, closely followed by the scheduler by machine interaction, and then the scheduler main effect. We can conclude that, if the user is concerned about the worst-case latency,

Factor	Effect (μs) Low 95%	Estimate	High 95%	p-value
machine	5,199.7	5,594.5	5,989.3	0.000
sched	1,849.3	2,244.1	2,639.0	0.000
vmstat	-947.9	-553.0	-158.2	0.021
load	-342.0	52.9	447.7	0.825
sched:machine	-5,087.5	-4,692.6	-4,297.8	0.000
sched:vmstat	-940.7	-545.9	-151.0	0.022
machine:vmstat	68.7	463.5	858.3	0.053
load:vmstat	-101.0	293.9	688.7	0.220
machine:load	-114.0	280.9	675.7	0.241
sched:load	-668.6	-273.8	121.1	0.254
sched:machine:vmstat	215.5	610.4	1,005.2	0.011
sched:machine:load	82.7	477.5	872.3	0.046
machine:load:vmstat	-636.5	-241.6	153.2	0.314
sched:load:vmstat	-232.6	162.3	557.1	0.499
sched:machine:load:vmstat	-602.827	-208.000	186.828	0.386

Table 3.4: Quantile regression coefficients for the 99th percentile.

then the BFS will provide better performance.

Figure 3.6 and Table 3.4 permit further conclusions. Latency, for most quantiles, is independent of the load factor; Figure 3.2a also shows this, where the difference between the latency measured in Experiment 3 (compile load) and 4 (encode load) was minimal. The difference between the two experiments is concentrated in the lower quantiles, where most of the load factor's effect is located. We can also observe that the VMStat factor, while largely indistinguishable from zero, becomes more pronounced after the 90th percentile. Table 3.4 shows that interactions with the VMStat factor are also significant at that quantile. Figure 3.2a for Experiment 5 already indicated this, where adding VMStat significantly decreased the worst-case latency of the CFS.

To summarize, quantile regression (1) allowed us to analyze a data set that linear regression failed to model correctly, and (2) provides a higher level of detail than linear regression. The first point is important because non-normality is common in computer science experimental data; linear regression yielded a poor model because of the non-normality and different variances in the latency data, while quantile regression provided insight into the data despite these properties. The second point is important because when researchers have more than a model of mean response (or indeed, simply a test of

whether samples originate from the same distribution), they can gain more insight from their experiments. For example, the fact that the scheduler has an increasingly larger effect as latency values increase — making it very important for worst-case latency analysis — would be impossible to detect through linear regression or any form of analysis that focuses on the response mean.

3.5 Discussion

While we conducted the experiments for this chapter, several experimentation pitfalls became obvious. This section discusses those pitfalls and their proposed solutions.

Autogroup as a Hidden Factor The experiment described in Section 3.2.5 demonstrated how a small, seemingly innocuous change of experimental conditions can significantly affect results. In that case, the addition of VMStat calls to Latt caused a significant change in latency behavior. This was caused by the autogroup option in the Linux kernel (named `CONFIG_SCHED_AUTOGROUP` in the configuration options), which causes the CFS to schedule processes in the same virtual terminal (or TTY) as a group.

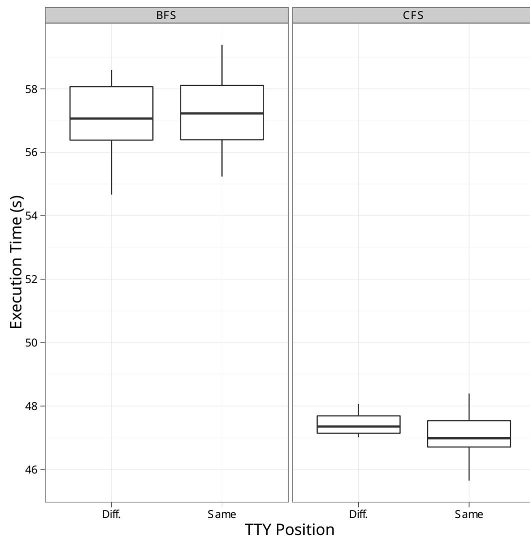


Figure 3.7: Effect of autogroup on the *ab* benchmark.

We confirmed this hidden factor with Apache's *ab* benchmark. The *ab* benchmark is an automated webserver client that measures how many requests per second Apache can

serve. To demonstrate this hidden effect, we executed Apache and *ab* in two configurations: (1) in the same TTY, and (2) in different TTYs. Figure 3.7 shows that, while there was no evidence of a change in the mean execution time (verified with a t-test, $P > 0.05$), executing *ab* in the same TTY as Apache caused the best observed execution time under the CFS to decrease by 1.37s, and the worse case to increase by 0.33s, that is, the spread of measured values seemed to increase. This was verified through an F-test, which showed that the ratio between the *variances* of the two CFS distributions was approximately 0.2. In other words, autogroup made the execution time variance measurably larger for the CFS. On the other hand, this does not hold for the BFS ($P > 0.05$ for both t and F-tests), as it ignores the autogroup kernel option. It is reasonable to assume that some server administrators are unknowingly measuring unrealistic performance because of this hidden scheduling artifact.

Incorrect Surrogate Metrics During our experiments, Latt's client throughput metric proved to be a poor surrogate for application throughput. We found that an execution of Latt could have a lower mean client throughput, but a higher end-to-end time than another execution, which can be easily interpreted as a fault in Latt. The surprising result is due to the fact that Latt does not tally time spent in the server thread, and therefore, the client throughput metric only captured a part of the total execution time of the benchmark. After discovering this, we started using the end-to-end time of a useful computation (compilation or video encoding) to provide a reliable metric of throughput.

Choice of Background Load "Clean room" experimentation may lead to incorrect or misleading results, as shown by the latency measurements for Experiments 1 and 2 in Section 3.2. By keeping the processor largely idle, the idle load yielded latency data that did not distinguish the two schedulers. Similarly, having a single Latt client perform no work failed to yield realistic latency data. Because of this, we encourage the exploration of the experimental space with a focus on determining not only what factors affect the response variable, but also how the response variable behaves at the different levels of those factors.

Chapter 4

Lowering the Cost of Reproducible Experiments

In Chapter 3, we demonstrated how to design and analyze a multi-dimensional experiment using rigorous statistics, even when facing highly non-normal computer performance data. Given the effort, expertise, and documentation that was required, it is unsurprising that this level of rigor is rare in computer science publications.

To lower the cost of statistically rigorous, reproducible computer performance experiments to a level acceptable by the community, this chapter presents DataMill, a distributed performance evaluation infrastructure. DataMill automates the factorial experiment design process demonstrated in Chapter 3, trivializes the exploration of hidden factors such as those discussed in Chapter 2, and provides hardware and software heterogeneity far beyond what is normally available to developers and researchers, leading to more reproducible and general results.

DataMill is a public infrastructure to which researchers gain access by contributing their own machines. This fosters the community-driven aspect of the infrastructure while improving the heterogeneity of the platforms available. Finally, DataMill also has built-in functionality for sharing experiment code and results data, allowing the original experiment submitter to re-run their own experiments on new platforms, and other researchers to "fork" experiments for result verification or comparison purposes.

4.1 DataMill: The User Experience

We first describe DataMill as seen by a user, and later describe the inner workings of DataMill in Section 4.2. The user experience starts with packaging the experiment for DataMill to execute, then defining a factorial experiment design from an array of hardware and software factors, and, finally, collecting and analyzing the resulting data.

4.1.1 Packages

Each experiment contains one or more packages. A single package experiment can quantify performance over a wide range of setups, while experiments with more packages allow performance *comparisons* over those setups.

Each package contains: (1) the source code and any input data for the experiment, and (2) auxiliary DataMill-specific scripts to set up, execute, and collect the results from the experiment. All package components are encapsulated in a compressed TAR file.

There is no restriction on experiment code, and users have administrator-level access to the machines that will execute the experiment – the *workers* – during experimentation. Users can generate free-form results data using arbitrary metrics, and then collect them via compressed packages. These features allow evaluating the performance of a wide range of software, ranging from user-space applications to kernel modules. Security concerns are minimal at this point, because participating users must contribute to the infrastructure and, therefore, are well known and trusted.

There are only two scripts that every DataMill package must contain: `run.sh` and `collect.sh`. These are the scripts that execute and collect data from the experiment, respectively. If the package requires a setup procedure (such as decompression, compilation, dependency installation, etc.), it may also contain a `setup.sh` script, which will be executed before `run.sh`. Finally, if there is the need for environment variables during execution, an `env` file may be included in the package. The contents of this environment file will be added to the experiment's environment before each execution.

To exemplify the construction of a DataMill package, consider the Dhrystone [101] benchmark. It consists of a single source file, `dry.c`, which compiles itself, and runs the benchmark. To execute it, one must first set its executable flag, which is done in the setup script shown in Listing 4.1. The script assumes the source file is located in the `/dry/` directory.

```
1 #!/bin/sh
2
3 cd /dry/
4 chmod +x dry.c
```

Listing 4.1: Setup Script

Listing 4.2 shows the `run.sh` script that executes Dhrystone, capturing its output (standard out and error) to the `/dry/results` file. The output is appended to the end of the file, allowing multiple executions before collection.

```
1 #!/bin/sh
2
3 cd /dry/
4 ./dry.c >> results 2>&1
```

Listing 4.2: Run Script

Finally, Listing 4.3 shows the script that packages the results file for collection. It simply compresses the result file using a unique name, and echoes the final archive's file name. This file will then be downloadable by the user once the experiment finishes executing.

```
1 #!/bin/sh
2
3 results=/dhrystone-`date +%Y%m%d_%H%M`.tar.gz
4 tar czf $results /dry/results > /dev/null 2>&1
5 echo $results
```

Listing 4.3: Collect Script

The user must package the experiment source code and DataMill scripts in a GZipped TAR file. All DataMill scripts should be in the root directory of this archive. For the Dhrystone package, the file structure of the package would be as shown in Figure 4.1.

To facilitate package creation, we provide users with a virtual machine image that mimics a DataMill worker. Users can develop, test, and debug their packages in a local environment until they are sure their packages are ready for production, at which point they can submit it for execution via our experiment creation interface.

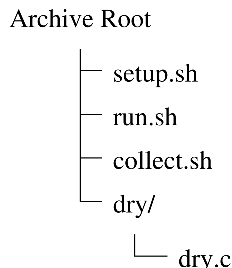


Figure 4.1: Dhrystone package directory structure.

4.1.2 Experiment Creation

Once the packages are ready, the user submits them for execution through a file upload interface. Users must then define an experiment for DataMill to execute. Experiment definition has three steps:

1. Package selection
2. Constraint definition
3. Experiment design definition

Package selection consists of choosing which packages will be executed; one or more packages may be selected. In the constraint definition step, users inform DataMill of any limitations to their experiment code. For example, if the experiment can only execute on the ARM9 architecture and requires at least 2GB of RAM, the user expresses these constraints through an intuitive web interface.

Finally, in the experiment design definition step, users select how many factors they wish to explore, and how many replicates to collect. The interface presents each dimension of a factorial experiment design, divided into hardware and software categories, using a unified percentage-scale. Non-zero percentages cause the addition of that dimension to the experiment space, and the percentage value represents what fraction of all available options should be *covered*, determining indirectly how many levels for that dimension the experiment will have. For example, users can define that their experiment must contain 100% of CPU available architectures and that 75% of GCC optimization flags be tested. This will cause DataMill to select machines such that every architecture in the infrastructure is represented (i.e., creating five levels for the "architecture" factor) and to compile the experiment with a randomly selected 75% of optimization flags (i.e., three levels to the

"optimization flag" factor). Setting any dimension to 0% results in the random selection of a single level for that dimension, effectively removing it from the factorial design. Finally, the number of replicates defines how many repeated measurements should be taken for each design point, in order to calculate variance, enabling linear or quantile regression.

After machines are selected according to the hardware dimension settings, all software dimensions selected are combined to generate each individual *job*. For example, if the user chose to explore three GCC optimization flags, on five machines, with five replicates, there would be five jobs generated per optimization-flag and machine pair, totaling 75. If, in conjunction with the optimization flags, both settings of the address randomization feature of Linux (on or off) should be tested, then the number of jobs grows to 150 (three flags times two address randomization settings times five machines times five replicates). Therefore, the number of jobs (displayed before experiment submission) for each experiment scales with the experiment design defined by the user, and care must be taken to avoid combinatorial explosion and an experiment with an excessive number of jobs.

4.1.3 Experiment Results

After the user designs the experiment and DataMill creates jobs, it will distribute the experiment's packages for execution, then collect the individual result files. The web interface dynamically updates experiment information as data arrives, allowing users to monitor their experiment's progress. In addition to the data collected by the collect script, DataMill collects additional metrics with minimal overhead [97], such as total execution time, and the number of page faults and cache misses. This data is also made available to the user.

Finally, once all jobs associated with an experiment are finished, users can download the full experiment results file, which contains the result file from every job. Examples of how to analyse large datasets generated by DataMill are provided in Section 4.3.

4.2 DataMill: The Infrastructure

Making the user experience described in Section 4.1 a reality requires considerable engineering effort. The DataMill infrastructure is composed of a *master node*, responsible for the distribution of experiment trials and the collection of results, and several *worker nodes*, which execute the experiment packages provided by the users. This section describes how DataMill was implemented and how its different parts interact.

4.2.1 Master Node

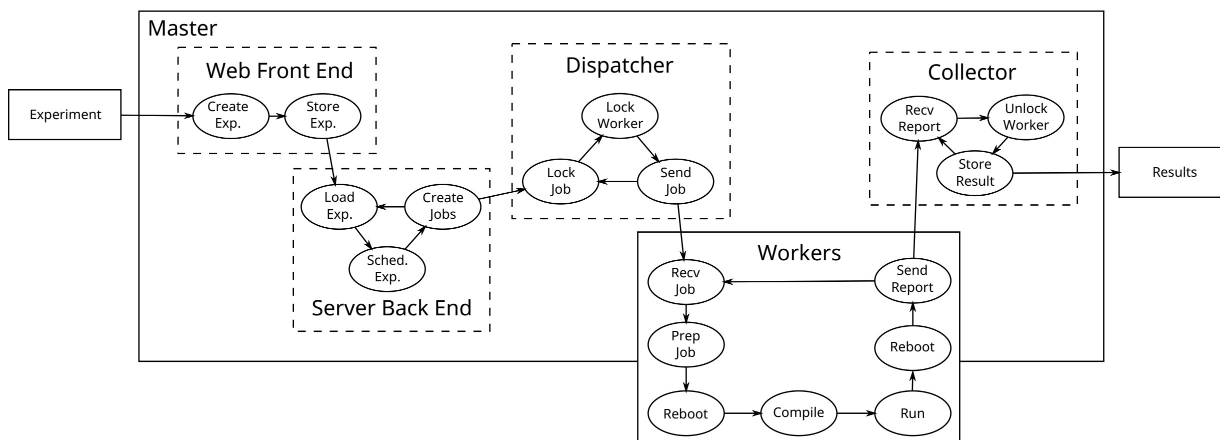


Figure 4.2: The DataMill infrastructure.

The responsibilities of the master node are split between four different daemons: the web front-end, the backend, the dispatcher, and the collector. The user interacts with DataMill through a web front-end. Creation of experiment configurations is based on the users' choices (e.g., experiment factors, experiment package, and desired platforms). The backend, which can consist of one or more instances, processes these experiment configurations and schedules individual instances of an experiment, which we refer to as *jobs,* on the workers. The dispatcher submits scheduled jobs to the workers. The collector accumulates results from individual jobs once they are completed on the associated worker. The web-interface provides access to these results as they arrive. In the following sections, we describe the individual components of the management layer in detail. Each of these daemons acts as a simple state-machine as shown in Figure 4.2. To ensure consistency of the state of the infrastructure, we use a database for central configuration management tasks.

The Web Front-End The *web front-end* is the interface through which the user interacts with DataMill. User submission of experiments triggers the backend daemon process, which processes package information and configurations and subsequently schedules jobs for worker nodes.

The web front-end also houses an XML-RPC [17] service. The service provides an API that workers call to inform the master node about state changes of the individual jobs.

Workers use the same API to register with the infrastructure. To separate control and data flows, this XML-RPC interface is only used to notify the master node of state changes. The dispatcher and collector handle the transfer of experiment data such as packages and results.

Backend The *backend* daemon, shown in Figure 4.2, is responsible for transforming the user-supplied specifications for a particular experiment into a set of specific trial configurations to run on the worker nodes. The creation of these configurations and the selection of the appropriate worker node to execute them is handled by an optimization solver.

The problem is then to select the minimum number of workers necessary to provide the desired factor coverage. For this, we specified an optimization problem, which the backend solves with the GNU Linear Programming Kit [27] (GLPK) for each submitted experiment. The optimization problem is of the form:

$$w_c = \min \sum_{x_i \in \mathcal{W}} x_i \tag{4.1a}$$

$$\text{s. to: } \sum_{x_i \in \mathcal{P}_j} x_i \geq l_j, \qquad l_j \in \mathcal{L} \tag{4.1b}$$

$$l_j \geq x_i, \qquad l_j \in \mathcal{L}, x_i \in \mathcal{P}_j \tag{4.1c}$$

$$\sum_{l_j \in \mathcal{F}_k} l_j \geq B_k, \qquad k = 1..f \tag{4.1d}$$

$$x_i \in \{0, 1\}, \qquad x_i \in \mathcal{W} \tag{4.1e}$$

$$l_j \in \{0, 1\}, \qquad l_j \in \mathcal{L} \tag{4.1f}$$

Where $\mathcal{W}$ is the set of workers, $\mathcal{L}$ is the set of factor levels, and $\mathcal{P}_m$ is the set of machines that *provide* level m. $\mathcal{F}_k$ is the set of factor levels for factor k (a subset of $\mathcal{L}$), B_k is the user-requested minimum coverage for factor k. Equation (4.1b) ensures picking a factor level means at least one machine with that factor level is added to the solution, while, conversely, Equation (4.1c) ensures that selecting a machine adds all factor levels it provides to the tally, which is checked in Equation (4.1d) for every factor. This last equation guarantees that all user-provided constraints are met. After a solution is found, each worker is individually selected by their respective x_i variable.

Dispatcher The dispatcher daemon in Figure 4.2 is responsible for transmission and triggering of the execution of the packages and configurations on the appropriate worker

as defined by the solution generated by the backend daemon. The dispatcher finds jobs whose associated worker is idle and transmits and calls for the execution of the associated package via secure shell.

Collector The collector daemon in Figure 4.2 is responsible for the collecting of the experiment results from the worker nodes which have completed their current job. Once the results have been collected from any particular worker the worker is marked as *idle* so that it may receive further jobs.

4.2.2 Worker Node

Each worker node is a separate machine running Gentoo Linux [91] with kernel version 3.3.8, and GCC 4.5.3. Gentoo Linux was chosen since it is a source-based distribution and thus supports a wide variety of architectures. Having the same distribution and toolchain on all the machines keeps their base software homogeneous, allowing for controlled variations when requested.

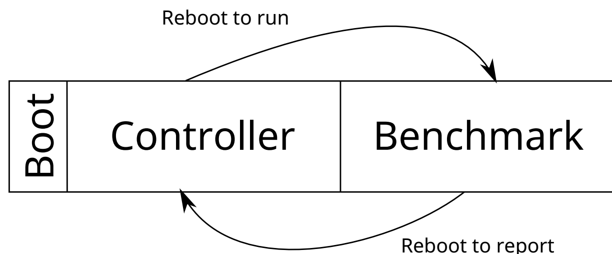


Figure 4.3: The worker node disk partitions.

The partition structure of a worker node is shown in Figure 4.3. The worker comprises two partitions, one with a minimal Gentoo installation, referred to as the *controller partition*, which is responsible for communication with the master, and one with a more complete software environment, referred to as the *benchmark partition* on which experiments will run. The state machines for each worker partition are shown in Figure 4.2, with the controller and benchmark portions of the state diagram separated by reboots.

The master node communicates with workers through secure shell. Communication is kept to a one-way push architecture, from master node to worker node, where possible. The workers do not have the ability to run code on the master barring the limited XML remote procedure calls that are provided to them. In this way, the communication between the master node and the worker nodes is strictly limited.

When a worker has been set up for the first time, it must notify the master of its existence. Upon its first boot, the worker collects information about itself and sends it, along with a registration request, to the master node. Once it is registered, it is marked as *idle* in the database and awaits further instruction.

When a worker receives a job from the dispatcher, the dispatcher executes a script on the worker via secure shell. The script reformats the benchmark partition, transfers the received package to it, and prepares the package for execution. The machine subsequently reboots into the benchmark partition. The benchmark partition's init process has three tasks to perform: changing the default boot partition, executing the received package, and finally rebooting to the controller partition.

Due to the size of DataMill, worker nodes are expected to fail. The master node detects such occurrences through communication timeouts, incorrect state machine progression, and faulty result files. The master node purposefully inserts redundancies in the job scheduling to minimize the impact of worker failures.

The DataMill infrastructure supports remote worker nodes to allow easy user contributions. Already included in the DataMill cluster are machines from Purdue University, the University of Pennsylvania, the University of Lugano, McMaster University, and the Federal University of Santa Catarina.

4.2.3 Factor Variation

DataMill facilitates execution of user supplied experiments using a wide variety of experimental setups. Knowledge of the factors affecting performance and the details of the mechanics required to vary them are not necessary to leverage the infrastructure. Nevertheless it is important to discuss some details of DataMill's factor manipulation.

To apply software-controlled factors — compilation flags, library versions, link orders, etc. — the worker node unpacks the configuration file sent by the dispatcher, applying each configuration individually before the setup or execution. Current supported factors include various GCC flags, C and C++ object link orders, the use of address space layout randomization (ASLR) in the Linux kernel, the addition of padding bytes to the POSIX environment, reboot behavior, dropping page, inode and dentry caches before execution, CPU frequency scaling, Linux's Autogroup function (as described in Chapter 3, the use of a swap partition, the root filesystem of the benchmarking partition, and the system clock.

While most factors can be easily applied (i.e., interacting with the `proc` filesystem to affect kernel options) some factors require more effort. For example, to support modifications to benchmark link order, we implemented a custom wrapper that intercepts calls to

GCC and the linker. This wrapper calculates the new order of objects according to the configuration received from the master (i.e., alphabetical, or reverse alphabetical according to object file names), and then forwards the call to GCC, using the new object file order.

