Measuring Empirical Computational Complexity

by

Simon Fredrick Goldsmith

B.S. (Carnegie Mellon University) 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Alex Aiken, Co-chair
Professor Koushik Sen, Co-chair
Professor Rastislav Bodik
Professor Dor Abrahamson

Fall 2009

Measuring Empirical Computational Complexity

Abstract

Measuring Empirical Computational Complexity

by

Simon Fredrick Goldsmith

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Alex Aiken, Co-chair

Professor Koushik Sen, Co-chair

Scalability is a fundamental problem in computer science. Computer scientists often describe the scalability of algorithms in the language of theoretical computational complexity, bounding the number of operations an algorithm performs as a function of the size of its input. The main contribution of this dissertation is to provide an analogous description of the scalability of actual software implementations run on realistic workloads.

We propose a method for describing the asymptotic behavior of programs in practice by measuring their *empirical computational complexity*. Our method involves running a program on workloads spanning several orders of magnitude in size, measuring their performance, and fitting these observations to a model that predicts performance as a function of workload size. Comparing these models to the programmer's expectations or to theoretical asymptotic bounds can reveal performance bugs or confirm that a program's performance

scales as expected.

We develop our methodology for constructing these models of empirical complexity as we describe and evaluate two techniques. Our first technique, BB-TrendProf, constructs models that predict how many times each basic block runs as a linear ($y = a + bx$) or a powerlaw ($y = ax^b$) function of user-specified features of the program's workloads. To present output succinctly and focus attention on scalability-critical code, BB-TrendProf groups and ranks program locations based on these models. We demonstrate the power of BB-TrendProf compared to existing tools by running it on several large programs and reporting cases where its models show (1) an implementation of a complex algorithm scaling as expected, (2) two complex algorithms beating their worst-case theoretical complexity bounds when run on realistic inputs, and (3) a performance bug.

Our second technique, CF-TrendProf, models performance of loops and functions both per-function-invocation and per-workload. It improves upon the precision of BB-TrendProf's models by using control flow to generate candidates from a richer family of models and a novel model selection criteria to select among these candidates. We show that CF-TrendProf's improvements to model generation and selection allow it to correctly characterize or closely approximate the empirical scalability of several well-known algorithms and data structures and to diagnose several synthetic, but realistic, scalability problems without observing an egregiously expensive workload. We also show that CF-TrendProf deals with multiple workload features better than BB-TrendProf. We qualitatively compare the output of BB-TrendProf and CF-TrendProf and discuss their relative strengths and weaknesses.

Professor Alex Aiken, Co-chair

Professor Koushik Sen, Co-chair

Dedication

To my wife Lili for her patience, love, and support.

Contents

Acknowledgments

I would like to thank the following people, in no particular order. Johnathon Jamison and Armando Solar-Lezama for trying TrendProf; Karl Chen for trying and helping debug TrendProf, a great help; John Kodumal for feedback about `banshee`; Scott McPeak for feedback about `elsa`; Adam Chlipala, Robert Johnson, Matt Harren and Jeremy Condit for feedback on early drafts of a paper about this work; Jimmy Su, Jonathan Traupman, and Joseph Dale for useful discussions; Daniel Wilkerson for our collaboration and for good advice; Alex Aiken for all the good stuff that advisers do; the Open Source Quality group at Berkeley for listening to and helping me improve many half-baked talks; the people who helped make grad school fun including Matt Harren, Jeremy Condit, John Kodumal, Tachio Terauchi, Wes Weimer, Scott McPeak, and Daniel Wilkerson; my wife and family for their support and encouragement.

Parts of this dissertation were previously published in the Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering [GAW07]. Chapters 2 and 4, as well as Appendices A and B are derived from material that appeared in that paper. That publication and thus those sections arose from joint work with Daniel S. Wilkerson and my adviser Alex Aiken. Daniel Wilkerson suggested powerlaw fitting and assisted with the architecture and implementation of BB-TrendProf.

Chapter 1

Introduction

Scalability is a fundamental problem in computer science. Recent trends towards more information, more computational units, richer media, smaller devices, and larger systems (with more potential for unanticipated component interactions) push issues of scalability to greater prominence. Unfortunately, scalability problems often do not manifest themselves until a program is run at scale: on large workloads, at heavy load, or on many nodes.

Computer scientists often describe the scalability of algorithms in the language of theoretical computational complexity, bounding the number of operations an algorithm performs as a function of the size of its input. The main contribution of this dissertation is to provide an analogous description of the scalability of actual software implementations run on realistic workloads.

We propose a method for describing the asymptotic behavior of programs in practice by measuring their *empirical computational complexity*. Our method involves running

a program on workloads spanning several orders of magnitude in size, measuring their performance, and fitting these observations to a model that predicts performance as a function of workload size. We rely on the user to provide *workloads* and describe them with *features*, some quantity upon which performance depends — for example, size in bytes, number of abstract syntax tree nodes, or number of edges in a graph. As we run the program on these workloads, we measure the number of operations performed by each location (e.g., basic block, loop, function) in the program. Finally, for every location, we automatically construct statistical models that predict number of operations as a function of workload features. Comparing these models to the programmer's expectations or to theoretical asymptotic bounds can reveal performance bugs or confirm that a program's performance scales as expected.

This work combines strengths of theoretical asymptotic analysis of algorithms with strengths of empirical profiling to yield a methodology that complements both. The strength of theoretical asymptotic analysis is its ability to reason about algorithms as the size of the problem on which they operate becomes large. However, how an algorithm fits into the context of a larger program and how the program's actual workloads exercise this algorithm are harder questions to approach analytically. In contrast, the strength of profilers like `gprof` [GKM82] is their ability to focus on an actual workload and account for how much of that workload's cost is attributable to which program locations. What profilers miss, though, is a sense of how the cost of a location changes as workloads change — they say nothing about workloads on which the program was not run. This dissertation's models of empirical computational complexity seek to combine the empiricism of a profiler with the

generality of a big-O bound: we focus on actual workloads and consider the performance of each location in the context of the rest of the program while creating performance models that predict performance on novel workloads.

According to a careful study that measured a large system [AKLW02], only a small portion of the input space tends to account for much of the work a program does. They offer the following scenario based on their experience.

> [T]he theoretical size of the input space might be 10^{50}, but even after recording every input that occurred during a 12 month period, the number of distinct inputs that were actually observed numbered only in the tens of thousands. Furthermore, it is not uncommon for only several thousand inputs to correspond to more than 99% of the probability mass associated with the input space.

Their observation underscores one of the recurring themes in this dissertation: that performance depends on the empirical distribution of workloads and that this distribution need not be uniform nor cause performance to conform to theoretical bounds. Indeed, we show several examples of complex algorithms whose empirical performance ranges from sometimes different to entirely different from its big-O bounds (Sections 3.5.5, 3.5.7, 2.4.4, 2.4.7).

Example

The following code illustrates how our combination of empiricism with generality leads to a useful, novel perspective.

```
node * last_node(node *n) {
    if (!n) return NULL;
    while (n->next) n = n->next;
    return n;
}
```

From a performance perspective, this programming idiom looks suspicious: it is finding the last element in a list in time linear in the list's length. Adding a pointer directly to the

last element in the list would admit an obvious constant time implementation. Of course, if the list's size is a small constant, the performance impact of the linear search is likely negligible, and adding the pointer might not be worth the cost in space or code complexity. On the other hand, if the lists tend to be long, and especially if their length increases with the size of the program input, then use of this idiom constitutes a performance bug.

The crucial information is how this list is used in the context of the rest of the program and how the workloads of the program exercise it. The code above is from a C parser used in a program analysis system [KA05] and is called from a list append function to construct lists of compound initializers. In practice the sizes of the lists increase as inputs grow larger, but unless an input makes extensive use of compound initializers, `last_node` will not be particularly high on the list of what a typical, `gprof`-style profiler reports. On a workload with lots of long compound initializers, however, the unreasonable performance of this little function suddenly becomes apparent. We call this phenomenon a *performance surprise*. In contrast, we found a similar linear-time list append in a C and C++ front-end [MN04] that turned out to be benign: the lists are so small in practice (and depend on a quantity that is unlikely to be large) that use of this idiom does not substantially contribute to the overall performance of the system. Our technique automatically distinguishes these two different situations.

Core Assumptions

Our approach to modeling empirical computational complexity makes the following assumptions about the programs we profile. To the extent to which these assumptions do not hold, this work is not applicable.

- *Workloads exist.* The user can provide discrete workloads on which to run their program. As few as thirty workloads are enough to characterize programs with well-behaved performance or find the general trends in more difficult programs. However, given the lack of guarantees in this space and the often noisy relationships of performance to workload features, we generally opt for several hundred workloads ranging in size from small to large.

- The *workloads are representative* of the distribution of all interesting workloads for the program. The assumption is not trivial [AKLW02], but any serious investigation of program's performance must be grounded in an understanding of the distribution of its workloads.

- *Workload features exist.* The user can provide some quantities that describe a workload and are easy to compute.

Design Constraints

There are two design constraints that bear mentioning since they rule out many techniques for constructing models.

- Models must be *interpretable*. A human must be able to grasp what a model says about code's performance and relate it to her understanding of her code.
- Model building must be *automatic*. Our technique must build thousands of models without human intervention.

The ultimate consumers of our models of empirical computational complexity are humans. As we show, the utility of our models is in their ability to describe the empirical performance trends of actual implementations on realistic workloads and enable a human user to compare these descriptions to her expectations. Thus, we are unwilling to consider statistical techniques that do not yield interpretable models.

Our experiments involve using our technique to build performance models for tens of thousands of locations (e.g., basic blocks, functions, loops) in a program and considering hundreds of thousands of models in all. Clearly, our technique cannot function at this scale if it relies on human intervention to adjust models, interpret statistical test results, or transform data. Instead we must use automatic approaches to building models and provide the user with enough data to assess the validity of each model should she wish to do so.

Core Hypotheses

Throughout this dissertation we investigate several core hypotheses; these have a direct bearing on the utility and applicability of our technique.

- The given *workload features predict performance*. There is some functional relationship between workload features and performance.
- Our *models are valid*: they capture the relationship, if any, between workload features and performance. If there is no relationship, our selected models decline to model performance as a function of this workload feature.

Contributions

In Chapter 2 we develop the idea of measuring empirical computational complexity and elaborate on our decision to use execution count in our models of scalability (Section 2.1). We go on to describe our first technique, BB-TrendProf, to measure empirical computational complexity (Section 2.3). BB-TrendProf models the total execution count of *clusters* of basic blocks whose performance varies together (Section 2.3.1) as linear ($y = a + bx$) and powerlaw ($y = ax^b$) functions of workload features (Section 2.3.2). Although the user interface is not the focus of this work, we discuss several techniques that we have found useful for presenting scalability information for large (tens of thousands of lines of code) programs.

Section 2.4 establishes the utility of models of empirical computational complexity with experiments on several large programs. We show that BB-TrendProf reports simple results for programs with simple performance behavior (Section 2.4.2), confirm that

desired performance behavior is realized in practice (Section 2.4.3), measure the empirical performance of complex algorithms (Sections 2.4.3, 2.4.4, and 2.4.7), and find a scalability bug (Section 2.4.5). We argue that BB-TrendProf reports the empirical computational complexity of a program succinctly (Section 2.4.1) and that it helps focus attention on performance and scalability critical code (Section 2.4.6).

Chapter 3 develops our second technique, CF-TrendProf. In essence, CF-TrendProf seeks to describe the relationship between program performance and workload features more precisely than BB-TrendProf by using the program's control flow to suggest more complex and potentially more precise models. In order to gather information about control flow for its model generation process, CF-TrendProf models empirical computational complexity at the granularity of loops and functions not only per-workload, but also per-function-invocation. In general, CF-TrendProf considers multiple models for each location and picks a best one based on a novel model selection criteria. Section 3.3 describes CF-TrendProf's measurements, annotations the user can add to their program to improve CF-TrendProf's precision; Section 3.4 describes its model generation and model selection procedures.

We evaluate CF-TrendProf by considering how its models characterize the performance of well-understood algorithms and data structures like matrix multiply, doubling lists, Dijkstra's algorithm, Fibonacci heaps, insertion sort, quicksort, and hash tables (Section 3.5). We further show CF-TrendProf's use in diagnosing data structure problems in situations where a troubling super-linear trend is apparent in its output, but no workload necessarily exhibits glaringly obvious performance problems (Section 3.6). Further experi-

ments on larger programs (Section 3.7) demonstrate CF-TrendProf's ability to identify functions that are crucial to scalability.

Although CF-TrendProf's call tree organization of performance is more verbose than BB-TrendProf's clusters, it makes the overall structure of the program's performance and scalability clearer. Furthermore, CF-TrendProf finds more precise performance models than BB-TrendProf and chooses more effectively among models in terms of different workload features. On the other hand, BB-TrendProf's clusters and log-log scatter plots are valuable tools for managing large programs with difficult to characterize performance. We compare our two techniques in Section 3.9.

Throughout this dissertation, we use TrendProf when we are discussing issues common to BB-TrendProf, CF-TrendProf, and any other tool modeling performance as a function of workload features. We specify BB-TrendProf or CF-TrendProf when our discussion applies to one and not the other.

Chapter 4 reviews threats to the validity of TrendProf's models and features of TrendProf that mitigate them; our understanding of these threats has informed the design of TrendProf. Chapter 5 discusses related work.

Chapter 2

Basic Block TrendProf

In this chapter we develop a technique, BB-TrendProf, for building models of empirical computational complexity that predict how many times a basic block executes as a function of workload features. The work in this chapter initially appeared as a separate paper [GAW07]. Our technique is as follows.

- *Choose a program* to profile.
- *Choose workloads* $\{w_1, \ldots, w_k\}$ for the program.
- *Describe the workloads* with numeric features $(f_1, \ldots, f_k), (g_1, \ldots, g_k)$, for example the number of bytes in an input file or the number of nodes in a graph.
- *Measure program performance*; run the program on each workload and record the cost of each each basic block, ℓ, as a k-vector: $(y_{\ell,1}, \ldots, y_{\ell,k})$.
- *Group* basic blocks whose performance is correlated into *clusters*.

- BB-TrendProf *predicts performance in terms of features*, fitting the performance measurements, y, to features of the program's input, f. We use linear models, $y = a + bf$, and powerlaw models, $y = af^b$.

2.1 Measuring Empirical Computational Complexity

In describing models of empirical computational complexity in general, we use the term *location* to refer to the parts of the program (e.g., basic blocks) and *cost* to refer to a location's performance (for instance, its execution count). We discuss our choice of counting the number of times each basic block executes as a measure of performance in Section 2.1.1.

After running and measuring k workloads, we have a k-vector of costs for each location (one measurement per workload) and a k-vector for each feature (one value of the feature per workload); these k-vectors are rows in the matrix below. Profilers such as `gprof` [GKM82] report results for one column of this matrix. In contrast, we predict the costs of locations in terms of features; i.e., we construct models to predict one row in terms of another. For example, we might predict the number of compares a bubble sort does in terms of a feature like the number of elements to be sorted.

		workloads			
		w_1	w_2	$\cdots$	w_k
locations	ℓ_1	$y_{1,1}$	$y_{1,2}$	$\cdots$	$y_{1,k}$
	ℓ_2	$y_{2,1}$	$y_{2,2}$	$\cdots$	$y_{2,k}$
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	ℓ_n	$y_{n,1}$	$y_{n,2}$	$\cdots$	$y_{n,k}$
features	f	f_1	f_2	$\cdots$	f_k
	g	g_1	g_2	$\cdots$	g_k

2.1.1 Execution Counts

Our focus on modeling scalability rather than exact running time led to our choice of execution counts as a measure of performance. The amount of time (or number of clock cycles) each basic block takes is another measure, but we chose basic block counts because of the following advantages:

- ACCURACY: Block counts are exact: issues of insufficient timer resolution do not apply.
- REPEATABILITY: If a program is deterministic, so is its measure. Our measurements do not depend on the operating system or architecture if the program's control flow does not.
- LACK OF BIAS: The mechanism of measurement does not affect its result. In contrast, the mechanism of measuring time distorts its own result. We do not sample, so there is no sampling bias.
- LOW OVERHEAD: Counting basic block executions by computing control-flow edge coverage [BL94] is cheap (Section 2.4).
- PORTABILITY: We rely only on `gcc`'s coverage mechanism [GCO] and not on platform-specific performance registers. Furthermore, because execution counts (in general) do not depend on architecture, results measured on one machine should generalize to others.

2.1.2 Other Notions of Location

Our notion of basic blocks as locations is useful, but is not the only sort of location we might measure. For instance, CF-TrendProf (see Chapter 3) models both the amount of work a function does in its own code and the transitive work that its callees do. Furthermore, it allows the user to distinguish invocations of the same function with different data parameters — effectively attributing these invocations to different locations. Also, the work of Ammons et al. [ACGS04] (see discussion in Section 5.1) measures the work of a sequence of nested function calls.

2.2 An Example

Before exploring our methodology in detail, we illustrate the use of BB-TrendProf with the following simple sorting code.

```
    // pre:  The memory at arr[0..n-1] is an array of ints.
    // post:  arr[0..n-1] is sorted in place from least to greatest.
    void bsort(int n, int *arr) {
1:      int i=0;
2:      while (i<n) {
3:          int j=i+1;
4:          while (j<n) {
5:              if (arr[j] < arr[i]) //compare
6:                  swap(&arr[i], &arr[j]);
7:              j++;
            }
8:          i++;
        }
    }
```

This code has eight *locations* (each of which happens to be exactly one line of code), numbered one through eight above. Each *workload* for `bsort` consists of an array of `n` integers. The size, `n`, is a *feature* of the workload. We ran `bsort` on 30 workloads: 3

arrays of random integers at each of the following sizes 60, 200, 500, 1000, 2000, 4000, 8000, 15000, 30000, 60000. We chose these sizes because they span a wide range, their logarithms span a wide range, and the smallest size is large enough that the high order terms dominate all other terms. We find that including very small workloads, for instance an array with 3 integers, serves only to add noise to the left of the plot. In subsequent sections we show the output of BB-TrendProf on this example.

2.3 Implementation of BB-TrendProf

We describe how BB-TrendProf builds and ranks clusters and how it models the performance of these clusters.

2.3.1 Summarizing with Clusters

Studying the performance variation of the thousands of basic blocks in a large program would be overwhelming. Fortunately, doing so is unnecessary for understanding the performance and scalability of a program. In practice, large groups of locations have executions counts that are very well correlated with each other: on a run of `bsort` where line 2 executes many times, lines 3 and 8 will also execute many times; when line 2 executes only a few times, lines 3 and 8 execute few times.

This observation leads us to divide the locations in a program into *clusters* of locations that vary linearly together. A *cluster* consists of one location, called the *cluster representative*, together with the set of locations that linearly fit the representative with $R^2 > 1 - \alpha$, where R^2 is a measure of goodness of fit (see Appendix A.1.5) and α is a

small constant such that $0 < \alpha < 0.5$. Every location belongs to at least one, and possibly multiple, clusters.

BB-TrendProf computes the set of cluster representatives together with computing cluster membership. Initially the set of cluster representatives is the set of user-specified features. We consider locations in descending order of variance (σ_ℓ^2) and add location ℓ to all clusters whose representative it fits. If ℓ fits no existing cluster representatives, ℓ becomes the cluster representative for a new cluster. Thus, when the cluster representative is a location and not a feature, it has higher variance than any other location in the cluster.

The choice of a value for α is a trade-off between how many clusters BB-TrendProf finds and how well the locations in these clusters fit each other. Lower values of α produce more, but tighter clusters. In this work we use $\alpha = 0.02$. This choice is somewhat arbitrary, but it is informed by the following intuition. As we show in Appendix B, this choice guarantees that all the locations in a cluster fit each other better than $R^2 > 0.92$; note that the converse does not hold. In our experience, many fits with $R^2 < 0.90$ do not convincingly demonstrate the sameness of the locations being fit. In choosing α, we err on the side of having a strong guarantee about the locations in a cluster at the cost of having more clusters.

We discard data for locations executing a constant number of times or showing very little variation ($\sigma_\ell < 10$) as they contain little information: for example, we would discard a location whose cost is always between 100 and 120.

The Meaning of Clusters

Clustering organizes the mass of information without compromising the ability to point to specific places in the code since the costs of locations in the same cluster vary together. The following theorem gives us a simple guarantee about what it means for a location to be in a cluster: if α is 0.02 and location x is in the same cluster as location y, then the performance of x is linearly related to the performance of y with an R^2 better than 0.92.

THEOREM: *If x, y, and p are vectors of length k such that x and y both fit p with $R^2 > 1-\alpha$ and $0 < \alpha < 0.5$, then x fits y with $R^2 > 1 - 4\alpha(1-\alpha)$.*

PROOF: See Appendix B.

Example

In the `bsort` example, BB-TRENDPROF breaks the locations in this code into three *clusters* we call COMPARES, SWAPS, and SIZE.

- COMPARES's representative is line 4; it contains lines $\{4, 5, 7\}$.
- SWAPS's representative and only location is line 6.
- SIZE's representative is line 2; it contains lines $\{2, 3, 8\}$.

If we specify the size of the input array, `n`, as a feature of the workloads, then BB-TRENDPROF uses the feature `n` as the representative for the cluster SIZE.

Notice that although lines 5 and 7 execute $0.5n^2 - 0.5n$ times and line 4 executes $0.5n^2 + 0.5n$ times, these lines are all in the same cluster. This behavior is desirable since for

Cluster	Max	Fit with n	R^2
COMPARES	1.1×10^{10}	$3.0\ n^{2.00}$	1.00
SWAPS	2.6×10^{9}	$3.1\ n^{1.93}$	0.99
SIZE	1.3×10^{6}	$22\ n^{1.00}$	1.00

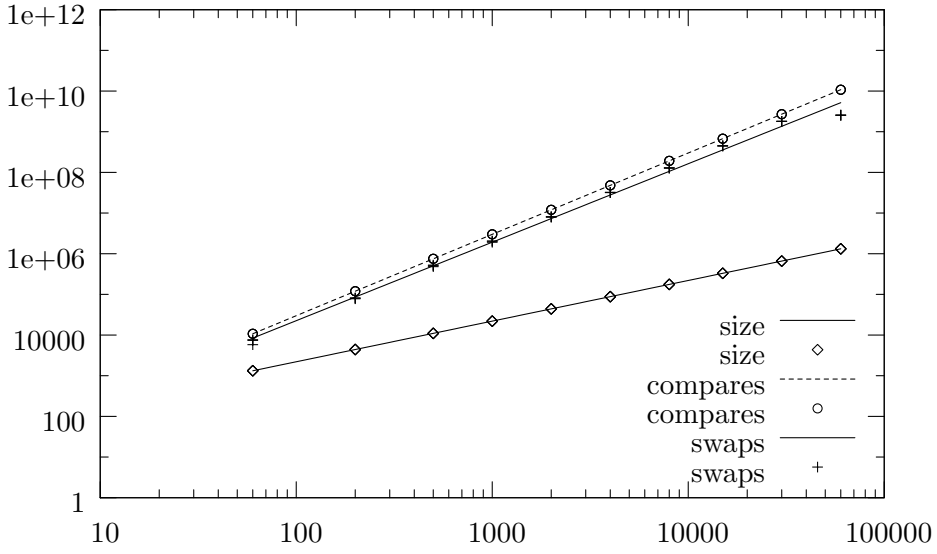

Figure 2.1: The table (top) shows powerlaw models predicting cluster costs for the Bubble Sort example. The graph (bottom) shows three powerlaw best-fit plots showing observed cluster costs for COMPARES, SWAPS, and SIZE (y axis) versus n (x axis) with their lines of best fit.

the values of n in our workloads, the quadratic term is the only important one for describing scalability.

2.3.2 Powerlaw Fits Measure Scalability

We define the *cost* of a cluster as the sum of the costs of all the locations in the cluster. BB-TRENDPROF measures the scalability of each cluster with respect to each feature, f, by powerlaw-fitting the cost of the cluster, C, to f; that is, BB-TRENDPROF finds a and b to fit $C = af^b$. The expression, af^b gives a concise, quantitative model of how the cost of the cluster increases as f increases. The summary output of BB-TRENDPROF

also includes the following for each feature/cluster pair.

- The R^2 goodness-of-fit statistic for the fit.
- *The best-fit plot*: a scatter plot of feature values versus cluster costs (f_i, C_i) on log-log axes with the line of best fit af^b. Recall that a true powerlaw looks like a line on log-log axes.
- *The residuals plot*: a scatter plot of f (x axis, log scale) versus the residuals $\log af^b - \log C$ (y axis, linear scale). The residuals plot is random if the powerlaw explains the data. Extra variation that the powerlaw does not account for, like a logarithmic factor or a lower order term, are often clearer in the residuals plot than the best-fit plot.
- Predicted cost at values of f larger than any actually measured. Define f_{95} as the 95th percentile value for f; that is if we have 1000 workloads and we sort the values for f, f_{95} is the 950th largest value. We show the model's predictions for $2f_{95}$ and $10f_{95}$ with a 95% confidence interval for each.
- A 95% confidence interval for a, the coefficient.
- A 95% confidence interval for b, the exponent.

We compute the confidence intervals mentioned above by means of a general statistical technique called the *bootstrap percentile method* [Ric06]. A detailed discussion of the bootstrap is beyond the scope of this thesis. In outline the bootstrap estimates the stability (such as the standard deviation or, in our case, confidence interval) of a function of the distribution of a random variable (such as median or mean or, in our case, the regression

coefficients or other predictions of our model). Bootstrap does this by 1) generating many "example" data sets, not from the distribution (which we do not know) but from the actual data set by repeatedly sampling with replacement, 2) computing the function in question on each example data set and collecting those results into a "function value" set and 3) simply measuring the stability of function value set (such as by throwing out the top and bottom 2.5% and calling the result the 95% confidence interval). The strength of the bootstrap method is that it makes no assumptions about any underlying distribution of the random variable (in our case, the regression coefficients). In BB-TrendProf we use one thousand iterations of the bootstrap.

A cluster that scales super-linearly (that is, has an exponent greater than one) has the potential to overtake higher ranked clusters on larger workloads. Thus, BB-TrendProf predicts situations where a cluster accounting for a modest portion of the cost of a program on medium sized workloads comes to dominate the performance cost on larger workloads.

The primary output of BB-TrendProf shows a list of clusters ranked by the maximum (over all workloads) cost of the cluster. This ranking draws attention to the clusters that cost the most on BB-TrendProf's workloads. Code that does not scale well and may cause performance problems is likely to be high on this list.

Since the logarithm of zero is not defined, BB-TrendProf ignores points where the observed execution count is zero when fitting to a powerlaw (the number of ignored points is reported). Thus, the models produced predict how many times a location is executed if it is executed at all. BB-TrendProf may be configured to suppress the display

Program	Description	Workloads
`bzip2 1.0.3` [BZ2]	Compresses files	Tarballs of preprocessed source code
`banshee 2005.10.07` [KA05]	Computes Andersen's alias analysis [And94] on a C program	Preprocessed C programs
`elsa` [MN04]	Parses, type-checks, and elaborates C and C++ programs	Preprocessed C++ programs
`maximus`	Ukkonen's suffix tree algorithm [Ukk90] for finding common substrings	C source code

Figure 2.2: We ran BB-TrendProf on these programs with workloads as described above.

Program	Workloads	Min – Max	Overhead	Time (h)
`bzip`	1000	3×10^7 – 2×10^{11}	22%	19 + 0.1
`banshee`	277	4×10^6 – 1×10^{10}	18%	0.7 + 1.1
`maximus`	910	3×10^4 – 8×10^9	10%	3.7 + 0.1
`elsa`	785	9×10^5 – 4×10^9	103%	3.3 + 7.4

Figure 2.3: Number of workloads, costs of the cheapest (Min) and most expensive (Max) workload (measured in number of basic block executions), geometric mean of overhead of edge profiling (Overhead), and BB-TrendProf's time in hours to run workloads and post-process data (Time).

of models constructed with few data points as such models are unlikely to make accurate predictions.

Example

In the `bsort` example we have only one feature, n, but it powerlaw-fits all cluster totals well. Figure 2.1 shows the scatter plot and lines of best fit for these powerlaws.

2.4 Results

We ran BB-TrendProf on the programs listed in Figure 2.2 with workloads as described in Figure 2.3. Figure 2.3 also mentions the average (geometric mean) overhead of

Program	Basic Blocks	Varying Basic Blocks	Clusters	Costly Clusters	Reduction Factor
`bzip`	1,032	721	23	10	103
`maximus`	1,220	496	13	9	136
`elsa`	33,647	22,382	1489	30	1122
`banshee`	13,308	11,891	859	26	512

Figure 2.4: For each benchmark we list number of basic blocks, number of basic blocks with $\sigma > 10$, number of clusters, number of clusters whose cost is ever more than 2% of the workload's total cost, and the ratio of Basic Blocks to Costly Clusters.

Cluster Rep		Max	Fit	R^2	Prediction
BYTES		35	77 BYTES$^{1.01}$	1.00	(470, 480)
blocksort.c	459	22	50 BYTES$^{1.03}$	0.95	(420, 580)
blocksort.c	416	16	34 BYTES$^{1.01}$	0.99	(210, 240)
blocksort.c	492	13	24 BYTES$^{1.04}$	0.94	(230, 340)
compress.c	241	3	4.0 BYTES$^{1.01}$	0.98	(23, 28)

Figure 2.5: The cluster representatives for the top clusters for `bzip`, the maximum observed cost of the cluster (in billions of basic block executions), the powerlaw fit of the cost of the cluster to BYTES, R^2 of this fit, a 95% confidence interval for predicted cluster cost (in billions of basic block executions) for a 5 GB workload.

running a workload with edge profiling enabled versus having it disabled (Overhead) and the total time in hours that our straightforward Perl implementation of BB-TrendProf takes to create a report on each program (Time). The Time column is broken down into two components: the first (left) time includes running the instrumented workloads and some minimal per-workload post-processing; the second (right) time includes the rest of BB-TrendProf's post-processing including clustering, model-fitting, and generation of plots and results pages. Once BB-TrendProf generates its results, they are browseable interactively.

2.4.1 Programs Have Few Clusters

For each of our benchmark programs, Figure 2.4 shows the number of basic blocks in the benchmarked program (Basic Blocks), the number of basic blocks whose standard deviation is greater than ten (Varying Basic Blocks), the number of clusters BB-TrendProf finds (Clusters), the number of clusters whose cost on any workload is more than 2% of the workload's total cost (Costly Clusters), and the ratio of basic blocks to costly clusters (Reduction Factor). These numbers illustrate a fundamental empirical fact about programs: that there are orders of magnitude fewer costly clusters than locations.

2.4.2 Simple Programs Have Simple Profiles

Running BB-TrendProf on `bzip` reveals that it scales linearly in the size of its input and that most of the locations vary together. Figure 2.5 shows the top several clusters of locations for `bzip`. The first cluster contains those basic blocks that linearly fit bytes, the number of bytes in the input, very well. The next several clusters all powerlaw-fit bytes very well with exponents very close to 1.0. Together these clusters account for 86% of the basic blocks in the program and (taking the geometric mean across all the workloads) more than 99% of the total number of basic block executions. Taken together, this output shows the number of bytes in the input is an excellent predictor of performance, that `bzip` scales nearly linearly in the size of its input, and that none of the code scales particularly worse than the rest. With BB-TrendProf a program with simple performance has a simple profile.

2.4.3 Confirming Expected Performance of the Implementation of a Complex Algorithm

Measuring the empirical computational complexity of a program using BB-TrendProf can verify that it scales as expected. Ukkonen's algorithm [Ukk90] finds common substrings in a string by constructing a data structure called a *suffix tree*. When implemented correctly, Ukkonen's algorithm creates a linear number of suffix tree nodes and edges. Faulty implementations of this tricky algorithm can cause performance with quadratic or worse scalability.

We ran BB-TrendProf on an implementation of Ukkonen's algorithm in a tool called `maximus`. A workload for `maximus` consists of a string. For each workload we specified three features: chars, the number of characters in the input string; nodes, the number of nodes in the suffix tree; and edges, the number of edges in the suffix tree. Feature chars is an easily measurable property of an input to `maximus`; after execution, `maximus` outputs nodes and edges and BB-TrendProf incorporates these features into its calculations. As expected, nodes and edges both linearly fit chars and thus wind up in its cluster. Figure 2.6 shows the relevant scatter plots and lines of best fit; chars is on the x axis and the two different styles of points and lines show nodes and edges.

The suffix tree representation of common substrings in a string is too compact to be comprehensible to a human, so `maximus` expands it to produce output. Operationally, for certain nodes in the suffix tree, the output routine must print something for each of the node's leaves and then recursively do the same thing for each of its children. This super-linearity is obvious in BB-TrendProf's output. The top ranked cluster scales as

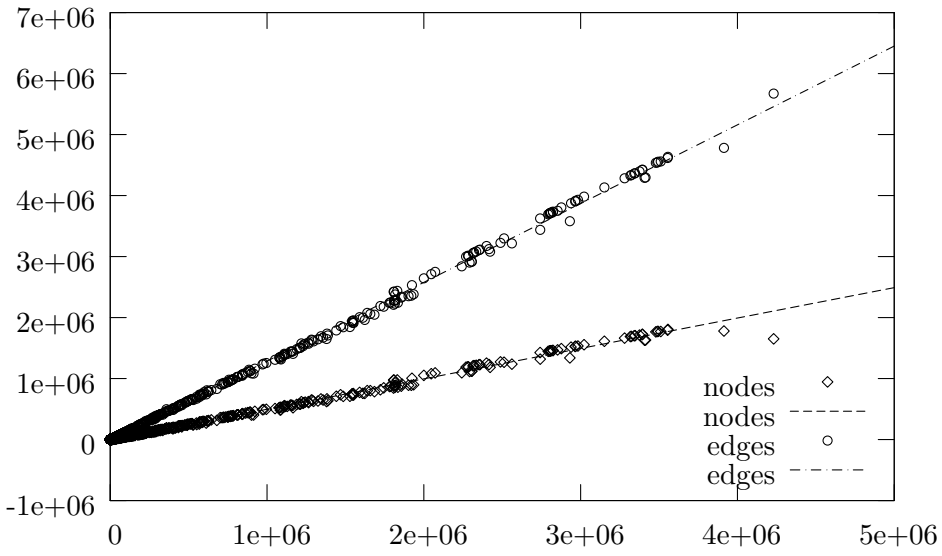

Figure 2.6: These two linear best-fit plots for `maximus` show that the number of suffix tree nodes and edges (y axis) grows linearly with the number of characters (x axis) in the workload.

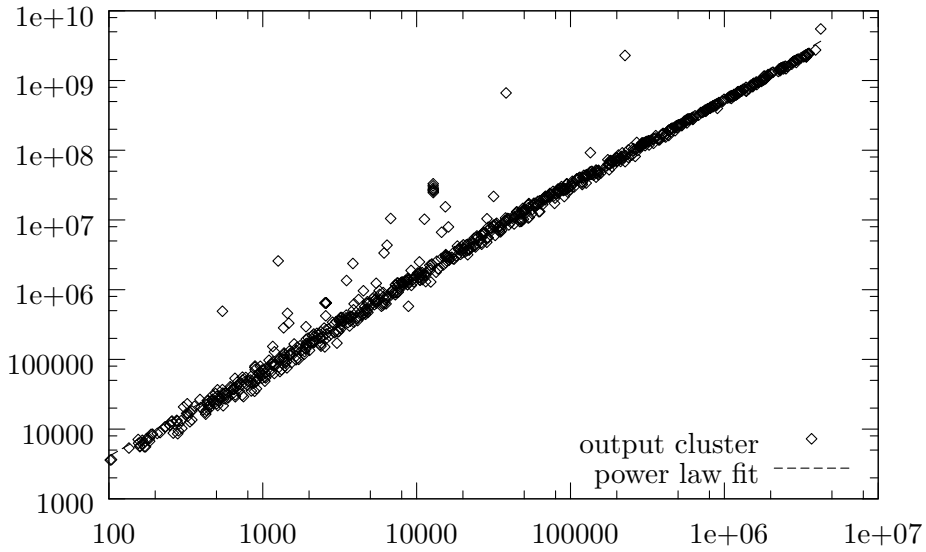

Figure 2.7: The crisp powerlaw fit in this best-fit plot for `maximus`'s output routines shows that their cost grows super-linearly in the number of characters in the input ($\hat{y} = 11\text{CHARS}^{1.29}$).

Cluster Rep		Max	Fit	R^2	Prediction
AST.c	34	800	0.9 BYTES$^{1.21}$	0.95	(400, 700)
regions.c	94	600	140 BYTES$^{1.01}$	0.99	(1900, 2100)
dhash.c	74	500	4 BYTES$^{1.05}$	0.98	(100, 200)
ufind.c	101	500	0.6 BYTES$^{1.18}$	0.88	(200, 300)
BYTES		200	50 BYTES$^{1.01}$	1.00	(700, 700)
AST.c	147	200	40 BYTES$^{1.02}$	1.00	(600, 700)
setif-sort.c	256	100	0.02 BYTES$^{1.25}$	0.86	(20, 40)
dhash.c	118	40	0.2 BYTES$^{1.03}$	0.95	(4, 7)
dhash.c	151	40	6 BYTES$^{1.03}$	0.99	(100, 100)
types.c	452	40	3 BYTES$^{1.05}$	0.98	(100, 100)
hashset.c	113	20	10^{-6} BYTES$^{1.72}$	0.87	(20, 60)
hashset.c	98	6	10^{-7} BYTES$^{1.91}$	0.77	(20, 50)

Figure 2.8: The top clusters for `banshee` with powerlaw fits and R^2. The maximum observed cost of each cluster and the 95% confidence interval for the model's prediction on a 128 MB workload are given in tens of millions of basic block executions.

11CHARS$^{1.29}$(R^2 = 0.99) and includes the output routines; Figure 2.7 shows the relevant best-fit plot.

The author of `maximus` was happy at the confirmation that the core of his implementation of this complex algorithm was in fact linear. Not being the object of his attention he was surprised at the super-linearity of the output routine; though obvious to him in retrospect, the use of BB-TrendProf was still required to find it.

2.4.4 Quantifying the Improvement of Heuristic Optimizations

At the core of our `banshee` benchmark is an implementation of Andersen's points-to analysis. Although this algorithm is cubic in the worst case, the workloads we measured scaled much better than that: no cluster scaled worse than n^2; Figure 2.8 shows the top several clusters. Realistic inputs often need not result in worst-case behavior; our measurements quantify the extent to which `banshee`'s optimizations take advantage of this fact.

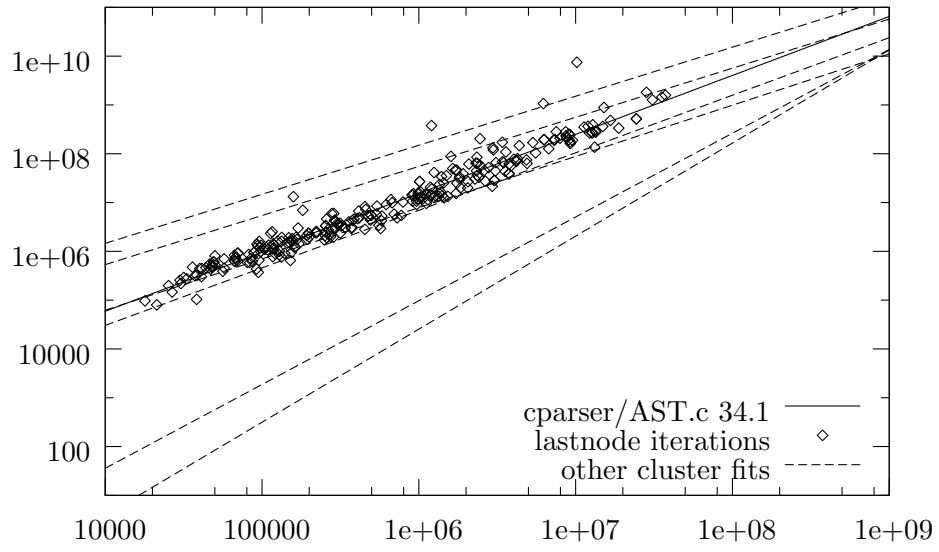

Figure 2.9: Powerlaw best-fit plot for the loop body of the performance bug in `banshee` ($\hat{y} = 0.87$ BYTES$^{1.21}$). We show lines of best fit for other cluster costs for reference.

2.4.5 This List Traversal is a Bug

As mentioned in earlier, we found a scalability bug in the C parser used by `banshee`. BB-TrendProf predicts that the `last_node` function (see below) is called roughly linearly in BYTES, the number of bytes in the input, and that the cost of the loop body scales as BYTES$^{1.2}$. These predictions suggest the average size of these lists grows as BYTES$^{0.2}$ and also that the three locations in this cluster account for more than 10% of the program's cost for inputs of 128 MB. Clearly, a pointer to the last node in the list is called for. Figure 2.9 shows the scatter plot of and powerlaw fit for this cluster together with the powerlaw fits for other top clusters (dotted lines) shown for comparison.

```
node * last_node(node *n) {
    if (!n) return NULL;
    while (n->next) n = n->next;
    return n;
}
```

2.4.6 Focusing on Scalability-Critical Code

The `elsa` benchmark is a parser, type-checker, and elaborator for C and C++ code. Running BB-TrendProf on `elsa` with C++ programs as input divides the roughly 33,000 basic blocks of `elsa` into fewer than 1500 clusters. Figure 2.10 shows the top several clusters and a few farther down the list with higher exponents along with their powerlaw fits to AST, the number of nodes in the abstract syntax tree for the workload, and 95% confidence intervals for our extrapolations when AST is 10 times larger than the 95th percentile value of AST for the workloads. Other features, notably BYTES, the number of bytes in an input, fit the cluster costs about as well as AST. The top several clusters contain code that is critical to the performance and scalability of `elsa` for large workloads.

2.4.7 An Empirical Measure of GLR Performance

Figure 2.11 shows the powerlaw fit and residuals plot for one of `elsa`'s top clusters (elkhound/glr.cc, line 362). Based on the scatter plot and residuals plot, the powerlaw fit with AST is a reasonable model for this cluster's cost. The 95% confidence interval for the exponent is (1.11, 1.15), and so it appears that the code in this cluster scales super-linearly with the number of AST nodes in the input. This cluster is largely concerned with GLR parsing and tracking and resolving ambiguous parse trees. As we would expect from a mostly unambiguous grammar and a well optimized parser generator [MN04], the measured empirical computational complexity is substantially better than the cubic worst case complexity of GLR parsing. Nonetheless, the slight super-linearity and the large coefficient suggest that this code is crucial to performance.

Cluster Rep		Max	Fit	R^2	Prediction
hashtbl.cc	44	100	6500 $(\text{AST})^{0.76}$	0.93	(40, 50)
ARGEXPR		70	260 $(\text{AST})^{1.11}$	0.97	(300, 300)
glr.cc	362	70	200 $(\text{AST})^{1.13}$	0.95	(300, 300)
cc_flags.h	139	70	490 $(\text{AST})^{0.865}$	0.84	(10, 20)
sobjset.h	28	60	65 $(\text{AST})^{0.997}$	0.84	(10, 20)
STMT		20	260 $(\text{AST})^{1.02}$	0.99	(70, 80)
hashtbl.cc	67	20	280 $(\text{AST})^{0.833}$	0.90	(4, 6)
lookupset.cc	154	4	0.008 $(\text{AST})^{1.35}$	0.65	(0.2, 0.5)

Figure 2.10: The top clusters for `elsa` with power law fits and R^2. We show the maximum observed cost of each cluster and a 95% confidence interval for the model's prediction on a two hundred thousand AST-node workload in tens of millions of basic block executions. The cluster representatives ARGEXPR and STMT are features that count particular kinds of AST nodes.

2.4.8 This List Traversal Is Not a Bug

The cost of the cluster whose representative is line 154 of `elsa/lookupset.cc` fits AST with the notably high exponent of 1.35. Figure 2.12 shows the scatter plot and powerlaw fit for this cluster's cost; it also shows the powerlaw fit for another top cluster whose representative is `elkhound/glr.cc` line 362 (also shown in Figure 2.11) for comparison. There is a lot of variance in the data and thus the fit is somewhat dubious, but two things are clear. For at least some kinds of inputs, this cluster's cost increases sharply as input size gets large. However, even if we follow the upper edge of the points, this cluster's cost will not overtake the cost of the other cluster for any reasonably sized input (recall that the y axis is on a logarithmic scale and that a factor of 100 is not particularly tall). We conclude that the code in this cluster is not crucial to performance.

The code in the aforementioned cluster consists of a function to add an object to a list in time linear in the length of the list. This pattern is exactly the sort of code that was a performance bug in the `banshee` benchmark, but here BB-TrendProf provided us

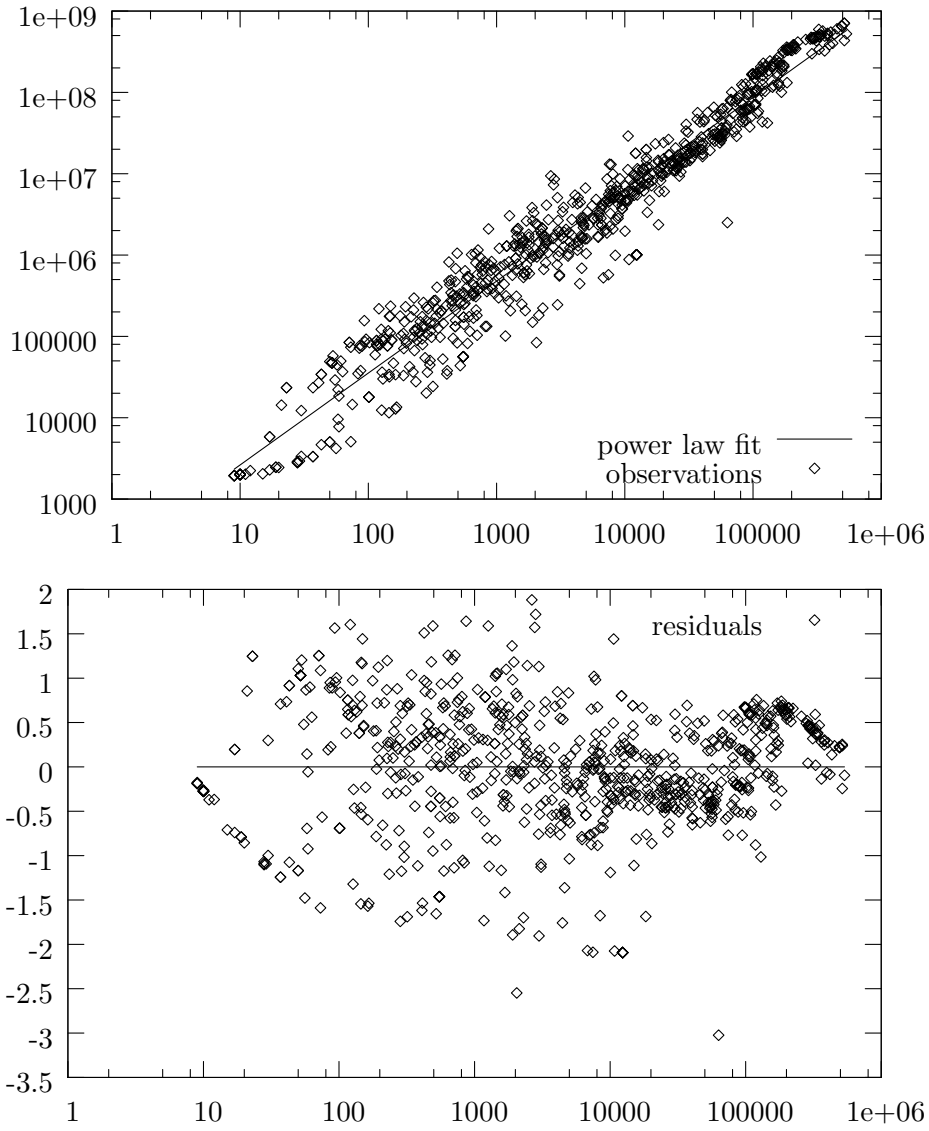

Figure 2.11: A powerlaw best-fit plot showing the slight super-linearity of `elsa`'s GLR parsing ($\hat{y} = 195$ AST$^{1.13}$, $R^2 = 0.95$) (top) and the corresponding residuals plot (bottom).

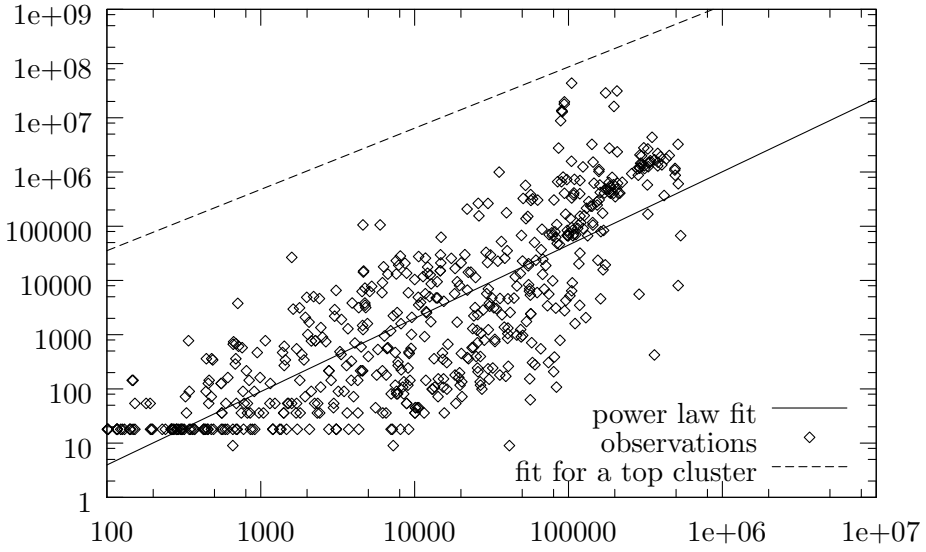

Figure 2.12: There is no clear relationship between feature AST and these points. Thus, the line of best fit (solid) is dubious. Nonetheless, comparing to the powerlaw fit from Figure 2.11 (dotted line) suggests that this cluster is not a scalability problem.

with enough information to conclude that it is not a serious scalability problem. The code is as follows.

```
void LookupSet::add(Variable *v) {
    for each w in this {
        if (sameEntity(v, w)) return;
    }
    prepend(v);
}
```

2.5 Assessment of BB-TrendProf

Given a program and workloads for it, BB-TRENDPROF 1) builds models of basic block execution count in terms of user-specified workload features and 2) provides the means to assess the plausibility and applicability of these models. By grouping related locations into clusters and modelling the performance of these clusters, TRENDPROF summarizes the performance of tens of thousands of lines of code with a few dozen of these models.

We cluster locations (and features) that vary together linearly; conversely, locations that vary somewhat independently end up in different clusters. For several programs and associated sets of workloads, we have empirically measured that there are many fewer clusters than locations; that is, empirically there is much linear correlation between execution counts of locations. Clustering dramatically reduces the number of degrees of freedom of the overall performance model; that is, clustering simplifies our presentation of program performance by dramatically reducing the number of program components whose costs we model. With only the clustered view of performance, however, it can be difficult to follow the control flow of the program or figure out how performance is distributed through the dynamic call graph.

BB-TrendProf fits each cluster's per-workload cost to a powerlaw. These models are not always accurate, but they often capture the general trend in the data as input size grows large. Managing multiple features is somewhat of a problem for BB-TrendProf: its powerlaw fits only support one feature and they are often not precise enough to really justify choosing one feature over another to describe a given cluster's performance. In the next chapter, we develop a technique that aims to relate performance more closely to the program's call graph and to increase model accuracy by trying a richer family of models that we generate based on the structure of the program's call graph and control flow.

Chapter 3

Control Flow TrendProf

3.1 Overview

Chapter 2 describes a technique and a tool, BB-TrendProf, for measuring empirical computational complexity by modeling the execution count of basic blocks as a function of workload features. This chapter presents a new technique and tool, Control Flow TrendProf (CF-TrendProf) that models program performance at the granularity of loops and functions. CF-TrendProf explores a different part of design space than BB-TrendProf and improves upon it in several ways as follows.

- CF-TrendProf models performance both per program run (like BB-TrendProf) and per function invocation.
- CF-TrendProf uses hints from the program's control flow and call graph to suggest performance models that are potentially more precise than BB-TrendProf's linear and powerlaw models (Section 3.4.2).

- CF-TrendProf considers many models for each location, including models in terms of multiple features; choosing the best from among these models (Section 3.4.3) potentially improves the precision of CF-TrendProf's models.

Section 3.9 contains a more thorough comparison of BB-TrendProf with CF-TrendProf.

A New Approach

Like BB-TrendProf, CF-TrendProf measures performance in terms of execution counts. Instead of modelling basic block executions, though, CF-TrendProf measures and models the costs of loops and functions. CF-TrendProf charges one unit of performance for either entering a function or going once around a loop. In addition to the per-workload view of performance inherent in BB-TrendProf, CF-TrendProf also measures and models performance per function invocation. Furthermore, CF-TrendProf models not just the per-invocation *self-cost* of each function (cost of the function excluding callees' costs), but also the per-invocation *transitive-cost* (cost of the function including callees' costs), and the per-workload *total-self-cost* and *total-transitive-cost* (computed as sums over all invocations). A single run of the program can potentially provide many per-invocation data points. Section 3.3.2 discusses the performance quantities that CF-TrendProf gathers.

Like BB-TrendProf, CF-TrendProf models the cost of a location (for example, a function's self-cost), by fitting measurements of that location's cost directly to workload features (Section 3.4.1); we call these BB-TrendProf-style models *direct mod-*

els. However CF-TrendProf also models performance more precisely with *derived models* that it forms by symbolically adding and multiplying other models based on hints from the program's control flow and call graph. The quantities CF-TrendProf measures and models are carefully chosen to enable computation of these derived models. Section 3.4.2 discusses the how CF-TrendProf builds derived models. This technique for deriving models allows CF-TrendProf's performance models to increase in complexity as the program they model increases in complexity.

One useful example of a derived model allows CF-TrendProf to model a function's self-cost by symbolically adding the models for the loops in the function. A function's self-cost is the sum of the costs of the loops in the function plus one for entering the function. CF-TrendProf measures the cost of each loop in the function and constructs direct models for these costs. By symbolically adding the models that it has constructed for each loop in the function (and adding 1 for entering the function), CF-TrendProf constructs a derived model for the function's self-cost.

This process of generating direct and derived models results in several candidate models for each performance quantity (e.g., self-cost, transitive-cost, total-transitive-cost, etc.). Section 3.4.3 discusses the process of choosing the best model from among the candidates based on a trade-off of model complexity versus model precision. The overall goal of CF-TrendProf's enhancements to the model selection procedure is to increase the precision of its performance models by considering (in a principled way) more complex performance models in terms of multiple features.

Multiple Views

The overall organization of CF-TrendProf's performance models differs from that of BB-TrendProf. While BB-TrendProf organizes its per-workload basic block models into clusters, CF-TrendProf models each function's per-invocation self-cost and transitive-cost, as well as its per-workload total-self-cost, total-transitive-cost, and call-count. Thus CF-TrendProf provides a view of performance for different chunks (self-cost versus transitive-cost) of a program at different levels of granularity (per-workload versus per-invocation). For example, considering the per-invocation transitive-cost of inserting an element into a doubling list (Section 3.5.3) is not enlightening, but considering the per-workload total-transitive-cost shows the linear trend: inserting n elements scales linearly in n. Section 3.5.7 illustrates another example. We consider the empirical scalability of Dijkstra's breadth first search (BFS) algorithm when it uses a Fibonacci heap as a priority queue. While the cost of each loop and helper function call is interesting to the developer of the Fibonacci heap and Dijkstra's algorithm, the bottom line for a consumer of this algorithm is what the entire breadth first search costs on some input graph. This cost is represented as a transitive-cost of the `dijkstra` function (Figure 3.27) and the cost of all BFS calls per workload is modeled as the total-transitive-cost of the `dijkstra` function. These views onto performance aim to capture the common decompositions programmers use to reason about program performance. If CF-TrendProf's decomposition of performance is insufficient, the user can add annotations to improve it (Section 3.3.1).

Evaluation

As with BB-TrendProf, CF-TrendProf is successful to the extent that it accurately models the performance of a program and facilitates a comparison of observed performance to expected performance. We evaluate CF-TrendProf by considering how its models characterize the performance of well-understood algorithms and data structures such as matrix multiply, doubling lists, Dijkstra's algorithm, Fibonacci heaps, insertion sort, quicksort, and hash tables in limited scenarios (Section 3.5). These micro-benchmarks show strengths and weakness of CF-TrendProf that we summarize below. As with BB-TrendProf, CF-TrendProf's models are about measured performance of a particular implementation run on a particular set of workloads; this focus can point to interesting trends in program inputs or faulty implementations of algorithms. In many cases, CF-TrendProf's models are more precise than any model BB-TrendProf can deduce. Expected case analysis and amortized analysis are powerful theoretical tools for characterizing asymptotic performance, but can be hard to apply to actual implementations and real-world workloads; CF-TrendProf effectively applies these techniques to the empirical distribution of workloads. On the other hand, CF-TrendProf's view is limited to an expected case view: its models do not provide upper or lower bounds.

Section 3.7 evaluates CF-TrendProf based on the insight it yields into larger programs. In general, the organization of CF-TrendProf's output based on the dynamic call tree allows one to follow the cost of a workload through the call tree. Many functions have moderate to good models that show that their cost scales linearly with workload features; absent concerns about other resources (which are beyond the scope of this disser-

tation) these functions can be safely ignored. Other functions scale super-linearly and are thus likely to be important to the scalability of the program.

Both the small and large benchmarks demonstrate CF-TrendProf's ability model performance precisely, sometimes perfectly, as a function of one or multiple workload features. In many cases, CF-TrendProf's model improves upon the precision of BB-TrendProf's. However, as we demonstrate in Section 3.5.5 and see again in Section 3.7.4, sometimes performance is not a simple function of workload features. The latter part of our evaluation of CF-TrendProf describes how it mitigates this issue and compares CF-TrendProf's approach to BB-TrendProf's. Section 3.9 summarizes this comparison.

3.2 Example

Interspersed with our discussion of the CF-TrendProf technique in Sections 3.3 and 3.4, we describe CF-TrendProf's measurements, processing, intermediate results, and output on a simple sorting algorithm, `bsort`, shown in Figure 3.1. The data we present is simplified and stylized for clarity of presentation, but the substance is not altered.

Workloads A workload for this simple example consists of an array, `arr`, full of n `int`s. The size of the array, n, is a workload feature: CF-TrendProf will model performance of `bsort` and its loops in terms of n. For a more complex program, however, we might choose to make n an invocation feature (Section 3.3.1), recording its (potentially different) value for each call to `bsort` and modeling the per-invocation performance of `bsort` in terms of n.

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

    // pre:  The memory at arr[0..n-1] is
    // an array of ints.
    // post:  The ints in arr[0..n-1] are
    // sorted in place from least to greatest.
    void bsort(int n, int *arr) {
1:      int i=0;
2:      while (i<n) {
3:          int j=i+1;
4:          while (j<n) {
5:              if (arr[j] < arr[i]) //compare
6:                  swap(&arr[i], &arr[j]);
7:              j++;
            }
8:          i++;
        }
    }

int main() {
    int n=..., *arr=..., i=-1;
    while (++i < n) arr[i] = ...;
    bsort(n, arr);
    return 0;
}
```

Figure 3.1: Code for the `bsort` example.

Function	self-cost	transitive-cost	call-count
`main`	$n+1$	$0.7n^2+1.5n+2$	1
`bsort`	$0.5n^2+0.5n+1$	$0.7n^2+0.5n+1$	1
`swap`	1	1	$0.2n^2$

Function	total-self-cost	total-transitive-cost
`main`	$n+1$	$0.7n^2+1.5n+2$
`bsort`	$0.5n^2+0.5n+1$	$0.7n^2+0.5n+1$
`swap`	$0.2n^2$	$0.2n^2$

Figure 3.2: Expected output of CF-TrendProf for the `bsort` example.

Loops This code has two loops: an outer loop at line 2, ℓ_o, and an inner loop at line 4, ℓ_i. With some thought we can see that ℓ_o goes around exactly n times per call and that ℓ_i goes around $\frac{1}{2}n(n-1)$ times per call.

Functions This code has three functions `main`, `bsort`, and `swap`. Characterizing the performance of `main` and `bsort` is easy, `swap` is harder because its total cost varies based on subtle properties (sortedness) of the input. Although it can vary widely, assume (as is the case for inputs generated uniformly at random) for the purposes of our example that `swap` is called about $0.2n^2$ times to sort an array of size n. Figure 3.2 summarizes the output we would expect from CF-TrendProf in this situation.

3.3 Gathering Data

The steps for using CF-TrendProf to measure empirical computational complexity are as follows:

- *Choose a program* to profile.

- *Annotate the program* with invocation features and contexts as necessary (Section 3.3.1).
- *Choose workloads* $\{w_1, \ldots, w_k\}$ for the program.
- *Describe the workloads* with numerical features $(f_1, \ldots, f_k), (g_1, \ldots, g_k)$, for example the number of bytes in an input file or the number of nodes in a graph.
- *Measure program performance*; run the program on each workload and record the loops costs, functions costs, and other measures (Sections 3.3.1, 3.3.2).
- *Build direct models* by fitting the performance measurements, y, to features of the program's input, f (Section 3.4.1). We use constant models, $y = a$, linear models, $y = a + bf$, and powerlaw models, $y = af^b$.
- *Generate derived models* for loop and function costs (Section 3.4.2).
- *Choose the best model* for each loop and function cost from among the direct and derived models (Section 3.4.3).

In our prototype implementation of CF-TrendProf, the last two steps are carefully interleaved to keep the working set small (Section 3.4.4). The following discussion proceeds in the order that CF-TrendProf processes its data.

3.3.1 Measuring Performance

In order to gather the performance data it requires, CF-TrendProf instruments the program to emit a program trace as it runs. The user may annotate the program to identify performance-relevant quantities or to distinguish calls of a function that ought to be

modeled separately. Running a workload yields a program trace that CF-TrendProf post-processes to extract the performance data it needs for subsequent analysis (Section 3.3.2). The details of the instrumentation and of the program trace are engineering concerns specific to our prototype implementation of CF-TrendProf: their goal is to obtain the data described in Section 3.3.2; other approaches are possible. For the purposes of a research prototype though, our clean separation of the program tracing and the data extraction has simplified our implementation efforts.

Preparing the Program to be Measured

Before instrumenting the program, CF-TrendProf assigns a unique ID to each function and each loop. It also records (for later stages) each loop's outer loop (if any) and the function in which each loop resides. As the instrumented program runs, CF-TrendProf's instrumentation does the following:

1. At the start of the `main` function, open a file to contain the trace.
2. At the start of a function, emit a start-of-function record that contains the function ID.
3. At the start of a function, reserve space for a counter for each loop and initialize these to zero.
4. Every time a loop goes around, increment its counter.
5. Before a function returns, emit an end-of-function record that contains the function ID and the counts for each loop.

User Annotations

The user may enhance the results of CF-TrendProf by adding annotations. Operationally, these annotations call CF-TrendProf helper functions that emit data into the program trace.

Workload features describe workloads. The user can specify them in a configuration file that associates workload features with workloads or they can call `tpRuntimeWorkloadFeature` with a name and a value. Essentially, if the easiest way to measure a feature of a workload is to have the program do it, this CF-TrendProf call saves the user some mechanism.

Contexts allow the user to partition the calls to a function. CF-TrendProf models the performance of the calls in each context separately, as if they were calls to separate functions that happen to have identical source code. The user identifies the context of a call by calling `tpRuntimeSetContext` with a context specifier and a flag to designate what is to be annotated. The context is a string the user provides, the function's caller, the entire call stack, or the currently executing (active) function. The context annotates either the active function, its immediate callees, or all the functions it calls transitively. Contexts are useful for apportioning performance cost of a library to different callers or even costs of data structure operations to different instances of the data structure.

Invocation features allow the user to identify a quantity that CF-TrendProf will use to predict performance of a function invocation by calling `tpRuntimeInvocationFeature` with a name and a value. For example, one might specify the size of a linked list as an

invocation feature to linked list operations. As we describe below, CF-TrendProf uses invocation features to predict all the quantities (loop counts, function costs, etc.) in the scope of the function.

3.3.2 Workload Data

Once the instrumented program has been run on a workload, CF-TrendProf is left with a program trace. In one pass over the trace, CF-TrendProf extracts data about the cost of the workload and the cost of each function invocation that happened during the workload. We refer to each of these quantities as *performance variables* or just *variables* when there is no potential for confusion.

For each invocation of function `F`, CF-TrendProf computes the following from the program trace.

- invocation features in `F`'s scope
- loop count for each loop in `F`
- average count of each inner loop (in `F`) per iteration of its outer loop: $\frac{\text{inner loop count}}{\text{outer loop count}}$
- `F`'s self-cost: one plus the sum of the loop counts of all the loops in `F`
- call-count for each direct callee
- transitive-cost for each direct callee, `G`: the sum of the transitive-costs of all calls to `G` during this invocation of `F`
- `F`'s pure-transitive-cost: the sum of the transitive-costs of all direct callees

- `F`'s transitive-cost: self-cost plus pure-transitive-cost

Once CF-TrendProf has read the end-of-function record in the program trace, it has enough information to compute all of the above for that invocation. After computing the function `F`'s transitive-cost, CF-TrendProf charges `F`'s caller for the cost of the call to `F`. It is by this mechanism of direct callees charging callers that CF-TrendProf computes per-direct-callee call-count and transitive-cost. We refer to this collection of data for a function invocation as a *frame*.

Even for modestly sized programs, this per-invocation data gathering leads to a massive number of frames. Attaining reasonable performance for CF-TrendProf requires that we compress these frames.

Subsequent steps of CF-TrendProf require that we be able to match, for example, an invocation feature to a loop count from the same invocation. However, the order of frames is not important; all the data we need to know about an invocation is recorded in its frame. Thus we may not split frames, but we may re-order them.

Empirically, there are many duplicate frames. Our compression strategy is to hash the frames and keep count of the occurrences of each frame—essentially re-ordering them to group duplicates. For convenience in subsequent steps, we output this data as a run-length encoded list of points for each variable in the frame (loop count, self-cost, etc.). Re-ordering by frame instead of treating each variable separately means that the i^{th} point in the list of data points for a function's self-cost and the i^{th} point in the list of data points for an invocation feature for that function refer to the same invocation in the same workload.

In addition to per-invocation data, CF-TrendProf gathers per-workload data

from the program trace as follows.

- total loop count for each loop: computed by summing over all invocations
- average count of each inner loop per iteration of its outer loop: $\frac{\text{total inner loop count}}{\text{total outer loop count}}$
- total-self-cost for each function: computed by summing over all invocations
- total-transitive-cost for each function: computed by summing over all invocations, taking care to avoid double-counting recursive calls to the same function
- call-count for each function

Example

Figure 3.3 shows the content of the program trace for a run of our `bsort` example for a workload with $n = 10$. In our actual implementation, the program trace is more terse, but we use names and labels here for clarity of presentation. Figure 3.4 shows the entire data record that CF-TrendProf extracts from the program traces after running workloads with $n = 10, n = 20, n = 1000$. The notation 1r23 is run-length encoding for 1 repeated 23 times. Since `main` and `bsort` are only called once each, their total (per-workload) figures are no different than their per-invocation figures. Since `swap` is called many times per workload, it has many data points in each column.

3.4 From Data to Models

CF-TrendProf's goal (like BB-TrendProf's) is to predict program performance as a function of workload features or invocation features. Recall that we refer to