This method of applying software factors enables making extensive modifications to experiment code without requiring users to implement these modifications themselves or to upload a series of different packages with pre-applied modifications. The list of supported factors is currently growing quickly, and we aim to support user-contributed factors in the future.

4.3 Case Studies

This section presents two case studies: we first perform a compression algorithm performance comparison to demonstrate how easy it is to conduct a performance experiment on DataMill, then we replicate the results from Mytkowicz et al. [70], the experiment that first revealed performance artifacts related to the link order of a binary (see Chapter 2 for details). This second experiment demonstrates the utility of DataMill for the scientific investigation of computer performance.

4.3.1 XZ vs. bzip2: Best Bang for Your Buck?

XZ [92] and bzip2 [47] are widely-used compression utilities for UNIX-like operating systems. While XZ uses the LZMA2 compression algorithm, bzip2 uses the Burrows-Wheeler algorithm. These two compressors will serve as stand-ins for a "baseline-vs.-proposed-approach" performance comparison, found in the majority of computer science papers that contain empirical performance evaluations. As bzip2 is the older of the two compressors, we will treat it as the baseline.

As described in Section 4.1, the only preparation step required for this experiment is the creation of two DataMill packages, one for XZ 5.0.4 and another for bzip2 1.0.6. The scripts themselves are omitted for brevity, but their contents are very similar to the ones for Dhrystone and comprise a total of 32 lines. We use the system-wide *emerge* command – Gentoo's package manager – to install both XZ and bzip2 in order to simplify installation, precluding the need to include their source code in each package.

In addition to the DataMill scripts, the packages contain the data to be compressed. For this experiment, we used the Maximum Compression [99] testset. This testset includes

Algo.	Size (kB)	Reduction (kB)	%
None	51,900	0	100.000
bzip2	13,232	38,668	25.496
XZ	12,120	39,780	23.353

Table 4.1: Compression rates for each algorithm.

various kinds of files (text, executable, graphics, etc.) and has been used to compare compression algorithms since 2003.

The metrics, which the `collect.sh` script collects, were execution time and compressed file size. Since each compression algorithm leads to a different archive size, the metric used for the comparison is the byte-per-second compression rate, calculated as bytes reduced/execution time. Table 4.1 shows the uncompressed data size, the resulting archive size under each compressor, the absolute reduction in size, and the resulting archive size as a percentage of the original file size. Note that both compression algorithms use deterministic algorithms, so the resulting compressed files are identical between runs and machines. Machine C, an ARMv7 which uses the ext3 filesystem, reports file sizes 20kB larger than the ones reported by all other machines, which use ext4. This small 0.1% discrepancy was ignored.

If absolute compression is the only metric of interest, then XZ is clearly the winner, due to its resulting archives' smaller size; however, if execution time or the rate of compression are of interest, then experimentation is necessary. By using DataMill, we can easily compare the two compressors, and measure their susceptibility to different factors. The DataMill experiment design was configured to include all machines, all link orders, all optimization flags, and address randomization on and off. The number of replications was set to 15 to allow the measurement of dispersion. This led to the generation of 6300 jobs, distributed between seven machines.[1] This experiment took approximately five days to complete on the slowest machine, a 600MHz ARMv7 Beagleboard xM. All other machines completed it in less time, and were free to continue with other experiments.

Figure 4.4 shows an overview of the data set resulting from this experiment. This facet plot is divided by optimization flag (top header) and machine (right-hand header). Machines are indexed with a capital letter, followed by their clock speed and CPU model. Each subplot contains boxplots for each of the compressors, bzip2 and XZ, with the same visual syntax as the ones from Chapter 3.

[1]Data for the "alphabetical" link order in the XZ was not generated, as that object order did not link successfully.

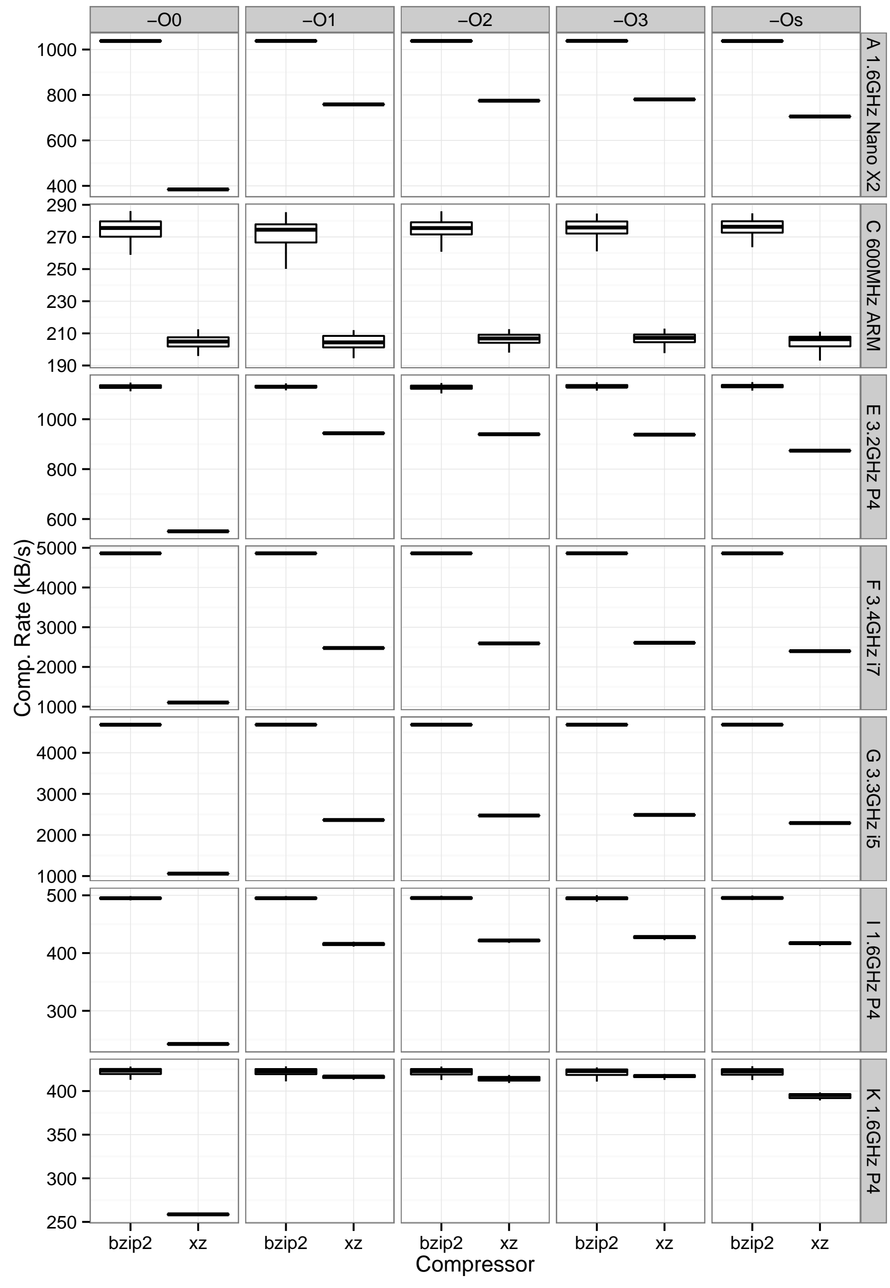


Figure 4.4: XZ vs. bzip2, compression rate.

Factor	Mean Sq.	p-value
Compressor	1.617e+12	<2e-16
Opt. Flag	1.258e+11	<2e-16
Link Order	7.397e+05	0.5807
Addr. Rand.	2.572e+04	0.8907
Compressor:Opt. Flag	1.883e+11	<2e-16
Compressor:Link Order	1.924e+06	0.2347
Compressor:Addr. Rand.	1.511e+06	0.2922
Opt. Flag:Link Order	1.312e+06	0.4624
Opt. Flag:Addr. Rand.	3.791e+06	0.0257
Link Order:Addr. Rand.	2.738e+05	0.8177
Residuals	1.360e+06	

Table 4.2: Reduced ANOVA table for XZ execution time on machine F.

The first conclusion is that bzip2 has a better compression rate for all machines under all link orders and all optimization flags tested. This would suggest that, for users interested in compression speed, bzip2 is the better alternative. Also of interest is the fact that bzip2 is unaffected by the optimization flag, which suggests it is entirely I/O-bound. XZ, on the other hand, has a marked performance increase from -O0 to -O1, but negligible differences in performance after that. The omitted experimental dimensions — link order and address randomization — did not significantly affect performance, which is demonstrated by the narrow grouping of all samples in the boxplots of Figure 4.4 (which contain data from multiple levels in the omitted dimensions).

We apply linear regression to analyze the data in more detail. Table 4.2 shows the ANOVA table for this experiment, fitting a model with up to two-factor interactions on the execution time data for machine F, a 3.4GHz Core i7. The Mean Sq. column shows the variability attributable to each factor, while the p-value column shows the statistical significance of each factor in respect to execution time. The model was a good fit, with $R^2 > 0.99$, meaning the model can be confidently used to analyze the data. The table shows that the compressor and optimization flags are significant in isolation ($P < 0.01$), but link order and address randomization are not ($P > 0.05$). Most interestingly, we can demonstrate the interaction between the compressor and optimization flag through their interaction term, which is also significant ($P < 0.01$), which confirms that only XZ, the more CPU-bound of the two algorithms, benefits from higher optimization levels.

Figure 4.5 shows the effect of GCC optimization flags on XZ's execution time, with the x-axis ordering the different optimization flags. While optimization flags are not a

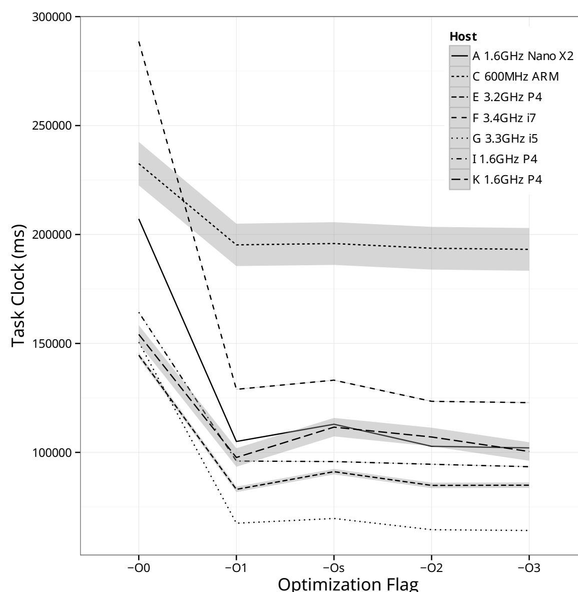


Figure 4.5: Effect of GCC optimization flags on XZ, by host.

continuous dimension, this ordering facilitates the visualization of the data and represents the activation of various individual GCC optimization options going from one optimization level to the next. According to GCC's manual, the -Os flag is located between -O1 and -O2 because it activates all options from -O2 that do not increase binary size, being, therefore, a middle point between the two. Data from each worker is plotted along an individual line, with 95% confidence intervals shown in light gray behind each curve. The plot shows a marked and general improvement in performance going from -O0 to -O1 (as shown in Figure 4.4).

This performance comparison demonstrates the utility of DataMill for users interested in evaluating performance: with just 32 lines of code, 6300 jobs were executed in under a week, exercising several dimensions that would normally be ignored, and leading to insight that would be unattainable through manual, one-factor-at-a-time experimentation.

4.3.2 Perlbench: Link Order Effect

We now demonstrate the use of DataMill for users interested in the science of computer performance evaluation. Mytkowitz et al. [71] report that the link order of a binary can be correlated with runtime performance, and that the optimal link order varies from host

to host. This is generally understood to be a consequence of different memory and cache layouts leading to different cache and page miss ratios. The authors showed that the performance of Perlbench — part of SPEC CPU 2006 [39] — can vary by more than 8% by simply modifying the link order.

Trying to reproduce their results, we created an experiment on DataMill to explore the effect of link order and address randomization on Perlbench performance. We encapsulated Perlbench and SPEC's "train" data set in a DataMill package, with scripts and environment files totaling 33 lines. Three link orders were explored (default, alphabetical and reverse alphabetical), with Linux address randomization on and off. If address randomization is on, one would expect that the affect of link order would be neutralized, since the memory layout will be randomized. In other words, the link order and the address randomization factors should be highly correlated. We chose a number of 15 replications of each configuration to calculate dispersion, generating a total of 630 jobs over 7 machines. DataMill took approximately 27 hours to finish the full experiment.

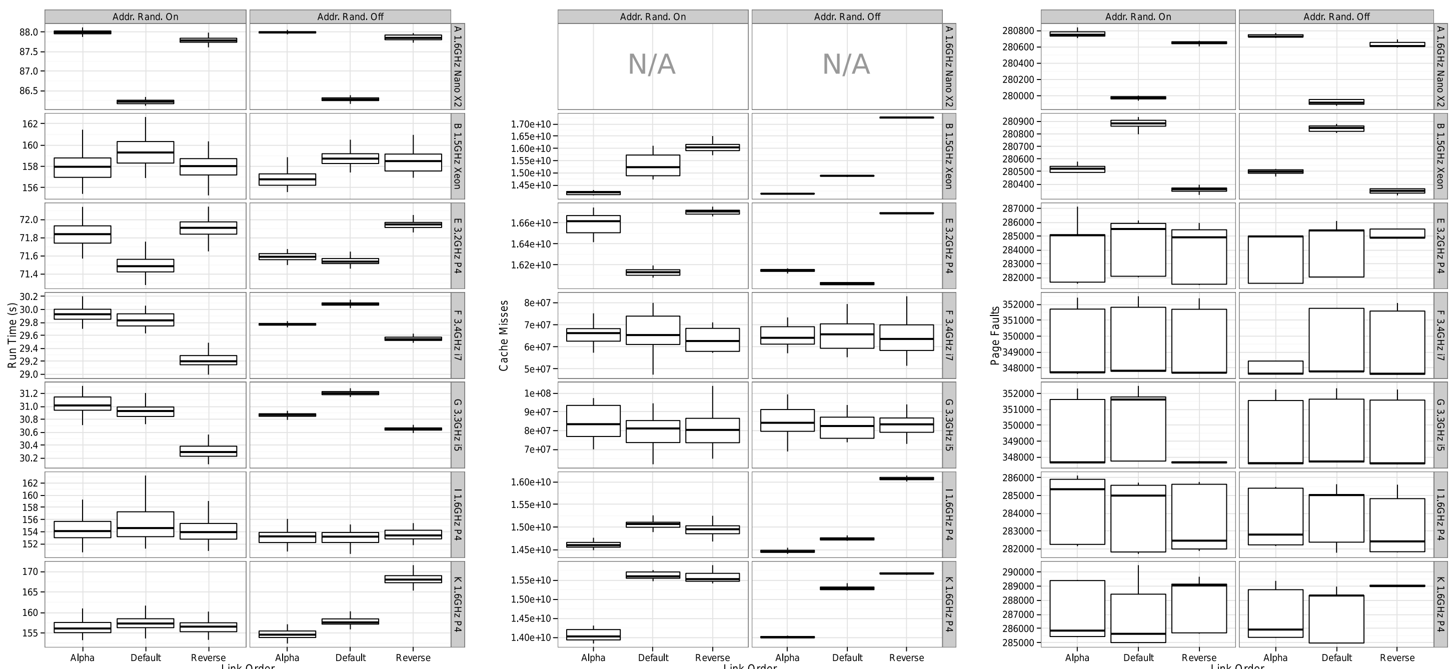
Figure 4.6 shows results for the different metrics for this experiment. These facet plots are divided by address randomization (top header) and host (right header). Each subplot contains three boxplots, one for each link order explored. The plots do not show the zero origin to demonstrate the small differences in results between experimental setups. Figure 4.6a shows execution time, and shows that there is indeed a small, but statistically significant change in execution time between the different link orders on most cases ($P < 0.05$). An exception to this rule is machine I, a 1.6GHz Pentium 4, where this effect is not statistically significant.

It is also clear that the link order effect does not depend on the address randomization feature of Linux being turned off; most machines show different execution times between link orders even when address randomization is turned on. However, machine K (shown in the bottom of Figure 4.6a) shows a link order effect only when address randomization is turned off, contrary to the other machines.

To help understand this effect, Figure 4.6b shows the cache misses for the experiment[2]. This plot shows that there appears to be a correlation between link order and cache misses for most machines, but they do not necessarily mirror the execution time effect seen in Figure 4.6a.

Finally, Figure 4.6c shows that the correlation between page faults and link order is statistically significant, yet practically negligible; even in the case of the Nano X2 and the

[2]Data is missing for the Nano X2 due to the lack of hardware performance counter support, needed for measuring cache misses.



(a) Execution Time

(b) Cache Misses

(c) Page Faults

Figure 4.6: Effect of link order on Perlbench execution time, cache misses, and page faults.

Xeon machines (top two subplots): less than 1%. A possible explanation for this is that the sum of code and data for the benchmark is small enough to fit within a page, no matter the order the object files are linked in.

Mach.	Factor	Mean Sq.	p-value
A	Link Order	135.77	<2e-16
	Addr. Rand.	0.07	0.0301
	Link Order:Addr. Rand.	0.04	0.0687
	Residuals	0.01	
K	Link Order	1856.5	<2e-16
	Addr. Rand.	1234.6	<2e-16
	Link Order:Addr. Rand.	1864.9	<2e-16
	Residuals	3.3	

Table 4.3: Reduced ANOVA table for Perlbench execution time on machines A and K.

Table 4.3 shows the reduced ANOVA table for machines A ($R^2 = 0.977$) and K ($R^2 = 0.855$). This table shows that the link order effect is significant for both of these machines ($P < 0.01$), but only in machine K do the address randomization factor ($P < 0.01$) and the interaction between address randomization and link order factors ($P < 0.01$) play a part. In machine A, both of these are not statistically significant at the 99% confidence level ($P > 0.01$ in both cases). This suggests that the effect of address randomization, which is present in machine K but not in others, is highly dependent on the machine.

Therefore, since neither cache misses or page faults correlate with the varying execution time, the correlation between link order and execution time still merits more investigation. DataMill is a powerful tool for researchers in performance evaluation, since it allows the systematic variation of correlated factors, such as link order and address randomization, and the simultaneous collection of multiple relevant metrics, such as cache misses and page faults.

4.4 Discussion

The implementation of the infrastructure and the execution of the case-study experiments raised several interesting questions.

Package Testing Debugging packages took considerably longer than expected, especially in the case of Perlbench. This was mainly due to the non-standard build system

distributed by SPEC, which requires manual configuration of target architecture parameters. Our experience with it led us to create a virtual worker image on which users can debug their packages. We are currently also investigating a special "debug" experiment type which would execute packages once on each available architecture to ensure it behaves as expected.

Firewalled Workers The addition of remote nodes (located in remote universities) was a challenge, mainly because of firewalls. Our current implementation uses two-way communication, which requires special treatment for firewalls that reject all incoming connections. In the near future we plan to move to a one-way communication design where the master node is entirely passive (i.e., all communication is inbound), sidestepping this issue and minimizing the effort needed to integrate new workers.

Worker Processing Power Even though small embedded targets can run Gentoo, their performance, particularly in compile phases, is prohibitively low for large experiment design spaces. In some instances of the compression experiment, we noticed that the compile time exceeded the execution time of the experiment by several orders of magnitude. In the future, we plan to investigate providing remote compilation support for experiments.

Chapter 5

Minimizing Performance Variability

Chapter 4 introduced DataMill, an infrastructure that dramatically lowers the cost of statistically rigorous computer performance evaluation. While DataMill provides researchers with tools to automatically explore hidden factors, using all factors in the same design can quickly lead to combinatorial explosion and to an excessive number of jobs. Therefore, researchers should know the extent to which hidden factors are *likely* to affect their performance metric.

In this chapter, we quantify the expected effect of hidden effects related to memory layout on the mean performance and on the variability of experiments, so that developers and experiment designers can prioritize what factors to explore. In case they do not have the time or ability to vary these factors and quantify their effects, they can choose sane defaults that minimize the variance in their metric of interest.

5.1 Experimental Setup

We leverage DataMill to quantify the effect of memory-layout-related factors on the execution time of a wide range of benchmarks, on a wide range of machines. The benchmarks chosen represent a large subset (27 out of 31) of the SPEC CPU 2006 benchmark suite [39], shown in Table 5.1a, selected because they are CPU-bound.

We are interested both in increases and decreases in execution time, the benchmarks' native metric. Table 5.1b shows the machines used. This large set of heterogeneous machines and benchmarks will provide a greater sample size of the machine population than

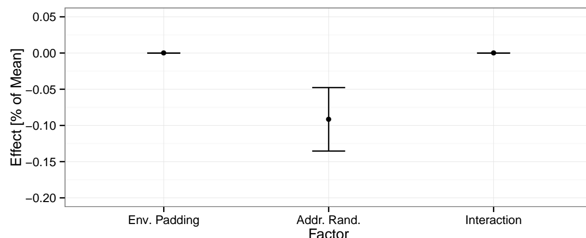


Figure 5.1: ASLR and environment padding effects on Bzip2 on machine 128.

previous research, painting a more comprehensive picture of memory-layout's inter-machine effects.

We control memory layout via five factors: Address Space Layout Randomization (ASLR), POSIX environment size, position-independent executables (the `-fPIE` compiler flag), performing warm-up runs before measuring performance, and, finally, rebooting each machine before collecting performance data. We describe each of these factors and how they affect performance in Sections 5.2 through 5.4. Note that we do not use all machines and all benchmarks for all experiments, due to technical issues: machine failures between experiments, and limited computing time. The figure headers in each section list which machines and benchmarks were used in each experiment.

Unless specifically stated, trials are conducted using the default DataMill strategy: random order, with reboots before every single trial. When any of our five factors is not the focus of an experiment, it is fixed to its "natural default", i.e., what the majority of Linux systems are deployed with: *Address Space Layout Randomization (ASLR) = on*, *environment padding = 0*, *`-fPIE` = off*, and *warmups = none*.

5.1.1 Data Analysis

In this chapter, we will graphically represent the results of ANOVA for each experiment, for easier comprehension of the large volume of data that we present. As an example, Figure 5.1 shows the effect size (i.e., the β_i coefficients) for the ASLR and POSIX environment size factors and their interaction on Bzip2 when executed exclusively on machine 128. The x-axis shows the source of variability, and the y-axis shows the size of the effect as a percentage of the mean performance, with error bars showing one standard error around the effect estimate. We normalize effect sizes to the mean so that effects from different benchmarks and machines can be directly compared to each other.