```
start main
  start bsort
    start swap
    end   swap
    ...
    start swap
    end   swap
  end bsort, outer_loop_count=10, inner_loop_count=45
end main, main_loop_count=10
```

Figure 3.3: The program trace for the `bsort` example when $n = 10$. For clarity, we replace numerical IDs with function and loop names and annotate the fields of records.

these functions as *models*. Thus we can restate CF-TrendProf's goal more operationally: CF-TrendProf seeks to choose a sensible model (for example, $a + bx + cx^2$ or ax^b) and then fit its observations to this model (that is, choose values for coefficients a, b and c that minimize some measure of error on our training data). The topic of the next few sections is choosing sensible models; Appendix A.1 discusses fitting observations to them.

There is absolutely no theoretical or practical reason why any code's performance must have anything to do with any easily discernible feature of the workload. Thus, if we are to fit code's performance to some model (such as a polynomial function of workload features), we must have some justification for doing so: either some prior belief that programs ought to behave as the model predicts, some hint from the structure of the program itself, or some pattern in the observed data. Simply having low error on the training data is insufficient: fitting a degree 6 polynomial to some program's performance may have reasonably low error, but why should we believe that this model will adequately predict performance for other workloads? Indeed, a degree 7 polynomial is almost guaranteed to produce lower error.

Variable	$n = 10$	$n = 20$	$n = 1000$
self-cost of `swap`	1r23	1r100	1r257808
pure-transitive-cost of `swap`	0r23	0r100	0r257808
transitive-cost of `swap`	1r23	1r100	1r257808
call-count of `swap`	23	100	257808
total-self-cost of `swap`	23	100	257808
total-transitive-cost of `swap`	23	100	257808
`bsort` inner loop count	45	190	499500
`bsort` outer loop count	10	20	1000
`bsort` $\frac{\text{inner}}{\text{outer}}$	4.5	9.5	499.5
total `bsort` inner loop count	45	190	499500
total `bsort` outer loop count	10	20	1000
`bsort` $\frac{\text{total inner}}{\text{total outer}}$	4.5	9.5	499.5
self-cost of `bsort`	56	211	500501
transitive-cost of `bsort` calling `swap`	23	100	257808
call-count of `bsort` calling `swap`	23	100	257808
pure-transitive-cost of `bsort`	23	100	257808
transitive-cost of `bsort`	79	311	758309
call-count of `bsort`	1	1	1
total-self-cost of `bsort`	56	211	500501
total-transitive-cost of `bsort`	79	311	758309
`main` loop count	10	20	1000
total `main` loop count	10	20	1000
self-cost of `main`	12	22	1002
transitive-cost of `main` calling `bsort`	79	311	758309
call-count of `main` calling `bsort`	1	1	1
pure-transitive-cost of `main`	79	311	758309
transitive-cost of `main`	91	333	759311
call-count of `main`	1	1	1
total-self-cost of `main`	12	22	1002
total-transitive-cost of `main`	91	333	759311

Figure 3.4: Data records for the `bsort` example for several workloads. The notation 1r23 is run-length encoding for 1 repeated 23 times.

Thus, there is a crucial question to answer: which models shall we consider and why those models and not others? The next few sections address this question as we describe our technique for choosing a model.

3.4.1 Direct Models

As a starting point, we are willing to entertain the notion that any loop's or function's performance grows as a linear or powerlaw function of some workload feature or invocation feature. Thus, we fit our observations for every variable (y) to linear ($\hat{y}(x) = a + bx$) and powerlaw ($\hat{y}(x) = ax^b$) models of every workload feature and invocation feature (x). We discuss the details of finding the coefficients of regression (a, b) for these models in Appendix A.1. Recall that our finding these coefficients for a model says nothing about the suitability of that model — assuming the model is a true description of the data, the values of a and b that we find minimize some measure of error, but the model may be utter nonsense.

It may be, however, that none of the features that the user provides is particularly useful for predicting performance. To cater to this case we consider a constant model for every variable. The constant model's prediction for a variable's value is just the mean of the observed data. Furthermore, in addition to the models already mentioned, we are also willing to predict the performance of an inner loop as a linear or powerlaw function of its outer loop's performance. A model in terms of an outer loop's count is not as informative as a model in terms of features, but such models can serve to explain several difficult variables in terms of one difficult variable.

We refer to these models as *direct* models since they directly fit performance mea-

surements to features. In the next section we discuss how CF-TrendProf uses evidence from the structure of the program to posit more complex, *derived* models. We discuss the process of *model selection*: choosing the "best" model from among many candidates in Section 3.4.3.

Fitting Workload Features to Per-Invocation Variables

To fit a workload feature (one performance measurement point per workload) to a per-invocation variable (one performance measurement per invocation—zero or more per workload), we pair the workload feature with every measurement for the variable on the corresponding workload. For example if a function's self-cost is measured at $\{5, 6\}$ on a workload with feature $n = 2$ and $\{11, 12, 12\}$ on a workload with feature $n = 4$, then we would treat the data set as $\{(2, 5), (2, 6), (4, 11), (4, 12), (4, 12)\}$ for the purposes of fitting this function's self-cost to workload feature n.

Example

Figure 3.5 shows some of the direct models for some of the variables for our `bsort` example. For brevity we show only the best direct model for each variable in this table. As we will see, some of these direct models are the best model for their variable, but CF-TrendProf replaces others with superior derived models.

3.4.2 Derived Models

As we saw in Chapter 2, direct models can do a reasonable job of predicting performance, but they are not always particularly accurate. This section considers a technique

Variable	Direct Model
self-cost of `swap`	1
call-count of `swap`	$0.2n^2$
total-self-cost of `swap`	$0.2n^2$
`bsort` inner loop count	$0.45n^{2.02}$
`bsort` outer loop count	n
`bsort` $\frac{\text{inner}}{\text{outer}}$	$0.5n - 0.5$
self-cost of `bsort`	$0.56n^{1.98}$
transitive-cost of `bsort` calling `swap`	$0.2n^2$
call-count of `bsort` calling `swap`	$0.2n^2$
pure-transitive-cost of `bsort`	$0.2n^2$
transitive-cost of `bsort`	$0.8n^{1.99}$
call-count of `bsort`	1
`main` loop count	n
self-cost of `main`	$n + 1$
transitive-cost of `main` calling `bsort`	$0.93n^{1.97}$
call-count of `main` calling `bsort`	1
transitive-cost of `main`	$0.93n^{1.97}$
call-count of `main`	1

Figure 3.5: Some direct models for the `bsort` example.

for deriving more precise models based on hints in the program's control flow. For example, if we can model the loops in a function, we can symbolically add these models to form a model for the function's self-cost; similarly if we can model the costs of a function's callees, we can symbolically add these models to form a models for the function's transitive-cost. This general intuition gives rise to several rules for constructing *derived models*.

This way of deriving models based on the program's control flow and call graph has several advantages. Most importantly, it gives us a principled way of considering complex models to explain the performance of complex programs; as the program grows more complex, the models we are willing to consider also grow more complex. Furthermore, generating many models (some simple, some complex) to explain the performance of a variable (for example, a function's self-cost) allows us to trade off model simplicity for model pre-

cision; in essence we only choose complex models when their complexity is paid for with extra precision.

Example

Consider the problem of finding a good model for the self-cost of function `F` with two loops `K` and `L`. Let $(l_1, \ldots, l_n)$, $(k_1, \ldots, k_n)$, and $(y_1, \ldots, y_n)$ be the data points we measured for `L`'s iteration count, `K`'s iteration count, and `F`'s self-cost on all the workloads; let $(w_1, \ldots, w_n)$ be the points for a workload feature in terms of which we'd like to predict `F`'s self-cost. Recall that by our definition of self-cost, $y_i \stackrel{\text{def}}{=} 1 + l_i + k_i$ for all $i \in \{1, \ldots, n\}$; indeed, this formula is how we compute self-cost from our program trace. Now suppose that we have (recursively) computed a best model for `L`, $\hat{L}(w)$, and a best model for `K`, $\hat{K}(w)$. Now we can form a *derived model* for `F`'s self-cost by symbolically adding these models: $\hat{y}(w_i) = 1 + \hat{L}(w_i) + \hat{K}(w_i)$. Thus we have arrived at a model for `F`'s self-cost by combining best models for its sub-components `L` and `K`. As we shall see, this intuition allows to model the self-cost of `bsort` perfectly (see Figure 3.8 and Figure 3.11) as the sum of one, its outer loop's cost, and its inner loop's cost:

$$\text{self-cost of } \texttt{bsort} :\approx 1 + (n) + (0.5n^2 - 0.5n) = 0.5n^2 + 0.5n + 1$$

If `F` had more loops, we would add more terms to our derived model for its self-cost. By construction this algorithm for computing a derived model for `F`'s self-cost results in models whose size is bounded by the number of loops in `F`. In more general terms, the complexity of the model for `F`'s self-cost is bounded by the number of sub-components of `F`'s self-cost. Of course like terms in $\hat{L}(w_i)$ and $\hat{K}(w_i)$ may combine; for example, if $\hat{L}(w_i)$ and

$\hat{K}(w_i)$ are both linear models in some feature w, then the derived model $(1+\hat{L}(w_i)+\hat{K}(w_i))$ is just a linear model in w. In general, though, $\hat{L}$ and $\hat{K}$ may be in terms of different features; for example, we might model `L`'s cost with features v and w and model `K`'s cost with features w and x, thus yielding the following derived model for y (`F`'s self-cost) in terms of all three features: $\hat{y}(v_i, w_i, x_i) = 1 + \hat{L}(v_i, w_i) + \hat{K}(w_i, x_i)$.

The general procedure we use to create derived models is as follows. First, decompose a variable (y_i) into a function of its sub-components ($y_i = 1 + l_i + k_i$). Then model each sub-component separately (find $\hat{L}(v_i, w_i)$ and $\hat{K}(w_i, x_i)$). Finally, reverse the decomposition by symbolically combining these models to yield a model of the variable ($\hat{y}(v_i, w_i, x_i) = 1 + \hat{L}(v_i, w_i) + \hat{K}(w_i, x_i)$).

Rules for Deriving Models

Figures 3.6 and 3.7 list the specific rules CF-TrendProf uses to create derived models. These rules use a different notation than the example above, defined as follows. In these lists of derived models we denote a model for variable v as $[v]$; we denote symbolic addition and multiplication of models for variables x and y with $[x] \oplus [y]$ and $[x] \otimes [y]$ respectively; we indicate that some expression, e over this language is a derived model for v with the notation $[v] \prec e$. We show the derived models for function `F` that calls functions $\{G_1, \ldots, G_m\}$ and has loops $\{\ell_1, \ldots, \ell_k\}$. Each of the variables for which we compute derived models are variables that CF-TrendProf measures directly and for which it constructs direct models (recall that we list these variables in Section 3.3.2). We use only direct models to explain variables that do not appear on the left of any of our derivation rules (for example,

$$[\text{inner loop count}] \prec [\text{outer loop count}] \otimes \left[\frac{\text{inner loop count}}{\text{outer loop count}}\right] \quad (3.1)$$

$$[\text{self-cost of } F] \prec 1 \oplus \bigoplus_{i=1}^{k} [\text{per-invocation cost of } \ell_i] \quad (3.2)$$

$$[\text{transitive-cost of } F] \prec [\text{self-cost of } F] \oplus [\text{pure-transitive-cost of } F] \quad (3.3)$$

$$[\text{total-self-cost of } F] \prec [\text{call-count of } F] \oplus \bigoplus_{i=1}^{k} [\text{per-workload cost of } \ell_i] \quad (3.4)$$

$$[\text{total-self-cost of } F] \prec [\text{call-count of } F] \otimes [\text{self-cost of } F] \quad (3.5)$$

$$[\text{total-transitive-cost of } F] \prec [\text{total-self-cost of } F] \quad (3.6)$$

Figure 3.6: Rules for generating derived models. CF-TrendProf derives these models on its first pass through the functions.

pure-transitive-cost of `F`, outer loop counts, and $\frac{\text{inner loop count}}{\text{outer loop count}}$).

Derived model 3.1 models the cost of an inner loop as a product of the cost of the outer loop and the average number of iterations of the inner loop per iteration of the outer loop. CF-TrendProf records and models these average iteration counts specifically for this derived model. The multiplication in this derived model allows inner loops to have models of higher degree than their outer loops. CF-TrendProf uses this rule for both per-invocation and per-workload loop counts. Notice that our `bsort` example uses this rule (Figure 3.8).

Derived model 3.2 is the one we developed in our example above. The self-cost of a function is one plus the sum of the costs of its loops.

Derived model 3.3 models the transitive-cost of a function by lumping the costs of all its callees into one quantity (pure-transitive-cost) that it models directly. Other derived models for transitive-cost rely on a finer decomposition of sub-components. This derived

$$[\text{transitive-cost of } F \text{ calling } G] \prec [\text{call-count of } G \text{ per invocation of } F] \otimes [\text{transitive-cost of } G] \quad (3.7)$$

$$[\text{transitive-cost of } F] \prec [\text{self-cost}] \oplus \bigoplus_{i=1}^{m} [\text{transitive-cost of } F \text{ calling } G_i] \quad (3.8)$$

$$[\text{total-transitive-cost of } F] \prec [\text{call-count of } F] \otimes [\text{transitive-cost of } F] \quad (3.9)$$

Figure 3.7: Rules for generating derived models. CF-TrendProf derives these models on its second pass through the functions.

model caters to the case where these models for sub-components are not very good; instead of a complex mess, we get one derived model that approximates some complex behavior. Indeed, the other decompositions may not be a useful view onto performance where this one may be. Essentially this model (and others like it) gives the model generation procedure a certain resilience: more complex derived models (that may contain bad sub-models) must compete with simpler derived models like this one and even simpler direct models; we can thus discard useless or counter-productive complex models in favor of simpler ones.

Derived model 3.4 is the per-workload analogue to derived model 3.2. It models total-self-cost of a function as the sum of the function's call-count (corresponding to the 1 in the self-cost's definition) and the per-workload sums of its loops. Derived model 3.5 models the total-self-cost of a function as its call-count times self-cost.

Derived model 3.6 guesses that a function's total-transitive-cost is the same as its total-self-cost. This guess is right if the function transitively makes no calls except to itself. This model goes against the established pattern in that it is more of a guess (that may be

wildly wrong) than a decomposition. Since it will be compared against other models, both direct and derived, this potential inaccuracy is not a particular problem: this model will be chosen as the best only if it is appropriate.

Derived model 3.7 is concerned with modeling the cost of all calls to G during a single invocation of F; it does so by multiplying the number of times F calls G (again per-invocation of F) by the transitive-cost model for G.

Derived model 3.8 decomposes the transitive-cost of a function as its self-cost plus the sum of the contributions of each callee. We model this per-callee term ([transitive-cost of F calling G]) both directly and with derived model 3.7 above. The fact that we also model this variable ([transitive-cost of F calling G]) directly gives this derived model resilience against bad or missing (see Section 3.4.4) transitive-cost models.

Derived model 3.9, analogous to derived model 3.5, models the total-transitive-cost of a function as its call-count times its transitive-cost.

Discussion

To be sure, these rules for generating derived models can create odd-looking models such as $3x^{2.0} + 9x^{1.9}$, or as we see in Section 3.7.2 and Figure 3.43, $0.24n^{2.49} + 30n^{1.73} + 3319n - 22912$. These models are not the standard fare of algorithms textbooks, but they are no less interpretable. Indeed, they convey more information: the terms arise as a consequence of the program's structure and so reflect behavior of inner loops or callees. The former model can only arise if there are two loops (or two callees) whose empirical scalability is similar but not identical; the $9x^{1.9}$ term dominates until $x = 3^{10} \approx 60000$ when the $3x^{2.0}$ term takes over. Furthermore, as we will see in the next section, models

Variable	Derived Model
[`bsort` inner loop count]	[`bsort` outer loop count] $\otimes \left[\texttt{bsort}\ \frac{\text{inner}}{\text{outer}}\right]$
[self-cost of `bsort`]	$1 \oplus$ [`bsort` inner loop count] $\oplus$ [`bsort` outer loop count]
[transitive-cost of `bsort`]	[pure-transitive-cost of `bsort`] $\oplus$ [self-cost of `bsort`]

Figure 3.8: Some derived models for the `bsort` example.

such as the former that consist of a sum of terms with similar exponents must compete against direct powerlaw models; if the extra terms do not add extra precision, we discard the model in favor of a simpler one. The latter model arises based on the behavior of two distinct callees. If we were to "round" the latter model to a cubic polynomial or a powerlaw, we would discard the information that one callee scales as $n^{2.49}$ while the other scales more slowly.

Of course performance need not decompose along the lines of control flow and the decompositions built into our derived models are not guaranteed to be the right ones to understand all programs' performance. However, CF-TrendProf is designed to try many feasible possibilities and can recover from bad decompositions at a higher level of the call tree, at the per-workload rather than per-invocation level, by using derivation rules such as derived models 3.3, 3.8, and 3.4. In any event, CF-TrendProf is an improvement upon any approach that chooses models from a small, a priori bounded family.

Example

Figure 3.8 shows some of the more useful derived models for some of the variables for our `bsort` example. Computing these derived models requires that we have computed the best model for sub-components.

3.4.3 Choosing the Best Model

So far we have seen that CF-TrendProf generates a set of candidate models for every variable. It considers direct models for all variables. Furthermore, it builds derived models for a variable based on the structure of the program and the best models for the variable's subcomponents (such as loops inside a function or callees).

There are two fundamental concerns to balance when choosing models: model precision, as measured by training set error, and model complexity. We would like a model with low error. As we argued before, more complex models with more terms have the potential to decrease error on the training data. However, there is a danger that if we allow our models to grow gratuitously complex, that they will *overfit* the training data and not generalize to other data. Put another way, the principle of Occam's razor says that we should pick a simple model absent evidence for a more complex one. Since any of our candidate models may be nonsense, we insist that any complexity in the model be justified by a sufficient decrease in training set error: derived models must exhibit lower error than direct models which must in turn exhibit lower error than constant models if they are to be chosen as the best model.

For every performance variable, CF-TrendProf chooses a best model from among the direct and derived models it produces by assigning each model a score (smaller is better) based on its standard error (Appendix A.1.5), and its complexity (see below). The model with the lowest score is the best model.

The following formula gives the score for a model, $\hat{y}(x)$, that explains observations from n workloads, $y_1, \ldots, y_n$, in terms of n vectors of k features each

$(x_{1,1}, \ldots, x_{1,k}), \ldots, (x_{n,1}, \ldots, x_{n,k})$; we define $\hat{y}_i$ as the model's estimate on workload i, (that is, $\hat{y}_i \stackrel{\text{def}}{=} \hat{y}(x_{i,1}, \ldots, x_{i,k})$) ; $\overline{y}$ is the sample mean of the observations $\frac{1}{n}\sum_{i=1}^{n} y_i$ and S_y is the standard error of the model $\sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n-2}}$ (see Appendix A.1.5).

$$\ell \stackrel{\text{def}}{=} \begin{cases} 0 & \hat{y}(x) \text{ is a simple linear model} \\ 1 & \text{otherwise} \end{cases}$$

$$V \stackrel{\text{def}}{=} \sum_{i=1}^{k} \begin{cases} 2 & x_i \text{ is a workload feature} \\ 3 & x_i \text{ is an invocation feature} \\ 20 & x_i \text{ is a loop count} \end{cases}$$

$$t \stackrel{\text{def}}{=} 0.01 \times (\text{the number of terms in the model})$$

$$\alpha \stackrel{\text{def}}{=} 1111$$

$$\text{model score} \stackrel{\text{def}}{=} \frac{100 S_y}{\alpha + \overline{y}} + \ell + V + t$$

This formula is ad hoc, but each term has a purpose. The units of the score formula are percentage points of deviation from the mean, $\overline{y}$, of the observations; the idea is that performance deviations in the thousands are serious when overall performance is, on average, in the ten thousands, but not so serious if overall performance is in the billions. The $\alpha > 0$ sets a threshold of performance that is too small to bother modeling precisely; when $\overline{y}$ is small compared to α (that is, performance just does not get very high), we prefer simpler models: the error term matters less and the complexity terms (ℓ, V, and t) matter more. The V term penalizes for extra independent variables; it causes CF-TrendProf to prefer workload features to invocation features and to penalize loop counts so seriously that models that include them must offer dramatic improvements over those with more interpretable

features. The ℓ term chooses a linear fit over other types when the errors are similar. The t term penalizes models with extra terms; its small magnitude means that t acts as a third tie breaker if the other terms are quite similar. In essence, this formula trades off precision (low standard error relative to $\overline{y}$) for model simplicity (fewer terms and features involved, constant is simpler than linear is simpler than powerlaw).

Nonetheless, this formula is mostly about error. The other terms only matter when two models are very close in error or when $\overline{y}$ and error are both very small and the α term causes the error part of the formula to be very small.

Which Models and Why

Earlier (Section 3.4), we posed the question of which models we will consider to explain the performance of a variable and why we are justified in considering them; we can now answer this question as follows. We consider direct (linear and powerlaw) models for every variable; these models are justified by our assumption that they may be reasonable and by the fact that they must compete with a constant model. For variables with more structure, such as a function's self-cost or an inner loop's count, we consider derived models whose complexity is bounded by the structure of what they are modeling; these models are justified by the structure of the program and furthermore by the fact that must compete with direct and constant models. All of the degrees of freedom in our models arise from apparent degrees of freedom in the program and each new term or feature in a model must prove its worth by reducing error sufficiently.

Comparison to Other Ways of Selecting Models

There is no best solution to the problem of model selection in general. A typical first-principles look at fitting [Ric06] and selecting [JB03] models to explain data begins with some assumptions about the nature of the data and the nature of any deviations from the model's prediction. For instance, the exposition of linear regression, adapted to our setting, is as follows. Suppose that we have a set of n workloads, $\{w_1, \ldots, w_n\}$ annotated with workload features $\{x_1, \ldots, x_n\}$ and as we run each workload, w_i, we measure some variable, y_i (for instance, the total-self-cost of function `F`) for each workload. We might consider the hypothesis that the values we observe for y are based on a linear function of x plus some noise that is beyond our model's explanatory power (e_i): $\hat{y_i}(x_i) = a + bx_i + e_i$. If we assume that the e_i are independent, have equal variance, and have some known distribution we can reason about the probability of observing some particular set of data points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ conditioned on the model being accurate. One approach to model selection hypothesizes a model with an extra term, say cx_i^2, and then does statistical tests [Ric06] on the hypothesis that $c = 0$. Brewer [Bre94], for instance, starts his models with many terms, computes coefficients for each term and confidence intervals for the coefficients; if any coefficient's confidence interval contains zero, he drops a term and re-assigns coefficients to those that remain. Alternately we could follow Jaynes's [JB03] Bayesian approach and assume some prior probability distribution for regression coefficients a and b and compare the odds of one model being true (conditioned on data and priors) to another model being true (conditioned on data and priors) — though the math for these comparisons is rather involved for even the case of comparing two simple models.

Unfortunately, it is not clear that our observations have equal variance; in many of our best-fit plots, residuals increase with features. For total-self-costs, the independence assumption seems reasonable, but for self-costs during the same program run, it seems ill-motivated. It is not at all clear what probability distributions to assign to either the errors or the model parameters; the normal distribution (with a mean of zero for the error terms) is standard, but again, residuals plots suggest that the sort of errors our approach must tolerate are anything but normally distributed. For instance, the models in Figures 3.15, 3.18, 3.22, 3.31, 3.32, and 3.43 are useful, but their errors are probably not independent, nor equal variance, nor normally distributed). Figures 3.9 and 3.10 show normal probability plots to evaluate the hypothesis that the residuals of several of CF-TrendProf's models are normally distributed; a detailed discussion of these plots is beyond the scope of this dissertation, but it suffices to understand that the residuals are normally distributed to the extent that the points in the normal probability plots form lines. Simply put, these residuals are not normally distributed.

Neither of these sources discuss a general framework for comparing multiple models. Furthermore, it is not clear how to incorporate, as our approach does, evidence from program structure into the model selection process.

Thus, deriving a model selection algorithm for our setting from statistical first principles would seem to involve several ill-motivated assumptions or difficult decisions about distributions and priors. We have opted instead to use domain knowledge (the structure of the program) and explicit, reasonable assumptions (constant, linear, powerlaw models explain performance) to generate plausible models and a special-purpose model selection

criterion to trade off model simplicity for low model error on training data.

As we have seen, model selection is a difficult problem. We have settled on a reasonable approach, but more work, both theoretical and empirical, remains. In Section 3.10.2 and Section 3.10.3 we consider some future directions for inquiry into selecting models for program performance.

Example

Figure 3.11 shows CF-TrendProf's best models for some of the variables in our `bsort` example. The rightmost "Source" column says "Direct" for direct models or shows the models symbolically combined to form derived models.

3.4.4 Interleaving Computation of Derived Models and Best Models

Derived models (for example, for self-cost) require models for sub-components (for example, loop counts). Since considering k models each for each of n subcomponents would entail considering n^k models, CF-TrendProf considers only the best model for each subcomponent while constructing derived models. Once all the direct and derived models for a variable are known, we can compute the best model for that variable.

In implementing our prototype of CF-TrendProf, we carefully manage the order in which we compute direct fits, derived fits, and best fits for each variable so that all the pieces (best models for other variables) we need are available for constructing derived models and best models with good locality. CF-TrendProf proceeds in two passes. Both passes process one function at a time in reverse topological order of the call graph (that is, leaf functions first and other functions only after all of their callees). When processing a function

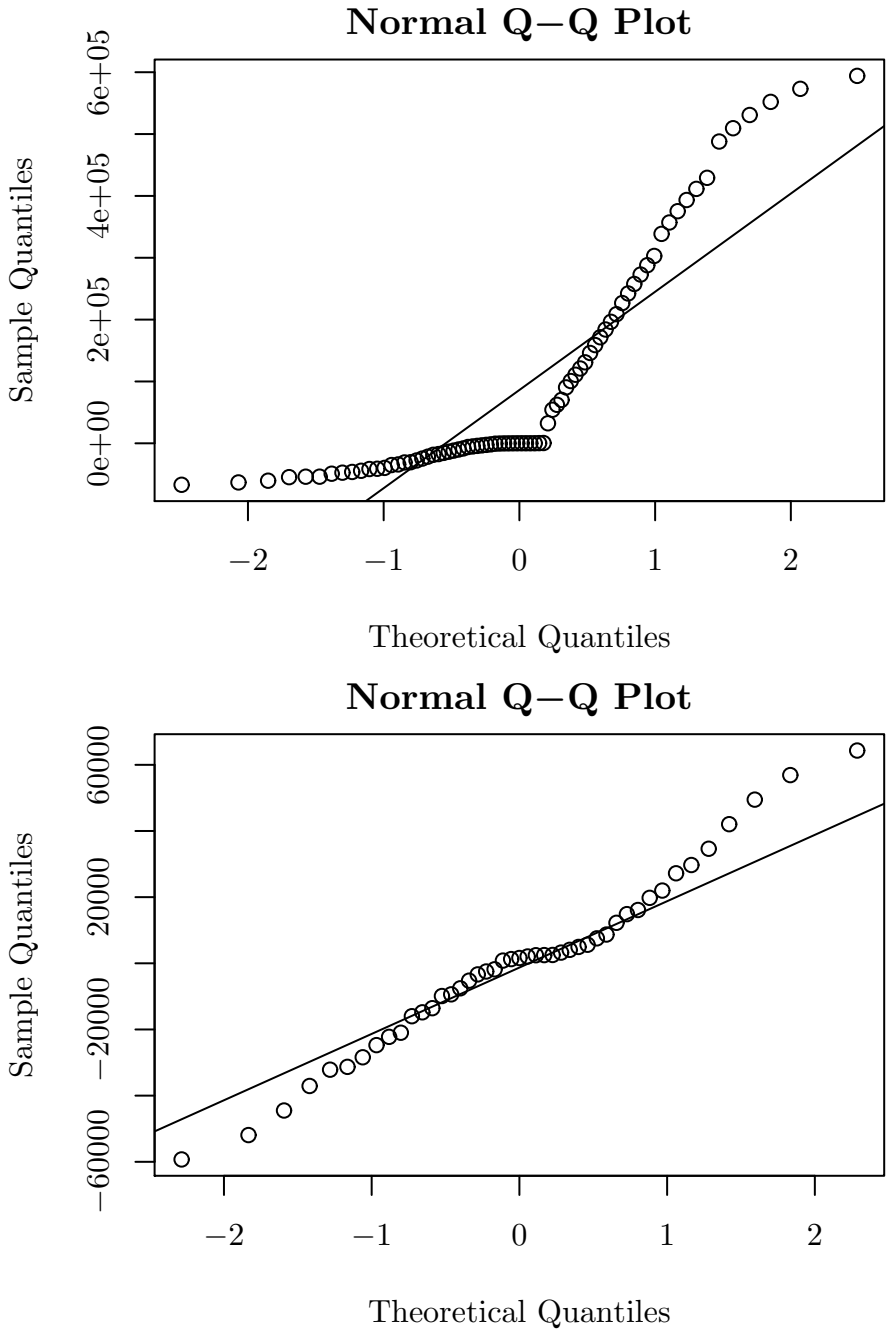

Figure 3.9: Normal probability plots comparing residuals for the models in Figure 3.15 (top) and Figure 3.18 (bottom) to quantiles of the normal distribution. To the extent that the points form a line, the residuals are normally distributed.

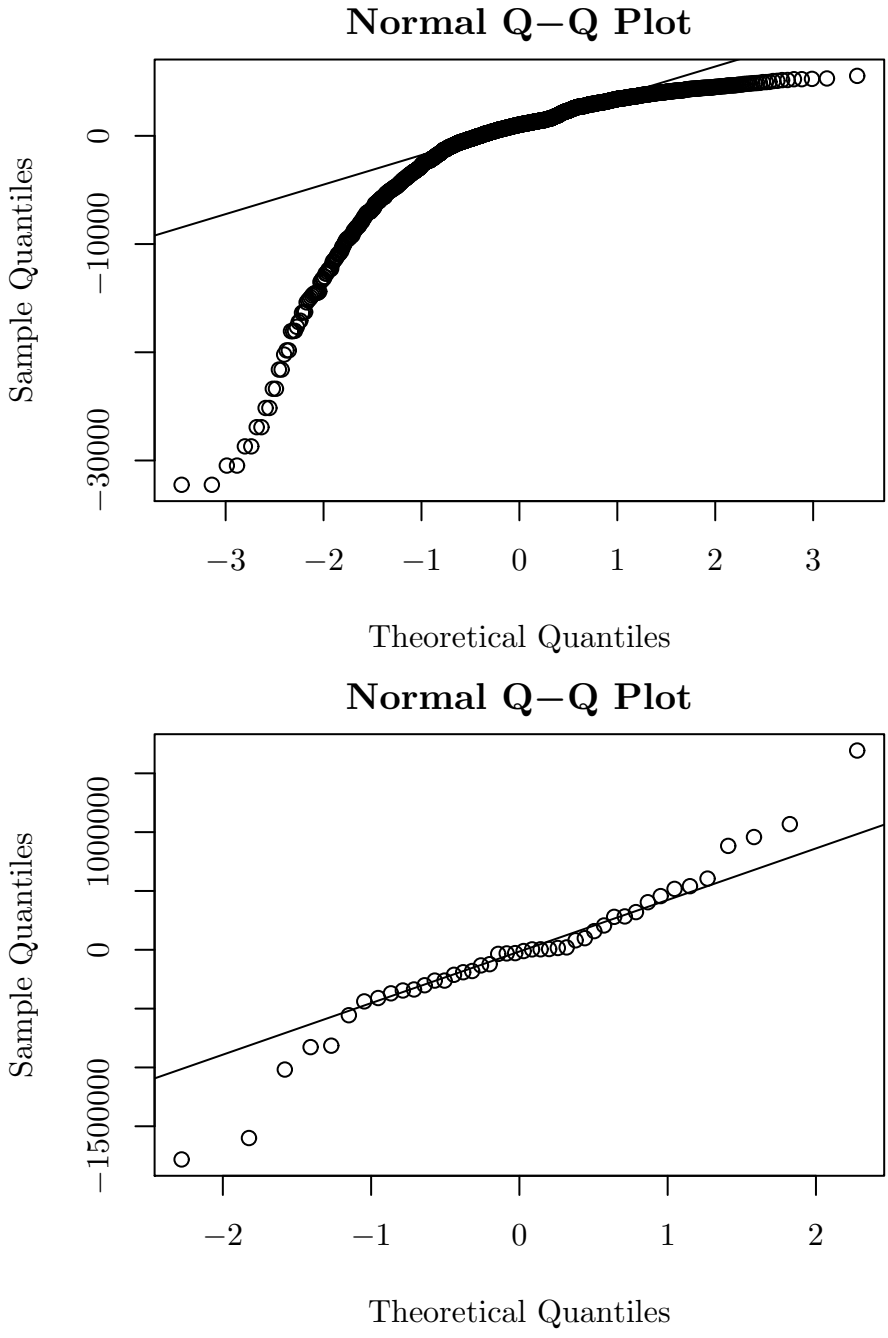

Figure 3.10: Normal probability plots comparing residuals for the models in Figure 3.31 (top) and Figure 3.43 (bottom) to quantiles of the normal distribution. To the extent that the points form a line, the residuals are normally distributed.

Variable	Best Model	Source
self-cost of `swap`	1	Direct
call-count of `swap`	$0.2n^2$	Direct
total-self-cost of `swap`	$0.2n^2$	Direct
`bsort` inner loop count	$0.5n^2 - 0.5n$	$(n) \otimes (0.5n - 0.5)$
`bsort` outer loop count	n	Direct
`bsort` $\frac{\text{inner}}{\text{outer}}$	$0.5n - 0.5$	Direct
self-cost of `bsort`	$0.5n^2 + 0.5n + 1$	$(0.5n^2 - 0.5n) \oplus (n) \oplus 1$
pure-transitive-cost of `bsort`	$0.2n^2$	Direct
transitive-cost of `bsort`	$0.7n^2 + 0.5n + 1$	$(0.5n^2 + 0.5n + 1) \oplus (0.2n^2)$
call-count of `bsort`	1	Direct
`main` loop count	n	Direct
self-cost of `main`	$n + 1$	Direct
transitive-cost of `main`	$0.7n^2 + 1.5n + 2$	$(n + 1) \oplus (1 \otimes (0.7n^2 + 0.5n + 1))$
call-count of `main`	1	Direct

Figure 3.11: Best models for some variables in the `bsort` example.

on the first pass, CF-TrendProf finds the best models in the following order: loop counts (outer loops first), total loop counts, self-costs, transitive costs, call counts, total self-cost, total transitive-cost; this order allows CF-TrendProf to compute all the derived models in Figure 3.6 for each of these variables. On the second pass, CF-TrendProf computes all the derived models in Figure 3.7; these models concern transitive-cost and total-transitive-cost.

The purpose computing best models in two passes instead of one is to handle a particular case that arises in the presence of recursion. Notice derived model 3.7 presupposes best models for the transitive-cost of each of a function's callees. On the first pass through all the variables, we compute a best-so-far model for every function's transitive-cost that encompasses direct models and all the derived models in Figure 3.6. Generally, the reverse topological order in which we process functions ensures that the derived models in Figure 3.6 are computed for callees as well. In the event of a recursive cycle in the call graph, though,

the first function we process from a connected component has only a best-so-far transitive-cost model for any of its callees from that component; in particular, its callees have not yet considered derived models from Figure 3.7.

Handling Recursive Functions

We have found it useful to manually annotate (using `tpRuntimeSetContext`) all recursive functions to be caller-sensitive. Clearly this annotation could be automated. The sensitivity means that CF-TrendProf is modeling the call-count, self-cost and transitive-cost of the initial invocation of the function (or entrance into the recursive component) separately from the subsequent recursive calls.

We emphasize, however, the resilience of derived models 3.7 and 3.8 in Figure 3.7. Suppose, for illustration, that `F` calls `G` and `H`. For every invocation of `F`, recall that CF-TrendProf measures the transitive-cost of `F` accounted for by its calls to `G` and similarly for `H`; CF-TrendProf considers both direct and derived models (derived model 3.7) for this quantity. Thus, even if CF-TrendProf's best model for `G`'s transitive-cost is complicated by recursion (or is otherwise bad), `G`'s cost when called from `F` can be captured with a direct model. Derived model 3.8 takes advantage of this per-callee modeling by deriving a model for `F`'s transitive-cost as the best model for `F`'s self-cost plus the best model for the portion of `F`'s transitive-cost accounted for by `G` plus the best model for the portion of `F`'s transitive-cost accounted for by `H`.

3.4.5 Output

CF-TrendProf's output organizes all of a program's functions according to their

dynamic call graph. It presents the best fits for each function's self-cost, transitive-cost, call-count, total-self-cost, and total-transitive-cost. For each variable, there is a more detailed view showing best-fit scatter plots and residuals scatter plots of all models for the variable (much like the output of BB-TrendProf) with standard errors and a breakdown of their score (used for computing the best fit).

For models that predict a variable in terms of more than one feature, we show a best-fit scatter plot that shows the predictions (x axis) versus the observed performance (y axis); in these plots the fit is good to the extent that the points lie on the line $y = x$. We also show a scatter plot of predictions (x axis) versus residuals (y axis) and a scatter plot for each feature that shows the feature (x axis) versus residuals (y axis).

In order to give a sense of the magnitude of each function's performance contribution, CF-TrendProf lists each function's maximum (over all workloads) total-self-cost and total-transitive-cost; we find that this maximum cost helps put scalability models and standard errors into perspective and is a useful tool for finding the important (and discarding the unimportant) parts of the call graph. Two alternate top-level output pages show all functions sorted by their maximum (over all workloads) total-transitive-cost, and maximum (over all workloads) total-self-cost respectively; these views quickly focus attention on the most expensive subtrees of the call graph and the most expensive functions respectively.

We report two measures of error for each model. The standard error (see Section A.1.5) gives a measure of the absolute magnitude of the deviations of observed values from predicted values; it is similar to the standard deviation, but with the model as a baseline instead of the mean. In order to give a sense of the error of the model relative

to magnitude of the data that it is modeling, we also report the standard error divided by the mean of the observed execution counts; we report this value as a percentage (0 % is a perfect model, 100 % means the standard error is equal to the mean, higher numbers are worse).

3.5 Micro-benchmarks

In this section we demonstrate both the power of CF-TrendProf and some of its limitations in a number of small, but realistic scenarios that focus on its analysis of several well understood algorithms and data structures. In each of these scenarios we have a clear hypothesis about how we expect performance to scale as a function of workload features. We evaluate CF-TrendProf by how clearly it supplies evidence to support or refute these hypotheses.

On a simple nested loop matrix multiply algorithm, CF-TrendProf derives an exact performance function (Section 3.5.1). Tiled matrix multiply is more complex, but CF-TrendProf derives a cubic model for its performance (Section 3.5.2). A look at doubling lists shows how amortized analysis is built in to CF-TrendProf's notions of total-self-cost and total-transitive-cost (Section 3.5.3). Similarly, our hash table benchmark shows how CF-TrendProf's measurement of real workload data serves the same purpose as an expected case analysis, not over a careful constructed theoretical distribution, but over the distribution of workloads that the user provides (Section 3.5.4). Insertion sort's cost and scalability depends on a deeper property of its input than its size (Section 3.5.5); this example illustrates a mixed blessing of the TrendProf technique that recurs throughout

this thesis: TrendProf's models intimately depend on the implementation's behavior given the empirical distribution of workloads that the user provides.

Establishing the scalability of Dijkstra's algorithm using a Fibonacci heap as a priority queue requires sophisticated analysis, but CF-TrendProf's models are a good approximation—they approximate a $O(n \log n)$ factor with a linear model (Section 3.5.7). Our quicksort benchmark (Section 3.5.6) combines several ideas from above: it has a complex theoretical performance analysis, CF-TrendProf approximates its expected case $O(n \log n)$ complexity with a linear model, and its performance ultimately depends on a deeper property than the size of its input. We also consider the problem of a quicksort algorithm with a deterministic choice of pivot and how CF-TrendProf finds the scalability problems this coding error can cause (Section 3.6.1) on certain distributions of workloads.

We show that CF-TrendProf is useful in diagnosing improperly implemented or improperly used hash tables. We consider the case of a subtly bad hash function (Section 3.6.2) and that of an overfull hash table (Section 3.6.3), two situations that are not hard to imagine occurring in the wild.

3.5.1 An Exact Bound for Square Matrix Multiply

Figure 3.12 shows code for multiplying two n by n square matrices, `A` and `B`, and storing the result in a third n by n square matrix, `C`. A workload consists of values for `A` and `B`. We specify n as the only workload feature. This code is clearly $\Theta\left(n^3\right)$.

CF-TrendProf derives the exact performance function, $n^3 + n^2 + n + 1$, for the `matmult` function. Figure 3.13 shows the best-fit plot (top) and residuals plot (bottom) for CF-TrendProf's model of the `matmult` function's cost. The residuals plot shows that

```
// C = A * B
void matmult(int *A, int *B, int *C, int n) {
    int i,j,k;
    memset(C, 0, n*n);
    for (i=0; i<n; ++i)
        for (j=0; j<n; ++j)
            for (k=0; k<n; ++k)
                C[n*i+j] += A[n*i+k] * B[n*k+j];
    return;
}
```

Figure 3.12: Code for matrix multiply.

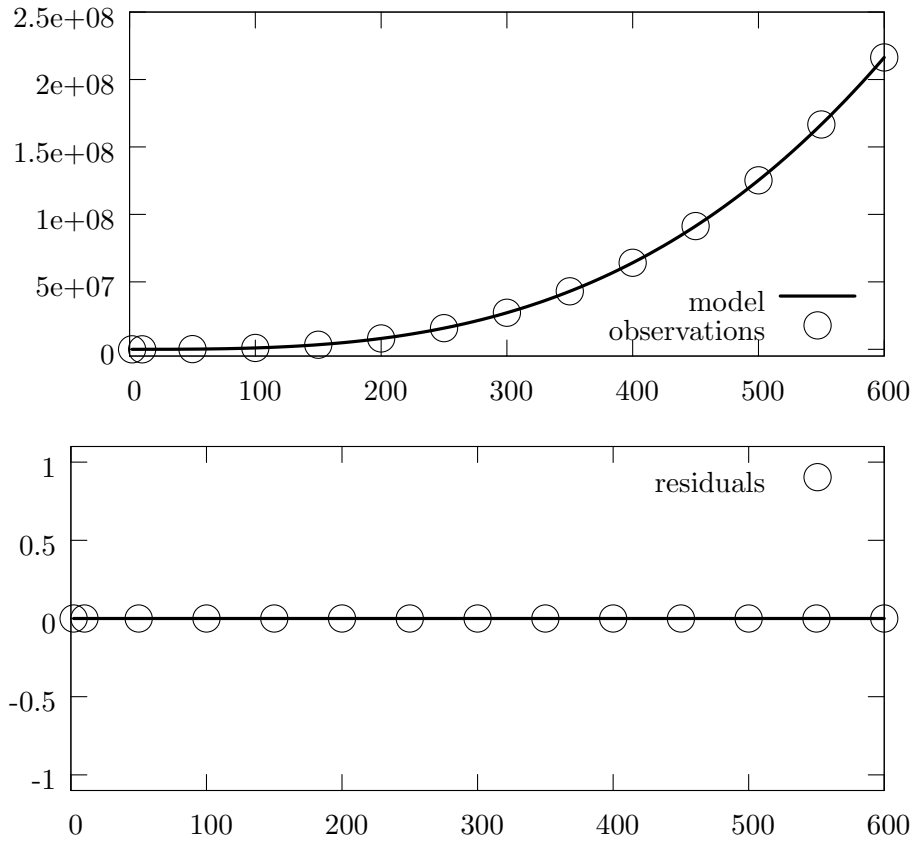

Figure 3.13: The best-fit plot (top) and residuals plot (bottom) for the cost of the `matmult` function for matrix multiplication. These plots show a perfect fit (self-cost := $n^3 + n^2 + n + 1$).

CF-TrendProf's model is a perfect fit to the observed performance.

This benchmark shows an ideal case for CF-TrendProf. Each loop's cost depends only on its outer loop and known workload features. To the extent that we can isolate contexts where loops cost roughly the same every time and provide workload features or invocation features that explain these costs, CF-TrendProf explains performance well.

3.5.2 Tiled Matrix Multiply is Cubic

Figure 3.14 shows an implementation of tiled n by n square matrix multiply. A workload consists of two n by n matrices, `A` and `B`, which `tmatmult` multiplies, yielding a third n by n matrix, `C`. We specify `n` as a workload features.

The complexity of this code is still $\Theta\left(n^3\right)$, but the exact performance cost of this code is more difficult to characterize than that of the simple matrix multiply code we saw previously. The gist of the algorithm is to compute the matrix multiplication by breaking `A` and `B` into `TileSize` by `TileSize` tiles and considering one pair of tiles at a time (resulting in better cache locality). The details are tedious, but for our purposes it suffices to understand that when the size of the matrix is not an integer multiple of the size of the tiles, there are fragments on the bottom and right of the matrix that do not fill a tile; for these fragments, at least one of `M`, `N`, and `K` is less than `TileSize` and the inner three loops have a different cost than the usual case when all three are equal to `TileSize`.

```
// C += A * B
void tmatmult (int n, int *A, int *B, int *C) {
    int ntiles = n / TileSize + (n%TileSize?  1 :  0);
    int bi, bj, bk;

    for (bi = 0; bi < ntiles; ++bi) {
        int i = bi * TileSize;

        for (bj = 0; bj < ntiles; ++bj) {
            int j = bj * TileSize;

            for (bk = 0; bk < ntiles; ++bk) {
                int k = bk * TileSize;
                int M = (i+TileSize > n?  n-i :  TileSize);
                int N = (j+TileSize > n?  n-j :  TileSize);
                int K = (k+TileSize > n?  n-k :  TileSize);

                int* AA = A + i + k*n;
                int* BB = B + k + j*n;
                int* CC = C + i + j*n;
                int ii, jj, kk;

                for (ii = 0; ii < M; ++ii) {
                    for (jj = 0; jj < N; ++jj) {
                        int ciijj = *(CC + jj*n + ii);

                        for (kk = 0; kk < K; ++kk) {
                            int aa = *(AA + ii + kk*n);
                            int bb = *(BB + jj*n + kk);
                            ciijj += aa * bb;
                        } //kk

                        *(CC + jj*n + ii) = ciijj;
                    } //jj
                } //ii
            } //bk
        } //bj
    } //bi
}
```

Figure 3.14: Code for tiled matrix multiply. The details of the traversal of `A`, `B`, and `C` are tedious, but fortunately unimportant for our discussion.

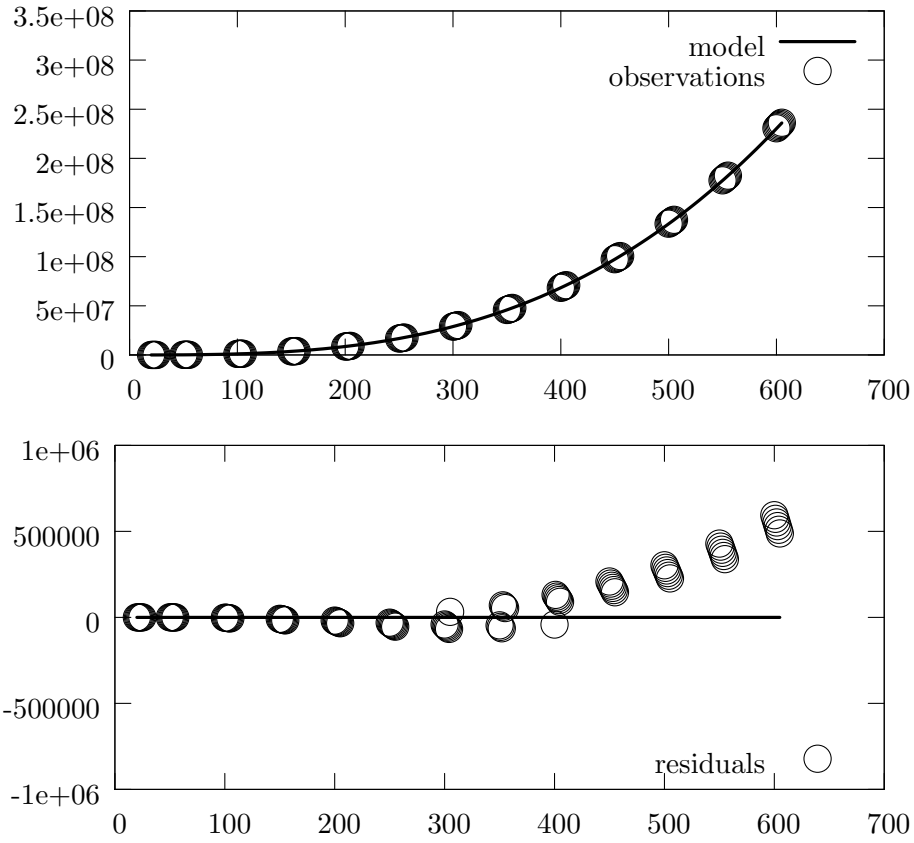

Figure 3.15: The best-fit plot (top) and residuals plot (bottom) for tiled matrix multiply as implemented in the `tmatmult` function (self-cost :$\approx n^3 + 0.11 * n^{2.91} + 0.10n^2 + 0.97n + 4.2$, SE $= 2.11 \times 10^5$). Notice that the scale of the residuals is substantially less than that of the performance.

CF-TrendProf arrives at a good approximation of `tmatmult`'s cost, $n^3 + 0.11 * n^{2.91} + 0.10n^2 + 0.97n + 4.2$. Figure 3.15 shows the best-fit plot (top) and residuals plot (bottom) for CF-TrendProf's model of `tmatmult`'s cost. The standard error (2.11×10^5) and residuals are small relative to the overall cost of the function. The fit is not exact (indeed it seems to have missed a quadratic term), but it is a good approximation that gets the overall cubic scalability right. This derived model beats the less precise, but simpler powerlaw model, $1.1n^{2.99}$ standard error $= 2.84 \times 10^5$. Despite the fact that the loops in `tmatmult` do not always iterate the same number of times, CF-TrendProf finds the performance trend.

3.5.3 Amortized Analysis of Doubling Lists

Perhaps the simplest example of amortized analysis is in the analysis of a doubling array list. Figure 3.16 shows an example implementation. Additions to the list usually take constant-time, but every once in a while the entire list must be moved to a larger array. A workload for this benchmark consists of n integers to pass to `insert`; n is the only workload feature. By an amortized analysis [CLR90], the insert operation takes amortized constant time; that is, n inserts take $O(n)$ time.

The self-cost of `insert`, shown in Figure 3.17 varies even at the same input size; there is some pattern, but the overall trend is unclear. However, CF-TrendProf predicts growth linear in n for the total-self-cost of `insert`: $4n - 2600$ (SE = 26900) for n inserts. Figure 3.18 shows the best-fit and residuals plot for this model.

It is not unequivocally clear from the best-fit plot that this implementation is amortized constant, but CF-TrendProf supplies evidence to support this hypothesis: the

```
int *list = NULL;
int capacity = 0;
int size = 0;

void insert(int e) {
    if (size == capacity) {
        capacity *= 2;
        int *newlist = (int*)malloc(capacity * sizeof(*list));
        int i=0;
        for (i=0; i<size; ++i) newlist[i] = list[i];
        for (; i<capacity; ++i) newlist[i] = 0;
        free(list);
        list = newlist;
    }
    assert(size < capacity);
    list[size] = e;
    size += 1;
}
```

Figure 3.16: Source code for a simple doubling list.

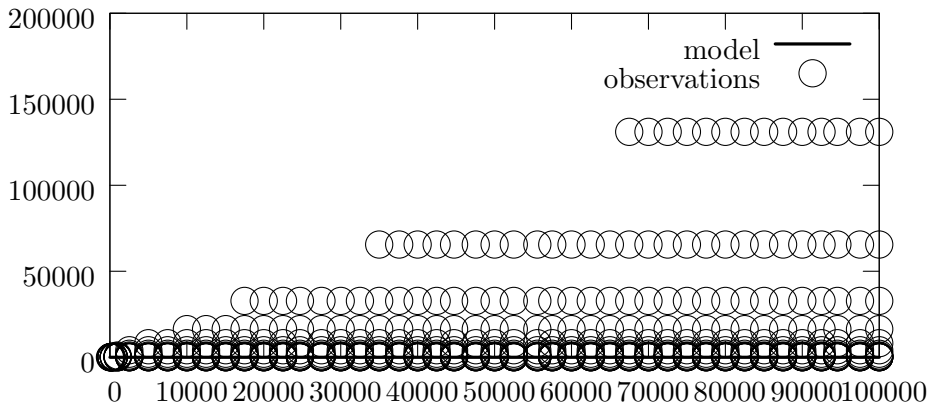

Figure 3.17: The best-fit plot for `insert`'s self-cost $:\approx 4$, SE $= 445$. We omit the residuals plot.

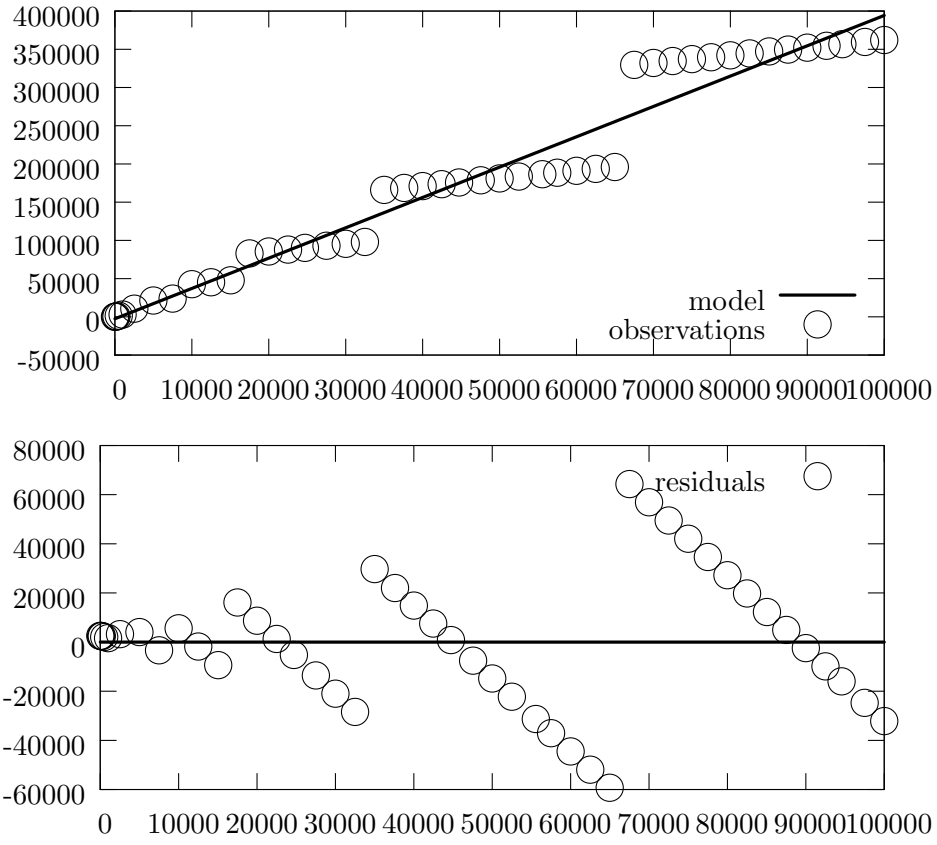

Figure 3.18: The best-fit plot (top) and residuals plot (bottom) for `insert`'s total-self-cost $:\approx 4n - 2600$, SE $= 26900$.

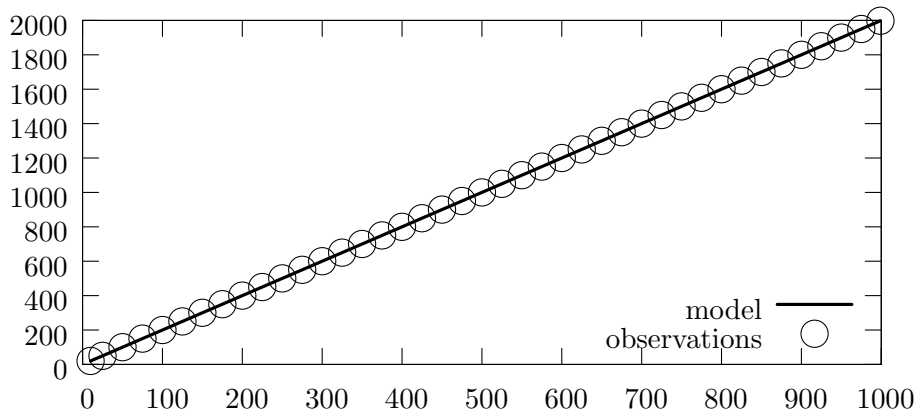

Figure 3.19: The total-self-cost := $2n$ of looking up in a chaining hash table each of the n items spread perfectly uniformly among its 2000 buckets. We omit residuals plots for this perfect fit.

total-self-cost for n averages out to a convincing linear trend and the best powerlaw fit has exponent 1.08. In this situation, the availability of the scatter plot and the fact the trend in the data averages out to linear growth is a great help in exploring the theoretical argument. By building models for functions' total-self-cost and total-transitive-cost, CF-TrendProf is essentially reporting their costs amortized over an entire workload. This sort of amortized analysis is crucial in reasoning about the theoretical computational complexity of many algorithms.

3.5.4 Empirical Performance of a Hash Table

For the purposes of this experiment, we consider a hash table with 2000 buckets that resolves collisions with chaining. A workload consists of n insertions and n lookups, one for each element that we insert. The only workload feature is n. For this experiment, we restrict our attention to $n < 1000$. Since our purpose with this experiment is to study the performance of hash table operations given hash functions with various properties, we

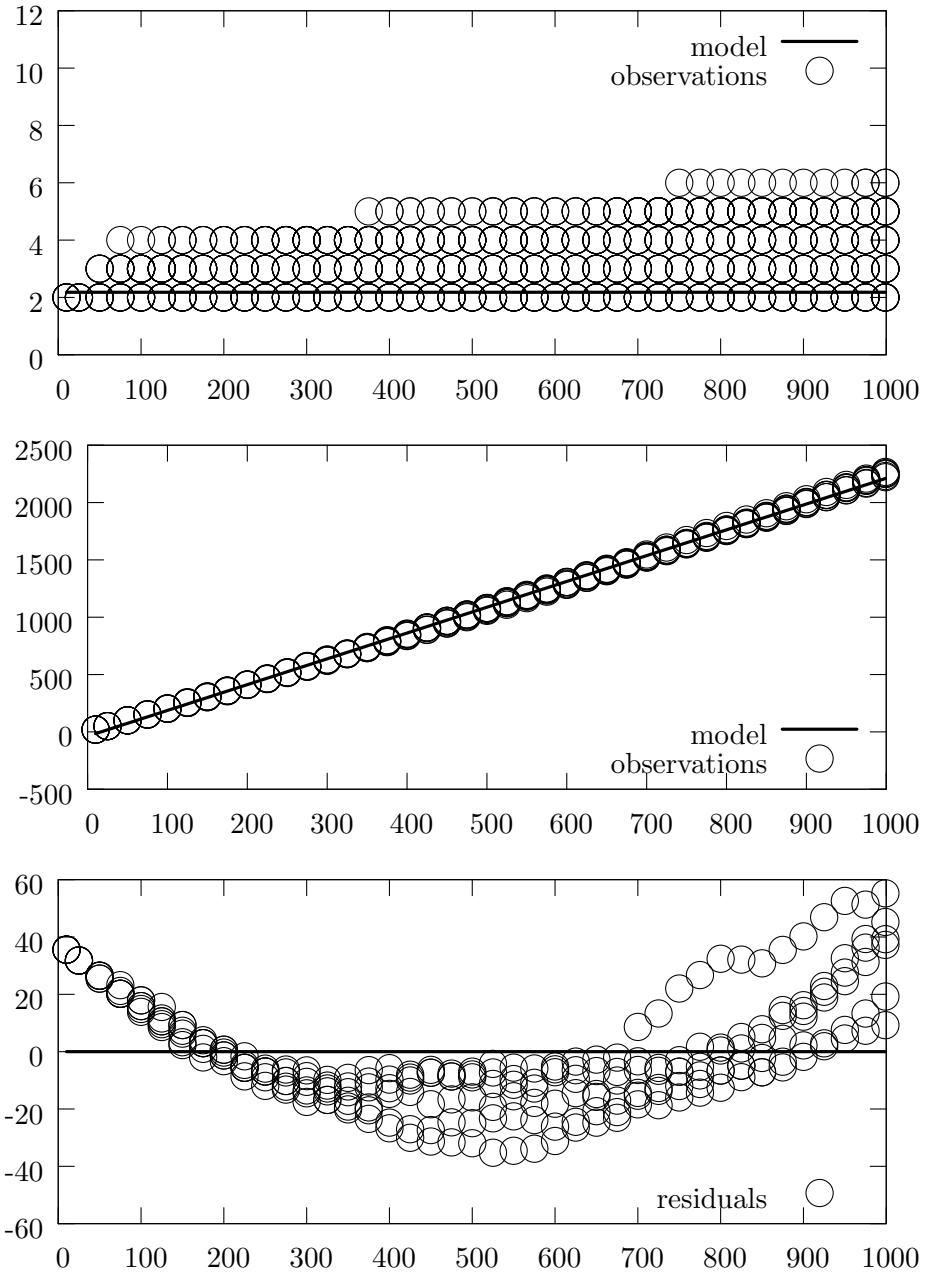

Figure 3.20: The self-cost $:\approx 2$, SE $= 0$ (top), total-self-cost $:\approx 2.25n - 38$, SE $= 19$ (middle), and total-self-cost residuals (bottom) of looking up in a chaining hash table each of the n items spread among its 2000 buckets by a uniform random hash function.

actually use the identity function as our hash function and generate inputs as if they had been hashed from a function with the desired properties.

A hash table with a good hash function provides constant time lookup in the expected case. In this section we examine CF-TrendProf's analysis of a hash table storing perfectly uniformly spread data (an ideal case) and data generated uniformly at random (a more reasonable ideal). In either case, we expect lookups to be constant on average.

There are no surprises: lookup is constant in both cases. Figure 3.19 shows the best-fit plot for the lookup operation's total-self-cost; the models predict performance exactly: the lookup operation has a self-cost of 2 and a total-self-cost of $2n$. Figure 3.20 shows CF-TrendProf's analysis of the lookup operation's cost in the case of uniform-randomly distributed items. The self-cost is still constant, though with an upward tendency as the hash table becomes more full. The total-self-cost averages out to a linear growth in n, but it has a slight, but undeniable upward bend as the hash table gets full.

In this micro-benchmark, CF-TrendProf shows a hash table behaving as it ought to. Theoretical analysis of hash tables requires one to reason about the expected case: assuming the hash function distributes keys uniformly at random (and the load factor is small), a hash table provides constant time lookup in the expected case. CF-TrendProf's models encompass this sort of reasoning automatically: they reflect the average over the empirical distribution of workloads.

As long as the load factor of the hash table is reasonably below 1, the lookup tends to be constant time. In a subsequent micro-benchmark we consider the performance