Benchmark	Area
Astar	Path-finding
Bwaves	Fluid Dynamics
Bzip2	Data Compression
CactusADM	Physics
Calculix	Structural Mechanics
GCC	Compiler
GemsFDTD	Electromagnetics
GobMK	A.I.
Gromacs	Molecular Dynamics
H264	Video Comp.
hmmer	Gene-Sequence Searching
Lbm	Fluid Dynamics
Leslie3d	Fluid Dynamics
Libquantum	Physics
mcf	Combinatorial Optimization
Milc	Physics
namd	Molecular Dynamics
Omnetpp	Simulation
Perlbench	Perl Programming Language
Povray	Ray-tracing
sjeng	Artificial Intelligence
Soplex	Optimization
sphinx3	Speech Recognition
Tonto	Chemistry
Wrf	Weather
xalancbmk	XML to HTML Translation
Zeus	Physics

(a) Benchmark Set.

ID	Processor	RAM
73	VIA Nano X2 1.6GHz	1.7GB
75	Pentium 4 3.20GHz	900MB
80	Core i7-2600K 3.40GHz	8GB
81	Opteron 8378 2.4GHz	32GB
84	Pentium 4 1.80GHz	1GB
88	Pentium M 1.70GHz	1GB
90	Pentium 4 3.20GHz	1GB
91	Pentium 4 2.40GHz	1GB
93	Pentium 4 3.40GHz	900MB
94	Pentium 4 1600MHz	500MB
96	Pentium 4 3.20GHz	2GB
97	Pentium 4 3.00GHz	900MB
98	Pentium 4 3.00GHz	900MB
99	Pentium 4 2.80GHz	900MB
101	Pentium 4 1.60GHz	900MB
104	Pentium 4 1.80GHz	500MB
105	Pentium 4 3.20GHz	900MB
106	Pentium 4 3.20GHz	500MB
125	Core i5-2500 3.30GHz	8GB
128	Pentium D 3.00GHz	2GB
129	Atom N270 1.60GHz	200MB
130	Pentium 4 3.20GHz	900MB
131	Celeron 131MHz	200MB
132	Athlon Processor 757MHz	700MB
135	Athlon 64 Processor 3500+	2GB
143	Xeon 5160 3.00GHz	2.5GB

(b) Machine Set.

Table 5.1: Experimental setup.

Figure 5.1 indicates that the effect size of environment padding and ASLR and their interaction is less than one tenth of one percent of the mean. Only address randomization has a statistically significant impact on performance with $P < 0.01$ (the environment padding effect and the interaction between the two effects are not statistically significant). However, significance only indicates that an effect may be distinguishable from noise. Pragmatically, an effect must also be large enough to be of importance; for example, a statistically significant effect of 0.1% of the mean is small enough to be negligible for the majority of applications.

In this chapter we use Kruskal-Wallis tests to compare the distribution of coefficients of variation between factors. For example, we execute several benchmarks on several different machines with $ASLR = on$ and $ASLR = off$, and calculate the coefficient of variation for each machine-benchmark pair, giving us a sample of coefficients of variation for both levels of ASLR. We then perform the Kruskal-Wallis test between these two samples to test if ASLR affects the coefficients of variations in this general (all benchmarks, all machines) case. If the test confirms a statistically significant difference between the distributions of coefficients, we can say that a factor adds or reduces non-determinism to the experiments at hand, which is important for i) developers that wish to ensure their software's performance is as consistent as possible and ii) researchers that wish to maximize the power of their statistical tests.

5.2 Experiment 1: ASLR vs. Environment Padding

Our first experiment executes benchmarks under different memory layouts by varying two factors: POSIX environment padding and Linux's ASLR feature. The environment size of a process affects the memory layout of that process since Linux places environment variables at the beginning of the process virtual address space. The larger the environment, the farther out in the address space the code section will be, and depending on the increments of this offset, cache or paging-related performance effects may occur. Similarly, Linux's ASLR feature may cause these effects to appear due to the heap, stack, and libraries of a process being mapped at different addresses.

The experiment follows a two-level full-factorial design [67], where both factors have two levels (or settings) and are explored concurrently (i.e., all possible combinations of factors are explored). The environment paddings used are zero bytes or 10 928 bytes (chosen because it does not divide evenly into pages), and ASLR is set to "on" or "off". Each factor combination is executed three times on each machine to allow for a measure of

variance, leading to a total of twelve $(2 \times 2 \times 3)$ jobs per machine, per benchmark, totaling 2244 individual jobs.

5.2.1 Results

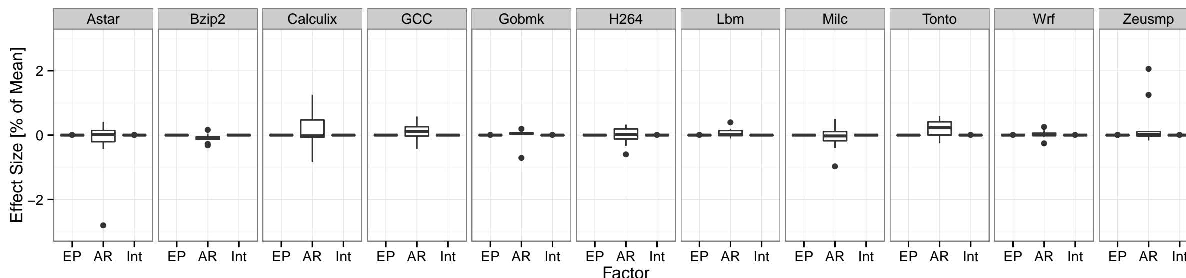


Figure 5.2: ASLR and environment padding effects by benchmark.

Figure 5.2 shows the relative effect sizes for environment padding (EP), ASLR (AR), and their interaction (Int), for each benchmark, with data from all machines grouped in a single box plot, following the same visual syntax as those in Chapter 3. If a benchmark had a systematic sensitivity to either factor or their interaction, one of these plots would show a significant deviation from zero. As the plots show, however, the most significant deviation is still well under 3%.

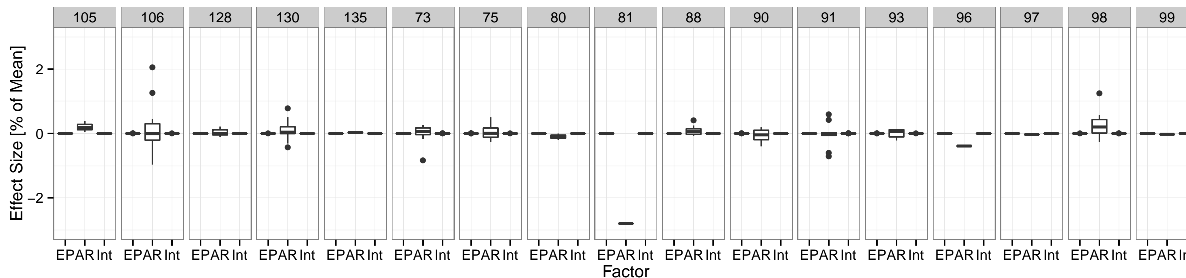


Figure 5.3: ASLR and environment padding effects by machine.

Figure 5.3 shows the relative effect sizes for each factor and their interaction for each machine, with data from all benchmarks grouped in a single box plot. Similarly to Figure 5.2, this plot would reveal a machine's systematic sensitivity to either factor or their

interaction (e.g., one machine would have all positive effects, another all negative ones), but, again, no such significant sensitivity exists.

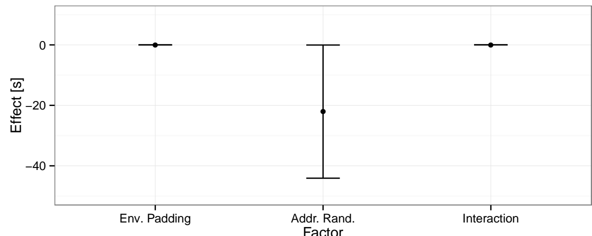


Figure 5.4: Worst case ASLR effect on astar on machine 81.

The previous figures show that, from a bird's eye view, the memory-layout-related effects that we have tested appear negligible. Figure 5.4 shows the absolute worst case observed in our experiments, Astar on machine 81. The x-axis shows the factor name, and the y-axis shows the *absolute* magnitude of the effect. On this machine-benchmark pair, the ASLR factor had an effect estimate of -22.06s, while the mean execution time of that benchmark on that machine was 787.3s (a relative effect of -2.8%). Note that the error bars straddle the zero axis, and therefore these effects are not statistically significant. Most importantly, even if we were to ignore statistical significance (running the risk of treating noise as a real effect), this effect is not *practically* significant, and therefore, not even our worst single observed case is cause for concern.

We now investigate the effect of these two factors on the variability of the experiments' metric. Figure 5.5 shows the coefficient of variation – a normalized measure of dispersion calculated by dividing the variance by the mean of the measurements – on the logarithmic x-axis, by machine on the y-axis. Each dot represents the coefficient of variation for a single point in the experiment design (e.g., ASLR on, Env. Padding 0, on machine X, benchmark Y). The figure shows that there is no discernible *pattern* of increased variation caused by the environment padding factor, and therefore controlling this factor does *not* lead to more deterministic experiments. There might be, however, a slight increase in coefficient of variation between ASLR on and off, seen by comparing the left-hand subplots with the right-hand ones. We test this with a Kruskal-Wallis test between the samples with ASLR on and off, and fail to reject the null hypothesis ($P = 0.49$), i.e., ASLR does not cause a statistically significant increase in variability.

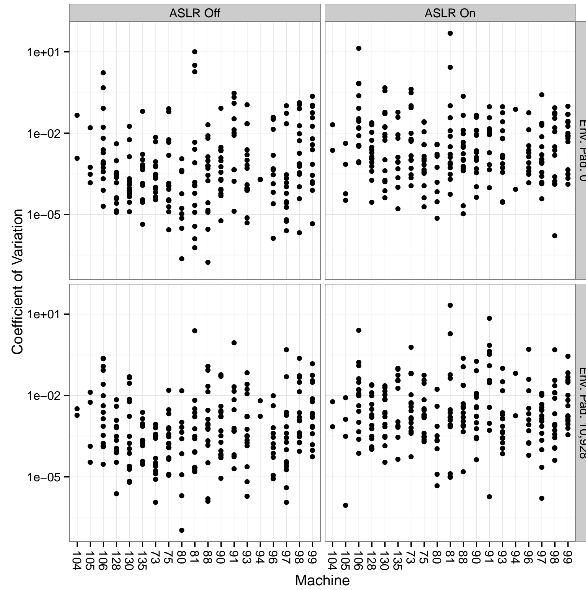


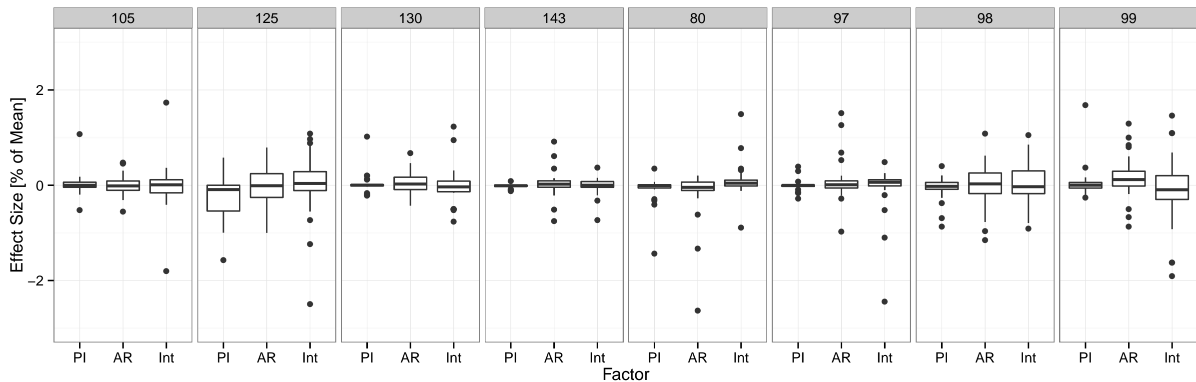
Figure 5.5: Coefficient of variation for Experiment 1, by machine.

5.3 Experiment 2: ASLR vs. Position-Independent Executables

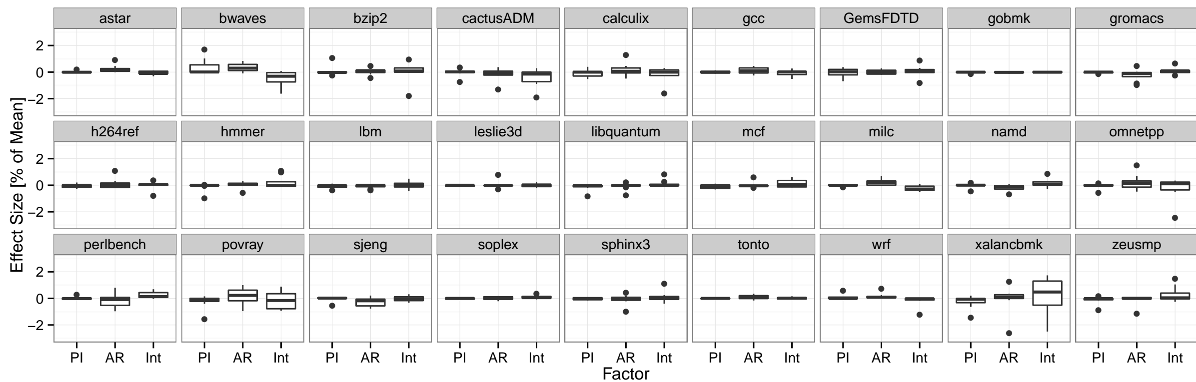
Our second experiment forces benchmarks to execute under different memory layouts by varying two factors: ASLR, used in the same way as in the previous experiment, and GCC's Position Independent Executable flag, or `-fPIE`. By using `-fPIE`, ASLR will affect not only the heap, stack, and dynamically linked libraries, but also the main executable section. This could reveal an effect on performance due to the positioning of executable code, such as loops' positions relating to cache or page boundaries. The implementation of `-fPIE` requires an additional register to maintain the base address of the executable, possibly causing run-time overhead. We are particularly interested in the interaction between these two factors, since `-fPIE` will only affect the base address of the binary if ASLR is on. This experiment's full-factorial design consists of both ASLR and `-fPIE` being set to "on" or "off".

5.3.1 Results

Figures 5.6a and 5.6b show the effect sizes for `-fPIE` (PI), ASLR (AR), and their interaction (Int), by machines and by benchmarks, respectively. Contrary to expectation, `-fPIE` in isolation had effect estimates very tightly packed around zero, meaning that the overhead added by the flag is very low in the benchmarks and machines we used. As for ASLR and the interaction effect, more variability in the effect estimates is observed, but all still fall within 3% of the mean. These results confirm the "performance neutrality" of ASLR in isolation, a result also found in Experiment 1,



(a) By Machine.



(b) By Benchmark.

Figure 5.6: ASLR and `-fPIE` effects.

Figure 5.7 shows that, again, variability is unaffected by `-fPIE`, but there appears to be

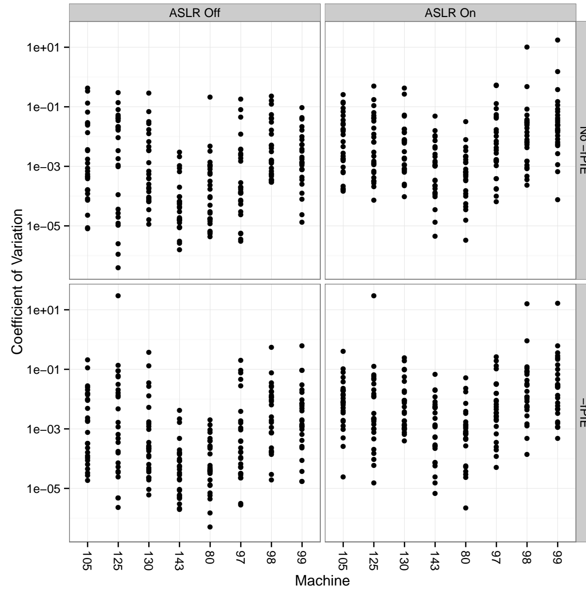


Figure 5.7: Coefficient of variation for Experiment 2.

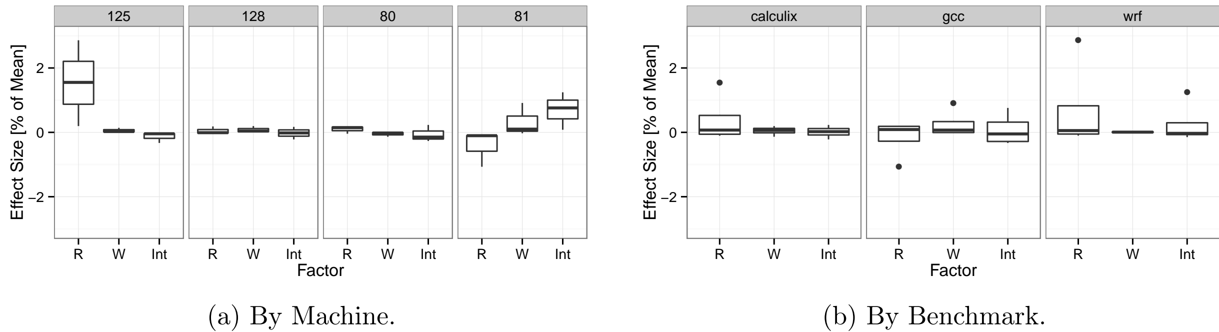
a minor increase in overall variability going from ASLR off to on. We test this variability for statistical significance using the Kruskal-Wallis test, and again find that this change is not statistically significant ($P = 0.491$).

5.4 Experiment 3: Warm-ups and Reboots

Our third experiment investigates the effect of warm-up runs and reboots on the performance of three of the benchmarks: Calculix, GCC, and Wrf, chosen due to their "high", "medium", and "low" susceptibility to the factors in Experiment 1. Note that none of the effects were statistically significant in Experiment 1, so "high" susceptibility here simply means the one with the largest observed effect estimate interval.

Warm-up runs (i.e., executing each benchmark a number of times before collecting performance measurements) aim to lower variability of the benchmark by ensuring that any caching mechanisms (from file-system caches up to main memory caches in the processors) are pre-populated with relevant data. In our warm-up tests, we executed each benchmark three times in immediate succession but only noted the execution time of the last execution.

Rebooting between replicates, DataMill's default policy, also aims to reduce variance



(a) By Machine.

(b) By Benchmark.

Figure 5.8: Reboot and warm-up effects.

by ensuring that the system is in a "clean" state, i.e., previous runs of other benchmarks (or any other processes) cannot interfere with performance. For reboot trials, the machine is rebooted immediately before the trial is conducted. For no reboot trials, all trials are collected in a single boot. These two factors were studied together in a full factorial design, allowing us to measure the effect of the interaction between warm-ups and reboots on performance.

5.4.1 Results

Figures 5.8a and 5.8b show the effect sizes for reboots (R) and warm-ups (W) and their interaction (Int), by machines and by benchmarks, respectively. Similar to Experiments 1 and 2, most effects are not statistically significant ($P > 0.05$ for 78% of effect estimates) and not practically significant, since *all* fall within 3% of the mean. Despite the small effects, Figure 5.8a has two stand-out cases, with effect sizes that do not straddle the zero axis: reboots on machine 125, and the interaction factor on machine 81. This suggests the need to experiment on as many heterogeneous machines as possible, in order to avoid incorrectly assuming such effects are general; a researcher with only those machines could assume such effect applied on every machine. Also of note is how narrowly concentrated around zero warm-up effect estimate is, this suggests that it had *no* effect on mean performance. In general, these results show that neither rebooting or warm-ups, in isolation or together, had a significant or substantial effect on the mean execution time of these benchmarks. On very short running benchmarks, warm-ups may be more beneficial due to time spent loading code and input data from disk being a larger part of overall running time; these benchmarks have more to gain from those files being preloaded in RAM. The same is

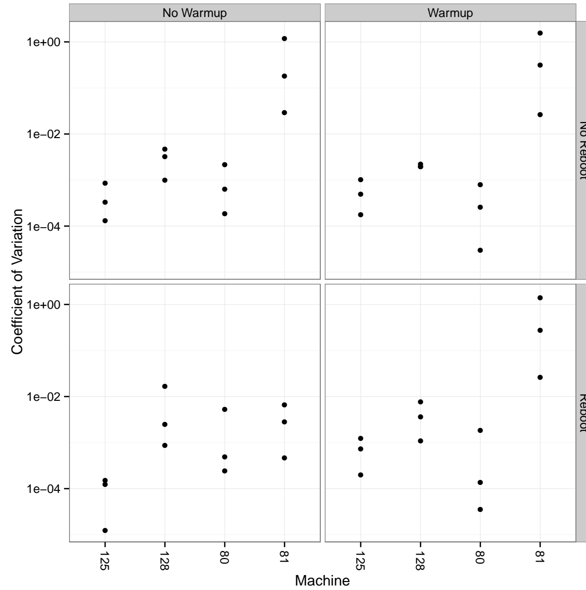


Figure 5.9: Coefficient of variation for Experiment 3.

true for some managed languages: if dynamic compilation is done in the warm-up phase, performance of the measurement runs will be improved, and possibly more deterministic.

Figure 5.9 shows the coefficient of variation for each benchmark on the y-axis, plotted by each machine on the x-axis, split by our factors of interest. Again, in general, there appears to be no pattern of increased or decreased coefficients as the factor levels change, leading us to believe that, in the general case, neither reboots or warm-ups contribute to lowering overall non-determinism. One exception is reboot vs. no reboot on machine 81 when no warm-up is conducted. In this case, rebooting leads to significantly less variation across all three experiments. Since this is a machine-specific effect, we suspect it may be caused by idiosyncrasies of the hardware or low-level software. Since the cost of automating the rebooting between trials is low, we feel this is evidence enough to endorse it as a way to reduce performance variance. Note that we made no effort to cause "artificial" non-determinism by running random processes before the "no reboot" trials.

	Original	Reproduction
Mean Env. Pad.	0.000032%	0.000045%
Max. Env. Pad.	0.00035%	0.00031%
Mean Addr. Rand.	0.23%	0.42%
Max. Addr. Rand.	2.8%	5.94%
Machines	17	13
Applications	11	9

Table 5.2: Reproduction summary.