```
// sort least to greatest
void isort(int n, int *arr) {
  int i=0;
  while (i < n) {
    // arr[0..i-1] is sorted least to greatest
    int val = arr[i];
    int j = i-1;
    // shift anything greater than val up by one position
    while (j >= 0 && arr[j] > val) {
      arr[j+1] = arr[j];
      --j;
    }
    // put val into the gap left by the shifting
    arr[j+1] = val;
    ++i;
  }
}
```

Figure 3.21: Code for insertion sort.

of misbehaving hash tables. As we shall see, a less than ideal hash function or a full table leads to measurable performance degradation.

3.5.5 Insertion Sort's Cost Depends on More Than Input Size

Consider the code for insertion sort shown in Figure 3.21. A workload for this micro-benchmark consists of an array of n integers to sort; n is the only workload feature.

While the outer loop always goes around exactly n times, the inner loop's cost depends on the sortedness of the array. If we consider only arrays that are already sorted or nearly so, the inner loop goes around a constant number of times and the cost of insertion sort scales linearly in n. In contrast, if we consider arrays that are sorted in reverse or are permuted at random, the cost of insertion sort scales quadratically in n. CF-TrendProf's

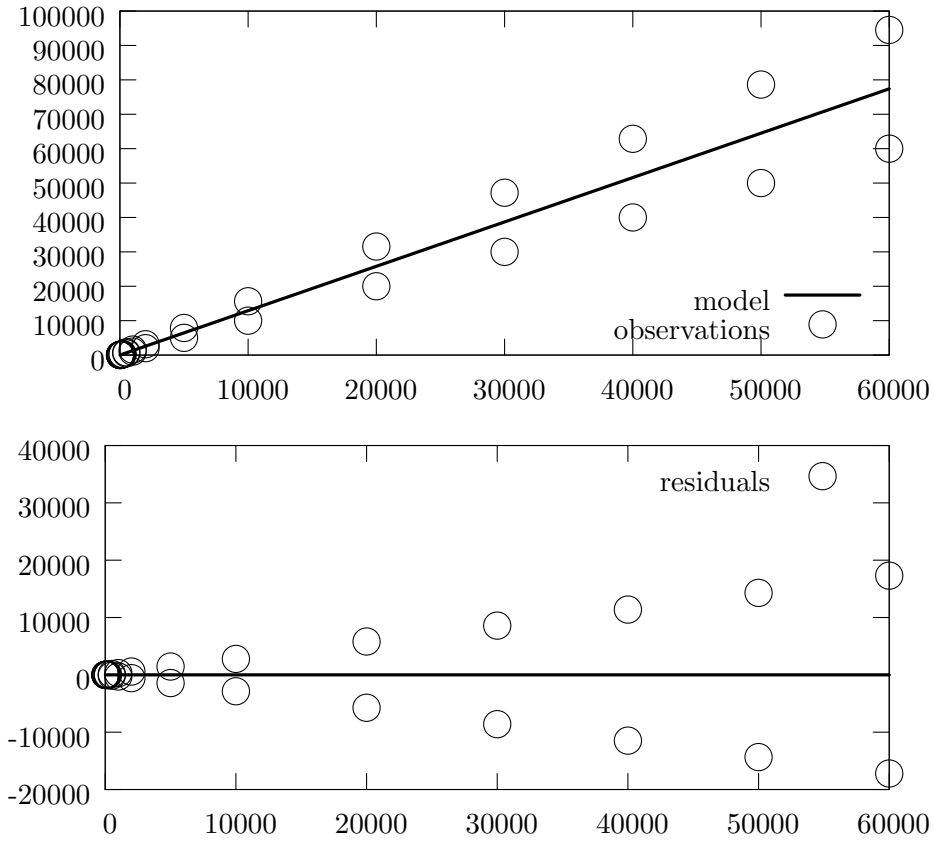

Figure 3.22: Best-fit plot (top) and residuals plot (bottom) for CF-TrendProf's model for `isort` self-cost $:\approx 1.3n$, SE $= 7330$ on sorted and nearly sorted inputs.

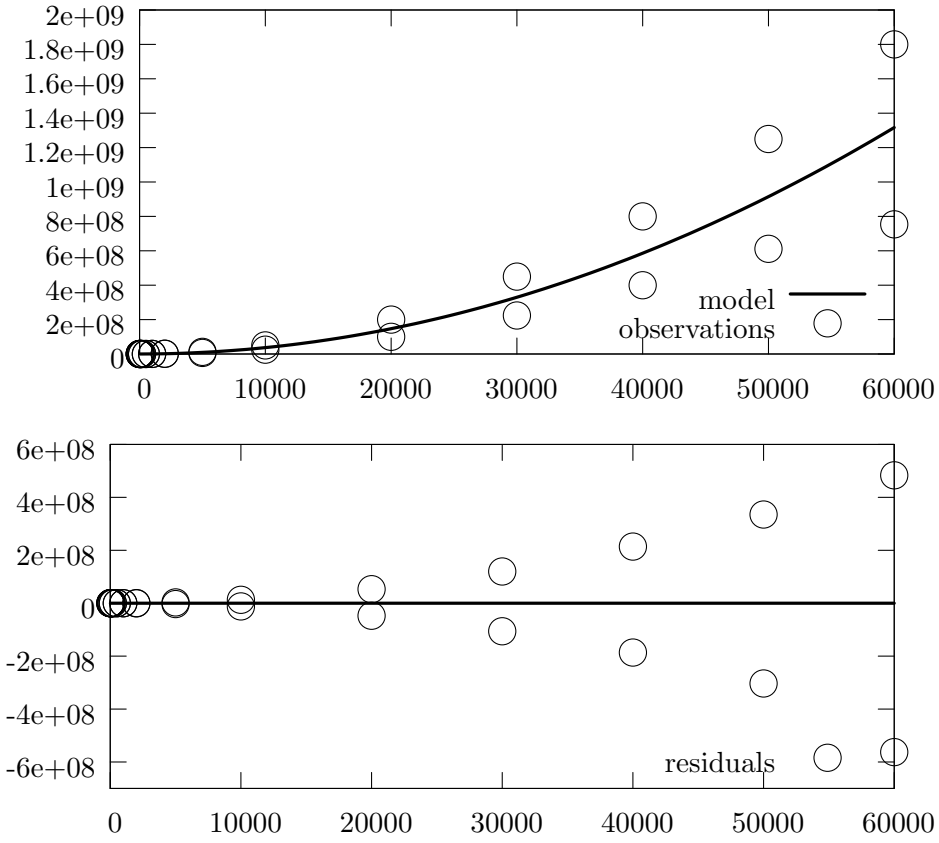

Figure 3.23: Best-fit plot (top) and residuals plot (bottom) for CF-TrendProf's model for `isort` self-cost $:\approx 0.36n^2 + 58n + 1$, SE $= 1.76 \times 10^8$ on random and reverse-sorted inputs.

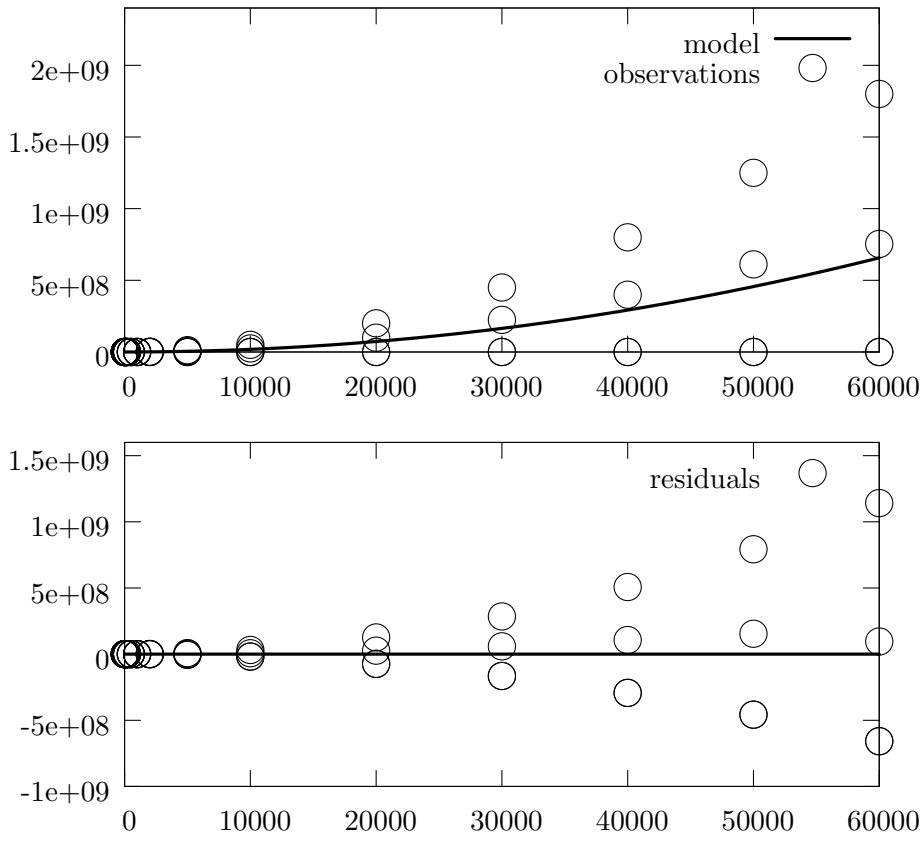

Figure 3.24: Best-fit plot (top) and residuals plot (bottom) for CF-TrendProf's model for `isort` $:\approx 0.18n^2 + 30n + 1$, SE $= 2.58 \times 10^8$ on sorted, nearly sorted, random, and reverse-sorted inputs.

results confirm this analysis.

Figure 3.22 shows CF-TrendProf's best fit plot and residuals plot for insertion sort on workloads (arrays of n integers) that are either sorted or have been sorted and had n adjacent elements swapped with each other (i.e., are nearly sorted). As predicted, CF-TrendProf's models show that `isort`'s cost scales linearly in n when run on these workloads. The upper line of points on the scatter plot corresponds to the nearly sorted inputs while the lower line corresponds to the sorted ones. It is not hard to imagine a distribution of workloads that would cause the scatter plot to fill out with points between the lower and upper lines of data points.

Figure 3.23 shows CF-TrendProf's best-fit plot and residuals plot for `isort` on workloads that are either permuted randomly or sorted in reverse order. Again, CF-TrendProf's models agree with our predictions: they show that `isort` scales quadratically on these workloads. Here the upper line of points are the reverse sorted workloads, the worst case for this code, and the lower line of points are the random workloads. Again, it is not hard to imagine this graph filled out with more data, nor unreasonable to conclude quadratic growth from it.

Obviously, the performance of `isort` on the union of these sets of workloads varies considerably. Figure 3.24 shows the models and scatter plots that CF-TrendProf computes for this situation. As always, CF-TrendProf reports the empirical average scaling behavior; in this case, the expensive workloads push the model toward quadratic.

This micro-benchmark demonstrates that the cost of a function on a particular workload can depend on very subtle properties of the workload, such as sortedness of the

array in this example. Indeed, it may be difficult to measure such properties without essentially running the function on it. This dependence of performance on such subtle properties means that the apparent scalability of an algorithm that CF-TrendProf measures is as much a consequence of the code being measured, as it is of the empirical distribution of workloads. This issue is one we see many times in this dissertation and it is both one of the greatest advantages of using TrendProf (Section 2.4.4, Section 2.4.7) and one of the biggest challenges to overcome in designing TrendProf (Section 3.7.4).

3.5.6 Approximating the Cost of Quicksort

Figure 3.25 shows the code for our quicksort benchmark [Lam]. The `qsort` function takes arrays of integers and sorts them using the quicksort algorithm. A workloads for this benchmark is an array of n randomly generated integers. The only workload feature is n; it ranges from ten to one hundred thousand. For this benchmark, we add a context annotation that distinguishes the first call to `qsort` from recursive calls.

The theoretical analysis of quicksort's performance is difficult [CLR90]. Like insertion sort, quicksort has a worst case complexity of $O\left(n^2\right)$ operations for sorting an array of n integers. In the expected case (over the uniform distribution of all permutations of the input array), though, it scales as $O\left(n \log n\right)$.

Figure 3.26 shows observations and CF-TrendProf's best model for the transitive-cost of the `qsort` function. For larger workloads (high n), there is more variation in performance: performance generally increases as a function of n, but the relationship is not exact. The model CF-TrendProf chooses is a linear one, $18.4n - 43000$, but the U-shaped residuals suggest some systematic error as if CF-TrendProf missed a factor.

```
void quickSort(int arraysize, int numbers[]) {
    qsort(numbers, 0, arraysize - 1);
}


void qsort(int numbers[], int left, int right) {
    int pivot, lhold, rhold;

    tpRuntimeSetContext(...);
    lhold = left;
    rhold = right;
    pivot = numbers[left]; // sub-optimal deterministic pivot choice
    while (left < right) {
        while ((numbers[right] >= pivot) && (left < right)) right--;
        if (left != right) {
            numbers[left] = numbers[right];
            left++;
        }
        while ((numbers[left] <= pivot) && (left < right)) left++;
        if (left != right) {
            numbers[right] = numbers[left];
            right--;
        }
    }
    numbers[left] = pivot;
    pivot = left;
    left = lhold;
    right = rhold;
    if (left < pivot) qsort(numbers, left, pivot-1);
    if (right > pivot) qsort(numbers, pivot+1, right);
}
```

Figure 3.25: Code for our quicksort micro-benchmark [Lam].

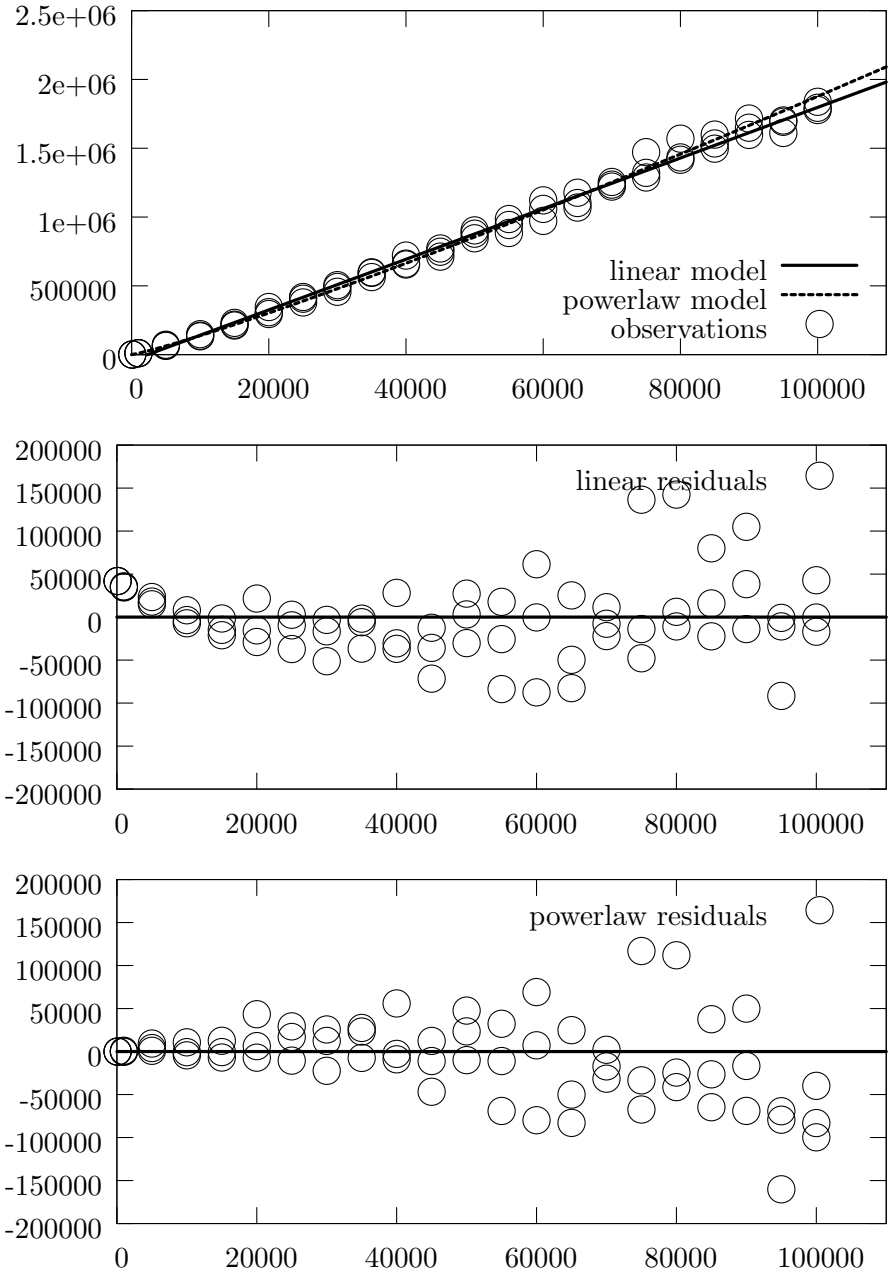

Figure 3.26: Scatter plot of the transitive-cost of initial, non-recursive call to `qsort` (y axis) versus n (x axis); the dotted line shows the best powerlaw model (transitive-cost $:\approx 4n^{1.13}$, SE $= 47900$) while the solid line shows the best model overall (transitive-cost $:\approx 18.4n - 43000$, SE $= 45300$) (top). The middle and bottom plots show the residuals (y axis) versus n (x axis) for the best (linear) model and the best powerlaw model respectively.

The powerlaw model, $4n^{1.13}$ is also shown in Figure 3.26. It has a slightly higher standard error (47900 instead of 45300), but the residuals have a more triangular shape: the model curves upward with the data. Both models approximate the noisy performance relationship reasonably.

It is likely that a model in terms of $n \log n$ term would predict `qsort`'s performance even better. If we were to include such models in CF-TrendProf's repertoire, however, it would try them for all variables. Because of the similarity between linear trends and $n \log n$ trends, noisy linear trends might be classified as $n \log n$ and vice versa. The trade-off we have taken with CF-TrendProf is towards a small number of possible models each of which is likely to convey useful information even if it is not a perfect fit. Inclusion of other families of models is worthy of further investigation, but it is not an obvious win.

3.5.7 Dijkstra's Algorithm Using a Fibonacci Heap

For this experiment we consider an implementation of Dijkstra's algorithm using a Fibonacci heap as a priority queue. Figure 3.27 shows pseudo-code [Sau]. A workload consists of a randomly generated sparse, connected graph with n ($n \in \{20 \ldots 1000\}$) nodes and e edges ($e \in \{n, \ldots, 5n\}$) on which we run Dijkstra's algorithm to find the shortest path from an arbitrarily chosen start node to all other nodes in the graph. We generate the graph by first generating a cycle with the n nodes (involving n edges) and then choosing $e - n$ more edges uniformly at random from all those that are possible; thus every node is reachable from every other node. All edges have randomly chosen, positive weight.

```
void dijkstra(Graph g, Node n0) {
    FiboHeap heap;

    foreach node in g.nodes {
        node.state = unseen;
        node.dist = infinity;
    }

    /* place n0 into the frontier set with a distance of zero */
    n0.dist = 0;
    heap.insert(n0, 0);
    n0.state = seen;

    while(n = heap.deleteMin()) {
        n.state = done;
        foreach edge in n.succs {
            w = edge.target;
            if(w.state != done) {
                dist = v.dist + edge.dist;
                if(dist < w.dist) {
                    w.dist = dist;
                    if(w.state == seen) {
                        heap.decreaseKey(w, dist);
                    } else {
                        heap.insert(w, dist);
                        w.state = seen;
                    }
                }
            }
        }
    }
}
```

Figure 3.27: Pseudo-code for Dijkstra's algorithm [Sau].

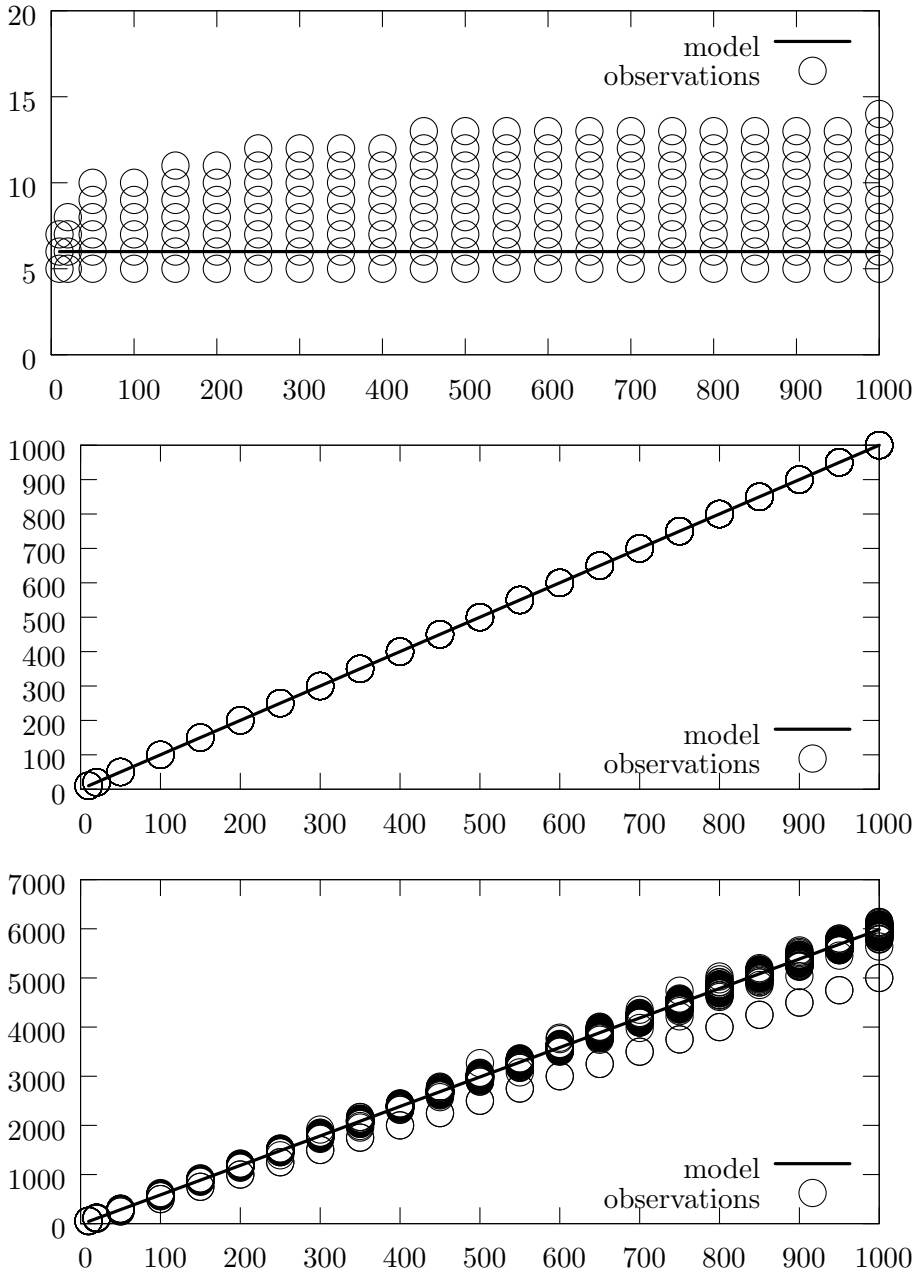

Figure 3.28: Best fit plots for the Fibonacci heap's `insert` operation during Dijkstra's algorithm. transitive-cost $:\approx 6.0$, SE $= 1$ (top), call-count $:= n$ (middle), total-transitive-cost $:\approx 6.0n - 15$, SE $= 99$ (bottom).

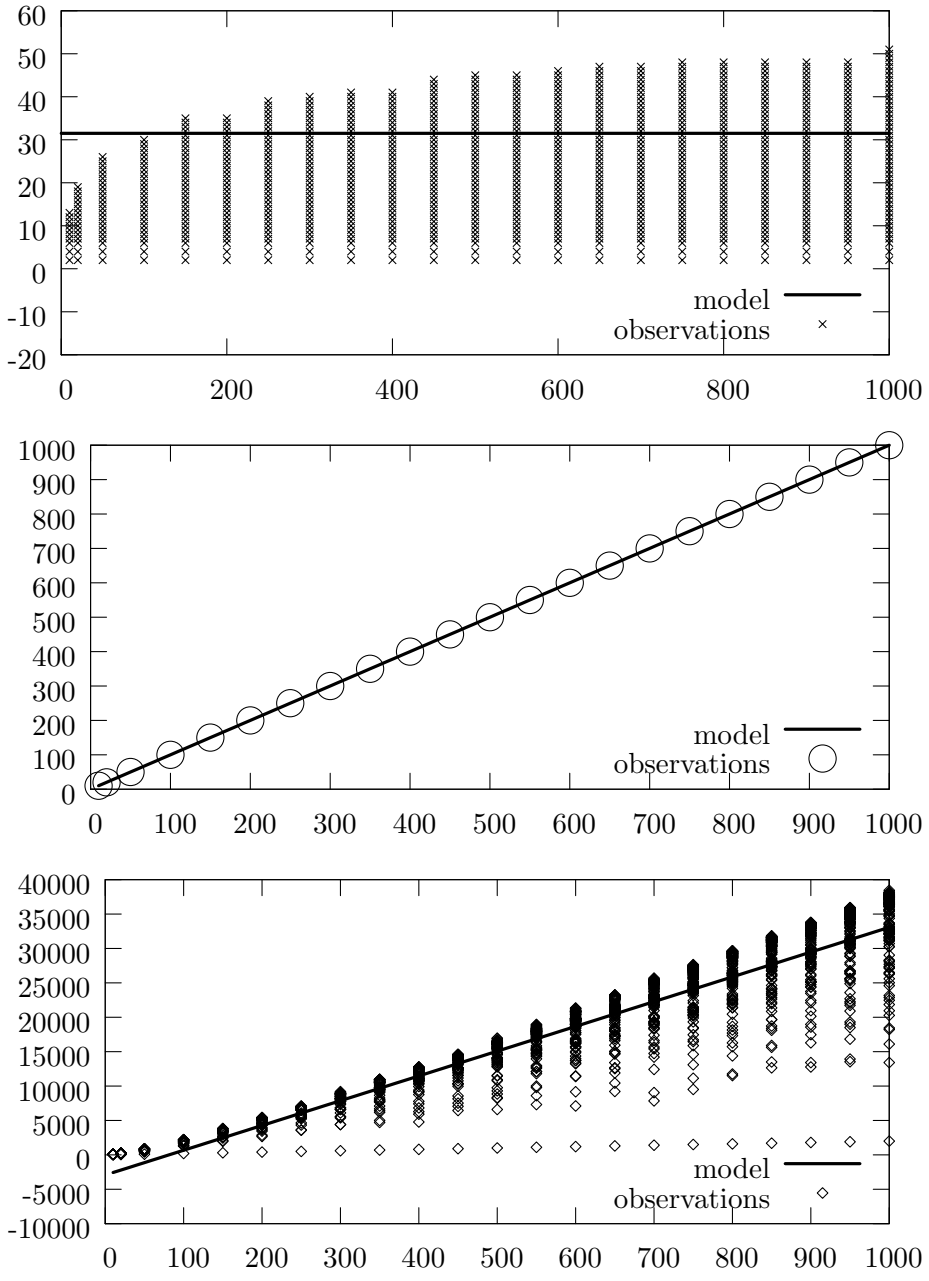

Figure 3.29: Best fit plots for the Fibonacci heap's `deleteMin` operation during Dijkstra's algorithm. transitive-cost $:\approx 32$, SE $= 9$ (top), call-count $:= n$ (middle), total-transitive-cost $:\approx 36n - 2900$, SE $= 4370$ (bottom).

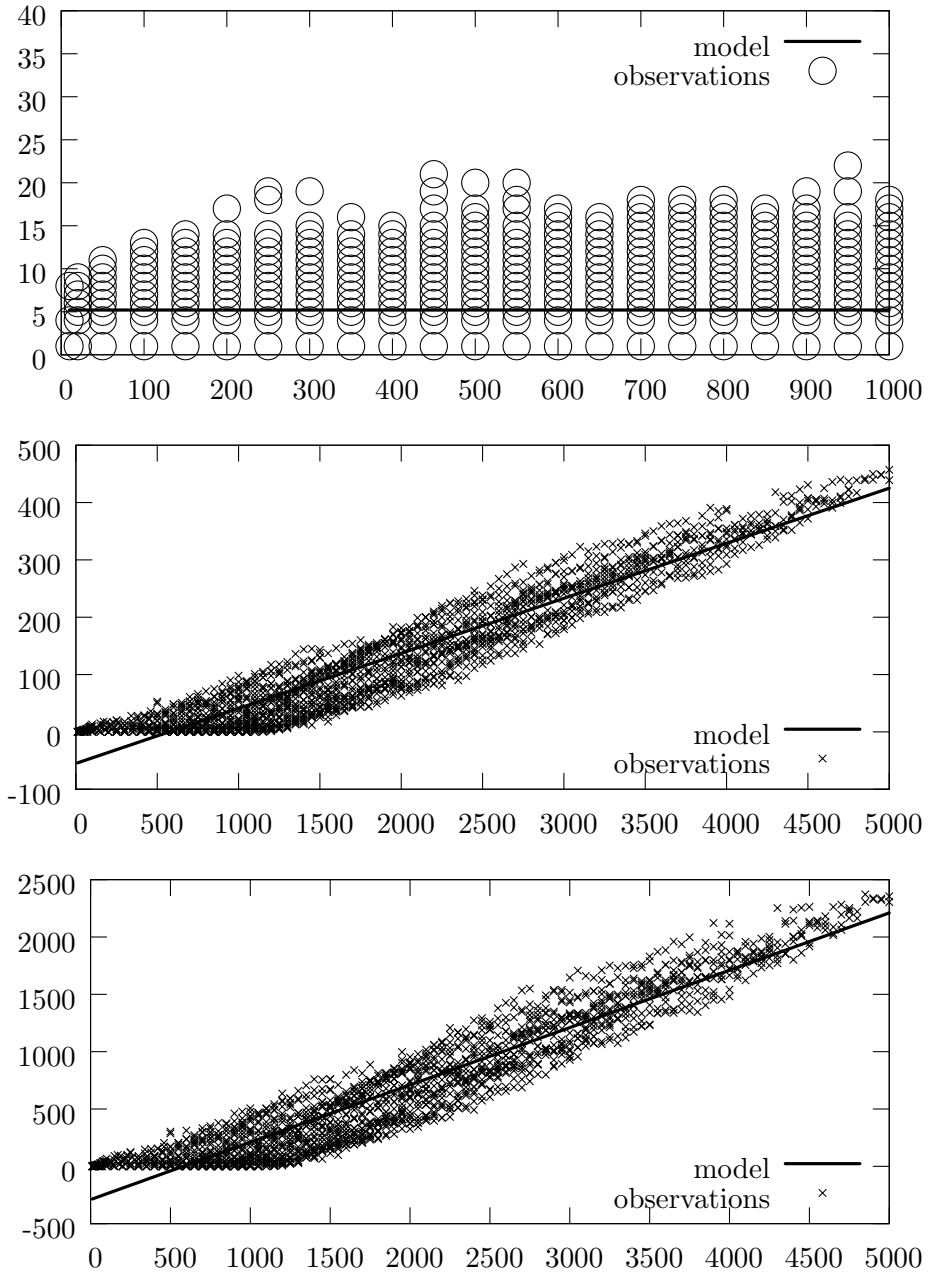

Figure 3.30: Best fit plots for the Fibonacci heap's `decreaseKey` operation during Dijkstra's algorithm. transitive-cost $:\approx 5.2$, SE $= 2$ (top), call-count $:\approx 0.096e - 55$, SE $= 35$ (middle), total-transitive-cost $:\approx 0.50e - 290$, SE $= 187$ (bottom).

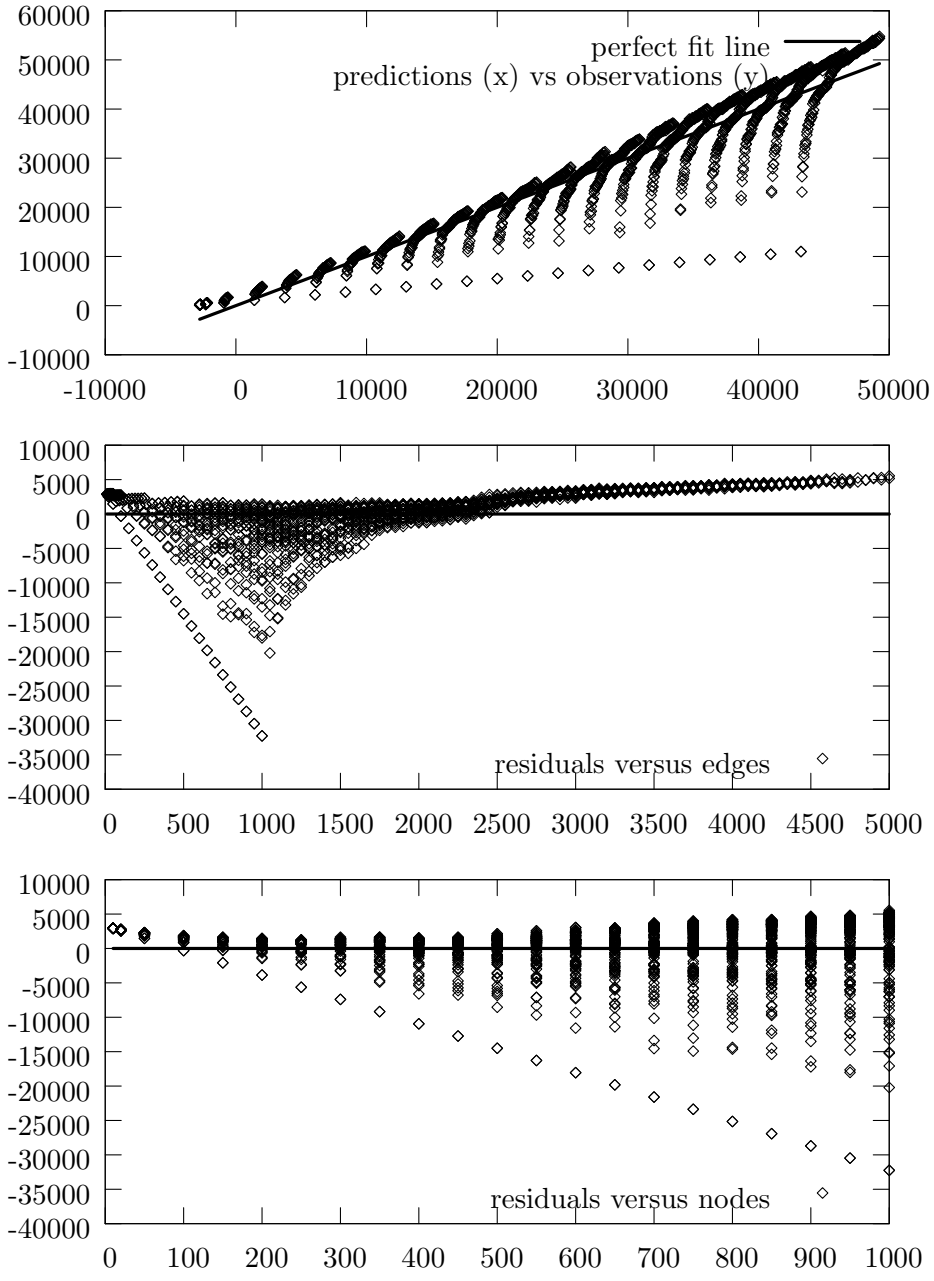

Figure 3.31: Best fit plot for transitive-cost :$\approx 1.5e + 45n - 3200$, SE $= 4500$ of `dijkstra`, Dijkstra's algorithm using a Fibonacci heap (top), residuals versus edges (middle), and residuals versus nodes (bottom).

Theoretical analysis of Dijkstra's algorithm tells us we can expect to do the following operations per workload [CLR90].

- exactly n calls to `insert`
- exactly n calls to `deleteMin`
- $O(e)$ calls to `decreaseKey`

Subtle theoretical analysis of the Fibonacci heap data structure bounds the costs of its operations [CLR90]. The details are beyond the scope of this thesis, but we summarize the results below. Based on these bounds, we can bound the contribution of each operation to the total cost of a workload.

- `insert` is $\Theta(1)$ and thus n calls should contribute $\Theta(n)$.
- `deleteMin` is amortized $O(\log n)$ and thus n calls should contribute $O(n \log n)$.
- `decreaseKey` is amortized $O(1)$ and thus e calls should contribute $O(e)$.

Therefore, the entire run of Dijkstra's algorithm is $O(n \log n + e + n)$. Notice that the bounds for `deleteMin` and `decreaseKey` are $O(\cdot)$ and not $\Theta(\cdot)$; thus, it is possible, depending on the properties of the workloads, that these operations account for less work.

Figure 3.28 shows that CF-TrendProf's models and measurements for the transitive-cost (top), call-count (middle), and total-transitive-cost (bottom) for the `insert` operation match well with theory. The transitive-cost is constant, the call-count exactly linear in n, and the total-transitive-cost is $\Theta(n)$.

Figure 3.29 shows CF-TrendProf's models for `deleteMin`. The call-count model (middle) shows that `deleteMin` is called exactly n times, as theory predicts. The scatter

plot for the transitive-cost is consistent with an amortized logarithmic operation, but ultimately inconclusive. CF-TrendProf chooses a constant model for transitive-cost (32) here because of the α term in the model scoring formula (Section 3.4.3) — we prefer simple models in situations such as this one where performance varies within small bounds. We might hope that CF-TrendProf's model of total-transitive-cost would mirror the theoretical prediction of $O(n \log n)$, but these models are not in CF-TrendProf's vocabulary; instead, CF-TrendProf chooses a linear model: $36n - 2900$, (SE = 4370). Based on this scatter plot and the curve at the top of the residuals plot, it is not hard to believe that the total-transitive-cost of `deleteMin` is $O(n \log n)$, but this pronouncement is beyond the scope of CF-TrendProf's power. Nonetheless, the scatter plots and different models that CF-TrendProf automatically generates are useful tools in analyzing the scalability of this code.

The empirical measurements for `decreaseKey` more closely track its theoretical upper bounds. Figure 3.30 shows CF-TrendProf's models and best-fit plots for transitive-cost, call-count, and total-transitive-cost. The cost of an individual invocation is constant and despite some noise, the call-count and transitive-cost seem to scale linearly with e.

Figure 3.31 shows CF-TrendProf's model for `dijkstra`'s transitive-cost as a function of n and e: $1.5e + 45n - 3200$. Since the three dimensional plots are much harder to judge, we include residuals plots versus edges (middle) and nodes (bottom). That the standard error (SE = 4500) and the spread of the points on the residuals plots are small compared to the magnitude of the performance show that CF-TrendProf's model is a reasonable model of noisy data. This model misses a logarithmic term, but is otherwise close

to theory. For this workload CF-TrendProf cannot definitively confirm that `dijkstra` scales as $O(n \log n + e + n)$, but the empirical truth that CF-TrendProf measures is at least close to this theoretical bound.

This micro-benchmark demonstrates CF-TrendProf's ability to analyze the performance of a complex algorithm. Its result, however, is not the same as what theory gives us. While theory can reason about upper, lower, and expected case (over some specified distribution of workloads) bounds on performance, CF-TrendProf measures the empirical cost of an implementation on particular workloads.

3.6 Diagnosing Data Structure Problems

In this section we use CF-TrendProf to diagnose performance problems. We construct a number of scenarios, state our expectations about the performance of the code involved and compare those expectations with CF-TrendProf's models. The deviations of the models from our expectations point to performance problems.

Our experiments in this section consider quicksort and hash tables in isolation: we provide a workload feature that describes the size of the data structure and consider one call to quicksort or one instance of a hash table. In order to realize such an ideal situation in the context of a larger algorithm, a user would likely have to provide CF-TrendProf with suitable context annotations and invocation features.

To find the performance problems we diagnose in this section with a tool such as `gprof` requires a (potentially large) workload that forces the cost of, say, hash table lookup to be a large percentage of the cost of the entire program. Even then, it may not be clear

exactly why hash table lookups account for such a large portion of performance cost nor what they ought to cost.

In contrast, CF-TrendProf requires only workloads that perform lookups on hash tables and that these hash tables span a range of sizes. Asymptotic bounds give a baseline with which to compare CF-TrendProf's models. Deviations suggest potential performance problems. As we discuss in Section 2.4.5, such deviations invite the user to imagine a workload that would lead to performance problems based on the observed scalability and perhaps to fix the performance bug before observing a workload that exercises it.

3.6.1 Deterministic Quicksort Pivot

Recall our quicksort micro-benchmark from Section 3.5.6. Implemented properly, quicksort sorts an array of n items in $\Theta(n \log n)$ steps in the expected case. However, if quicksort chooses a pivot without randomness, say the first element, and its inputs are suitably permuted, say sorted in reverse, then it can consistently take $\Theta\left(n^2\right)$ steps to sort its input.

Figure 3.32 shows the results of running CF-TrendProf on such an implementation of quicksort on arrays of n integers that are sorted in reverse. Quicksort's transitive-cost fits a quadratic, $0.5n^2 + n$, quite well. Although the residuals plot suggests that CF-TrendProf's model misses a linear term roughly proportional to $0.5n$, we can safely ignore it because it is quite small compared to the quadratic term. The quadratic scaling is quite clear from CF-TrendProf's models. Thus we see CF-TrendProf finding a performance bug.

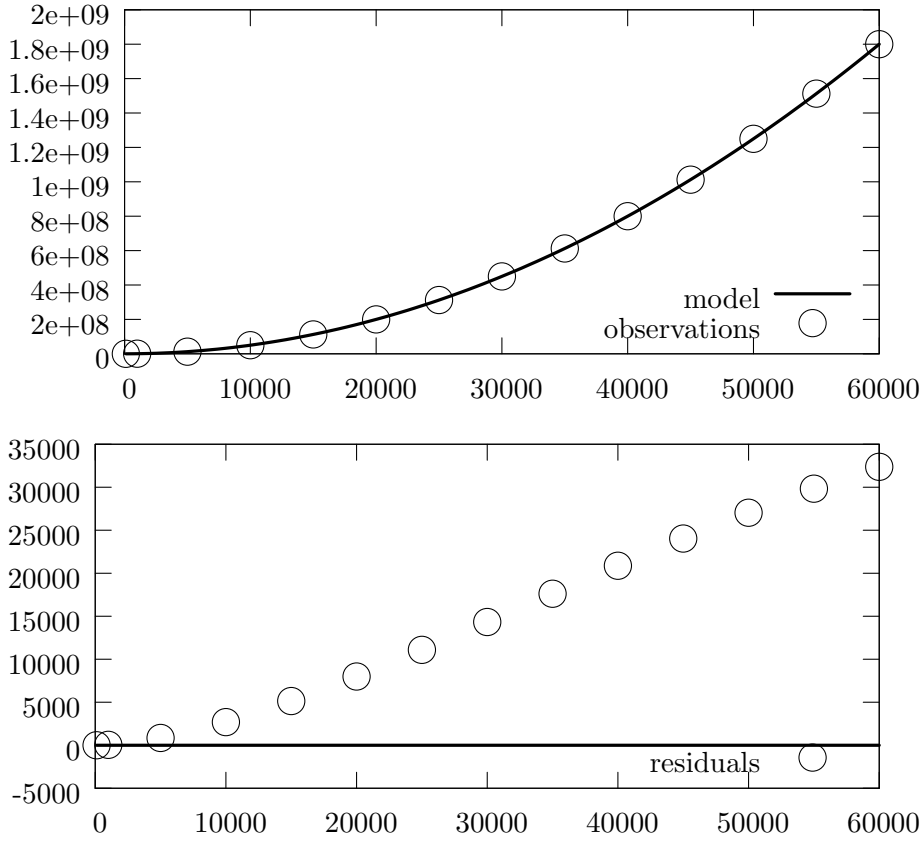

Figure 3.32: Best-fit plot(top) and residuals plot (bottom) for the transitive-cost of a flawed implementation of quicksort transitive-cost $:\approx 0.5n^2 + n$, SE $= 19200$ (it chooses the first element in its list as the pivot) run on reverse-sorted inputs. We obtained this model by separating the calls to quicksort by callee – effectively splitting off the initial call from the recursive ones. The systematic deviation in the residuals plot is not too worrying because of its small scale compared to the actual performance.

3.6.2 Bad Hash Function

A bad hash function can lead to a performance surprise: the hash table may behave well on inputs that do not make extensive use of it, but its lousy scalability will cause it to dominate performance on workloads that make heavy use of it. In this experiment we consider the problem of bad hash functions causing degraded hash table performance. We consider two bad hash functions: the pathologically bad hash function that hashes everything to 0 and a clustering hash function that favors buckets around a central mean. We consider a hash table that resolves collisions with chaining and one that uses linear probing. A workload consists of adding $n \in \{10 \ldots 1000\}$ items to a hash table with 2000 buckets and then looking each item up. As we saw in Section 3.5.4, a well-behaved hash table averages constant time lookups and thus n lookups in $\Theta(n)$ steps,

As with our well-behaved hash table micro-benchmark, we use the identity function as our hash function and generate inputs as if they had been hashed from a function with the desired properties. That is, for our first hash function, the items in each input are all zero. For our clustered hash function, the items are the average of thirteen random numbers between 0 and 1999; these items tend toward the middle buckets with high probability and the outer buckets near 0 and 1999 with low probability.

The clustering hash function has some effect on the chaining hash table. Figure 3.33 shows the best-fit plot and residuals plot for the total-transitive-cost $:\approx 2.89x - 140$, SE $= 69.3$ of looking up the x items in the table. The concave-up bend in the data points (especially evident in the residuals plot) indicates a slight super-linear trend, but we see a largely similar effect with a well-behaved hash function such as that in Figure 3.20. It

seems reasonable to attribute the bend to the table's filling up. There is no substantial performance loss in this case and CF-TrendProf does not suggest one.

The linear-probing hash tables fares much worse with the clustering hash function. Although CF-TrendProf's model is linear, the high standard error and the high absolute values of the slope and intercept suggest further inspection. A glance at the best-fit plot in Figure 3.34 (top) shows that the linear fit is nonsense and that the code's scalability is clearly super-linear. The best powerlaw fit to this data (not shown) is $0.0092x^{2.4}$ and even this fit does not adequately capture the steep increase. Manually plotting the data on linear x axis, logarithmic y axis (bottom of Figure 3.34) shows that the data is not quite growing exponentially either (the plot is not quite a line). Although the exact relationship of performance to table size is unclear, CF-TrendProf has told us all we need to know: look at the best-fit plot and notice the obvious super-linearity.

The degenerate, one-bucket hash function causes both implementations to go quadratic. Figure 3.35 and Figure 3.36 show the best fit plots and residuals plots for our x hash table lookups. The exceptionally good fits leave little doubt: there is a serious problem with the lookup routine.

3.6.3 Overfull Hash Table

In this section we consider the performance of overfull hash tables. Again we consider a chaining hash table and a linear-probing hash table. Again workloads consist of adding x elements and then looking each element up; we expect performance to be linear in x. Also as before, we use the identity function on integers as our hash function and generate inputs uniformly at random, simulating use of a good hash function.

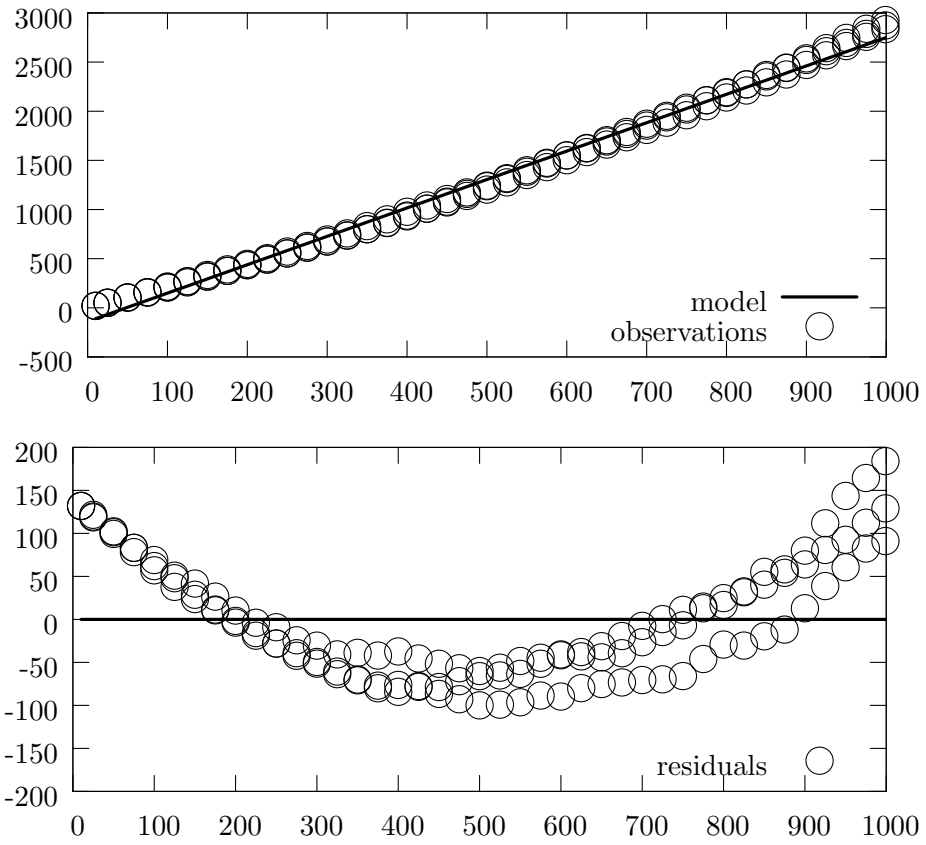

Figure 3.33: Best-fit plot (top) and residuals plot (bottom) for the total-transitive-cost $:\approx 2.89x - 140$, SE $= 69.3$ of looking up in a *chaining* hash table each of the x items spread among its 2000 buckets by the *clustering* hash function. The concave-up bend in the data points (especially evident in the residuals plot) indicates a slight super-linear trend.

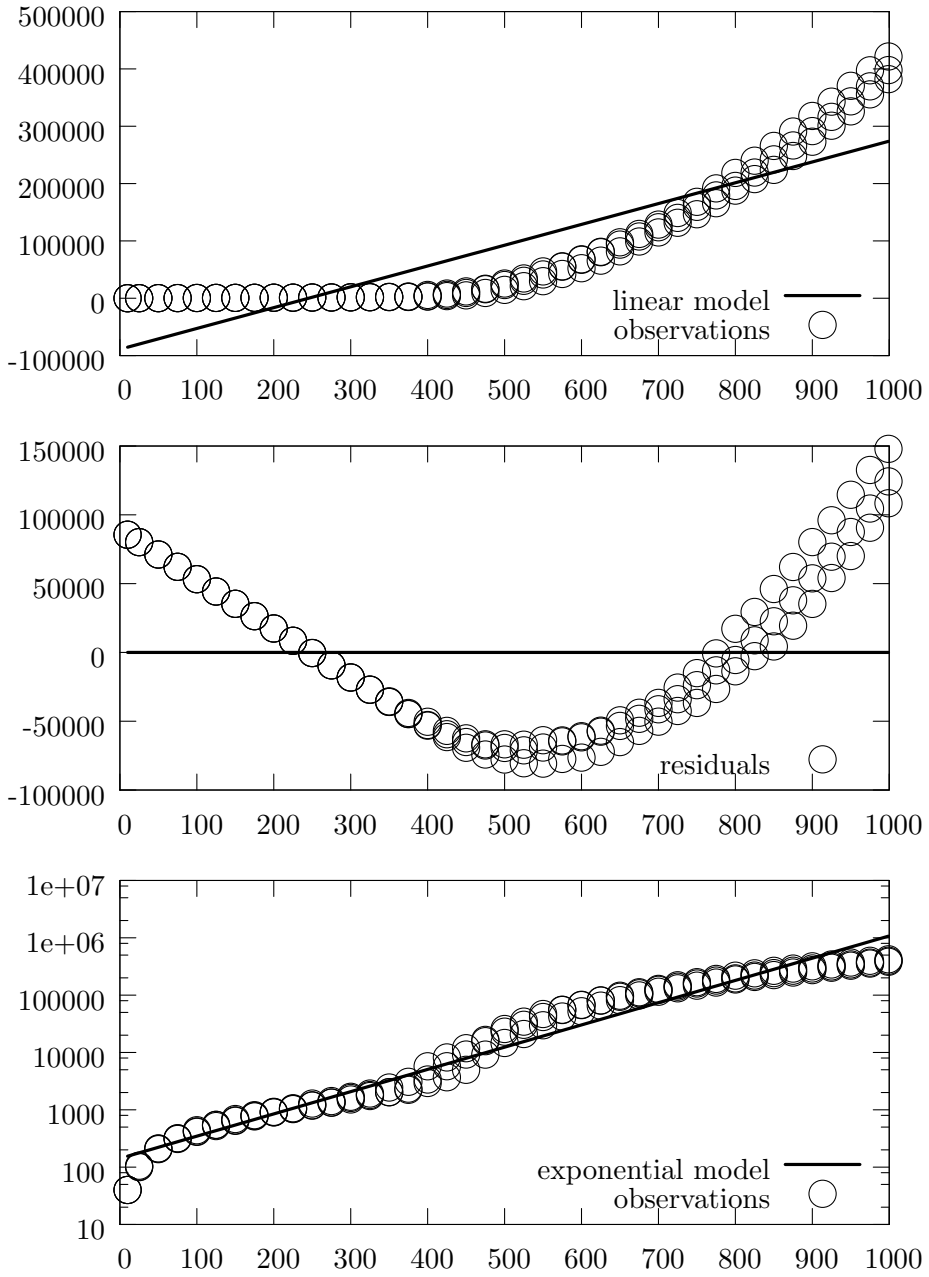

Figure 3.34: Best-fit plot (top) and residuals plot (middle) for the total-transitive-cost $:\approx 363x - 89000$, SE $= 58400$ of looking up in a *linear-probing* hash table each of the x items spread among its 2000 buckets by the *clustering* hash function. The linear fit is clearly nonsense as the high standard error and high absolute values of the slope and intercept suggest. The bottom plot shows the same data with a logarithmic y axis and the best fit of $\log$ (total-transitive-cost) to x: total-transitive-cost $:\approx 142e^{0.0089x}$.

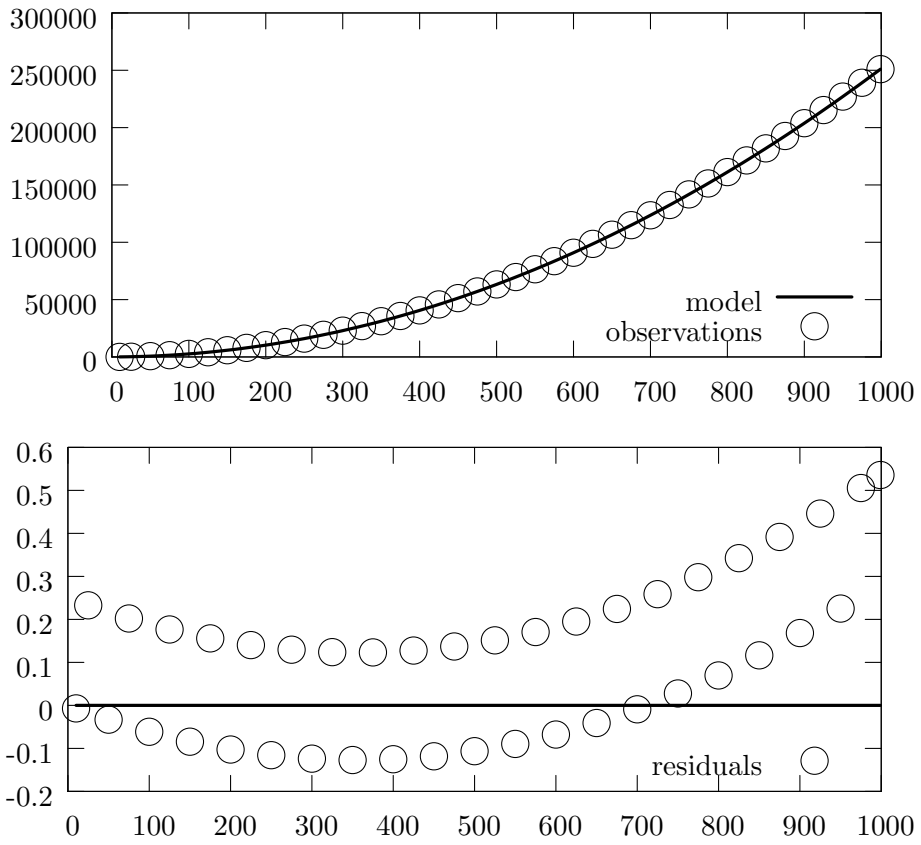

Figure 3.35: Best-fit plot (top) and residuals plot (bottom) for the total-transitive-cost $:\approx 0.25x^2 + 1.5x$, SE $= 0$ of looking up in a *chaining* hash table each of the x items that have been dumped into the same bucket by a terrible hash function. The excellent quadratic fit clearly indicates a problem.

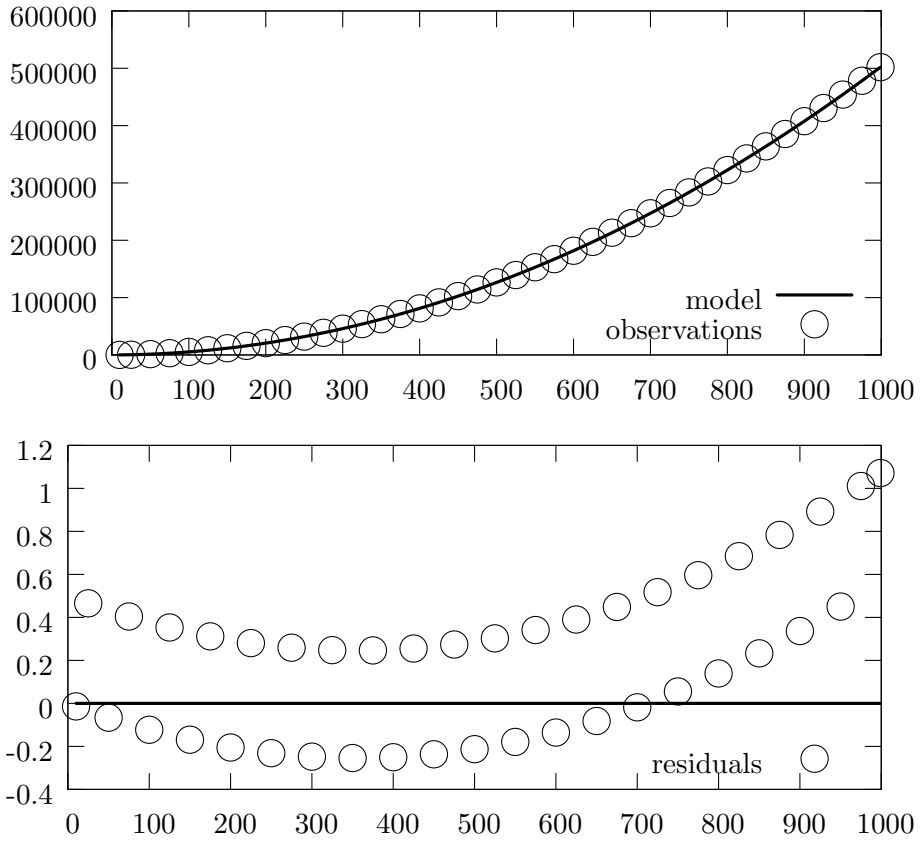

Figure 3.36: Best-fit plot (top) and residuals plot (bottom) for the total-transitive-cost $:\approx 0.5x^2 + 3.0x$, SE $= 0$ of looking up in a *linear-probing* hash table each of the x items that have been dumped into the same bucket by a terrible hash function. The excellent quadratic fit clearly indicates a problem.

Chaining hash tables can hold arbitrary numbers of elements, though at some cost to performance; it is not hard to imagine careless code evolution leading to very full chaining hash tables. Figure 3.37 shows the best-fit and residuals plots for a 100 bucket chaining hash table holding up to 1000 items. This model is not perfect, but it points to the clear super-linearity in the performance of the chaining hash table.

Hash tables that use any sort of open addressing, including our linear probing example, cannot store more elements than they have buckets. The performance of these tables degrades drastically as they become full as the best-fit plot in Figure 3.38 shows. Again, the the high standard error and the high absolute values of the slope and intercept indicate trouble; inspection of the best-fit plot shows that the linear fit is clearly nonsense. It is not clear exactly what sort of relationship the performance of this linear probing hash table has to its input size, but it is clearly super-linear.

3.7 Large Benchmarks

We evaluated CF-TrendProf's ability to analyze large programs by running it on several larger benchmarks. Section 3.7.1 explains the setup of the experiments and the programs and workloads we ran. We show that CF-TrendProf meets its design goals of building precise models, potentially in terms of multiple features (Section 3.7.2) and of enabling reasoning about how performance flows through the call graph (Section 3.7.3). Section 3.7.4 considers how CF-TrendProf deals with the fact that performance, especially of the innards of heuristically optimized complex algorithms run on general workloads, is not a clean function of workload features. Section 3.7.5 discusses a related issue: how the

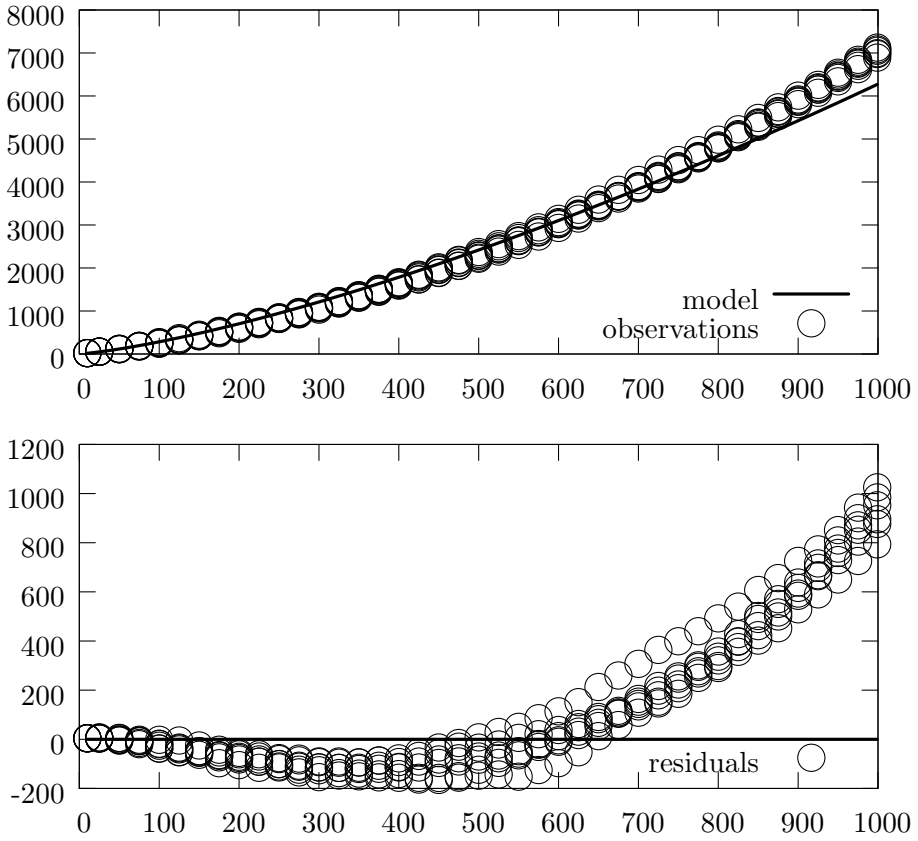

Figure 3.37: Best-fit plot (top) and residuals plot (bottom) for total-transitive-cost $:\approx 0.22x^{1.46} + x$, SE $= 326$ of looking up in a chaining hash table each of the x items spread among its 100 buckets. The model is obviously flawed, but the super-linear trend is clear.

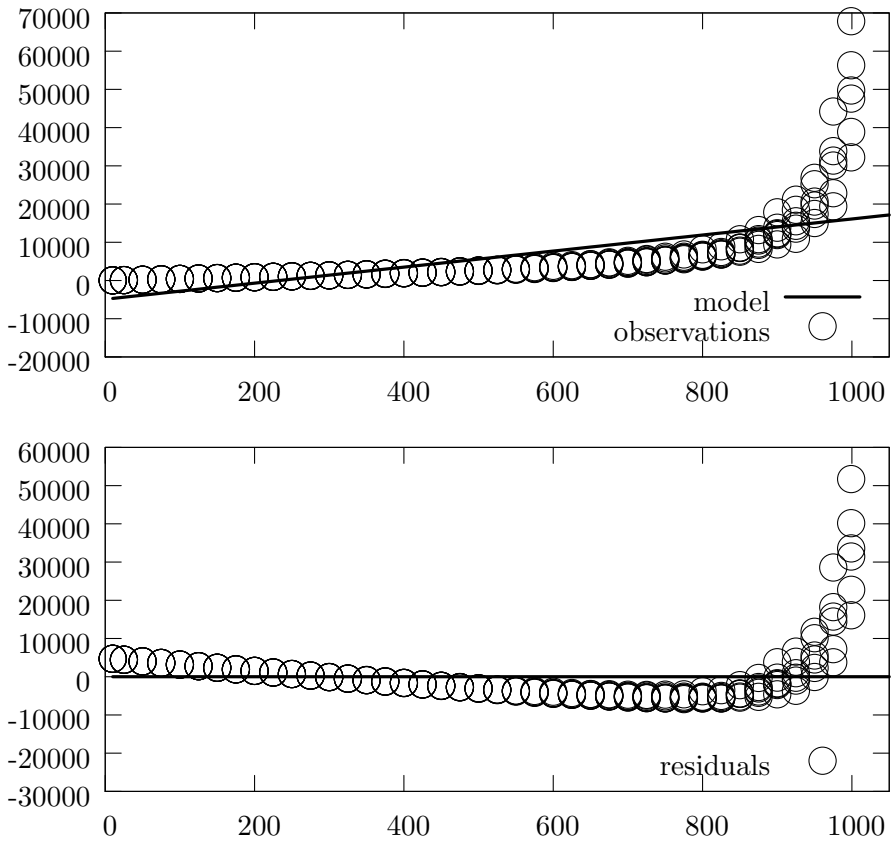

Figure 3.38: Best-fit plot (top) and residuals plot (bottom) for total-transitive-cost $:\approx 21x - 4900$, SE $= 7030$ of looking up in a linear probing hash table each of the x items spread among it 1000 buckets. The model is obviously flawed, but the high absolute values of the slope and intercept indicate trouble.

Program	Description	Workloads
`bzip2 1.0.4` [BZ2]	Compresses files	Tarballs of preprocessed source code
`banshee 2005.10.07` [KA05]	Computes Andersen's alias analysis [And94] on a C program	Preprocessed C programs
`dot from graphviz 2.14.1` [Gra, GN00]	Renders directed graphs, avoiding edge crossings and minimizing edge length	Randomly generated connected, directed graphs
`lp_solve 5.5.0.10` [LPS]	Solves mixed integer linear programs	Linear Programs from MIPLIB 2003 [AKM06], Mittelmann [Mit], Mészáros [Més]

Figure 3.39: We ran CF-TrendProf on these programs with workloads as described above.

Program	Workloads	Min – Max	Overhead	Time (h)
`bzip`	524	4×10^6 – 8×10^9	609%	35 + 2.7
`banshee`	116	1×10^6 – 1×10^9	1928%	82 + 5.4
`dot` (simple: $e = n$)	100	9×10^4 – 4×10^7	2677%	? + 0.5
`dot` (complex: $n \leq e \leq 1.3n$)	175	9×10^4 – 1×10^8	2520%	? + 1.2
`lp_solve`	215	3×10^3 – 2×10^{10}	149%	22 + 2.6

Figure 3.40: Number of workloads, costs of the cheapest (Min) and most expensive (Max) workload (measured in loop and function counts), geometric mean of overhead of CF-TrendProf's instrumentation (Overhead), and CF-TrendProf's user+system time in hours to 1) run workloads and post-process the program trace and 2) fit models and produce output (Time).

distribution of workloads affects the models that CF-TrendProf computes.

3.7.1 Workloads and Experimental Setup

We ran CF-TrendProf on the programs listed in Figure 3.39 with workloads as described in Figure 3.40 and elaborated below. We did not repeat our `elsa` and `maximus` benchmarks from Chapter 2 because shortcomings in our instrumentation infrastructure did not allow us to handle the C++ templates in these benchmarks. The Overhead column of Figure 3.40 reports the average (geometric mean) overhead of running a workload with

CF-TrendProf's tracing versus having it disabled: (user + system time instrumented) divided by (user + system time uninstrumented), reported as a percentage; these overhead measurements are based on 20 randomly selected workloads rather than the entire set. The Time column reports the total (user + system) time in hours that our straightforward Perl and C implementation of CF-TrendProf takes to create a report on each program; the first (left) time includes running the instrumented workloads and post-processing the trace data; the second (right) time includes the rest of CF-TrendProf's post-processing including model-fitting, and generation of plots and results pages. We did not measure the time to run the exact set of workloads for the `dot` benchmarks, though a comparable set of workloads to the complex ($n \leq e \leq 1.3n$) set took on the order of several days to run and post-process. Running the workloads can take a long time, but once CF-TrendProf generates its results, they are browseable interactively. We ran these experiments on an Intel Xeon with two 2.8 GHz CPUs and 3.7 GB of RAM. Because CF-TrendProf's measurements do not depend on time, scheduling, or system load, we made no effort to run these experiments on an unloaded system and sometimes ran multiple experiments simultaneously. The design of CF-TrendProf, like BB-TrendProf, generally spends extra computer time to save human time (for example, by generating all models and plots in advance instead of on demand).

Workloads for `bzip`

A workload for `bzip` consists of a tarball of pre-processed source code ranging in size from 25 thousand bytes to 61 million bytes. The only workload feature we specified for `bzip` is B, the size of the input in bytes. CF-TrendProf did not instrument two functions

in the `bzip` code because they have irreducible control flow (`unRLE_obuf_to_output_FAST` and `BZ2_decompress`); these functions, however, seem to only concern decompression and our benchmark only exercised compression. We instrumented only those `bzip` functions with loops or recursion as other functions' costs are accounted for by callers.

Workloads for `dot`

A workload for `dot` consists of a directed graph with n nodes and e edges that `dot` renders in its default output format so as to minimize edge crossings and edge lengths. We generate a random workload with n nodes and e edges as follows. The first node starts with no predecessors. As we add each of the next $n-1$ nodes, the new node chooses a predecessor uniformly at random from those nodes already in existence; this process results in a tree of n nodes and $n-1$ edges. We then pick enough edges uniformly at random from all non-existent, non-self edges to bring the graph to e edges.