5.5 Experiment 1 Revisited

While the goal of statistical methods such as factorial experiment design and linear regression is to ensure reproducible results, we have repeated the experiment described in Section 5.2 one week later to evaluate how reliable those measurements were, and how easily the results from a DataMill experiment could be repeated at a later date.

Table 5.2 presents a comparison between the original experiment and the reproduction experiment. The numbers presented for mean and maximum effects are based on the absolute values of the effects, such that negative effects do not cancel out positive ones. Due to a string of hardware malfunctions related to a building-wide power failure, only 13 of the original 17 machines were available for the repeated experiment, and only 9 of the original 11 applications ran to completion for the full duration of the experiment. As seen in the table, the mean estimates of the repeated experiment were similar to the original experiment (that is to say, effects were negligible), but the single worst case effect of the address randomization factor grew significantly. However, much like the 2.8% worst case effect of the original experiment, the 5.94% effect of the repeated experiment is also statistically insignificant, and therefore indistinguishable from noise. The equivalent plots to Figures 5.2 and 5.3 are suppressed due to space constraints, but are largely indistinguishable from the original plots.

Although precisely the same conditions were not recreated as a result of a power failure that led to hardware failures, we were able to reach the *result* from the first experiment. This is due to the built-in reproducibility of results on a wide array of machines and benchmarks. This broader strategy of reproduction is more desirable than simply rerunning experiments on a single platform, since results derived from it are more likely to be reproducible by other researchers on other platforms.

5.6 Average Case Variability

Given the breadth of machines and benchmarks explored here, we feel confident in saying that memory effects, as controlled by ASLR, two levels of environment padding, `-fPIE`, warm-ups and reboots, are not significant enough to warrant widespread concern for DataMill users. Exceptional cases, where these effects are more pronounced, or where a sub-3% performance difference is crucial, may require closer attention, but for the majority of computer performance researchers these effects appear negligible.

This also means that, at least on DataMill, these factors are not particularly useful for the reduction of non-determinism. Given the low variability we observed (coefficients of variation ranging from 10^{-5} to 10^{-1} on the vast majority of cases), there was not much improvement to be made. Therefore, we believe that the strategy of using a clean, dedicated system, with reboots between trials, is sufficient to collect reliable performance measurements.

While our results point to a lack of sensitivity to memory layout performance effects in the general case, and we are confident that the average developer should not be worried, the fact remains that a given experiment could be susceptible to them. How can we determine whether or not memory layout effects will be a concern for a given benchmark-machine pair? Currently, there is no simple answer other than to perform exploratory experimentation.

DataMill can provide this functionality, by varying known nuisance factors in a small number of screening trials. If developers find no evidence that their experiment is susceptible to a hidden factor, they can drop the additional dimensions from their experiment design and continue exploring the factors they are directly interested in.

5.7 Discussion

While we conducted the experiments for this chapter, we learned a few lessons concerning the repetition of experiments on DataMill. This section discusses those observations.

Reproduction of Data vs. Reproduction of Results Given the fast pace of computer technology, measuring performance on a single platform only goes so far. By reproducing an experiment on various platforms and under various operating conditions, researchers improve the generality of their results. The fact that our hidden factor worst

case results differ from those found by others in the literature confirms that we must use more than one experimental condition.

In [71], Mytkowicz *et al.* find that link-order and POSIX environment size have a significant effect on performance, and that they can cause very dramatic regressions or speedups in performance (57% and 300% worst cases respectively). The paper concludes saying that we must be careful not to run experiments against only a small set of conditions, and that we need to be cautious of the impact of small variations in our experimental environment, as they can be deceptively influential.

Although our results point towards much lower worst case memory layout effects — 2.8% and 5.94% worst case effects for Experiment 1 and its reproduction respectively — this does not contradict data found by other researchers, nor does it contradict their results. Instead, our data improves our understanding of memory layout performance effects, specially regarding how widespread they are. We observe negligible effects in different benchmarks, and on different hardware configurations.

The process of reproducing the *data* from an experiment allows researchers to ensure that the experiment is calibrated appropriately, and to work out issues in the experimental setup. Once the experiment yields the data previously observed in the original study, performing sanity checks by varying additional, unrelated factors can shed light into how dependent on the original setup the results are.

Reliability of Systems In our attempt to repeat our own results at a later date, we found that several of the machines used in the previous experiments had ceased to function as the result of a power failure. Experiences like this highlight the importance of reproducible *results* as opposed to reproducible *data*. Although we were unable to replicate the precise conditions of the experiment due to our machine losses, we *were* able to reproduce our previous *results*. This would be impossible without robust statistical methods.

Interaction Between Memory-Layout Factors Since we did not investigate all factors *together* (due to combinatorial explosion), there could conceivably be some interaction effect we failed to detect. For example, the performance effect of `-fPIE` may be amplified by padding the POSIX environment, but since these factors were not investigated together, their interaction is not measurable in our data. Given their negligible effect in isolation, it is unlikely that such an interaction will be significant, but more exploration is warranted nonetheless.

Part II

Predicting Computer Software Performance

“Essentially, all models are wrong, but some are useful.”

— George E. P. Box

Chapter 6

Predicting Performance Changes During Software Development

The approaches in Part I of this thesis constitute a foundation for low cost, low variance, statistically rigorous empirical computer performance evaluation. However, as discussed in Chapter 1, there are cases where direct performance measurement is justifiably infeasible, even using dedicated performance evaluation infrastructures such as DataMill. One such case is performance regression testing in software projects with high development rates.

Ideally, every single code change in a software project would be put through a performance test suite before being admitted to revision control, so that developers would know immediately, if and when a performance regression was introduced. However, executing a statistically rigorous experiment for every single commit is infeasible due to the turn-around time of a properly designed experiment, and the excessive cost of dedicated infrastructure.

In this chapter we present Perphecy, a lightweight performance-change prediction approach that builds on the low-variance experimentation methods presented in Chapter 5. Perphecy maintains near-zero performance regression detection latency while adding limited testing overhead by predicting which code changes will cause performance regressions for which performance tests.

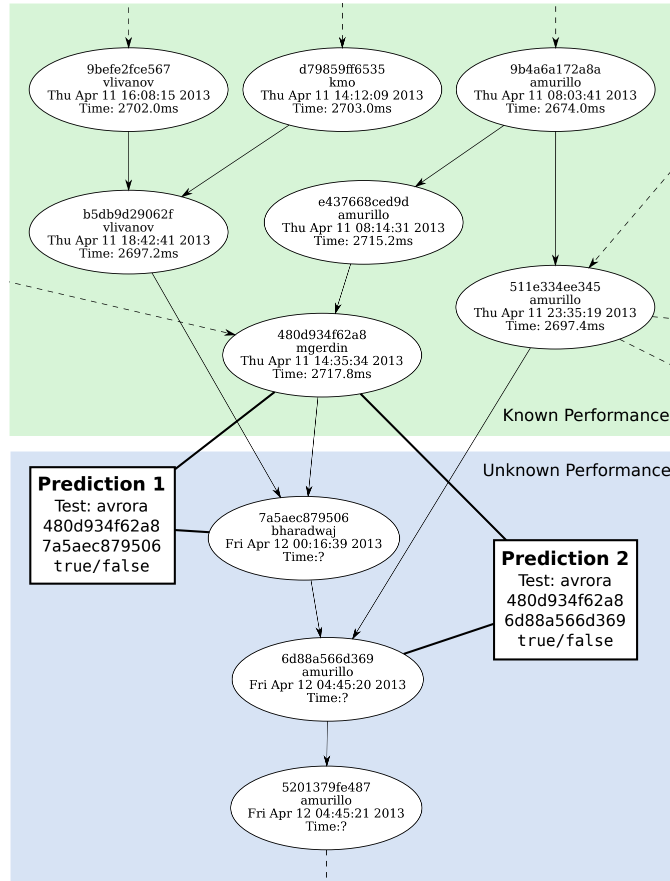


Figure 6.1: Graph of a period in the development of the HotSpot JVM, with performance data from the DaCapo avrora benchmark.

6.1 Problem Definition

Software development consists of a series of code changes, or *commits*, through one or more development branches, that ultimately converge into a production-ready version of the software. As development progresses, developers execute a performance test or benchmark suite to detect any performance regressions. Figure 6.1 shows a small set of commits in the development history of the HotSpot JVM. The ellipses represent individual commits, and contain the commit identifier, the author, the time of the commit and, where available, the performance of avrora, one of the benchmarks in the DaCapo suite. The arrows represent parent-child relationships between commits, dashed arrows connect to/from commits

outside the figure. A commit with multiple arrows leaving it is a branch in development, while a commit with multiple inbound arrows is a merging of two branches. The figure divides commits in two sets: those for which performance of avrora (one of the HotSpot benchmarks) has been measured, and those for which it is has not.

If HotSpot developers wish to avoid executing every benchmark at every commit, while still detecting performance changes as they are introduced, they must somehow predict which commits will cause a performance change for which benchmarks. This question is posed on per-benchmark, per-commit-pair basis, such as Prediction 1 in Figure 6.1. This prediction will be `true` if the performance of avrora (the benchmark) is expected to change between commits 480d934f62a8 and 7a5aec879506 (the commit pair), and `false` otherwise. To make this prediction, developers can use static data about the commits (i.e., code differences) and also dynamic data – profiling data – from the previous commit (i.e., code that avrora reaches).

If Prediction 1 returns `false`, developers will skip that particular benchmark and continue development assuming their change did not affect the performance of avrora. When they make a new commit, 6d88a566d369, they will make new predictions about the performance effect of the new change. Since performance results are unknown for 7a5aec879506, the predictor must be able to "skip" it, rooting its prediction on a commit for which dynamic data from a run of avrora is available, 480d934f62a8, to make Prediction 2. This process continues along with development, with new performance data being added as benchmarks execute.

We now formalize the problem of predicting which commits will affect the performance of a given benchmark. We first define what performance changes are of interest:

Performance Change: A statistically significant difference in mean execution time (or other performance response metric) of a given benchmark between two versions of the same software that differ in execution flow.

The predictor is concerned only with performance changes that are repeatable, i.e., statistically significant, and come about due to differences in the code that is executed. We do not consider performance differences that arise from unreachable code changes, such as memory layout effects [70]. Furthermore, this performance change can be either a speedup or a slowdown, since developers are interested in both cases. Developers may prefer to consider only performance changes above a certain threshold, but we consider changes of *any* magnitude, for generality. We define the task of a predictor as follows:

Problem Statement: Given a set $A = \{a_i : 1 \leq i \leq n\}$ of software versions for which dynamic and static data are available, and one version a_u for which only static data are available, predict which benchmarks in a benchmark set $W = \{w_j : 1 \leq j \leq m\}$ will incur a performance change from each a_i to a_u.

This definition does not restrict the relationship between the versions of the software being investigated; they need not have a direct parent-child relationship in version control, they need only to be able to run the same benchmarks. This allows developers to avoid running benchmarks for a commit, yet keep making predictions about its descendants in relation to the latest known version a_n.

While our problem statement allows using dynamic information from prior versions (which is available in a repository) it does not allow us to use dynamic information from a_u, since that is exactly what we are looking to avoid. We refer to the set A as the set of *known* versions, since profiling data and execution times are available for all a_i, and version a_u is the *unknown* version, for which a predictor will estimate if there will be a performance change. The prediction set consists of the set of benchmarks $W_p \subseteq W$ whose performance is expected to change from a given a_i to a_u, or:

Prediction: A boolean value for a given (a_i, a_u, w_j) tuple, where `true` means that the performance of w_j is expected to be different on a_i and a_u, `false` otherwise.

Predictors may have additional outputs such as the confidence in such a change or a prediction of the magnitude of the change, but we focus on binary predictions (run w_1, do not run w_2, etc.) since, despite their simplicity, they suffice for the developer to decide which benchmarks to run on a_u.

6.2 Anatomy of a Predictor

Figure 6.2 shows a general, abstract model of a performance predictor, based on the problem statement defined above. Boxes denote artifacts, ellipses denote processes, and arrows denote input/output relationships.

A static analyzer $stat(A, a_u)$ takes the set A of known versions and the unknown version a_u. The static analyzer extracts a set of static data D_s from these inputs without executing them and stores them in a repository to be recalled when predictions are made. Examples of static data of interest include: source code and binary deltas between each a_i and a_u, the differences in number of static floating point instructions between a_i and a_u, or

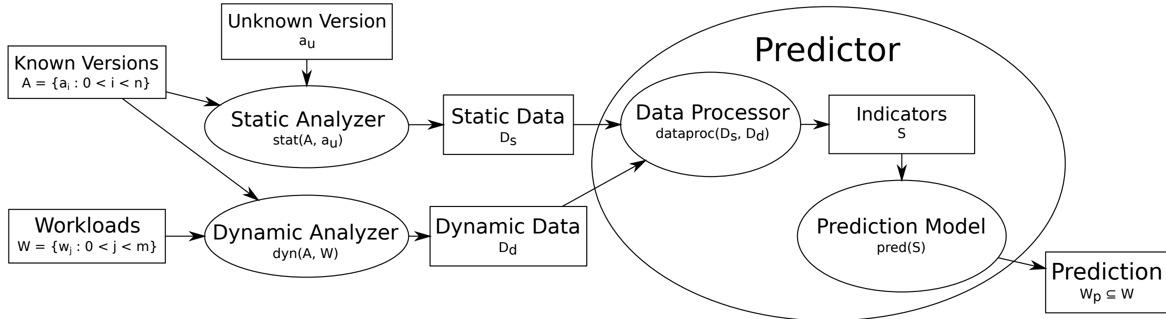


Figure 6.2: Abstract model of a performance change predictor.

the differences between compiler parameters. The static analyzer, as the only component with access to both known and unknown versions, generates all information regarding the *difference* between them.

Optionally, a dynamic analyzer $dyn(A, W)$ can store performance data from the set of benchmarks W on the set A of known versions, in a repository of dynamic data D_d. Examples of dynamic data are a list of functions reached by each benchmark, the total number of cycles spent in each function, and the full call graph. Analyzers can collect this data as the new versions are selected for benchmarking, and then recall it as needed when making predictions.

The predictor is composed of two processes, a data processor and the predictor engine. The data processor $dataproc(D_s, D_d)$ combines static and dynamic data into a set S of *indicators*. We define indicators as:

Indicator: A scalar value derived from D_s and D_d, where high values are expected to correlate with a performance change in a benchmark w_j between an a_i and a_u.

For example, the indicator "number of functions changed in source" is derived from the static data alone, and the expectation is that higher values are more likely to correlate with overall performance changes[1]. For naturally boolean indicators such as "the compiler parameters have changed", values of 1.0 and 0.0 can represent `true` and `false`, respectively.

Finally, the predictor takes the processed indicators S as input to a prediction engine $pred(S)$ and emits a boolean prediction for each (a_i, a_u, w_j) of interest. Possible implementations of such a predictor include an expert system, a machine learning system, and a statistical model.

[1]We will empirically confirm or refute these expectations in Section 6.4.2.

Expert systems [80] follow strict rules as to when predictions should be `true` or `false`, e.g., "return `true` if the number of changed functions is greater than 10". Such predictors are effective for problems that can be exhaustively enumerated, but may not generalize well. Statistical regression models [67] relate the value of indicators to the value of the performance metric, and its coefficients are calculated by a fitting process that takes a set of training data. A fitted model can then make predictions as to the magnitude of a change, and, if the predicted change is different from zero with the desired statistical confidence, a value of `true` is returned. Statistical models perform well when the relationship between the indicators and the performance metric is simple (linear, or low-order polynomial), but those cases are rare in unnatural systems such as computer software [20]. Finally, there are a multitude of machine learning [81] techniques ranging from decision trees to neural networks that can transform indicator values into a prediction. Each of the techniques discussed above have their own strengths and weaknesses, but given the nature of indicator data and the complex relationship between indicators and their correlation with overall performance changes, we believe the most successful approaches will involve machine learning. Thus, we pursue a simple machine learning approach throughout the rest of the paper.

6.3 Case Studies

Software	Description	Commits	Tests	$\|W\|$	$\|A\|$	Sig.
Git	Version control system, short-running	11.78/day	init, add, commit, diff, clone	5	201	13
glibc	C library, pervasive, OS-facing	4.94/day	libc-bench suite	6	98	0
HotSpot	Java virtual machine, variable running lengths, generates dynamically compiled code	4.99/day	DaCapo suite	14	50	10
MongoDB	Database management system, daemon	10.18/day	insert, insert then select	2	80	27

Table 6.1: Software projects used in experiments.

Before designing a predictor, we must determine that the potential for reducing wasted work actually exists, and establish a data set on which to evaluate our predictor design. We chose the software projects described in Table 6.1 for being performance-sensitive, actively

developed by multiple programmers, open source, and, most importantly, widely varied in their functionality. The commits chosen were a continuous string going back from the start of April 2013. The table shows the name and description for each software project, the average number of daily commits for the year 2013, the benchmark sets we executed on each of them, the number $|W|$ of individual benchmarks in each set, the number $|A|$ of commits compiled and benchmarked, and, finally, "Sig.", the number of (a_i, a_u, w_j) tuples, where a_u is a direct child of a_i, with a performance change, the ground truth that our predictor will try to match.

Git [31] is a version control system widely used in industry and the open-source community alike (users of Git include Google, Facebook, and Twitter). Our custom benchmarks consist of five operations frequently used by developers, on a relatively large data set: "init" is the initialization of an empty repository, "add" is the addition of the entire Linux kernel v3.15 source code into the staging area of an empty repository, "commit" records the addition of the Linux source code to a clean repository, making a snapshot of the working tree, "diff" consists of the generation of the code delta between a Linux commit from May 4th, 2011 to a commit in June 10th, 2014, and "clone" consists of cloning a local copy of the Linux repository to another directory.

The GNU C Library [32] (glibc) implements the user-space portion of system calls (e.g., `fork()`, `read()`) and other basic functions (e.g., `strlen()`) for the C language. Given its widespread use (the majority of Linux systems), its performance is extremely important. We used the libc-bench suite [58], a pre-existing benchmark suite that exercises memory, string, thread, and I/O functions in the library.

HotSpot [72] is a Java virtual machine implementation maintained by Oracle, and it is the most popular JVM available today [26]. We use the DaCapo benchmark suite [5], a suite of fourteen Java benchmarks designed by a collaboration of academic communities to improve on SPEC Java benchmarks.

MongoDB [66] is a NoSQL DB, built for "scalability, high performance and high availability", so its performance is highly important for its developers. The custom benchmarks used are "insert", which makes one million insertions, and "insert and select", which makes one million insertions and then retrieves them. The load generator executes on the same machine as MongoDB itself.

We compiled a number $|A|$ of commits for each of software project, then executed five or more replicates of each benchmark on each commit, to estimate variance and then perform formal statistical tests. The metric for each benchmark was its externally measured total execution time for all cases except DaCapo, which reports its own internal execution time. Two machines executed all experiments, each with a four-core 3.60GHz Core i7-3820 CPU,

one with 24GB of RAM and the other with 32GB of RAM. To minimize variability in the data, we applied the methodology described in Chapter 4: rebooting the machines after each individual replicate, in addition to randomizing the order in which the benchmarks and commits were executed.

The first notable result is that the number of significant performance changes, shown in the Sig. column of Table 6.1, is low. In the case of the HotSpot dataset, by comparing the results of 10 benchmarks on 57 parent-child version pairs (even though we only evaluated 50 versions, 57 parent-child pairs exist due to branches and merges in the development tree), we found only 10 out of 570 (a_i, a_u, w_j) tuples to have a performance change. This means that the opportunity to reduce wasted work is large: 98.2% in this case. Unfortunately, this also means that missing a single performance change would represent missing 10% of all significant changes.

Also note that for all the glibc commits investigated, *none* incurred a performance change on *any* of the benchmarks. Over all 285 individual (a_i, a_u, w_j) tuples investigated, only two reachable binary functions ever changed, `__init_cpu_features` and `_IO_file_open`, and neither code change caused a significant performance effect on the benchmarks. Since glibc is by far the oldest software project investigated here – its initial release was in 1987 – it is to be expected that its user-facing, performance-sensitive functions (i.e., the ones exercised by our benchmark suite) will not change frequently, but this will pose a challenge to our prediction strategy.

Given the rarity of performance changes, we will proceed conservatively, favoring the detection of all performance changes over reduction in wasted work whenever a choice must be made. The rationale for this is that a false negative is a larger disruption to development (e.g., a performance regression goes unnoticed until the next periodic all-test run, becoming more expensive to fix), while a false positive incurs the overhead of on execution of one benchmark.

6.4 Perphecy

Implementing a concrete version of the predictor described in Section 6.2 involves deciding what static and dynamic analyses to perform, what indicators to derive, and finally, how to combine the available indicators in a prediction. This is a non-trivial task, given the practically infinite possible combinations of data and processes. In this section we present the approach used to design Perphecy, a general, lightweight, effective, and reliable performance change prediction strategy.

6.4.1 Static and Dynamic Analyses

During the development process, the static and dynamic analyzers perform their data collection, storing information that will be later used to make predictions. Given the effectively infinite amount of data that developers *can* collect, we restricted ourselves to what we can collect and process through lightweight processes.

The static analyzer collects the raw source and binary code for all versions investigated. From this data, we can later derive source code deltas, binary code deltas, tool-chain parameter changes (e.g., compiler or linker options), and more. Collecting all source and binary code is computationally trivial, and ensures any changes made by the developers can be detected, since all other aspects of the system (e.g., benchmarks, auxiliary libraries, hardware) will be held constant. By collecting compiled binaries, we are able to detect changes that are not reflected in the source code itself, such as the in-lining of a function.

The dynamic analyzer executes two types of profiling runs, one with Pin [61], and another with Perf [75]. The Pin profiling runs collect per-function call and instruction counts, while the Perf runs collect per-function cache misses, branch counts and cycle counts. We used Pin for Git, glibc and HotSpot, and perf for MongoDB. We decided against collecting more complex dynamic data such as full call graphs or instruction-level traces due to the cost of collection and the large amount of data these would generate. The dynamic dataset allows us to calculate what parts of the code are reachable by each benchmark, and which functions are "hot" or relatively long-running. Note that this information is only available for the known versions, so there is an implicit assumption that this information will still be useful in the unknown version (i.e., not *all* functions will behave completely differently or have vanished altogether). Section 6.4.2 shows that this is the case for the projects we investigate.