We ran two experiments with `dot`, both on graphs generated randomly. In the first experiment, which we refer to as complex `dot`, we ran `dot` on graphs containing between 20 and 837 nodes and, for each number of nodes, n, four graphs containing $\{n, 1.1n, 1.2n, 1.3n\}$ edges respectively. In the second experiment, which we refer to as simple `dot`, we ran `dot` on the subset of the graphs from the first experiment with an equal number of nodes and edges ($e = n$).

Workloads for `banshee`

A workload for banshee consists of one or more pre-processed C files that constitute an entire binary; these C files are drawn from the Debian Linux archive [Deb].

During the course of a workload, `banshee` parses the C files and performs Andersen's alias analysis [And94] on them. We specify four workload features:

- `files`, the number of files in the input.
- `bytes`, the total number of bytes in all the input files.
- `vars` an internal metric that `banshee` outputs.
- `nonemptySets`, an internal metric that `banshee` outputs.

Workloads range in size from 30 thousand bytes to 24 million bytes, 1 to 181 files, 41 to 33 thousand `vars`, and 0 to 14 thousand `nonemptySets`.

Because of the prevalence of mutual recursion in `banshee`, we set all functions in `banshee` to be caller-sensitive. The `yyparse` function in `banshee` has irreducible control flow and so was not automatically instrumented. We manually instrumented it to report its loop costs as part of the cost of its caller, `compile_file`.

Workloads for `lpsolve`

A workload for `lpsolve` consists of a linear program: a set of variables, constraints (on the variables), and an objective function (in terms of the variables) to optimize (by picking values for the variables subject to the constraints). We specify three workload features.

- `bytes`, the size of an input in bytes.
- `rows`, the number of rows in `lpsolve`'s matrix for the linear program; each constraint in the linear program occupies a row

- `columns`, the number of columns in `lpsolve`'s matrix for the linear program; each variable in the program occupies a column

Workloads range in size from 442 bytes to 37 million bytes, 3 to 59 thousand rows, and 3 to 123 thousand columns.

We instrumented only those `lpsolve` functions with loops as other functions' costs are accounted for by callers. The only recursive function in `lpsolve` (that our workloads exercised) calls itself at most once and accounts for negligible work; we did not instrument it.

3.7.2 Precise Models in Terms of Multiple Features

CF-TrendProf's model generation and selection algorithms enable it to find precise models and to choose effectively among models in terms of different features. Thus, the user can provide CF-TrendProf with multiple features and have confidence that it will choose the most suitable. We illustrate this point with several models from our benchmarks.

The total-transitive-cost of the `hash_table_copy` function from `banshee` scales as $10.8 \cdot \text{files} \cdot \text{vars} + 260 \cdot \text{files}$ (SE $= 8.44 \times 10^5$). Figure 3.41 shows the multi-feature best-fit plot (predicted values on the x axis versus observed values on the y axis). This model tells a story: for every file, `banshee` copies two hash tables (call-count $:= 2 \cdot \text{files}$) whose sizes increase with the number of `vars` in the workload. The next best model, total-transitive-cost $:\approx 2.3 \cdot \text{bytes} - 1.5 \times 10^6$, SE $= 2.02 \times 10^6$, has more error and less explanatory power.

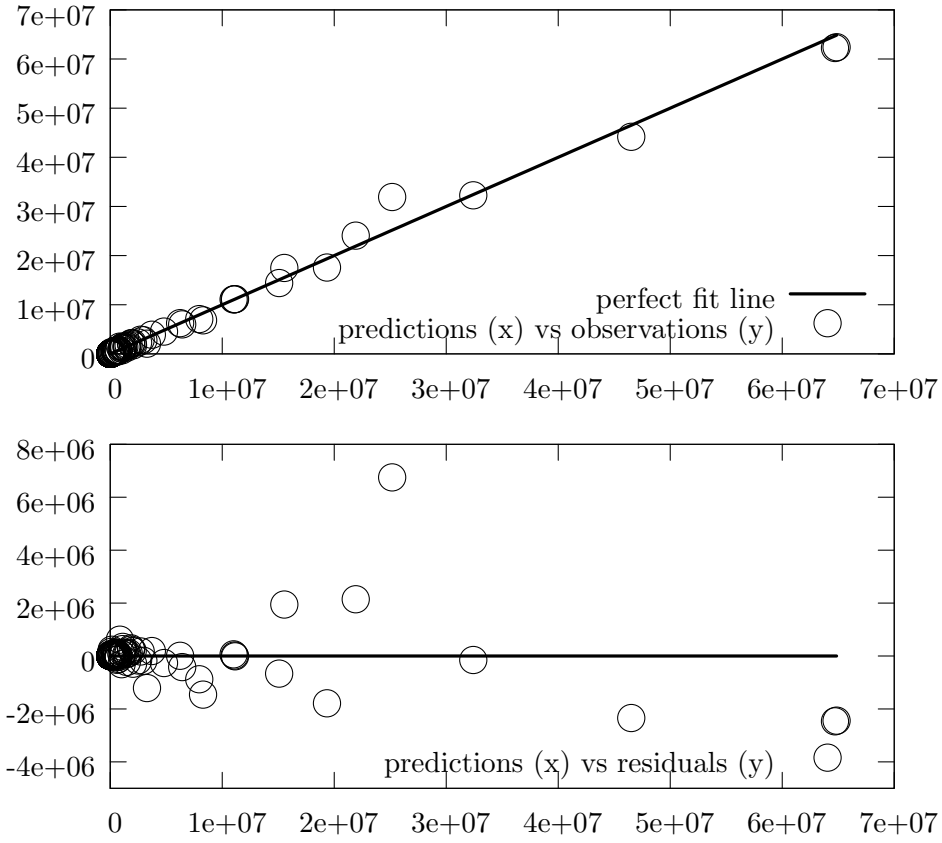

Figure 3.41: This scatter plot shows CF-TrendProf's predictions (x axis) versus the measured values (y axis) for `banshee`'s `hash_table_copy` total-transitive-cost $:\approx 10.8 \cdot \text{files} \cdot \text{vars} + 260 \cdot \text{files}$, SE $= 8.44 \times 10^5$. The fit is perfect to the extent that the points lie on the line $y = x$. The bottom plot shows the residuals (y axis) plotted versus the predicted values (x axis).

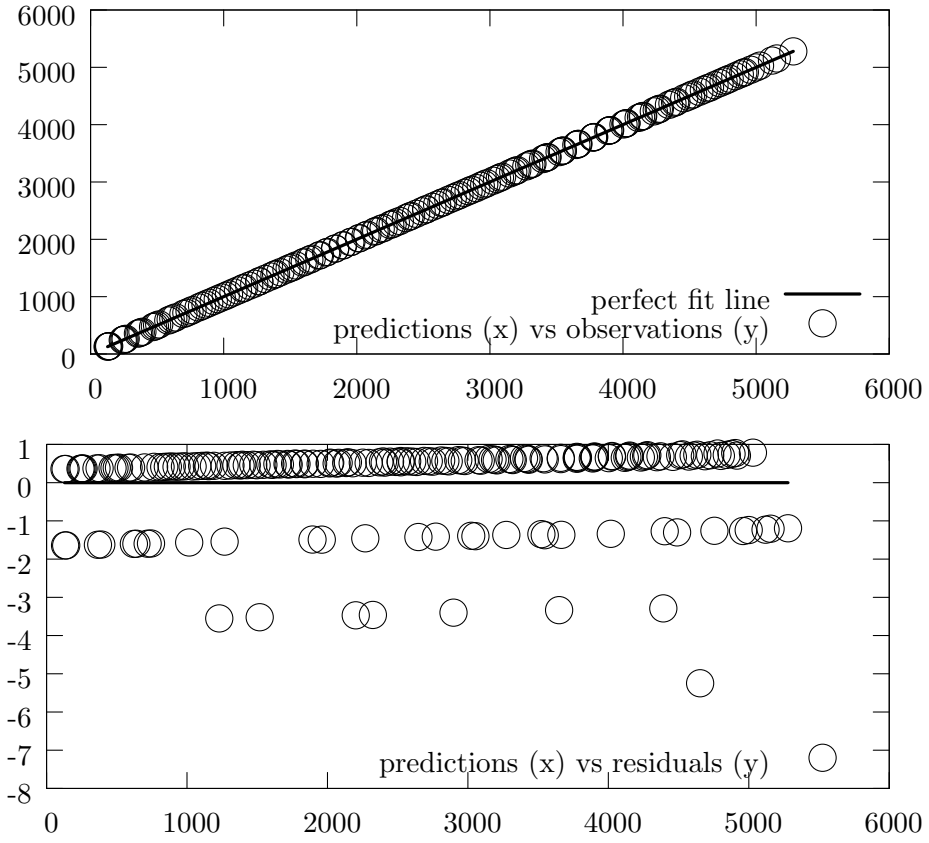

Figure 3.42: Complex `dot`'s `init_rank` transitive-cost $:\approx 4n + 2e + 7.66$, SE $= 1$. This scatter plot shows CF-TrendProf's predictions (x axis) versus the measured values (y axis) on top and predictions (x axis) versus residuals (y axis).

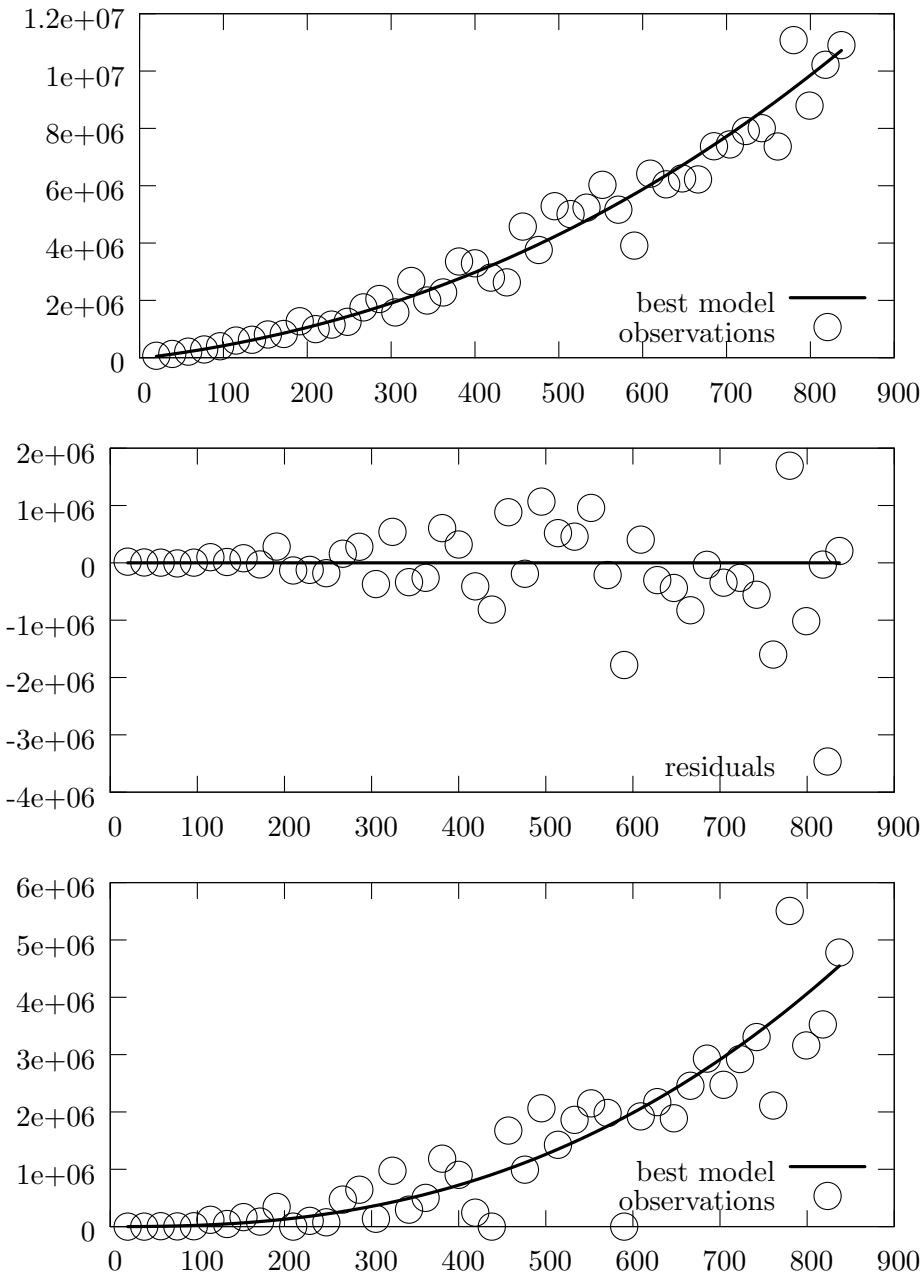

Figure 3.43: Best-fit plot (top) and residuals plot (middle) for simple `dot`'s `gvLayoutJobs` total-transitive-cost :$\approx 0.24n^{2.49} + 30n^{1.73} + 3319n - 22912$, SE $= 6.34 \times 10^5$. The bottom plot is the best-fit plot for `gvLayoutJobs`'s callee `mincross_step`'s total-transitive-cost :$\approx 0.24n^{2.49}$, SE $= 5.89 \times 10^5$.

Figure 3.42 shows another example of a precise model: complex `dot`'s `init_rank` function whose transitive-cost $:\approx 4n + 2e + 7.66$, SE $= 1$. Again, CF-TrendProf has chosen a precise model in terms of two features.

On the simple ($e = n$) `dot` benchmark, the `gvLayoutJobs` function has total-transitive-cost $:\approx 0.24n^{2.49} + 30n^{1.73} + 3319n - 22912$, SE $= 6.34 \times 10^5$; shown at the top of Figure 3.43. To be sure, the powerlaw terms in this model look a little strange, but this model is superior to the best powerlaw direct model ($692n^{1.41}$) in two respects. First, it has smaller error. More importantly, though, it preserves the high exponent (2.49) that arises from a transitive callee: `gvLayoutJobs` calls `dot_layout` calls `dot_mincross` calls `mincross` calls `mincross_step` whose total-transitive-cost scales as $0.24n^{2.49}$. Its high maximum total-transitive-cost (5.5×10^6) and high exponent suggest that the total-transitive-cost of `mincross_step`, and thus the total-transitive-cost of `gvLayoutJobs`, is likely to remain quite high.

Thus we see that CF-TrendProf's model generation and selection can result in more precise models that mirror the control flow of the program. That is not to say that every function has such a complex performance model. Indeed, in most cases the direct linear or powerlaw models have low error and are quite adequate. The model selection criterion penalizes models for including extra features so that these features must justify their presence by reducing the error of the model. This ability to choose more complex models when appropriate and reject them otherwise is an asset in describing the scalability of programs and an improvement over BB-TrendProf.

`bzip compressStream`			
	model	SE	SE/mean
total-self-cost	$0.0002B + 2.5$	0	0 %
total-transitive-cost	$126B - 2.4 \times 10^7$	7×10^7	11 %
call-count	1	0	0 %
self-cost	$0.0002B + 2.5$	0	0 %
transitive-cost	$126B - 2.4 \times 10^7$	7×10^7	11 %

`bzip BZ2_bzWrite`			
	model	SE	SE/mean
total-self-cost	$0.000592B - 80$	81	2 %
total-transitive-cost	$125B - 6.3 \times 10^7$	8×10^7	13 %
call-count	$0.0002B + 0.5$	0	0 %
self-cost	2.89	12	1 %
transitive-cost	569000	8×10^6	1375 %

Figure 3.44: CF-TrendProf's output on several `bzip` functions.

3.7.3 Following Cost through the Call Graph

With CF-TrendProf, one can follow performance through the call graph. The call-graph view of functions together with the flat list of functions (sorted by maximum, over all workloads, total-self-cost) allows the same sort of reasoning that `gprof` [GKM82] enables: starting at `main`, one can top-down explore sub-trees with high total-transitive-cost or bottom-up start at functions with high total-self-cost and see what calls them and how many times. CF-TrendProf's models surpass the `gprof` view in several ways that we illustrate with examples from our benchmarks.

Finding `bzip`'s Main Loop

CF-TrendProf finds `bzip`'s main loop. Its output shows that `main` (total-transitive-cost $:\approx 126B - 2.4 \times 10^7$, SE $= 7 \times 10^7$) calls `compress` (same total-transitive-cost model) which calls `compressStream` which calls `BZ2_bzWrite`. Figure 3.44

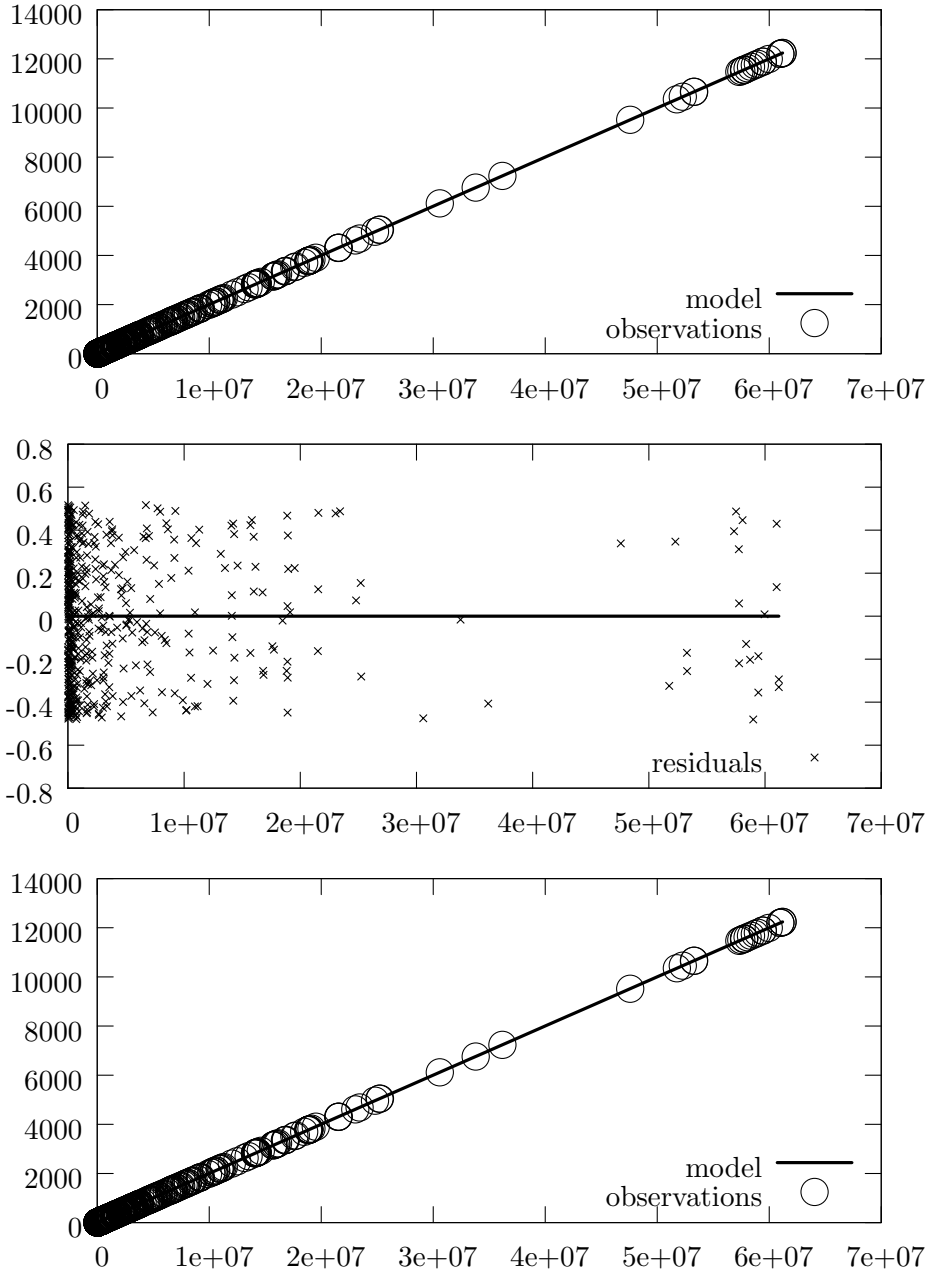

Figure 3.45: Plots that illustrate `bzip`'s overall performance structure. `compressStream` total-self-cost $:\approx 2.5 + 0.000200B$, SE $= 0$ (top) and residuals (middle). `BZ2_bzWrite` call-count $:\approx 0.5 + 0.000200B$, SE $= 0$ (bottom; residuals not shown, but similar to middle plot).

Function	total-transitive-cost Model	SE	Maximum total-transitive-cost
`dot_layout`	$0.24n^{2.49}+30n^{1.73}+3300n-23000$	634000	1×10^7
`dot_mincross`	$0.24n^{2.49} + 310n - 15000$	615000	6×10^6
`dot_position`	$30n^{1.73}$	252000	4×10^6
`dot_splines`	$2670n - 7900$	10100	2×10^6
`dot_init_node_edge`	$167n + 17$	19	1×10^5
`dot_rank`	$139n + 95$	313	1×10^5

Figure 3.46: CF-TrendProf's total-transitive-cost models for `dot_layout` and its more expensive callees. These models are based the simple ($e = n$) `dot` benchmark.

shows CF-TrendProf's models for these latter two functions. From these models, the overall structure of `bzip`'s performance is clear. The linear scaling of `compressStream`'s self-cost and the linearly scaling call-count of its callee, `BZ2_bzWrite`, suggest (and quick inspection of the code confirm) that `compressStream` iterates over 5000-byte blocks of input and calls `BZ2_bzWrite` to operate on them. Figure 3.45 shows the relevant best-fit plots (`compressStream`'s total-self-cost and `BZ2_bzWrite`'s call-count); the models fit the data quite well. The constant models for subsequent functions' self-cost and transitive-cost (not shown) indicate that their per-invocation cost does not scale up with input size while the linear models for their call-count, total-self-cost, and total-transitive-cost show that their total per-workload cost scales linearly with input size because they are called a linear number of times. This view provides a quick overview of the broad performance structure of `bzip`: it iterates over its input in fixed-size blocks and does a varying amount of work for each block, but this per-block cost does not grow with the number of blocks.

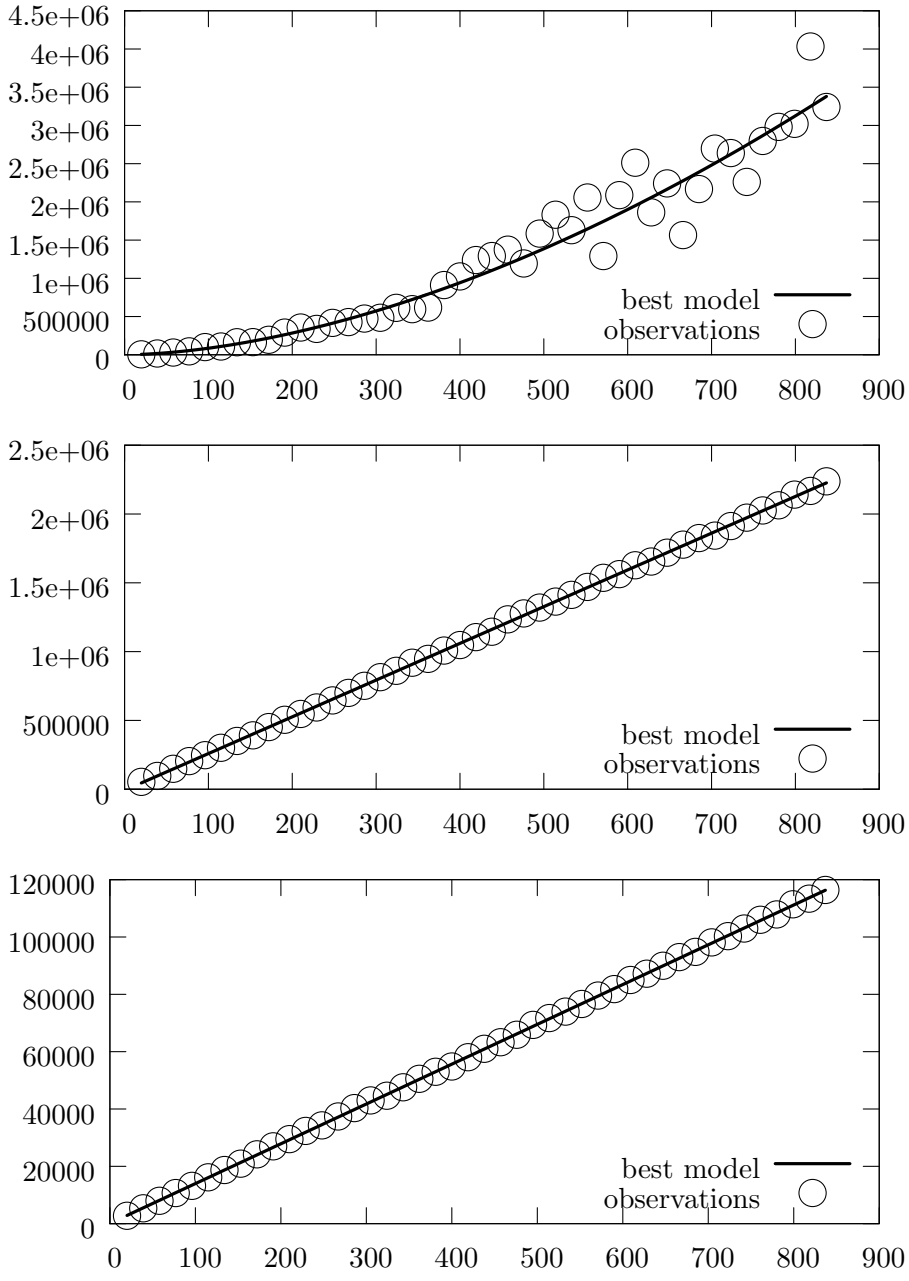

Figure 3.47: Best-fit plots for the total-transitive-cost three functions from the simple `dot` benchmark: `dot_position` (top), `dot_splines` (middle), and `dot_rank` (bottom). Figure 3.46 shows the models.

Ignoring the Cheap Stuff

CF-TrendProf's call-graph output helps us follow call trees with high or potentially high cost and ignore those that do not matter to scalability. We show one example, the `dot_layout` function and its callees from the simple `dot` benchmark, but this sort of reasoning applies more generally. Figure 3.46 lists the best total-transitive-cost models for `dot_layout` and its more expensive callees. Each of these functions has a linear or sub-linear total-self-cost and is called exactly once per workload, so we show only CF-TrendProf's total-transitive-cost models.

The high maximum total-transitive-costs (the maximum total-transitive-cost of `main` is 1.2×10^7) and reasonable looking super-linear models clearly indicate that the callees of `dot_mincross` and `dot_position` (top of Figure 3.47) merit further investigation. The `dot_splines` function (middle of Figure 3.47) is interesting: its maximum total-transitive-cost is quite high (about a fifth of the maximum total-transitive-cost of `main`), but its total-transitive-cost scales linearly; this high maximum total-transitive-cost suggests that `dot_splines`'s callees account for a reasonable chunk of performance, but the model suggests that they will become less important for larger workloads. The other callees can be safely ignored: they have maximum total-transitive-costs that are about a factor of one hundred off from that of `main` and very good models that show linear scaling (bottom of Figure 3.47).

Finding Inner Loops in `dot`

Again, we focus on the simple ($e = n$) `dot` benchmark. Along the `dot_position` call tree we find some nested loops. A function which is called exactly twice, `rank`, calls

`rank`			
	model	SE	SE/mean
call-count	1	0	0 %
total-self-cost	$0.87n - 21$	28	2 %
total-transitive-cost	$18n^{1.80}$	254000	20 %

`update`			
	model	SE	SE/mean
call-count	$0.87n - 22$	28	2 %
total-self-cost	$0.87n - 22$	28	2 %
total-transitive-cost	$3.1n^2$	156000	22 %

`dfs_range` (initial call)			
	model	SE	SE/mean
call-count	$0.87n - 22$	28	2 %
total-self-cost	$4.1n - 110$	136	5 %
total-transitive-cost	$1.8n^{2.04}$	107000	20 %

`dfs_range` (recursive calls)			
	model	SE	SE/mean
call-count	$0.78n^2$	33900	19 %
total-self-cost	$2.3n^2$	102000	19 %
total-transitive-cost	$2.3n^2$	102000	19 %

Figure 3.48: CF-TrendProf's models for three functions from the simple ($e = n$) `dot` benchmark.

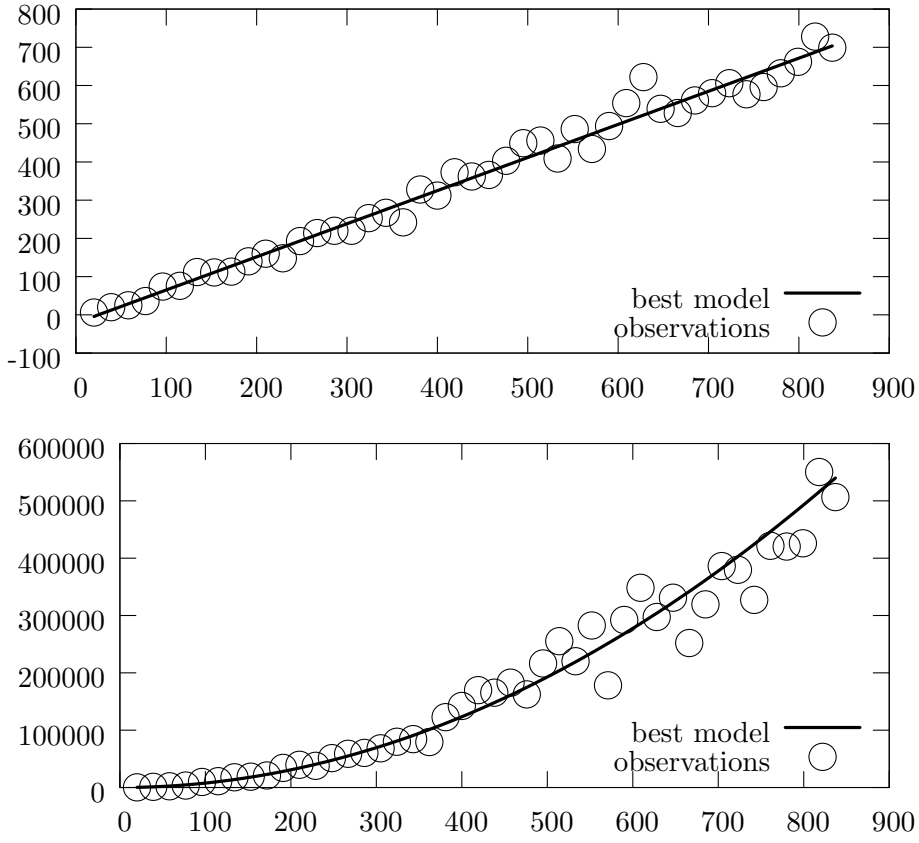

Figure 3.49: Best-fit plots for two functions from the simple `dot` benchmark: call-count of `update` (top) and recursive calls to `dfs_range` (bottom). Figure 3.48 shows the models.

`banshee compile_file`			
	model	SE	SE/mean
total-self-cost	$3.44 \cdot \text{bytes} - 270000$	1×10^6	21 %
call-count	files	0	0 %
self-cost	$10.4 \cdot \text{vars} + 160000$	2×10^5	60 %

Figure 3.50: A `banshee` function whose self-cost increases with workload features; its call-count increases only modestly, but its total-self-cost increases because its self-cost does.

`banshee env_hash`			
	model	SE	SE/mean
total-self-cost	$1.81 \cdot \text{bytes} + 340000$	1×10^6	27 %
call-count	$0.2 \cdot \text{bytes} + 42000$	1×10^5	24 %
self-cost	8.96	6	1 %

`banshee yylex`			
	model	SE	SE/mean
total-self-cost	$1.3 \cdot \text{bytes} - 29000$	1×10^5	5 %
call-count	$0.223 \cdot \text{bytes} - 10000$	6×10^4	13 %
self-cost	5.88	6	1 %

Figure 3.51: Functions in `banshee` whose total-self-cost increases because they are called more often as input size gets larger.

`update` (maximum total-transitive-cost of 2.3×10^6) a linear number of times in n; thus `update` is in `rank`'s inner loop. Since `rank` is called exactly twice from two separate parts of the code, we distinguish its caller with `tpRuntimeSetContext` and so it behaves as if it were two different functions — the context we discuss here is by far the more expensive one.

Furthermore, `update` calls `dfs_range` (maximum total-transitive-cost of 1.6×10^6) a linear number of times in n. Then `dfs_range` calls itself recursively; based on its call-count model, we conclude that these recursive calls result in its scaling quadratically in n. Figure 3.48 shows the call-count, total-self-cost, and total-transitive-cost models for these functions with the initial call to `dfs_range` accounted for separately from the subsequent recursive calls. Figure 3.49 shows the best-fit plots for the call-count models.

More Calls or More Cost?

Some functions, such as `bzip`'s `compressStream` (discussed above) increase in total-self-cost because they iterate over the whole input (or an increasing part of it). These functions have a self-cost that grows with input size. Figure 3.50 shows CF-TrendProf's models for such a function from `banshee`.

Often, however, a function's self-cost does not grow larger with the size of its input (though in some cases its maximum or variance goes up), but its total-self-cost scales up because it is called more (Figure 3.51). Looking at CF-TrendProf's models for call-count and total-self-cost clarify this situation: if the total-self-cost model is a constant multiple of the call-count model, then the increased cost of the function on larger inputs is probably because of the increased number of calls. If a function has a self-cost model that grows with some workload feature or if the total-self-cost model is of a higher degree, then it is safe to assume that an invocation of this function touches an ever increasing chunk of the input. Again we see that CF-TrendProf shows not only how a function's cost increases with bigger workloads, but why: more calls, more loop iterations, or more work done by callees.

3.7.4 Performance of Complex Algorithms in Large Programs

One of the more exciting results of BB-TrendProf is its ability to analyze the performance of complex algorithms in the context of large programs (Section 2.4). We demonstrated in Section 3.5 that CF-TrendProf can analyze the scalability of complex algorithms precisely and furthermore demonstrated in Section 3.7.3 that CF-TrendProf enables reasoning about how performance moves through the call graph of large programs.

Function	Model	SE	SE/mean	maximum total-self-cost
`generateMTFValues`	$114B - 24 \times 10^6$	7×10^7	12 %	7×10^9
`mainSort`	$2.86B + 210000$	1×10^5	1 %	2×10^8
`bsW`	$2.02B - 140000$	4×10^5	4 %	1×10^8
`mainGtU`	$1.57B - 47000$	3×10^5	3 %	1×10^8
`mainSimpleSort`	$1.43B - 81000$	7×10^4	1 %	9×10^7
`sendMTFValues`	$1.19B - 40000$	1×10^5	2 %	7×10^7
`copy_input_until_stop`	$1.00B + 2.2$	3	0 %	6×10^7
`copy_output_until_stop`	$0.968B - 98000$	3×10^5	5 %	6×10^7
`BZ2_blockSort`	$0.488B - 140000$	6×10^5	23 %	3×10^7
`mainQSort3`	$0.186B + 67000$	2×10^5	20 %	1×10^7
`BZ2_hbMakeCodeLengths`	$0.184B + 110000$	5×10^4	4 %	1×10^7

Figure 3.52: Top several functions in the `bzip` benchmark, ranked by maximum total-self-cost.

Function	Model	SE	SE/mean	maximum total-self-cost
`reorder`	$0.059n^{2.60}$	3×10^5	45 %	3×10^6
`dfs_range`	$2.3n^2$	1×10^5	19 %	2×10^6
`left2right`	$0.0078n^{2.84}$	2×10^5	53 %	2×10^6
`dfs_enter_inedge`	$1.5n^2$	1×10^5	40 %	1×10^6
`rerank`	$2.2n^{1.82}$	6×10^4	35 %	6×10^5
`connecttris`	$582n - 2600$	2×10^3	1 %	5×10^5
`out_cross`	$0.36n^2$	5×10^4	56 %	4×10^5
`in_cross`	$0.36n^2$	5×10^4	55 %	4×10^5
`dttree`	$430n - 3600$	1×10^3	1 %	4×10^5
`Bezier`	$420n - 300$	645	0 %	4×10^5
`ccw`	$300n - 1900$	2×10^3	1 %	3×10^5

Figure 3.53: Top several functions in the simple ($e = n$) `dot` benchmark, ranked by maximum total-self-cost. Rows for the following functions include only recursive calls: `dfs_range`, `dfs_enter_inedge`, `rerank`.

Function	Model	SE	SE/mean	maximum total-self-cost
`dfs_range`	$10100e - 2.4 \times 10^6$	4×10^6	171 %	4×10^7
`dfs_enter_inedge`	$3600e - 870000$	2×10^6	200 %	2×10^7
`ccw`	$3700e - 720000$	1×10^6	131 %	1×10^7
`connecttris`	$3900e - 670000$	1×10^6	118 %	1×10^7
`in_cross`	$5100e - 1.1 \times 10^6$	1×10^6	99 %	1×10^7
`out_cross`	$5100e - 1.1 \times 10^6$	1×10^6	98 %	1×10^7
`reorder`	$5e^2 - 260e + 5n - 34000$	9×10^5	59 %	1×10^7
`rerank`	$2900e - 670000$	1×10^6	144 %	9×10^6
`dfs_enter_outedge`	$1900e - 420000$	9×10^5	176 %	8×10^6
`_routesplines`	$2400e - 490000$	9×10^5	133 %	7×10^6
`left2right`	$0.33e^{2.34}$	6×10^5	61 %	7×10^6
`leave_edge`	$160e + 0.24e^{2.17} - 16000$	2×10^5	76 %	2×10^6

Figure 3.54: Top several functions in the complex ($n \leq e \leq 1.3n$) `dot` benchmark, ranked by maximum total-self-cost. Rows for the following functions include only recursive calls: `dfs_range`, `dfs_enter_inedge`, `rerank`, `dfs_enter_outedge`.

Function	Model	SE	SE/mean	maximum total-self-cost
`last_node`	$8.2 \cdot \text{bytes} - 78000$	2×10^7	119 %	3×10^8
`clear` 1	$11 \cdot \text{bytes} + 95000$	4×10^6	14 %	3×10^8
`clear` 2	$11 \cdot \text{bytes} + 200000$	3×10^6	14 %	3×10^8
`compile_file`	$3.4 \cdot \text{bytes} - 270000$	1×10^6	21 %	9×10^7
`env_hash` 1	$1.8 \cdot \text{bytes} + 340000$	1×10^6	27 %	4×10^7
`yylex`	$1.3 \cdot \text{bytes} - 29000$	1×10^5	5 %	3×10^7
`clear` 3	$1.1 \cdot \text{bytes} + 55000$	5×10^5	19 %	3×10^7
`dhlookup`	$0.76 \cdot \text{bytes} + 140000$	9×10^5	50 %	2×10^7
`TGETC`	$0.74 \cdot \text{bytes} - 24000$	6×10^4	4 %	2×10^7
`AST_set_parent_list`	$0.63 \cdot \text{bytes} - 110000$	4×10^5	30 %	2×10^7
`env_hash` 2	$0.60 \cdot \text{bytes} - 14000$	3×10^5	27 %	1×10^7
`env_compare`	$0.41 \cdot \text{bytes} + 52000$	8×10^5	81 %	1×10^7

Figure 3.55: Top several functions in the `banshee` benchmark, ranked by maximum total-self-cost. For brevity, we omit the caller-context for these functions.

Function	Model	SE	SE/mean	maximum total-self-cost
`LU1FAD`	49×10^6	6×10^8	1150 %	8×10^9
`prod_xA2`	$3.6v_1 + 92 \times 10^6$	2×10^8	123 %	4×10^9
`prod_xA`	$3.4v_2 + 30 \times 10^6$	2×10^8	256 %	3×10^9
`get_colIndexA`	$10000c + 6.9 \times 10^6$	2×10^8	259 %	3×10^9
`LU6LT`	$5.0v_3 + 19 \times 10^6$	2×10^8	296 %	3×10^9
`LU6UT`	$18000r + 13 \times 10^6$	2×10^8	267 %	2×10^9
`LU6U`	$2.4v_4 + 8600r + 5.3 \times 10^6$	1×10^8	164 %	1×10^9
`LU1MAR`	$130v_5 + 1.1v_6 + -3.4 \times 10^6$	4×10^7	363 %	1×10^9
`LU1GAU`	$86v_7 + 1.3v_8 + -60000$	5×10^7	499 %	9×10^8
`LU7ZAP`	$2300r + 3.2v_9 + 4.2 \times 10^6$	6×10^7	137 %	7×10^8
`my_daxpy`	5.7×10^6	6×10^7	999 %	7×10^8
`LU6L`	16×10^6	6×10^7	375 %	7×10^8

Figure 3.56: Top several functions in the `lpsolve` benchmark, ranked by maximum total-self-cost. We use r for feature `rows`, c for feature `columns`, and v_1 through v_9 for loop count variables.

In considering its results on our large benchmarks, however, we found that some of the models for the functions with high total-self-cost, the functions that account for much of the performance cost of our benchmarks, were not very good. For the rest of this section, we assess the top total-self-cost models for our benchmarks and consider factors that can lead to 1) there being no clean relationship between performance and input size and 2) to CF-TrendProf's choosing inadequate models even if performance and input size are related.

Assessing the Models