6.4.2 Predictor: Data Processor

The data processor must "boil down" the data collected by the analyzers into indicators. This process reduces unformatted, uncorrelated data (e.g., source code changes, list of reached functions for each w_j) to a scalar related to the difference between a known and an unknown version of software (e.g., how many reachable functions have changed). Without expert knowledge about the software projects being studied and the benchmarks being executed, we proceed using our intuition as to what indicators *could* correlate with performance changes in general, and then evaluate the indicators to confirm or refute their usefulness.

Indicator	Description	Rationale	Source Data
New functions	Returns the number of functions in a_u that are not in a_i	A large number of new functions can indicate significant refactoring, which can lead to overall performance changes	D_s
Functions deleted	Returns the number of functions in a_i that are not in a_u	Same as above	D_s
Reached func. deleted	Returns the number of functions w_j reached in a_i that are not in a_u	Same as above	$D_s + D_d$
Top changed function by instruction length	Functions are ranked from 0 to 100 by the number of dynamic instructions they execute over a whole run of the benchmark. Returns the maximum rank between changed functions.	Changes in the longest-running functions are more likely to correlate with overall performance changes, due to Ahmdahl's Law	$D_s + D_d$
Top changed function by call count	Functions are ranked from 0 to 100 by the number of times they are called. Returns the maximum rank between changed functions.	Performance-affecting changes in a function are multiplied by the number of times that function is called	$D_s + D_d$
Top changed function w/ ≥10% instr. delta by call count	Same as above, but functions' instruction length must have changed by 10% or more	Same as above	$D_s + D_d$
Top function length change	Returns the largest instruction length change between functions in %	Large changes to functions are more likely to affect performance than small ones	D_s
Top reached function length change	Same as above, restricted to functions reached by w_j	Same as above with the additional likelihood of reachability	$D_s + D_d$
Top changed function by performance counter	Functions are ranked from 0 to 100 by the count of a hardware performance counter event. Returns the maximum rank between changed functions.	A change in a function with high value for performance counters may correlate with overall performance changes	$D_s + D_d$

Table 6.2: Indicators investigated.

Table 6.2 lists the indicators derived at the data processor, the rationale as to why they would correlate with performance changes, and if they are derived from static data alone or from a mix of static and dynamic data. All of the indicators listed are derived from lightweight operations on the binary static data alone, or a mix of binary static data with dynamic data. The reason behind this is twofold: it ensures that our predictor is as general as possible (detecting which functions changed at source level requires programming language knowledge, for example), and it allows us to easily connect run-time information with static information without having to translate binary symbol names back into their original source counterparts.

Since these indicators return values in widely different ranges, we reduce them to boolean values in order to evaluate them uniformly. For example, the "functions deleted" indicator value can range from zero to the number of functions in a_i (i.e., all were deleted). This indicator function can be reduced to a boolean indicator of the form "x or more functions were deleted", where x is a threshold parameter (discussed later).

Once we determine what the boolean versions for each indicator are, we determine their values at several different thresholds, and evaluate these indicators with respect to two metrics: *hit rate* and *dismiss rate*. Let H be the set of (a_i, a_u, w_j) tuples with a confirmed performance change, and H_s the set of (a_i, a_u, w_j) for which a indicator returns `true`. An indicator's hit rate is defined as follows:

Indicator Hit Rate: $|H_s \cap H| / |H|$

Intuitively, the hit rate is a value between 0.0 and 1.0 that indicates the fraction of performance changes the indicator *correctly* detected, therefore higher values are better.[2] An indicator that simply returns a `true` value independent of input will, by definition, have a perfect hit rate of 1.0. A *hit* is, therefore, a (a_i, a_u, w_j) tuple that an indicator correctly detects as having a performance change. We also say an indicator *covers* a hit if it correctly detects it.

Let D be the set of (a_i, a_u, w_j) tuples with no measurable performance change, and D_s the set of (a_i, a_u, w_j) for which an indicator returns `false`. Dismiss rate, the competing metric to hit rate, is defined as:

Indicator Dismiss Rate: $|D_s \cap D| / |D|$

[2] This is equivalent to the recall metric in pattern recognition.

The dismiss rate is a the fraction between 0.0 and 1.0 of performance changes the indicator correctly dismissed[3], again, a higher value is better. An indicator that always returns `false` will have a perfect dismiss rate of 1.0. An optimal indicator will have *both* a hit rate and a dismiss rate of 1.0, but such an indicator will be difficult to create for any software project. An effective indicator will incur the smallest amount of false positives (i.e., highest dismiss rates) for covering hits, allowing a predictor to have both a high hit and dismiss rate once indicators are composed.

Figure 6.3a shows the hit rate (solid lines) and dismiss rate (dashed lines) of these indicators for the HotSpot dataset. As expected, as thresholds increase, so do dismiss rates, and hit rate decreases and dismiss rate increases. The figure also shows that each indicator has its own hit and dismiss rate curves, meaning that some indicators are more effective at detecting certain *types* of hits than others.

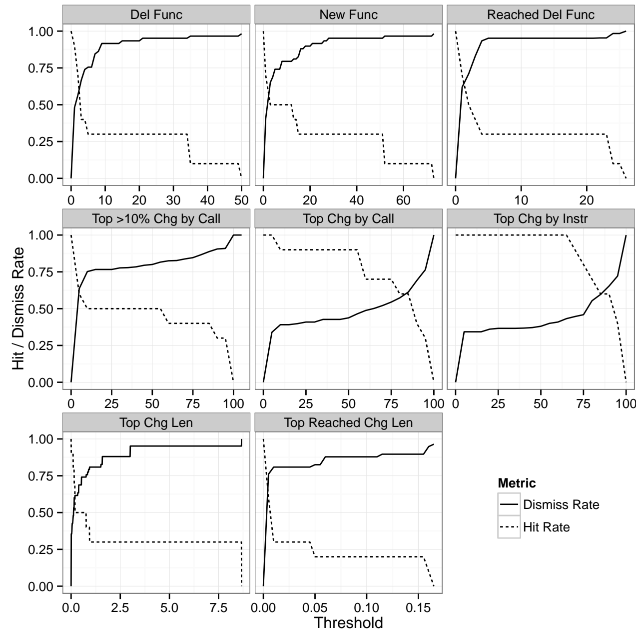
The individual indicator that has the highest dismiss rate at 1.0 hit rate (i.e., the cheapest way to detect all commit pairs with performance changes) for HotSpot is "Top changed function by instruction length" ("Top Chg by Instr" in Figure 6.3a) with its threshold set at 65, and a dismiss rate of 0.57 (i.e., only 43% of commits would be sent for benchmarking by this indicator). "Top reached function length change", a less effective indicator individually, has an immediate drop-off in hits as the threshold increases. It is not without value, however, since at a low enough threshold, the "cost" for its hits – the number of false positives incurred per true positive – is low. If enough indicators like this are combined, they can add up to a predictor that has a high hit rate with a low cost per hit, i.e., a high dismiss rate.

Figure 6.3b shows the hit and dismiss rates of the same signals for Git. The metrics for each indicator are very different from the HotSpot results, which suggests that different software projects will require different predictors. Also of note is the "Reached func. deleted" indicator, which has a hit rate of 0.0 as soon as its threshold is raised from zero to one (thresholds for this indicator can only be integers), making it either `true` for all changes or `false` for all changes, and, therefore, effectively useless for Git. This suggests that not all indicators will be useful for all software projects.

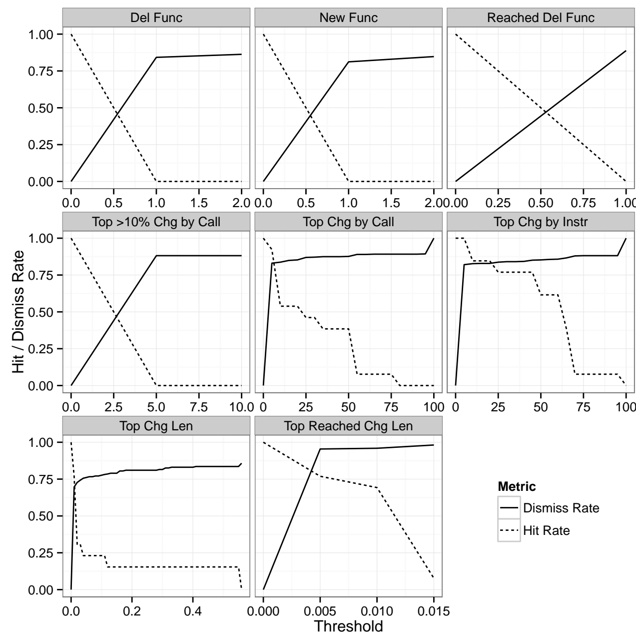
6.4.3 Predictor: Predictor Engine

The job of the predictor engine is to mix indicators, targeting a hit rate of 1.0 while maximizing dismiss rate, so that the minimal performance testing needs to be done while

[3]We chose dismiss rate over precision ($|H_s \cap H| / |H_s|$) due to precision's non-monotonicity as $|H_s|$ varies.



(a) HotSpot



(b) Git

Figure 6.3: Indicator performance at various thresholds for HotSpot and Git.

retaining zero performance bug detection delays. The simplest way to connect indicators is to first fix their thresholds (as we did for their evaluation in isolation) and then perform a logical OR operation of their boolean values. An example of such a predictor is "Top changed function by instruction length > 85 `OR` Top changed functions by call count > 85". We can then apply the same metrics used to evaluate indicators – hit and dismiss rate – to evaluate predictors as a whole.

Mixing indicators with logical ORs can improve dismiss rates of individual indicators because mixed indicators can "divide" the hit set among themselves. Each indicator's threshold can be higher (raising their dismiss rates, and allowing their hit rates to drop below 1.0), as long as the misses incurred by one indicator are covered by another. If the indicators are sufficiently dissimilar, i.e., they cover hits in different orders, the raised thresholds will cause the *aggregate* dismiss rate to be higher than that of any of the indicators individually, while maintaining the hit rate at 1.0. The problem then becomes selecting thresholds for each indicator in a way that maximizes aggregate dismiss rate without lowering the aggregate hit rate.

Algorithm 1 is a greedy heuristic solution to the problem of setting thresholds for a set of n_{ind} indicators given a set H of hits that must be "covered" while minimizing the "cost", or false positives. Given a software project, developers can determine thresholds to use for indicators using this algorithm on a *training set*, i.e., a set of known versions, and then apply the resulting predictor – the logical ORing of those indicators at the thresholds returned by the algorithm – as development continues. The key assumption is that the causes of the performance changes seen in the training set will continue to cause changes in future development, and that new causes will be covered by the current thresholds. In Section 6.5, we show that this assumption generally holds for the projects we investigate.

The algorithm executes in polynomial time ($O(|H|n_{ind} + |H|^2)$), and requires four auxiliary functions: `perfdiff(version a, version b)`, which returns `true` if there is a performance change between the two versions, `maxthresh(hit h, indicator s)`, which returns the maximum threshold for indicator s that still covers hit h, `falsepos(indicator s, threshold t)`, which returns the number of false positives incurred by indicator s at threshold t, and `allhits(indicator s, threshold t)`, which returns the set of hits covered by indicator s at threshold t. The algorithm begins by calculating the minimum number of false positives (or *price*) required to cover each hit (lines 3-14), storing the "cheapest" available indicators and thresholds in the "`min_price`" structures. Then, until no hits are left uncovered, it finds the hit with the highest minimum price (lines 20-26), fixates the associated indicator's threshold to its highest possible value (line 27), and removes all hits covered by that indicator at that threshold from the hit set (line 28). Since a single indicator at a given threshold may cover multiple hits, the hit set often decreases

Data: $H = \{(a_i, a_u) : \text{perfdiff}(a_i, a_u) == true\}$, $S = \{s_j : 1 \leq j \leq n_{ind}\}$
Result: $T = \{(s_k, t_k) : 1 \leq k \leq n_{ind}\}$

```
1  T = ∅;
2  for h ∈ H do
3      min_price[h] = ∞;
4      min_price_ind[h] = null;
5      min_price_thresh[h] = null;
6      for s ∈ S do
7          thresh_for_hs = maxthresh(h, s);
8          price_for_hs = falsepos(s, thresh_for_hs);
9          if price_for_hs < min_price[h] then
10             min_price[h] = price_for_hs;
11             min_price_ind[h] = s;
12             min_price_thresh[h] = thresh_for_hs;
13         end
14     end
15 end
16 while H ≠ ∅ do
17     max_min_price = 0;
18     target_ind = null;
19     target_thresh = null;
20     for h ∈ H do
21         if min_price[h] > max_min_price then
22             max_min_price = min_price[h];
23             target_ind = min_price_ind[h];
24             target_thresh = min_price_thresh[h];
25         end
26     end
27     T = T ∪ {(target_ind, target_thresh)};
28     H = H \ {allhits(target_ind, target_thresh)};
29 end
```

Algorithm 1: Threshold selection for a hit and indicator set.

by more than one hit at a time.

This algorithm covers all hits in the training set (its hit rate will be 1.0), but it is not guaranteed to maximize dismiss rate (it does not arrive at an optimal solution). Since developers will use the resulting predictor for making predictions on a different input set, optimality of the solution regarding the training set is not essential, and an optimal one may indeed be overfit to the training data. The sub-optimal predictors generated by Algorithm 1 will have lower (i.e., sub-optimal) dismiss rates than the absolute optimal predictors would, but conversely may cover more hits in the unknown data than the optimal predictor, due to their thresholds not being fitted as tightly as possible to the hits in the training set. For example, a threshold for the “new functions” indicator may be set at 5 when 10 would suffice for the training set, but when making predictions this difference would cause a performance-changing commit pair with 6 new functions to be a hit, when an optimally fitted predictor would miss it.

This algorithm depends on a non-empty set of hits to operate correctly. If $H = \emptyset$, such as in the glibc data introduced in Section 6.3, the resulting predictor will be trivial: the thresholds for each of its indicators will be at their maximum possible value, such that its dismiss rate equals 1.0 over the training set, and no matter what code change the predictor is used on, it will return `false`. It is essential, therefore, that developers ensure their training sets include as many hits as possible to avoid such overfitting.

6.5 Evaluation

For Perphecy to be useful for developers, they need to be able to trust it to predict the majority of the hits, while saving them time by dismissing the majority of the performance-neutral changes. To evaluate Perphecy's ability to dismiss performance neutral commits given perfect information, we measure the dismiss rate given the full input data. Then, to evaluate its predictive abilities, we train it on a portion of the data, and make predictions on the rest. We also determine if a predictor should be trained on a per-benchmark basis, per software-project basis, or if there exists a one-size-fits-all predictor. To inform developers as to how large their training sets should be and how often they should add new data to it, we investigate the effect of the training set size and “age” on the quality of the predictions. Finally, we investigate the effectiveness of generating artificial training data by synthesizing commits out of arbitrary version pairs in a repository.

Algorithm Effectiveness: The first evaluation we performed is of the threshold selection algorithm, determining how high a dismiss rate it achieves when given the full $|A|$ as

	Full		Per Test		All Tests	
Software	Hit	Dism.	Hit	Dism.	Hit	Dism.
Git	1.0	0.83	0.77	0.84	0.85	0.83
glibc	n/a	1.00	n/a	1.00	n/a	1.00
HotSpot	1.0	0.89	0.40	0.88	0.70	0.47
MongoDB	1.0	0.46	0.74	0.46	1.0	0.46

Table 6.3: Performance of the algorithm in isolation, and K-fold cross validation.

input. The columns under the "Full" heading in Table 6.3 show the average hit and dismiss rates achieved by the algorithm when applied to each benchmark. For Git, HotSpot and MongoDB, the hit rate is 1.0 by construction (since the algorithm covers all hits). The dismiss rate column shows that, for Git and HotSpot, the algorithm is able to dismiss more than 80% of the performance-neutral commits, while the predictor for MongoDB, which relies on performance-counter-based indicators, has a comparatively low dismiss rate of 46%. Since the glibc training set contains no hits, the resulting trivial predictor has a dismiss rate of 1.0.

Predictor Effectiveness: We now measure the effectiveness of the strategy in actually *predicting* performance changes by using k-fold cross-validation [67]. The method of k-fold cross-validation divides a data set into k randomly constructed subsets, uses the union of $k-1$ of those subsets as training data, then makes predictions for the unused k^{th} subset, or "unknown", data set. We calculate the hit rate and dismiss rate for those predictions in the same way we did for indicators in Section 6.4.2, and repeat the process using each of the k subsets as "unknown" data. This validation strategy demonstrates the validity of the approach because, if satisfactory predictors are generated no matter what the training set, then the strategy will most likely generalize to other unknown data.

The hit and dismiss rate columns under the "Per Test" heading in Table 6.3 show the results of k-fold cross validation, where $k = 10$ (i.e., train on 90% of data, predict remaining 10%), where one predictor is generated for each individual benchmark, i.e., each predictor's H contains only the hits for the benchmark it is designed to cover. The results show that the predictors are not perfect – hit rates range from 0.77 to 0.4 – but they do still dismiss between 46 and 80% of performance-neutral commits.

The hit and dismiss rate columns under the "All Tests" heading in Table 6.3 show predictors generated from all benchmarks at once, i.e., hits from all benchmarks are included in H, and a single predictor is created for each software project. These predictors have uni-

versally lower dismiss rates than their per-benchmark counterparts, yet their hit rates are improved. Note that these predictors still make predictions on a per-benchmark basis, and are not restricted to simple "run every benchmark" or "run no benchmark" predictions. For HotSpot, the aggregated predictor correctly dismisses 47% of performance-neutral commits, while covering 70% of significant commits, a marked improvement from the 40% achieved by per-benchmark predictors. The consequence of creating one predictor for all benchmarks, as opposed to one predictor per benchmark, is that the single predictor must be less aggressive (indicator thresholds must be lower) to cover the aggregate set of hits. While this may also lower dismiss rates, we suggest that developers use this strategy to train predictors for real-world use due to the improved hit rates.

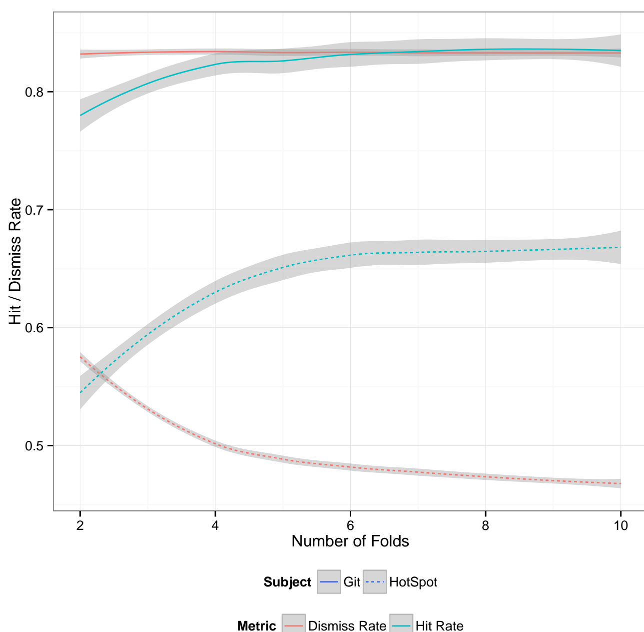


Figure 6.4: Predictor hit and dismiss rates by size of training dataset.

Training Set Size: To determine the size of training set needed to generate a good predictor, we use k-fold cross-validation once again. Since the size of the training set grows as the value of k grows, we can investigate the effect of the training set size on the quality of the predictors. Figure 6.4 shows smoothed curves for the hit and dismiss rates for each project by k, where $2 \leq k \leq 10$. When $k = 2$, the training sets are 50% of the data, and predictions are made for the other 50%, and when $k = 10$, the training sets are 90% of

the data and predictions are made for the remaining 10%. The figure shows that, as the training set grows, there is a marked improvement on the hit rate, accompanied by a minor decrease in the dismiss rate. In the case of Git, the improvement to the hit rate levels off at $k = 8$, or approximately 175 commits, suggesting that number as ideal for a training set size.

Cross-Project Prediction: To evaluate if a single predictor can be used effectively for all software projects, we test the transferability of predictors between the projects (MongoDB is excluded due to its indicators being incompatible with the other projects, and glibc is excluded due to its data set not containing any hits). The hit rate and dismiss of the predictor trained on Git data has a hit rate of 1.0 and a dismiss rate of 0.34 on HotSpot data, a conservative predictor, while the reciprocal predictor, trained on HotSpot, has a 0.0 hit rate and a dismiss rate of 0.88, a very aggressive predictor that covers none of the hits. These results show that using a predictor trained on one software project to make predictions about another is ineffective at best.

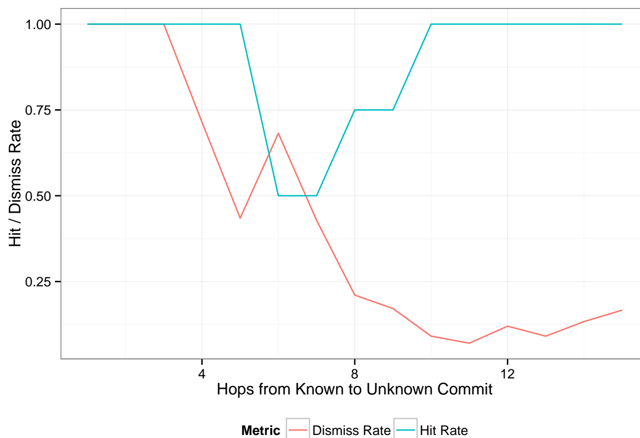


Figure 6.5: Predictor hit and dismiss rates by hops between known and unknown commit, for Git.

Distance to Last Known Commit: As development continues, if the predictor indicates that that none of the incoming code changes will incur a performance change, developers may make multiple subsequent commits without running a benchmark. In such a scenario, the last commit with a known performance change has a continuously growing

distance from the latest commit with unknown performance. To determine the effect of this distance on predictor performance, Figure 6.5 shows the hit and dismiss rates of a predictor trained on the 25% oldest (by time stamp) commits of the Git dataset, making a series of predictions (a_i, a_u, w_j) where a_i is the latest commit in the training set, to each a_u in the 75% of commits not used in the training step. The figure shows that, for this data set, there is a correlation between hop distance and dismiss rate. Between one and three hops, *all* commits are correctly dismissed (hit and dismiss rates equal 1.0), but after that the dismiss rate decreases nearly monotonically with hops until reaching 0.16 at 15 hops. This happens because the longer the distance between a_i and a_u, the larger the difference between the resulting binaries (all changes in the intervening commits are accumulated), and the more indicators will have their thresholds surpassed. No such effect is observed for hit rate, which varies between 0.5 and 1.0 with no pattern relating to hop distance. This result suggests that there may be a natural effect limiting the length of a string of commits for which the predictor will return `false`. This effect is desirable for developers, because it bounds the amount of hops a missed performance change will go unnoticed. In the scenario shown in Figure 6.5, the predictor starts returning `true` as soon as four hops from a_i, which would have "reset" the latest known commit, updating the developers' knowledge about their application's performance.

Software	Hit	Dism.
Git	1.00	0.08
HotSpot	0.99	0.03
glibc	1.00	0.29

Table 6.4: K-fold cross validation for synthetic training sets.

Synthetic Training Sets: Since there is no restriction on the relationship between versions a_i and a_u involved in a prediction, we can generate "synthetic commits" to achieve a larger training set. We test this by exhaustively combining commits in A, creating a delta from one to the other: from $|A|$ known versions, we create $\binom{|A|}{2}$ version pairs, greatly expanding the inputs to the predictor generation algorithm. Table 6.4 presents the results for k-fold cross validation ($k = 10$) on this exhaustive set of commit pairs for Git, HotSpot and glibc. The data shows that the predictors trained on such synthetic training inputs are conservative, with perfect or nearly perfect hit rates, and dismiss rates of less than 0.3. Since these artificial training sets include even the most distant commit pairs (in all metrics of distance: hops, commit date, *and* code delta size), the frequency and nature of hits they contain is different from what occurs in regular development. As the hit set

grows, the threshold selection algorithm can only make the predictor more conservative, which suggests developers should limit the size of their training set once their dismiss rate levels off (as in Figure 6.4) or starts decreasing. Interestingly, the artificially generated training set for glibc included 398 hits in 14259 prediction tuples, allowing a non-trivial predictor to be derived. The generation of synthetic commits can be, therefore, a viable option for augmenting training sets with an insufficient amount of hits, such as glibc's.

This section showed that, for Git, Perphecy can predict as much as 85% of the performance-changing commits while avoiding executing 83% of benchmarks. In the case of HotSpot, which complicates analysis by generating dynamic binary code during runtime, the predictor detects 70% of performance-changing commits, and dismisses nearly half of performance-neutral code changes. These predictors are based on very lightweight processes: language and architecture independent static analysis of binary code, and lightweight profiling of benchmarking runs, that collects nothing but (1) the number of times each function is called, and (2) how many instructions are executed in each function. In the case of Git, for example, all static processing operations needed to make a prediction (which are what developers would carry out at every commit) executed in under $350ms$. The predictor for MongoDB, which used a different set of indicators based on hardware performance counters, detected 100% of performance-affecting changes, while dismissing 46% of performance neutral changes, suggesting that these indicators are effective in conservative predictors.

6.6 Discussion

While we designed and evaluated Perphecy, we noted some observations, possible pitfalls, and lessons learned. This section discusses those points.

The Data Dependency Problem Imagine a software project with a global boolean variable called `do_sanity_check`, set to `false` in development, that determines if an expensive sanity checking procedure is to execute or not. If a commit changes that value to `true` (or makes an auxiliary set-up function set it to `true`), only the data section of the resulting binary (or a short-running function that is only called once) will change. None of the indicators described in Table 6.2 are well suited to detect performance regressions caused by this type of data dependency, which, while not common in our case studies, could be frequent in other software projects. Indicators designed to detect this type of change would involve measuring the differences in the data sections of the binaries, and possibly some more complex form of static analysis.

Applying Perphecy Retroactively If developers detect a performance regression in the very last commit before deployment, they will need to bisect the development tree to find the commit that introduced the regression. Since Perphecy does not require a_u to be a descendant of a_i in a prediction, developers can use that last commit, for which performance information is available, as a_i, and walk the commit graph backwards using each previous commit as the unknown commit a_u, predicting which commit introduced the change *retroactively*. This can be useful if executing any performance tests at all during development is infeasible or otherwise undesirable.

Perphecy on DataMill Given the need for months-long, exclusive dedication of the platforms involved in the experiments presented in this chapter, we chose to use machines outside of DataMill for these measurements. Once a developer has a trained predictor for their project, however, performance testing should be done on an infrastructure such as DataMill to ensure maximum platform heterogeneity.

Each Test is Unique Commits that affect performance of one benchmark are not guaranteed to affect all others. In fact, we observed a strong separation between benchmarks, i.e., the majority of commits affected the performance of exactly zero, one, or two benchmarks in our suites. This suggests that predicting that a commit causes a performance change in the *general case* (i.e., all benchmarks) will most often be incorrect, and predictions should be made on a per-benchmark basis. There are two corollaries to this observation: one is that making predictions on the basis of static data alone will miss the differences between benchmarks (i.e., code reachability, function call counts), and the other is that the components of a benchmark suite should have as little overlap as possible in the code they exercise, so they can be effectively selected depending on what code is changed.

Chapter 7

Predicting Application Performance on Unavailable Platforms

Chapter 6 presented a solution for performance-change prediction in cases where the target platform is available, yet the volume of performance evaluation is impractical or impossible. We now address cases where the target platforms are inaccessible, such as developers comparing various candidate target platforms, which they cannot acquire due to cost or consumer availability.

This chapter presents performance fingerprinting, a statistical performance modeling approach that allows developers to directly predict their performance metrics of interest on platforms that they do not have direct access to. Performance fingerprinting leverages DataMill to detect correlations between application and benchmark performance over a wide range of heterogeneous machines, then constructs predictive statistical models based on these correlations.

7.1 Problem Definition

Benchmark suites generally attempt to contain the most diverse set of individual applications possible, in order to represent as many production applications as possible. For example, the SPEC CPU 2006 [39], DaCapo [5], and PARSEC [4] suites all claim to represent a wide range of possible applications by containing several individual benchmarks with different performance-related properties. While this is a valid approach to creating a benchmark suite that represents *as many applications* as possible at once, it may

Benchmark	Area
Astar	Path-finding
Bwaves	Fluid Dynamics
Bzip2	Data Compression
CactusADM	Physics
Calculix	Structural Mechanics
GCC	Compiler
GemsFDTD	Electromagnetics
GobMK	A.I.
Gromacs	Molecular Dynamics
H264	Video Comp.
hmmer	Gene-Sequence Searching
Lbm	Fluid Dynamics
Leslie3d	Fluid Dynamics
Libquantum	Physics

(a)

Benchmark	Area
mcf	Combinatorial Optimization
Milc	Physics
namd	Molecular Dynamics
Omnetpp	Simulation
Perlbench	Perl Programming Language
Povray	Ray-tracing
sjeng	Artificial Intelligence
Soplex	Optimization
sphinx3	Speech Recognition
Tonto	Chemistry
Wrf	Weather
xalancbmk	XML to HTML Translation
Zeus	Physics

(b)

Table 7.1: Benchmarks.

be counter-productive if the user has a *specific application* whose performance they are interested in predicting.

To illustrate this problem, Figure 7.1 shows the mean performance of each SPEC CPU 2006 benchmark listed in Table 7.1, executed five times, on each machine listed on Table 7.2. The y-axis shows the execution time in seconds (as reported by SPEC's run scripts), and the x-axis shows each machine, positioned according to the geometric mean of its execution time of all of the benchmarks. The x-axis is organized this way to order machines by overall SPEC performance, which is the metric a developer would use to compare possible target platforms. The black line represents the geometric mean performance of SPEC over all machines, drawn here as a perfect log curve due to the special ordering of the x-axis and the log scale of the y-axis. Intuitively, each machine occupies one "column" on the plot, e.g., all data from machine 80 is located at $x = 358.96$. For legibility, the figure highlights only three of the benchmarks.

To demonstrate the danger of using the mean performance of a benchmark set as indicator of performance, imagine cactusADM is the application whose performance we are interested in predicting. Figure 7.1 shows that it follows the general trend of the SPEC geometric mean, which agrees with intuition. What is of interest, however, are the individual cases where it *does not* track the overall mean from machine to machine. CactusADM's

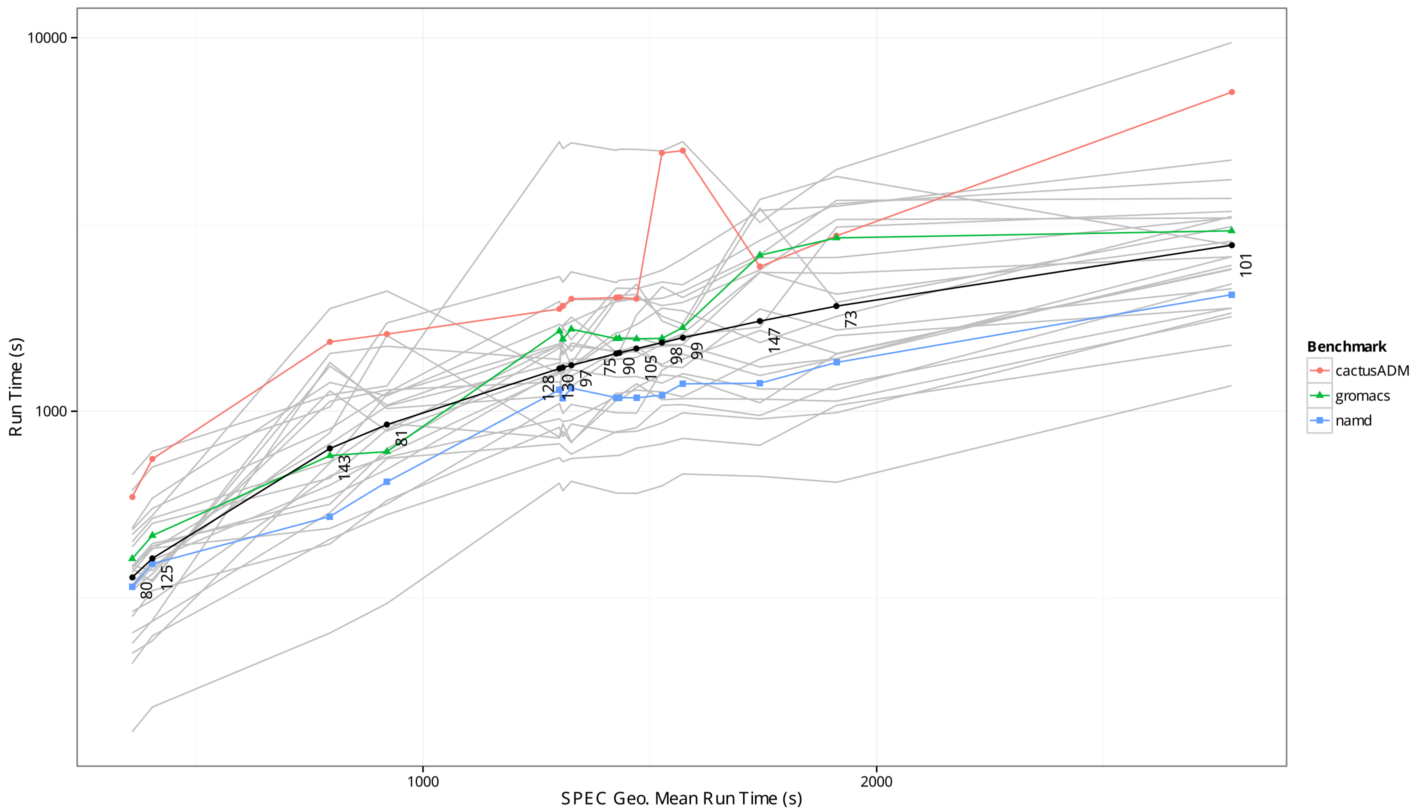


Figure 7.1: Individual SPEC benchmark performance by SPEC geometric mean performance.

ID	Processor	Clock (GHz)	RAM (MB)	SPEC Mean (s)
101	Pentium 4	1.60	900	2782.29
73	VIA Nano X2	1.60	1700	1911.51
147	Pentium M	1.70	881	1742.53
99	Pentium 4	2.80	900	1572.36
98	Pentium 4	3.00	900	1526.78
105	Pentium 4	3.20	900	1470.72
90	Pentium 4	3.20	1000	1432.64
75	Pentium 4	3.20	900	1425.74
97	Pentium 4	3.00	900	1326.82
130	Pentium 4	3.20	900	1308.20
128	Pentium D	3.00	2000	1300.57
81	Opteron 8378	2.40	3200	920.57
143	Xeon 5160	3.00	2500	794.92
125	Core i5-2500	3.30	8000	403.64
80	Core i7-2600K	3.40	8000	358.96

Table 7.2: Machines.

performance line correlates with (i.e., has an approximately linear relationship to) the geometric mean from machine 80 to machine 105, but from that machine to the next (98), its execution time has a severe increase, which is maintained through to machine 99, and then decreases again for machine 147, contrary to what one would expect from the suite's geometric mean. In this case, and all others cases where the derivative of the performance line is negative from machine to machine, using the overall performance of SPEC would cause the developer to make an incorrect decision: in this example, he would pick machine 99 over machine 147 and experience surprisingly inferior performance.

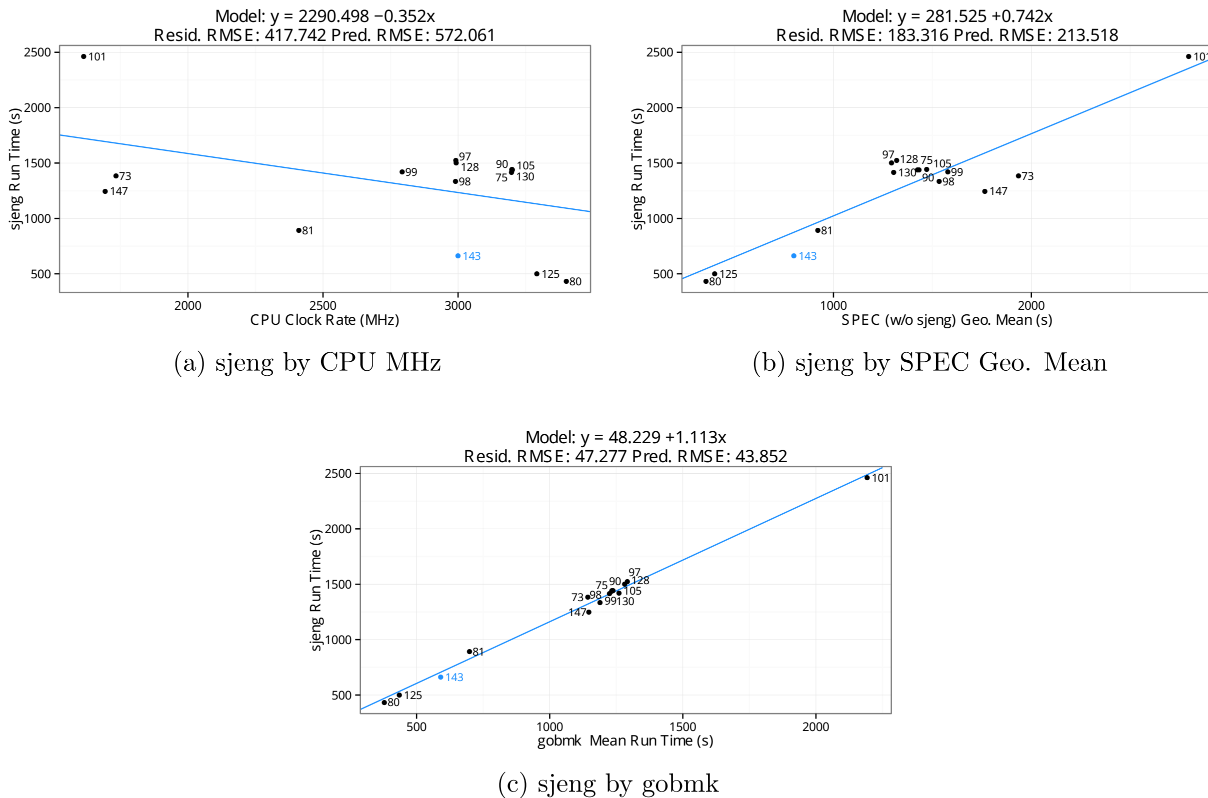
How, then, can developers use a benchmark suite to predict their application's performance? In the case of CactusADM, the application's performance is not sufficiently correlated to the mean performance of the suite, or to any individual benchmark in the suite. In cases where a one-to-one correlation does not exist, a predictive solution must involve data from multiple benchmarks in the suite. These cases appear to be the norm; in Figure 7.1, only gromacs and namd rank machines by performance in exactly the same order, albeit with varying relative performance ratios between machines.

The problem, then, is defined as follows: given performance results of an application and a benchmark suite on a set of machines, predict the performance of the application on a machine for which only the benchmark suite's performance results are available, or

safely fail when accurate prediction is impossible.

7.2 Using Benchmark Performance as Statistical Predictors

In this section we propose *performance fingerprinting*, a statistical performance prediction approach that models application performance using benchmark performance. Performance fingerprinting uses benchmark scores as predictors in linear models, completely abstracting away hardware characteristics (e.g., architecture, CPU clock rate, memory bandwidth, etc.), and allowing precise performance predictions on platforms for which only standard benchmark performance results are available.



(a) sjeng by CPU MHz

(b) sjeng by SPEC Geo. Mean

(c) sjeng by gobmk

Figure 7.2: Models of sjeng execution time.

To exemplify the approach, Figure 7.2 shows three single-factor linear regression models of sjeng execution time, trained over the machines in Table 7.2, whose IDs appear in black, predicting the performance for machine 143, whose ID appears in light blue. Both the target application – sjeng – and the target machine – 143 – were selected at random. In each of these plots, the y-axis shows execution time in seconds (the y in our model), while the x-axis shows the predictor variable for the model (the x in our model), and dots denote the mean performance for the sjeng in each machine, whose number is denoted next to the dot. A blue line shows the model curve, which can be interpreted as the "best guess" of the y-axis value each model would make for each x-axis value. Finally, the resulting model's equation and fit metrics appear above the plot itself. These three models have the same goal: map a characteristic of the machines to the performance of a particular application, sjeng. To use any of these models for prediction, developers would "plug in" the relevant characteristic of the machine they are targeting as the x value, coming up with an estimate of the execution time of their application as the y value. To directly compare these models, we will use the root mean square error (RMSE), a measure of the distance between model estimates and the observed data. The RMSE is calculated as follows:

$$MSE = \frac{SSE}{n} \tag{7.1a}$$

$$RMSE = \sqrt{MSE} \tag{7.1b}$$

where n is the number of observations, the mean square error (MSE) is the error sum of squares (Equation 1.3b) averaged over all individual observations, and finally, the RMSE is the MSE expressed in the metric's own units. The *residual* RMSE is calculated over the data used to fit the model (the black dots in Figure 7.2), and the *prediction* RMSE is calculated over the "unknown" data (the blue dots). The lower the residual RMSE, the better the fit of the model to the data. The lower the prediction RMSE, the better the predictions made by the model. Note that the prediction RMSE can only be calculated a posteriori, once the model has been fitted and performance on the target platforms is known.

Figure 7.2a shows a (purposefully naïve) model of sjeng execution time by CPU clock rate in MHz. Despite the correlation between a processor's clock rate's and processing performance (as demonstrated by the blue line generally following the dot pattern) it is not the only relevant factor; details such as cache size and memory hierarchy bandwidth are also major components of performance. Since this model ignores many relevant factors, it does not have a good *fit* to the dataset, with a residual RMSE of $417.742s$. Figure 7.2b

shows a model of sjeng based on the geometric mean of the execution time of all SPEC (excluding sjeng itself). The figure shows that modeling the performance of an application on the full benchmark suite is a significant improvement over the CPU MHz model, with a 56% reduction in the prediction RMSE.

Finally, Figure 7.2c shows a model created with the best single benchmark predictor, gobmk, exemplifying the goal of performance fingerprinting: finding the subset of the benchmark suite that *best* represents the application. This model, with a predictive RMSE of 43.852, shows an improvement of 74% over the model based on the SPEC geometric mean, demonstrating that while the suite represents average behavior, individual benchmarks can drastically improve prediction performance of individual applications.

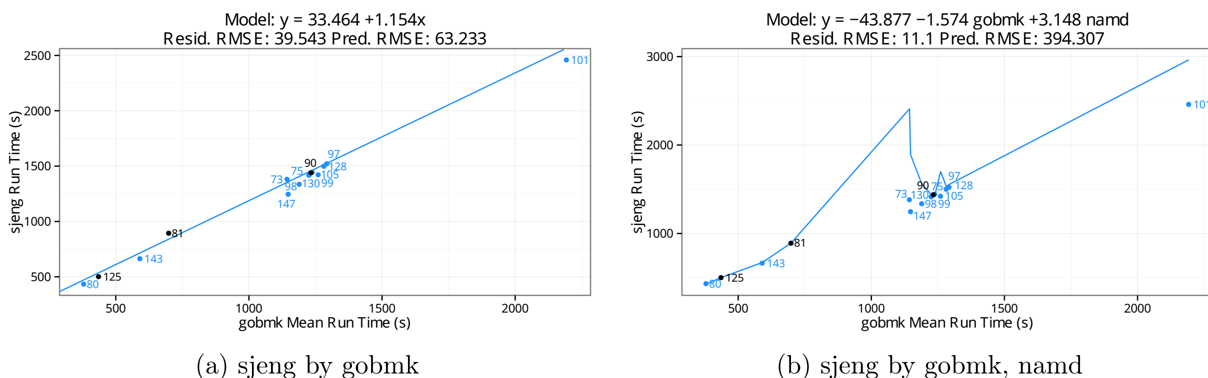
It is clear that performance fingerprinting — statistical modeling of application performance using components of a benchmark suite — *can* yield very accurate predictive models, a large number of training machines are available and the application has representative predictors in the benchmark. Next, we address potential pitfalls concerning the number and relative performance heterogeneity of machines used in fingerprinting, and, in Section 7.4, we introduce techniques to select which benchmarks to use as predictors for an application.

7.3 Potential Pitfalls

Figure 7.3: sjeng by gobmk, small data set.

The quality of the predictive model depends on the number of machines used in its fitting, and the relation between those machines and the target platforms. Figure 7.3 shows a different model of sjeng by gobmk fitted only on data from machines 147, 98, and 97,

chosen for having relatively similar sjeng execution times, and therefore poorly representing the wide spectrum of machines in Table 7.2. Although gobmk is fundamentally a good predictor of sjeng performance – as evidenced by Figure 7.2c – attempting to fit the exact same model on those three machines yields a prediction RMSE of 394.705, much worse than the 213.518 and 43.852 from the models from Figures 7.2c and 7.2b, and only somewhat better than 572.061, from using CPU MHz alone as a predictor. This makes it clear that the *number* and *heterogeneity* of the machines used during model-fitting is vital to produce good predictive models, making DataMill ideal for collecting training data.



(a) sjeng by gobmk

(b) sjeng by gobmk, namd

Figure 7.4: Example of overfitting.

Even when the machines available for fingerprinting are representative of the target machines, selecting *which* benchmarks to use as predictors is a non-trivial problem. The 27 individual benchmarks in SPEC alone allow the creation of $27^2 = 729$ different predictor combinations, only one of which developers will use for fitting and, subsequently, performance prediction. Simply picking the model with the least residual RMSE between all alternatives may lead to *overfitting*, a problem demonstrated in Figure 7.4. Figure 7.4a shows a model of sjeng by gobmk fitted on data from machines 125, 81, and 90, yielding a well-fitting model with comparable residual and prediction RMSEs to that of Figure 7.2c. This model has a much better prediction RMSE than the one in Figure 7.3 due to the performance "spacing" between machines: its prediction targets are always in the vicinity of one of the machines used in fitting the model. Figure 7.4b shows a model of sjeng by gobmk *and* namd, fitted on the *same machines*, which has a significantly lower residual RMSE, yet a significantly higher prediction RMSE: the blue line significantly deviates from the blue dots. This is a traditional case of overfitting, where the increase in model complexity (i.e., the added predictor) allows for the fitting process to further minimize *residual* RMSE, at the expense of predictive power. Indeed, a property of least-squares

model fitting is that the addition of predictors will *always* reduce the residual RMSE [67], yet, as shown, this does not necessarily lead to better *prediction* RMSE.

Using benchmark performance as a predictor for application performance in linear statistical models is a promising approach, but two problems remain: choosing the best model for prediction purposes, and selecting machines that best represent the target.

7.4 Model Selection

In this section, we discuss three strategies for selecting a predictive model given a set of machines, while avoiding the overfitting problem: using the geometric mean as the sole predictor (i.e., the baseline approach), using correlation clustering techniques to subset the benchmark suite, and regression subsetting.

Suite Geometric Mean: Using the mean performance of a benchmark suite as the sole predictor (as in Figure 7.2b) avoids overfitting by limiting the number the predictors to one. It does, as shown in Figure 7.2, lead to sub-optimal models, since it is possible that a subset of the same benchmark suite better represents the application. If the prediction RMSE of models fitted with this strategy is acceptable, however, it may be a sufficiently accurate option.

Cluster-Based Subsetting: If two or more predictors have a strong linear correlation to each other, all but one can be dropped from the model with little loss of explanatory power [67]. If such correlations exist in the benchmark set (i.e., the performance of two or more benchmarks correlate with each other as the machine varies), dropping correlated predictors can aid in avoiding overfitting. Such correlations are expected – indeed, this chapter's approach depends on the existence of correlations between the performance of applications and benchmarks – and, in the case of SPEC CPU 2006, have been reported by Phansalkar et al. [77].

Figure 7.5 shows a complete-linkage clustering dendrogram of the benchmarks in Table 7.1, grouped by correlation in performance, which can guide subsetting of the benchmark set. Clusters of linearly correlated performance are identified by "cutting" the tree at any height in the diagram, and then one representative of each cluster will be used as a predictor. The further down the tree the cut is made, the higher the number of predictors will be, increasing the risk of overfitting.

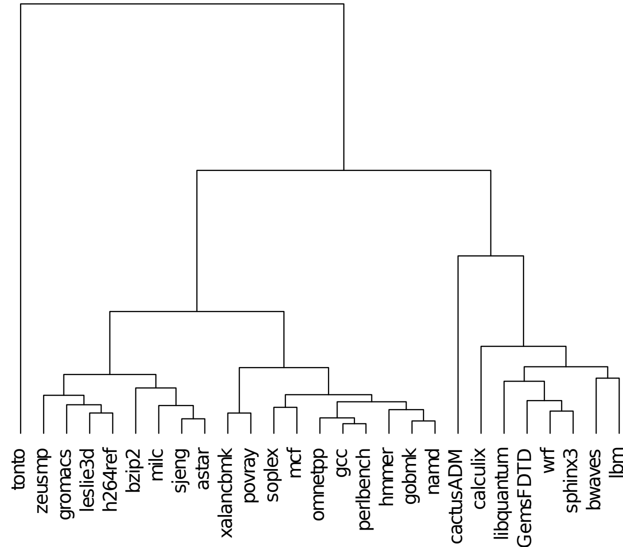


Figure 7.5: Clustering of SPEC CPU 2006 benchmark performance.

Regression Subsetting: This statistical method attempts to subset a large set of *candidate predictors* by either starting with the full set of predictors and removing them one at a time as long as a model's quality does not degrade (backward selection), starting with an empty model and adding predictors as long as the model's quality improves (forward selection), or exhaustively searching the model space. Despite the combinatorial explosion on the number of predictors mentioned in Section 7.2, exhaustive selection is possible in modern computers using branch-and-bound algorithms.

Since developers will apply the resulting model to make predictions, model selection must avoid overfitting by favoring small predictor sets. In regression subsetting, this entails replacing residual RMSE as a measure of model quality with an alternative metric that does not automatically increase with the number of predictors; otherwise, the model with the maximum number of predictors would always be selected. There are multiple such metrics in statistics, two of which we utilize here due to effectiveness and simplicity: R^2_{adj} – adjusted R^2 – and Mallow's C_p.

R^2_{adj} is a modified version of the regular R^2 (Equation 1.4a), ranging from 0.0 to 1.0, calculated by dividing the unexplained variability by the total variability, while taking into

account the number of predictors in the model:

$$R^2_{adj} = 1 - \frac{SSE/n-1}{SST/n-p-1} \tag{7.2a}$$

where SSE is the error sum of squares (Equation 1.3b), SST is the total sum of squares (Equation 1.3a), n is the number of observations, and p is the number of predictors in the model. R^2_{adj} takes into account the number of predictor variables, and its value will only increase with the addition of a predictor if the increase in explained variability is higher than what is expected by chance [67].

Mallow's C_p is an estimator of what the average mean squared prediction error will be, and is based on a model's residual sum of squares, the variability in the data, and the number of predictors:

$$C_p = \frac{SSE}{MSE_f} - n + 2p \tag{7.3a}$$

where MSE_f is the mean square error (Equation 7.1a) for the model with *all* predictors, n is the number of observations, and p is the number of predictors. As with R^2_{adj}, it does not automatically improve with the addition of predictors.

7.5 Evaluation

To assess the quality of the model selection strategies described in Section 7.4, we apply them to the data set in Figure 7.1. One by one, we select a benchmark from Table 7.1 to represent the target "application" by first removing it from the suite, and then creating predictive models for the machines in Table 7.2. To ensure unbiased machine selection, we exhaustively explore all machine combinations for a given training set size (e.g., three machines out of 15 in the training set generates $\binom{15}{3} = 36855$ models). We report each model's RMSE in percent of the mean execution time, similar to the normalization to the mean in Chapter 5, to make models for benchmarks with widely varying absolute execution times comparable with each other.

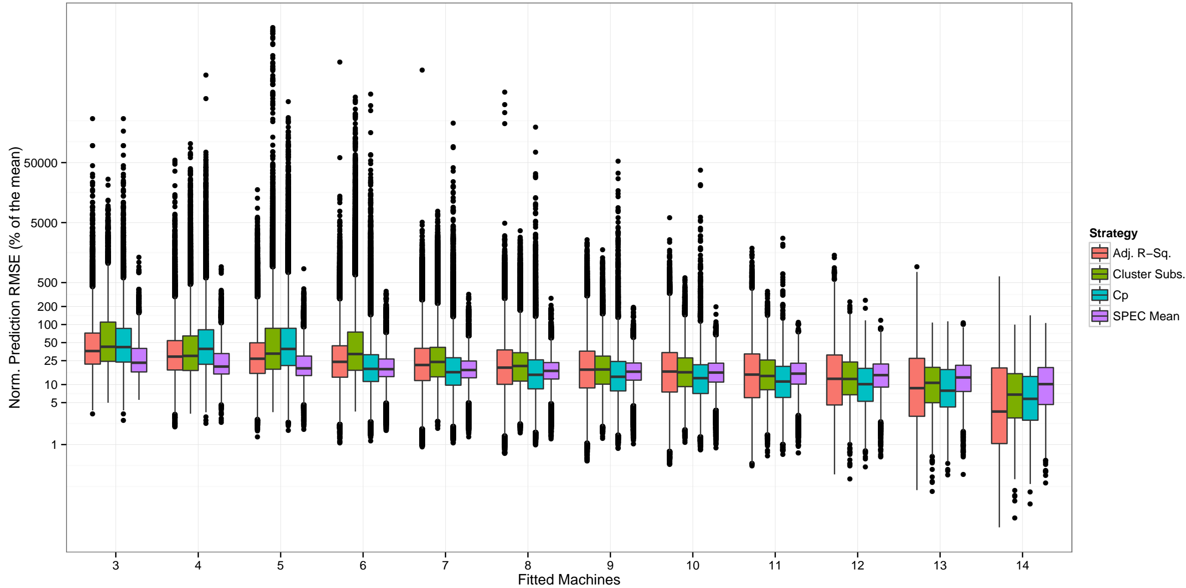


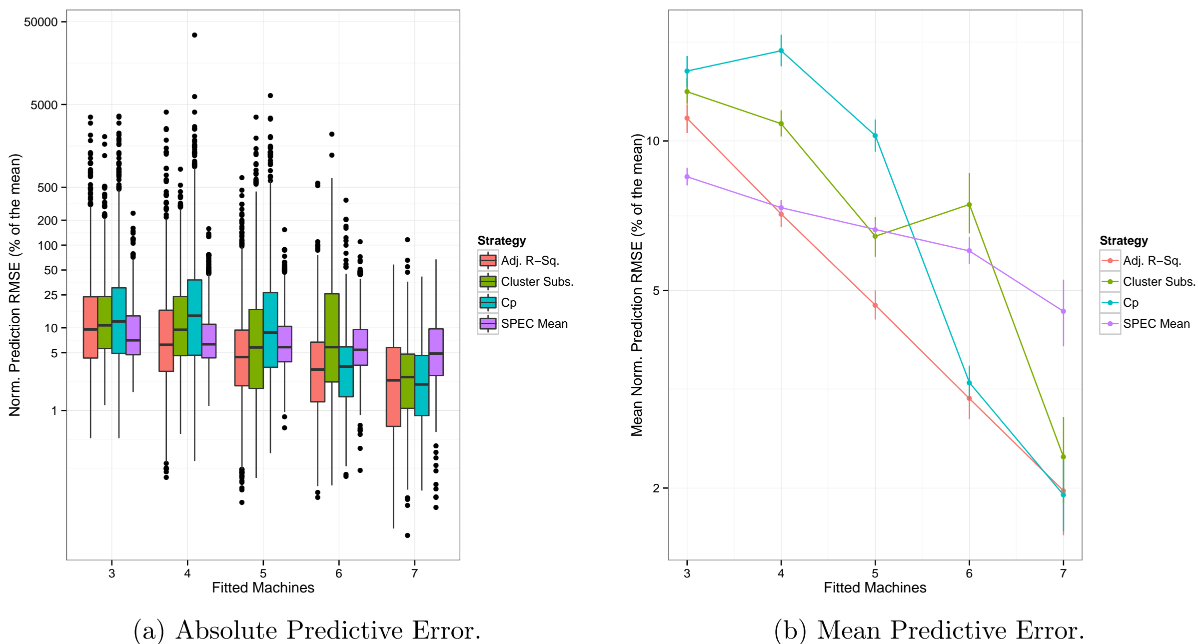
Figure 7.6: Predictive model error by training set size.

Sensitivity to Machine Number: First we investigate the relationship between the quality of the models and the number of machines used in fingerprinting. Figure 7.6 shows all models' normalized RMSE in boxplots on the logarithmic y-axis, by the number of machines used in fitting the model on the x-axis. Each strategy is identified by a different color. The models generated by the regression subsetting strategies had no restriction on how many predictors they could use. For the cluster subsetting strategy, five clusters were represented, by tonto, lbm, namd, cactusADM, and astar. Each model's fit quality was not measured before using them for predictions. We address limiting the number of predictors and checking models for error later in this section.

There is a clear decrease in error as the number of fitted machines increases, which is expected for two reasons: the model fitting strategies have improved performance when given more information, and the likelihood of a training machine being similar to a predicted machine is higher, avoiding the kind of error exemplified in Figure 7.3. It is not until the training set contains 6 machines that the majority of models is below the 25% error mark for all strategies, as demonstrated by the median line on the boxplots. Only at 14 machines does the upper quartile fall below that same 25% mark. The worst case errors are worrying even at 14 machines, reaching nearly 100% of the observed mean for all strategies. This indicates that creating a single model to predict performance across

machines of such disparate configurations, and ignoring all internal fit indicators (e.g., the R^2_{adj} of each model) is not a safe strategy for real-world use.

Also of note is the performance of the Suite Geometric Mean strategy, which performed better than at least one other strategy on both median and worst-case error until 10 machines were in the training set. This suggests that its "natural" avoidance of overfitting makes it the best candidate for very small training sets (although its worst-case performance is still unacceptable for most cases), although it becomes significantly worse on average than all other strategies as soon as seven machines are available.



(a) Absolute Predictive Error.

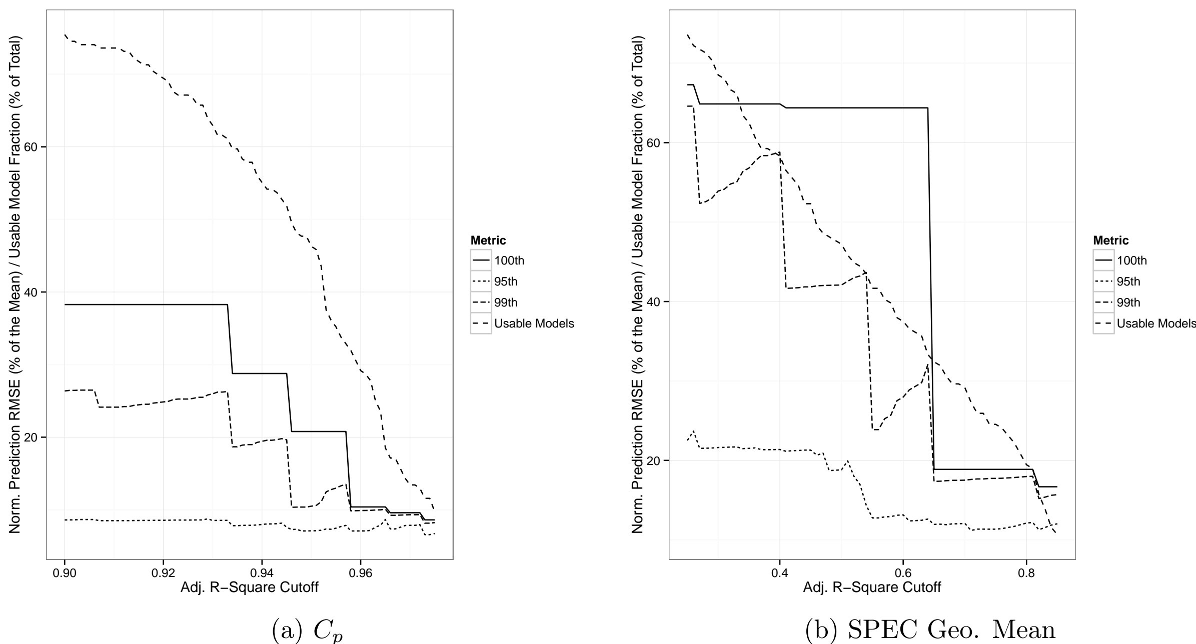
(b) Mean Predictive Error.

Figure 7.7: Model performance, mid-range machines only.

Machine Selection: We now investigate the effect of constraining training machines to those similar to the target machine. Figure 7.7a shows normalized error by training set size, with a training set limited to the following mid-range machines: 128, 90, 75, 98, 105, 130, 99, 97, with mean SPEC performance ranging from 1300.57s to 1572.35s. These machines were chosen due to their similar mean SPEC performance, and, since they belong to the same or subsequent generations of the x86 family, share architectural features. Limiting targets to machines similar to the training set significantly improves both the median and

worst-case errors. For the C_p strategy with 7 machines, the 75th percentile of error is now 4.64%, the 95th percentile is 9.89%, and the absolutely worst case is 41.69%.

Figure 7.7b shows the mean normalized prediction RMSE by training set size, with whiskers extending to the 95% confidence interval around the mean. At seven training machines, the mean error for all strategies is below 5%. These results show that training models on machines that are similar to the target has a large impact on both the average and worst cases.



(a) C_p

(b) SPEC Geo. Mean

Figure 7.8: Model performance, mid-range machines only.

Discarding Ill-Fitting Models Previous analysis considered all models, regardless of their fit. Developers may wish to discard ill-fitting models – those with high residual error, an indication that the application is not well represented by the benchmark set – to avoid making poor predictions. To evaluate this approach, we discard 7-worker models from Figure 7.7a with an R^2_{adj} below a set of thresholds, and measure the 100th (worst case), 99th, and 95th percentiles of error for the remaining models. Figure 7.8 shows these different percentiles and the percentage of remaining models (i.e., approximately the odds that a model can found for a given application), shown on the y-axis, with a range of

R^2_{adj} thresholds, shown in the x-axis, for the C_p and Suite Geometric Mean model selection strategies. Figure 7.8a shows that applying a threshold is an effective way to reduce the worst case error, but the trade-off is that a number of models with low predictive error are also discarded. At a threshold (x-axis value) of 0.958, the 100% percentile, or absolute worst case error, is reduced to 10.39%, but only 31.94% of the models remain. The 95th percentile of error hovers around the 8.5% mark independent of the threshold.

Figure 7.8b shows that applying these thresholds to the Suite Geometric Mean strategy is not as effective; at a threshold of 0.65, 32% of the models remain, but worst case error is still a relatively high 18.87%. This is because this strategy is limited to a single predictor, and even its best models have a significantly lower R^2_{adj} than the other strategies due to their "coarser" nature. A similar effect was observed for the cluster subsetting strategy. The worst-case error of the R^2_{adj} strategy cannot benefit from a R^2_{adj} cutoff threshold, as all models generated by it have maximum or near-maximum R^2_{adj}.

Sensitivity to Number of Predictors: To further minimize the chance of overfitting, developers can artificially limit the number of predictors available to the model selection strategies. For the cluster subsetting strategy, this entails "cutting" the dendrogram at height r, where r is the number of desired predictors, and picking one benchmark to represent each cluster. For the C_p and R^2_{adj} strategies, this entails limiting the exhaustively-explored model space to include only models with r predictors or less. The Suite Geometric Mean strategy cannot benefit from this, as it already uses only a single predictor.

Figure 7.9 shows the normalized error for models created by all strategies on the y-axis by the maximum number of predictors in the x-axis. Each subplot in the figure represents different training set sizes, ranging from three to seven machines. Again, only the mid-range machines first shown in Figure 7.7a are used here, and the residual error of the models is not checked before predictions are made. Between one and six workers in the training set, the worst and median cases are generally improved by limiting the number of predictors. A notable exception is the cluster subsetting strategy with six training workers, whose median performance improves as predictors are added from one to four, and then degrades with the fifth. When seven training machines are used, limiting the predictor count has a less pronounced effect on worst-case error, and actually has a negative effect in various cases. This shows that imposing a hard limit on predictor count is effective in reducing median and worst-case error for smaller training sets, but as more information is available to the model selection strategies, there is less risk of overfitting and more complex models are more successful.

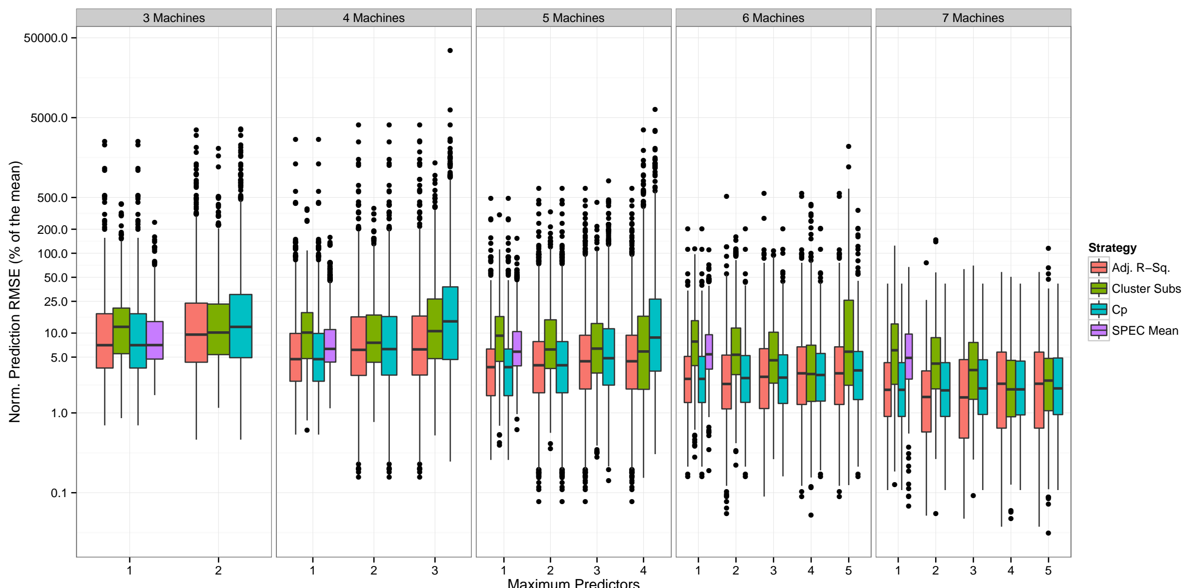


Figure 7.9: Predictive model error by predictor number limit.

Insight Into Worst Cases: There are two main causes for bad predictive models: either the application is not well represented by the benchmark set, or the target machine is not well represented by the training machines. To investigate which of these causes of prediction error is more prevalent, we analyze the worst models, between those fitted by C_p, with seven training workers, in the mid-range machine data set (i.e., the rightmost C_p data from Figure 7.7a). We chose this data-set for having the lowest worst-case error before the application of R^2_{adj} thresholds, with a view to improving it further.

Tables 7.3a and 7.3b show the incidence of target machines and target benchmarks. respectively, in the top decile of models by normalized RMSE (i.e., the worst 22 models). The models' error ranges from 7.66% to 41.69%. All eight machines appear in Table 7.3a, each being the target of at least one model, meaning that they all share the "blame" for the worst models. While it may be the case that machines that appear four times are harder to predict for than ones that appear only once due to their higher appearance among the worst models, this disparity is minor and does not seem to indicate a trend. However, a different scenario is true for the benchmarks, shown in Table 7.3b: only eight of the 27 possible benchmarks appear, only two of which are responsible for over half of the worst models (all unlisted benchmarks had no models in these 22 worst cases). This indicates that these benchmarks are specially poorly represented by the other members

Machine	Models
105	4
128	4
99	4
97	4
130	2
98	2
75	1
90	1

(a) Target Machine

Benchmark	Models
bwaves	8
cactusADM	5
calculix	2
mcf	2
milc	2
hmmer	1
lbm	1
zeusmp	1

(b) Target Benchmark

Table 7.3: Details of worst decile of models.

of the benchmark set. In fact, by discarding models of bwaves, cactusADM, calculix, mcf, and milc (i.e, the benchmarks with more than one model in the worst decile, and only 5 out of 27 benchmarks), the absolute worst case error is reduced from 41.69% to 9.59%. This indicates that a large and varied benchmark set is essential to ensure good predictive models for applications, while the machine selection, once constrained around the prediction target, plays a much smaller role.

7.6 Discussion

During the design and evaluation of the performance fingerprinting approach, we noted interesting properties of the approach. This section discusses those points.

Toward Representative Benchmark Suites As discussed in [35], effective performance fingerprinting requires applications to be represented by the benchmark set. If they are not, there might be no well-fitting model available, or worse, model selection may produce a well-fitting model that makes poor predictions. Luckily, the problem of creating a general benchmark suite appears tractable, since even SPEC CPU 2006, which was created with variety in mind, has several internal redundancies. This is exemplified by the effectiveness of the cluster subsetting strategy, which uses only five of the suite's benchmarks to model all others. Since none of the model selection strategies presented in Section 7.4 degrades as the benchmark suite grows, we believe that there should be an iterative effort in creating a generally representative benchmark suite: as developers find

applications that are poorly represented by a suite, a benchmark version of that application should be added to the suite. As more unique benchmarks are added, more applications will be represented. It is still important to limit the size of the benchmark suite as much as possible, to keep benchmark running time for each new platform from becoming excessive. Benchmark suite maintainers can implement internal redundancy checks (such as the correlation tests described in [77]) to limit benchmark suite sizes.

Leveraging DataMill for Performance Fingerprinting Section 7.5 shows that all of the model selection strategies benefit from (i) a large amount of training machines and (ii) training machines that are similar to target machines. It is already impractical for small developers to fingerprint their application on more than a handful of machines; restricting training machines to those similar to the target machine can make fingerprinting infeasible. Since DataMill already contains a wide variety of machines of multiple architectures for which performance results of several benchmark suites is already available, we believe that DataMill can provide developers with an effective performance fingerprinting service. As long as new machines are added to the service as technology evolves, updating an application's fingerprint is as simple as executing it on new machines as they come online in the infrastructure.

Chapter 8

Conclusion and Future Work

Empirical performance evaluation is a cornerstone of computer science and industry. Despite its importance, it has become evident in recent years that the status quo of performance evaluation is unsustainable: the lack of statistical rigor and experiment reproducibility puts into question our conclusions. As former president of the ACM Peter Denning notes, "[t]he science paradigm has not been part of the mainstream perception of computer science." This has prompted several papers bringing the issue to the attention of the community, and several efforts by publications to raise the bar in empirical computer science, but current practice still trails many other scientific disciplines.

It is clear from the current state of empirical experimentation in the field that the cost of statistically rigorous performance evaluation is more than what the community is willing to incur. The first part of this thesis presents our efforts in justifying and lowering the cost of statistically rigorous performance evaluation.

Chapter 3 demonstrates the dangers of ignoring statistical design and analysis tools in computer performance evaluation, using a Linux processor scheduler comparison as an example. We applied quantile regression to answer questions such as "what is the effect of the interaction between the choice of processor scheduler and system load on the 95th percentile of scheduling latency?", which are far beyond the reach of simple t-tests and even linear regression, with the added benefit of not imposing restrictions on the distribution of the data set.

Chapter 4 presents DataMill, a public performance evaluation infrastructure designed with statistical rigor and ease of use in mind. DataMill automates the process of factorial experiment design and provides hardware and software heterogeneity that is generally beyond the reach of any one researcher, while incurring the small overhead of packaging

the experiment for execution. We shows that with only 32 lines of shell code and a few clicks on a web interface, DataMill produced 6300 data points spanning seven machines, five compiler optimizations, three link orders, and both ASLR settings, exhaustively combining dimensions in a factorial design, and with plentiful replicates for rigorous statistical analysis, all in under a week.

Chapter 5 shows that the clean-room policy employed by DataMill minimizes performance variability caused by memory layout effects to a point where they are not worrisome in practice. Exploration of five memory layout factors, on 27 benchmarks, on up to 26 machines – a process trivialized by DataMill – shows that our current policy of restarting machines between every individual performance sample is sufficient to reduce memory-layout-related performance effects to negligible, sub-3% levels on average, and the absolute single worst case below 6% of the mean.

These three chapters aim to present a starting point in enabling widespread adoption of rigorous performance evaluation. The tools and approaches presents can be expanded to further reduce the complexity of empirical performance evaluation. Some of the foreseeable improvements and future work in this area follows:

1. Automatic Factor Screening: DataMill still requires some expert knowledge and user involvement in determining which hidden factors affect or do not affect the results of an experiment. We believe this process should eventually be fully automated;

2. User-Contributed Factors: Currently, the DataMill team develops and integrates each individual factor, but since factors can be the focus of a scientific experiment as often as the applications or benchmarks themselves (such as in Chapter 5), support for user-submitted factors could increase the target audience of DataMill significantly;

3. Scheduling According to Desired Statistical Power: Currently, data analysis is left to the user after all observations are collected. By prompting the user for the goal of the experiment during the experiment submission step (e.g., "the hypothesis is that approach A has mean run time at least 10% shorter than approach B, and I wish to prove that with a 95% confidence level"), DataMill itself could scale the number of replicates to the minimum number until the hypothesis can be satisfactorily proven;

4. Integration with Research Publication Venues: Community adoption of rigorous performance evaluation depends, to some extent, on incentives given by their target publications. Since DataMill already provides a result publication mechanism, and easy "cloning" functionality, we envision direct integration with journals and conferences as a milestone in the future of performance evaluation infrastructures.

In the second part of this thesis, we addressed cases where the ideal robust evaluation we advocate earlier is not a feasible option, either because the volume of experimentation is infeasible – as in high commit-rate software projects, requiring hundreds of independent experiments a day – or because the target platform is unavailable – as in cases where developers must choose between various prospective target platforms but cannot acquire them due to cost or availability.

Chapter 6 presents Perphecy, a tool that reduces performance regression detection latency to nearly zero while incurring small performance testing overhead, by predicting which code changes will affect the performance of which tests during development. Perphecy is based on robust performance evaluation techniques, lightweight static analysis and simple machine learning, and is fast enough to run at every commit, complementing periodic executions of the full performance test suite. In the case of Git, a widely-used, performance-sensitive software revision system, Perphecy predicted 85% of all performance-changing commits, while avoiding 83% of full performance testing.

Chapter 7 presents performance fingerprinting, a DataMill-based method of formally correlating application performance with that of standard benchmarks, generating statistical performance models that are accurate enough to predict the user's response metric on computers that they do not have access to. Performance fingerprinting removes any implied assumption from the use of standardized benchmarks, while fit metrics alert the user in case their application is not well represented by the benchmarks available. By constraining the prediction targets to machines similar to those in the training set, we achieved mean prediction error of under 2% of the mean for the components of SPEC CPU 2006, with a 95th error percentile below 10% of the mean.

These two approaches demonstrate the utility of the concept of performance prediction in a production setting. We envision that they can be further improved through the following future work:

- Perphecy:

    1. Other Machine Learning Approaches: Perphecy uses a simple heuristic to combine simplified indicators into a prediction. We applied this due to its simplicity, effectiveness, and the easy comprehensibility of complete predictors by users. Now that the indicator concept is demonstrated and the simple predictor has been shown to work, we believe that more general machine learning techniques can further improve the accuracy of predictors, specially because indicators would not need to be simplified down to thresholded boolean values.

2. Historical Indicators: While the performance of Perphecy for the HotSpot JVM was satisfactory (70% of changes predicted for 53% of full testing), managed languages are still a potential challenge to the strategy. "History-aware" indicators could predict performance changes caused by modified JIT or garbage-collector code after they have been linked to past performance changes, even though those features are responsible for only a small fraction of total run time.

- Performance Fingerprinting:

1. Statistically Sound Benchmark Suites: Some benchmark developers such as DaCapo [5] quantify the self-similarity in their benchmark suites in order to demonstrate their effort in eliminating redundancy. Performance fingerprinting also demonstrates that the "design space" of performance profiles should be uniformly covered so that as many applications are represented as possible. We believe that future research effort should be directed toward designing benchmark suites that not only avoid redundancies, but also strive to exhaustively represent performance-relevant features of software.
2. I/O-bound and Mixed Loads: The evaluation of performance fingerprinting focuses on SPEC CPU 2006, and, therefore, exclusively CPU-bound loads. While this suffices to demonstrate the concept for CPU-bound loads, real-world applications can also be I/O-bound or mixed. In order to support these loads, the benchmark suite must include I/O-bound benchmarks as well, so that the speed of each machine's storage subsystem is represented in the training data set. We believe that the performance of mixed-loads can be accurately modeled by a model with both CPU- and I/O-bound benchmarks as predictors, but demonstrating this remains as future work at this point.

Appendix A

Statement of Contributions

The following is the list of publications I have co-authored and made use of in this thesis. For each publication, I present here a list of my contributions.

- Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2013. Why you should care about quantile regression. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13)*. ACM, New York, NY, USA, 207-218.
  - Defined the case study;
  - Designed and conducted the experiment;
  - Analyzed the data;
  - Wrote the paper with feedback from co-authors.
- Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. 2013. DataMill: rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*, Seetharami Seelam (Ed.). ACM, New York, NY, USA,
  - Co-designed DataMill;
  - Designed and implemented the optimization-based job scheduler;
  - Implemented the worker-side job running script;

  - Designed, packaged, and conducted the case studies, with co-author assistance;
  - Analyzed the data;
  - Wrote portions of the paper.

- Augusto Born de Oliveira, Jean-Christophe Petkovich, and Sebastian Fischmeister. 2014. How Much Does Memory Layout Impact Performance? A Wide Study. In *Proceedings of the International Workshop on Reproducible Research Methodologies (REPRODUCE)*, Orlando, USA, 23-28.

  - Designed the experiments;
  - Conducted the majority of experiments;
  - Analyzed the data;
  - Wrote the majority of the paper.

- Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2015. Perphecy: Predicting Code Commit Performance Effects Without Running Performance Tests. Under submission.

  - Designed and implemented the approach;
  - Designed and conducted the case studies;
  - Analyzed the data;
  - Wrote the paper with feedback from co-authors.

- Augusto Born de Oliveira, and Sebastian Fischmeister, Performance Fingerprints: Using Benchmarks to Accurately Predict Application Performance Across Machines. Under submission.

  - Designed and implemented the approach;
  - Designed and conducted the case studies;
  - Analyzed the data;
  - Wrote the paper with feedback from co-author.

References

[1] Andrea Adamoli and Matthias Hauswirth. Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 73–82, New York, NY, USA, 2010. ACM.

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeño in java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 314–324, New York, NY, USA, 1999. ACM.

[3] Jens Axboe. Latt benchmark. http://git.kernel.dk/?p=latt.git;a=summary.

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[6] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM*, 51(8):83–89, August 2008.

[7] Matt Brubeck. Better automated detection of firefox performance regressions. http://limpet.net/mbrubeck/2013/11/10/improving-regression-detection.html, November 2013.

[8] Brian S. Cade and Barry R. Noon. A gentle introduction to quantile regression for ecologists. *Frontiers in Ecology and the Environment*, 1(8):412–420, 2003.

[9] Roy Campbell, Indranil Gupta, Michael Heath, Steven Y. Ko, Michael Kozuch, Marcel Kunze, Thomas Kwan, Kevin Lai, Hing Yan Lee, Martha Lyons, Dejan Milojicic, David O'Hallaron, and Yeng Chai Soh. Open CirrusTMcloud Computing Testbed: Federated Data Centers for Open Source Systems and Services Research. In *Proceedings of The 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[10] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.

[11] Tim Chen, Leonid I. Ananiev, and Alexander V. Tikhonov. Keeping kernel performance from regressions. In *Proceedings of the Linux Symposium*, OLSâĂŹ 07, pages 93–102, June 2007.

[12] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 448–459, New York, NY, USA, 2010. ACM.

[13] Evaluate Collaboratory. Experimental Evaluation of Software and Systems in Computer Science. http://evaluate.inf.usi.ch/. Accessed August 7th, 2012.

[14] Coveralls. https://coveralls.io/, 2014.

[15] Mark E. Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Trans. Netw.*, 5(6):835–846, December 1997.

[16] Charlie Curtsinger and Emery D. Berger. STABILIZER: statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 219–228, New York, NY, USA, 2013. ACM.

[17] D. Winer. XML-RPC Specification. http://www.xmlrpc.org/spec. Nov. 1999.

[18] DaCapo. http://dacapo.anu.edu.au/regression/perf/head.html, 2014.

[19] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. Why you should care about Quantile Regression — to appear. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, New York, NY, USA, 2013. ACM.

[20] Augusto Born de Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Why you should care about quantile regression. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[21] Augusto Born de Oliveira, Jean-Christophe Petkovich, and Sebastian Fischmeister. How much does memory layout impact performance? a wide study. In *Proceedings of the International Workshop on Reproducible Research Methodologies (REPRODUCE)*, page 23–28, Orlando, USA, February 2014.

[22] Augusto Born de Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. DataMill: rigorous performance evaluation made easy. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 137–148, New York, NY, USA, 2013. ACM.

[23] Peter J. Denning. Is computer science science? *Commun. ACM*, 48(4):27–31, April 2005.

[24] F. Desprez, G. Fox, E. Jeannot, K. Keahey, M. Kozuch, D. Margery, P. Neyron, L. Nussbaum, C. Perez, O. Richard, W. Smith, G. von Laszewski, and J. Voeckler. Supporting Experimental Computer Science. Technical report, Argonne National Laboratory Technical Memo, 2012.

[25] Drone. http://drone.io, 2014.

[26] Richard Finger. The java virtual machine (jvm): The platform that powers the world. http://www.forbes.com/, October 2013.

[27] Free Software Foundation. GLPK (GNU Linear Programming Kit). http://www.gnu.org/software/glpk/. Accessed Sep. 17th, 2012.

[28] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[29] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 367–384, New York, NY, USA, 2008. ACM.

[30] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. A microbenchmark case study and lessons learned. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 297–308, New York, NY, USA, 2011. ACM.

[31] Git. http://git-scm.com/, 2014.

[32] glibc. http://www.gnu.org/software/libc/, 2014.

[33] Gmane. Linux Kernel Mailing List - BFS vs. mainline scheduler benchmarks and measurements. http://thread.gmane.org/gmane.linux.kernel/886319.

[34] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in jvm performance. In *Component And Middleware Performance workshop, OOPSLA*, 2004.

[35] John L Gustafson and Rajat Todi. Conventional benchmarks as a sample of the performance spectrum. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 7, pages 514–523. IEEE, 1998.

[36] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press.

[37] Mor Harchol-balter. The effect of heavy-tailed job size distributions on computer system design. In *Proc. of ASA-IMS Conf. on Applications of Heavy Tailed Distributions in Economics*, 1999.

[38] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. Our troubles with linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 2:1–2:5, New York, NY, USA, 2011. ACM.

[39] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[40] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 114–122. ACM, 2006.

[41] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *NIPS*, pages 883–891, 2010.

[42] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71, New York, NY, USA, 2014. ACM.

[43] Elliot Jaffe, Danny Bickson, and Scott Kirkpatrick. Everlab: A Production Platform for Research in Network Experimentation and Computation. In *Proceedings of the 21th Large Installation System Administration Conference*, pages 203–213, 2007.

[44] Raj Jain. *The Art of Computer Systems Performance Analysis.* Wiley professional computing. Wiley, 1991.

[45] Jenkins. http://jenkins-ci.org/, 2014.

[46] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 155–170, New York, NY, USA, 2011. ACM.

[47] Julian Seward. bzip2 and libbzip2. http://www.bzip.org/. Accessed Sep. 17th, 2012.

[48] T. Kalibera, L. Bulej, and P. Tuma. Automated detection of performance regressions: the mono experience. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 183–190, Sept 2005.

[49] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance*

Evaluation of Computer and Telecommunications Systems (SPECTS), pages 853–862. SCS, 2005.

[50] Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals. Technical Report 4–12, University of Kent, June 2012.

[51] Tomas Kalibera and Richard E. Jones. Rigorous benchmarking in reasonable time. In Perry Cheng and Erez Petrank, editors, *ISMM*, pages 63–74. ACM, 2013.

[52] Tomas Kalibera, Jakub Lehotsky, David Majda, Branislav Repcek, Michal Tomcanyi, Antonin Tomecek, Petr Tuma, and Jaroslav Urban. Automated benchmarking and analysis tool. In Luciano Lenzini and Rene L. Cruz, editors, *VALUETOOLS*, volume 180 of *ACM International Conference Proceeding Series*, page 5. ACM, 2006.

[53] Roger Koenker and Jr. Bassett, Gilbert. Regression quantiles. *Econometrica*, 46(1):pp. 33–50, 1978.

[54] Roger Koenker and Kevin F. Hallock. Quantile regression. *Journal of Economic Perspectives*, 15(4):143–156, Fall 2001.

[55] Con Kolivas. FAQS about BFS. v0.330. http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

[56] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGOPS Oper. Syst. Rev.*, 40(5):185–194, October 2006.

[57] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[58] Libcbench. http://www.etalabs.net/libc-bench.html, 2014.

[59] D.J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2005.

[60] LKP. https://01.org/lkp/, 2014.

[61] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[62] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[63] Michael Larabel. OpenBenchmarking.org. http://openbenchmarking.org/. Accessed Sep. 17th, 2012.

[64] Michael Larabel. Phoronix Test Suite. http://www.phoronix-test-suite.com/. Accessed Sep. 17th, 2012.

[65] Ingo Molnar. Design of the CFS scheduler. http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.

[66] MongoDB. http://www.mongodb.org/, 2014.

[67] D.C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.

[68] Mozilla. http://graphs.mozilla.org/, 2014.

[69] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We have it easy, but do we have it right? In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –7, april 2008.

[70] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 265–276, New York, NY, USA, 2009. ACM.

[71] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Notes*, 44(3):265–276, March 2009.

[72] Oracle. The HotSpot Group. http://openjdk.java.net/groups/hotspot/, 2014.

[73] Larry Paterson and Timothy Roscoe. The Design Principles of PlanetLab. *Operating Systems Review*, 40(1):11–16, January 2006.

[74] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Trans. Netw.*, 3(3):226–244, June 1995.

[75] Perf. https://perf.wiki.kernel.org/, 2014.

[76] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences Building PlanetLab. In *Proceedings of The 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[77] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Subsetting the SPEC CPU2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(1):69–76, March 2007.

[78] PlanetLab. PlanetLab Bibliography. http://www.planet-lab.org/biblio visited 2012-09-28.

[79] PLDI AEC. http://pldi14-aec.cs.brown.edu/, 2014.

[80] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[81] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[82] Shared Hierarchical Academic Research Computing Network. SHARCNET. https://www.sharcnet.ca. Accessed Sep. 17th, 2012.

[83] Sameh Sharkawi, Don Desota, Raj Panda, Rajeev Indukuru, Stephen Stevens, Valerie Taylor, and Xingfu Wu. Performance projection of hpc applications using spec cfp2006 benchmarks. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[84] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 149–156, Washington, DC, USA, 2001. IEEE Computer Society.

[85] SPEC CPU 2006 Documentation. https://www.spec.org/cpu2006/Docs/, 2014.

[86] SPECjvm2008. http://www.spec.org/jvm2008/, 2014.

[87] S. Stefanov. Yslow 2.0. Presented at the CSDN Software Development 2.0 Conference, 2008.

[88] Talos. https://wiki.mozilla.org/Buildbot/Talos, 2014.

[89] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[90] Telemetry. http://telemetry.mozilla.org/, 2014.

[91] The Gentoo Foundation. Gentoo Linux. http://www.gentoo.org/. Accessed Oct. 5th, 2012.

[92] The Tukaani Project. XZ Utils. http://tukaani.org/xz/. Accessed Sep. 17th, 2012.

[93] Walter F. Tichy. Should Computer Scientists Experiment More? *IEEE Computer*, 31(5):32–40, 1998.

[94] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Systems Software*, 28:9–18, 1995.

[95] Travis. https://travis-ci.org/, 2014.

[96] VideoLAN. x264. http://www.videolan.org/developers/x264.html.

[97] Vince Weaver. Perf Event Overhead Measurements. http://web.eecs.utk.edu/~vweaver1/projects/perf-events/benchmarks/rdtsc_overhead/. Accessed Sep. 17th, 2012.

[98] Jan Vitek and Tomas Kalibera. Repeatability, Reproducibility, and Rigor in Systems Research. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 33–38, New York, NY, USA, 2011. ACM.

[99] W. Bergmans. Maximum Compression. http://www.maximumcompression.com/data/files/index.html. Accessed Sep. 17th, 2012.

[100] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 0:215–224, 2013.

[101] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *ACM Communications*, 27(10):1013–1030, October 1984.

[102] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[103] Xiph.Org Foundation. Xiph.org Video Test Media. http://media.xiph.org/video/derf/.

[104] Leo T. Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 40–, Washington, DC, USA, 2005. IEEE Computer Society.

[105] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.