Figures 3.52, 3.53, 3.54, 3.55, and 3.56 show the top ten functions, ranked by maximum total-self-cost, for each benchmark. These top fits range from quite good (especially `bzip` and the simple ($e = n$) `dot` benchmark, to mediocre (due to noise or outliers, a common issue with the `banshee` models), to quite bad (especially the `lpsolve` models).

Identifying Bad Models

As we discuss elsewhere, the error measures, and to a greater extent, the best-fit and residuals scatter plots make it clear when a model does not fit the data well. Models with high error are generally bad, though sometimes a single outlier is enough to disrupt a model and cause it to have high error. The residuals plot can reveal systematic error in the model. When CF-TrendProf chooses a constant model or a model in terms of loop counts (as it often does for `lpsolve`) instead of workload features, this choice is a sign that there is no tight relationship between execution count and workload features: adding a workload feature to the model does not decrease error enough to justify its inclusion. Figure 3.57 shows some functions whose total-self-cost CF-TrendProf does not fit well.

Finding Better Features

If we have recognized that there is no clear relationship between performance of a function and a given set of workload features, it would be nice to have some way of coming up with better features or understanding what factors cause performance to behave as it does. CF-TrendProf provides some clues in its detailed analysis of a program's control flow. Examining the models for all variables (not just functions, but also loops and call counts) can show which loops are predictable in terms of features and which are not thus allowing one to localize the parts of the program that depend on more than the provided workload features. Furthermore, even if some views onto performance, for instance a function's self-cost, do not make sense, perhaps considering the transitive-cost or total-self-cost yields better models and more insight into performance.

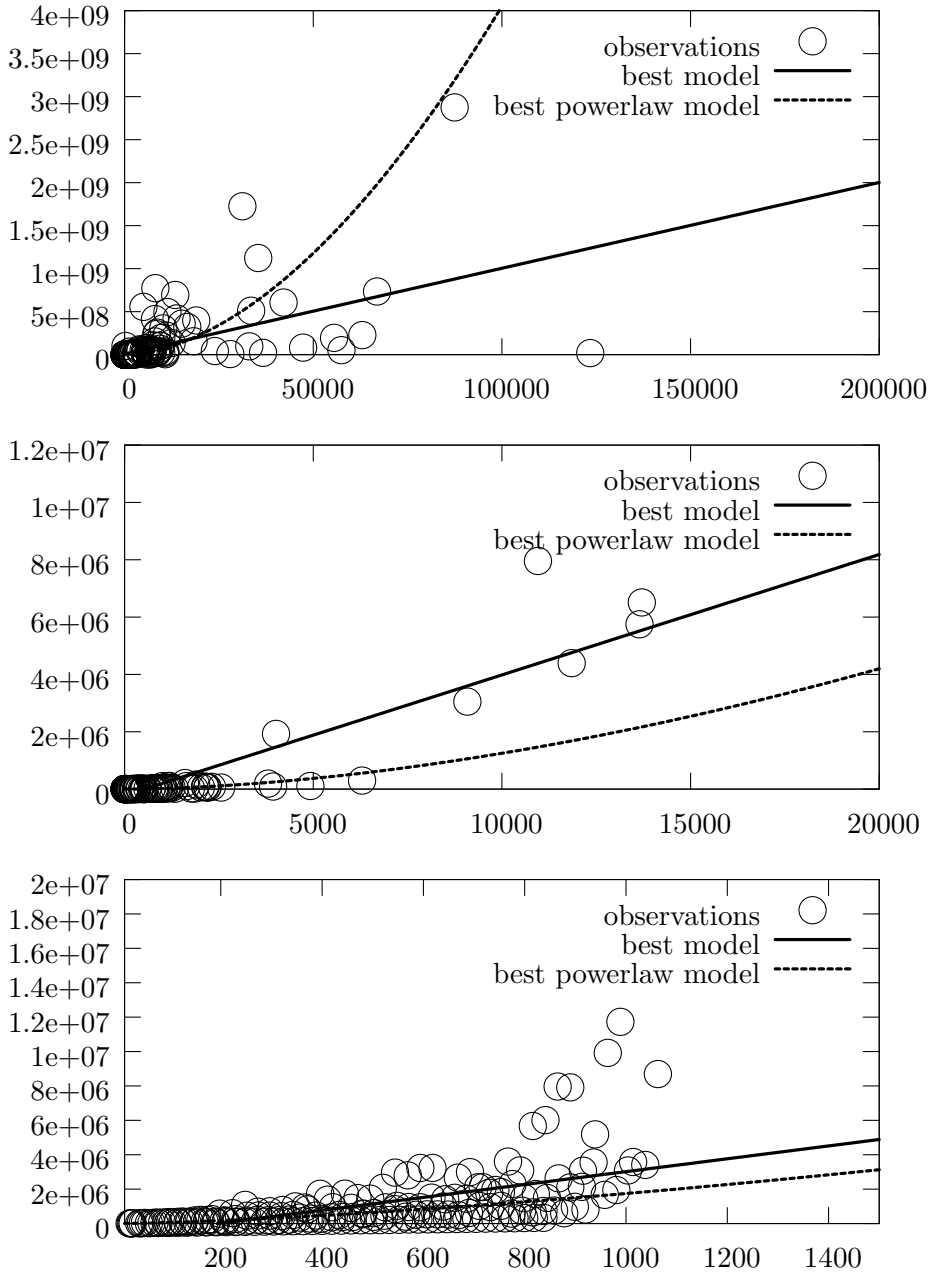

Figure 3.57: Best fit plots for three functions with difficult performance: `lpsolve`'s `get_colIndexA` total-self-cost $:\approx 9980 \cdot \text{columns} + 6.9 \times 10^6$, SE $= 2.26 \times 10^8$ (top), `banshee`'s `member_or_insert` total-self-cost $:\approx 420 \cdot \text{nonemptySets} - 220000$, SE $= 5.3 \times 10^5$ (middle), and complex `dot`'s `ccw` total-self-cost $:\approx 3740e - 720000$, SE $= 1.46 \times 10^6$ (bottom).

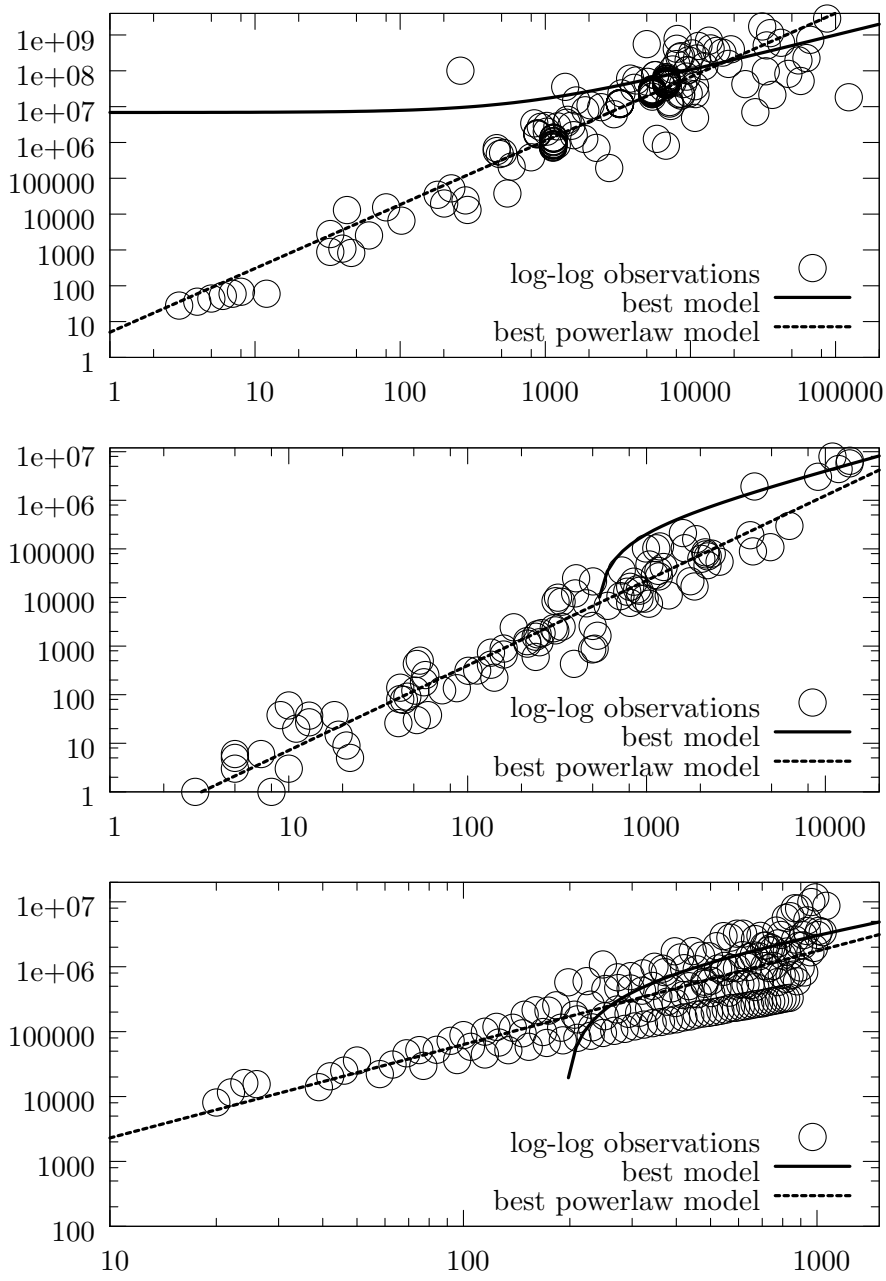

Figure 3.58: The same scatter plots from Figure 3.57, but on log-log axes.

The clustering of BB-TrendProf offers different insights toward finding better features. Even if BB-TrendProf cannot precisely explain the performance of a cluster, it can at least say how many clusters there are — rather than leaving the user with hundreds of basic blocks whose performance remains cryptic, BB-TrendProf might find three clusters. Indeed, since all the locations in a cluster vary together, a cluster captures some facet of how a program's performance changes with workloads. Thus, clusters allow the program to act as a feature detector for the workloads: if one can quantify the features of a workload that make the locations in a cluster execute, perhaps one can better explain the performance of the program. Furthermore, a feature that explains the performance of one location in a cluster is likely to explain all of them. For these benchmarks, however, it is not entirely clear what features might explain performance better.

3.7.5 Performance Trends Depend on Workload Distribution

One factor that can lead to noisy performance relationships is the fact that the performance that CF-TrendProf observes and thus the models that it produces depend on the distribution of workloads. We saw this phenomenon in our insertion sort micro-benchmark in Section 3.5.5 and it also holds true of our larger benchmarks as well.

Comparing the performance of our `bsort` example (Figure 3.1) to that of our `isort` micro-benchmark (Figure 3.21) is instructive. Ignoring the cost of `swap`, `bsort` mechanically does $0.5n^2 + 0.5n + 1$ for any workload of size n. On the other hand, `isort` is more clever: it recognizes easy cases, cases where the input is already partially sorted, and does less work in these situations. For some distributions of workloads, `isort` even does asymptotically less work.

Real programs are generally more like `isort` than like `bsort`. Where they can, programmers find ways to avoid doing extra work rather than mechanically living down to the worst case on every workload. With enough of this heuristic optimization of the cases that seem to occur in practice, a program's performance on some ad hoc subset of inputs improves. On the whole though, this sort of optimization makes the performance on the broader space of possible inputs less predictable. To the extent that the workloads on which we train TrendProf are typical, TrendProf can measure the effect of these heuristic optimizations on the given subspace of program inputs. Workloads whose performance does not fit the model well are interesting: perhaps the heuristic optimizations do not apply or there are other factors that make them expensive and difficult. On the other hand, the performance of a broader set of workloads is harder to characterize.

How Workload Distribution Affects Our Benchmarks

For `bzip` we consider a narrow subspace of workloads: tarballs of source code. It is not a big surprise, then, that we observe clear trends in `bzip`'s performance. On the whole, CF-TrendProf's models for `bzip` are quite good.

In the complex `dot` experiment, where we allow the number of edges to vary from n to $1.3n$ (arguably considering larger and more dense graphs than `dot` was meant to render), we see some cloudy models. Exactly how `dot` grows expensive on these denser graphs is not clear, but the performance of many helper functions is quite clear (e.g., Figure 3.42). Constraining the subspace of inputs to connected graphs with an equal number of nodes and edges, as we do in the simple ($e = n$) `dot` experiment, yields a tighter relationship between features and performance and thus nicer fits. Initially we hypothesized that `dot` would

exhibit linear scaling on the simple set of workloads and only scale super-linearly on denser input graphs. CF-TrendProf's models make it abundantly clear that this hypothesis is false.

The space of possible C programs is enormous, yet the space of real-world C programs, the space from which we pick our workloads for our `banshee` benchmark, is much smaller. Furthermore, the sort of points-to graphs that these C programs induce are not the sort that necessarily induce the worst case cubic performance of Andersen's analysis. In fact, `banshee` is optimized for dealing with these common cases in such a way that it improves performance and avoids bad scalability on the sort of inputs that are likely to arise in practice. While it is true that a carefully constructed points-to graph, or perhaps even a carefully constructed C program, could cause banshee to exhibit worst-case performance, such an input is not likely to arise in practice nor is it important in understanding the empirical scalability of `banshee` on typical workloads. Although CF-TrendProf provides some precise models, we argue elsewhere (Section 3.9) that BB-TrendProf does a better job of presenting the big picture of a program's scalability.

Our `lpsolve` benchmark is the hardest case for CF-TrendProf. Not only is the possible space of inputs huge, but the particular workloads we chose are an eclectic subset with no obvious commonality. Hence, the relationship between features and performance is cloudy and the models are not very accurate. It is notable, however, that CF-TrendProf often presents constant models and models in terms of loop counts for `lpsolve`'s functions instead of nonsense models in terms of workload features; these models indicate that the provided workload features are inadequate to describing `lpsolve`'s performance.

The Benefit of Empiricism

That TrendProf's models depend on the distribution of workloads that it observes is ultimately a double-edged sword. As we saw, it can lead to cloudy results. On the other hand, though, others [AKLW02] have observed that the possible space of inputs can be quite different from those that are probable (and thus important in practice). CF-TrendProf finds the trends that these empirical distributions of workloads induce in the wild: on actual implementations of algorithms as they exist within large programs. As we have shown, particularly with BB-TrendProf, algorithms often beat their worst-case bounds for realistic distributions of workloads.

3.8 Count versus Time

In this section we briefly compare CF-TrendProf's models and rankings to the output of `gprof` on a subset of the workloads for our large benchmarks. For these experiments, we have compiled our benchmarks without optimizations to ensure that the loops and functions that CF-TrendProf sees are the same as those that `gprof` sees.

We emphasize that we do not intend that the execution counts we measure and model be a proxy for execution time per se. Instead, we seek to characterize, in a robust, platform-agnostic way, how the number of operations code performs scales with workload features. Characterizing scalability in terms of number of operations (count) is a necessary part of understanding usage of other resources. Measuring count is enough to point to unexpected asymptotic scalability problems.

Of course, if count had no relation whatsoever to time, measuring and modeling it would be useless. However, the following tables show that count is a reasonable, if imperfect predictor of time (as measured by `gprof`. Thus if one considers models whose predictions are within a factor of one hundred (or so) from the top observed costs and ignores others, one can have some confidence that one is focusing on the code that is important to scalability.

Rankings for `bzip`

CF-TrendProf reports linear scaling in B for the total-self-cost of 19 functions with coefficients between 0.0002 and 114; other total-self-cost models are constant. Figure 3.59 shows the top 10 functions reported by `gprof` and the total-self-cost models that CF-TrendProf computes for them. Of the functions not shown, none has a B coefficient that is higher than 0.1. The coefficients of the linear models, `gprof`, and maximum total-self-cost all rank these functions similarly.

Rankings for Simple `dot` ($e = n$)

Figure 3.60 shows the top ten functions reported by `gprof` (above the line) as well as some other functions with high maximum total-self-cost (below the line). Again, we see that the models, `gprof`, and maximum total-self-cost all rank these functions similarly.

3.9 Comparing CF-TrendProf with BB-TrendProf

Our evaluation shows that both BB-TrendProf and CF-TrendProf make valuable contributions to characterizing and organizing the scalability of actual software implementations run on realistic workloads. Neither technique is strictly superior to the

Function	Percent of time reported by `gprof`	Model for total-self-cost	Maximum total-self-cost
`generateMTFValues`	49.45 %	$114B - 24 \times 10^6$	717
`mainSort`	24.43 %	$2.86B + 210000$	17.5
`sendMTFValues`	8.16 %	$1.19B - 40000$	7.34
`mainGtU`	5.84 %	$1.57B - 47000$	9.61
`mainSimpleSort`	3.91 %	$1.43B - 81000$	8.73
`copy_input_until_stop`	2.91 %	$1.00B + 2.2$	6.12
`bsW`	1.84 %	$2.02B - 140000$	12.5
`copy_output_until_stop`	1.71 %	$0.968B - 98000$	6.01
`mainQSort3`	0.75 %	$0.186B + 67000$	1.24
`BZ2_blockSort`	0.61 %	$0.488B - 140000$	3.19

Figure 3.59: Comparison of `bzip` functions. We show (left) the percentage of time for which the function accounts (according to `gprof`'s estimate based on the sum of the samples from 20 randomly chosen workloads), (middle) CF-TrendProf's model for the function's total-self-cost, and (right) the maximum (over all workloads) total-self-cost, measured in tens of millions (10^7) of executions.

other: they have complementary strengths. Our comparison in this section naturally leads to our discussion of future work in the following section.

Both techniques offer tools for identifying scalability-critical code and eliminating unimportant code from consideration. BB-TrendProf organizes locations into clusters; clusters whose maximum cluster total is low and whose scalability is linear or sub-linear can generally be ignored. CF-TrendProf provides a call tree view; subtrees whose maximum total-transitive-cost is low and whose scalability is linear or sub-linear can generally be ignored. Furthermore, both provide diagnostics (error measures, best-fit scatter plots, and residuals scatter plots) to assess the quality of their models.

Function	Percent of time reported by `gprof`	Model for total-self-cost	Maximum total-self-cost
`dfs_enter_inedge`	11.21 %	$1.5n^2$	12.3
`reorder`	9.05 %	$0.059n^{2.60}$	28.5
`dfs_range`	8.19 %	$2.3n^2$	16.5
`left2right`	6.90 %	$0.0078n^{2.84}$	16.4
`dttree`	6.90 %	$428n - 3600$	3.56
`rerank`	5.17 %	$2.2n^{1.82}$	5.95
`_routesplines`	4.74 %	$189n + 710$	1.61
`connecttris`	4.31 %	$582n - 2600$	4.84
`ccw`	4.31 %	$398n - 1900$	3.31
`exchange`	3.88 %	$270n - 45000$	3.00
`Bezier`	2.59 %	$423n - 300$	3.54
`out_cross`	0.86 %	$0.36n^2$	4.09
`in_cross`	2.16 %	$0.36n^2$	3.58
`leave_edge`	1.72 %	$342n - 28000$	2.93
`dfs_enter_outedge`	1.72 %	$128n - 13000$	2.07

Figure 3.60: Comparison of `dot` functions for the simple run ($e = n$). We show (left) the percentage of time for which the function accounts (according to `gprof`'s estimate based on the sum of the samples from 20 randomly chosen workloads), (middle) CF-TrendProf's model for the function's total-self-cost, and (right) the maximum (over all workloads) total-self-cost, measured in hundreds of thousands (10^5) of executions. The first ten functions (above the horizontal line) are the top ten reported by `gprof`; subsequent functions (below the line) are those with high maximum total-self-cost.

Managing the Complexity of Large Programs

CF-TrendProf's call-graph view allows one to find which functions are the inner loops of which others, to follow performance through the call graph, and to explain to what extent a function's total cost increases on larger workloads because it is called more or because it does more work per call. BB-TrendProf has a complementary strength. By grouping locations with related performance into clusters, BB-TrendProf summarizes the performance behaviors of the entire program. The scalability of these clusters gives a concise overview of how the program scales: considering models for a few dozen costly clusters is easier than considering hundreds of functions. Furthermore, clusters group similar unknown behaviors. In a setting where performance need not be a clean function of workload features, reducing the number of unknown entities is useful.

Modeling Performance

By powerlaw fitting every cluster total, BB-TrendProf provides a coarse and sometimes imprecise, but ultimately concise measure of each cluster's scalability. Each of BB-TrendProf's powerlaw fits to cluster totals constitutes a hypothesis about the cluster's scalability; error measures, scatter plots, and residuals plots provide the evidence to accept or reject the hypothesis. Of course, these powerlaw models may be confused by lower order terms and do not handle multiple features elegantly. In contrast, CF-TrendProf prefers to fit a line to data unless that data is compellingly curvy. These linear models compose into cleaner derived models, but can miss the curve in the data. CF-TrendProf's model generation and selection algorithms solve (for our problem domain)

two problems that are difficult when posed in generality [Ric06]: how to decide what model describes a given set of data points and how to build models involving highly correlated features. CF-TrendProf can fit precise models involving multiple features when these models are appropriate.

Relative Error is More Useful Than Absolute Error

Comparing CF-TrendProf's models to BB-TrendProf's makes clear the importance of how one measures, minimizes, and visualizes error. CF-TrendProf evaluates models based on their standard error, a quantity that scales up with the data's squared deviation from the model's predictions, $\sum_i (y_i - \hat{y}_i)^2$ (see Section A.1.5). While this view of error seems to be the starting point for discussion of regression (see for instance [Ric06]), it is not clear that it is entirely suitable for our purposes. BB-TrendProf implicitly makes a different choice: by fitting its cluster totals to powerlaws (see Sections 2.3.2 and A.1.3), BB-TrendProf chooses to minimize relative error, $\sum_i \left(\log \frac{\hat{y}_i}{y_i}\right)^2$.

This relative view of error leads to models with a different sort of guarantee. Consider what it means for a model to have low absolute error, versus low relative error, as cost increases. Roughly speaking, a model has low absolute error to the extent that the squared difference between its predictions and actual performance is small; for instance, most of a good model's predictions would be off by less than one hundred ($\hat{y} - 100 < y < \hat{y} + 100$) and very few would be off by more than one thousand. A model has low error by the relative notion of error to the extent that its predictions are within a small factor of actual performance; for instance, most of a good model's predictions would be within a factor of two ($0.5\hat{y} < y < 2\hat{y}$) and very few would be off by more than a factor of ten. So a model

with low relative error corresponds to the familiar notion in theoretical complexity of being within a constant factor of actual performance (though of course TrendProf's models are more like averages than bounds).

To make this discussion more concrete, compare the view of absolute error implicit in the linear-linear plots in Figure 3.57 to the view of relative error implicit in the log-log plots of the same data in Figure 3.58. The vertical distance to the line of best fit in the linear-linear plots corresponds to absolute error; the vertical distance in the log-log plots corresponds to relative error. There are many situations where the absolute error of CF-TrendProf's models increases as the cost they model increases, but as these log-log plots show, the relative error often (but not always) stays relatively fixed. The log-log plots have the additional advantage of showing the performance trend in the data across all orders of magnitude while the linear-linear plots really focus only on the largest points.

Given these results, we find the relative notion of error more appealing. In fairness, we must acknowledge that Brewer [Bre94] advocates regression models that minimize relative error to describe the performance of programs; we pursued a more standard approach to regression with CF-TrendProf because we found his arguments unpersuasive compared to the difficulty and extra machinery they require, but we now share his belief. Rather than trying for (and failing to achieve) models that predict error to within a small absolute distance, it is more valuable for our purposes to have a model that characterizes how performance grows asymptotically to within a constant factor — this tolerance for constant factors is built in to the notion of big-O and big-theta bounds.

Unfortunately, simply changing CF-TrendProf's linear models to minimize relative error, though a useful step, is insufficient to yield any drastic improvements in models. While small changes might yield a modest improvement for some models, the evidence and experience accumulated in this thesis suggests a more thorough re-consideration. As we elaborate in Section 3.10, there are several opportunities for improving our methodology. A real solution ought to put powerlaw models and linear models on equal footing: they should minimize the same notion of error; too many of CF-TrendProf's models approximate a curve with a line because this difference in error puts powerlaw models at too much of a disadvantage. However, such a change is not trivial: without our trick of doing linear regression on $(\log x,\ \log y)$, fitting powerlaw models requires a potentially unstable iterative optimization process. Furthermore, other issues beg for a solution as well (Sections 3.10.3, 3.10.4, and 3.10.5) and impede any improvement based purely on improving the notion of error.

In any event, the exact notion of error is not the central thrust of this thesis. Even with a better notion of error, the good models will stay good and the hopeless data sets will still be hopeless. A change in our notion of error will most strongly affect those models that have some predictive power, but also have a flaw: outliers, a missing lower order term or logarithmic factor, or just general noisiness. A different notion of error will affect how the model fitting adjusts the model in the presence of the flaw. Other approaches, like considering a wider class of models or improving our ability to distinguish distinct performance contexts might more directly address the flaw.

Complementary Strengths

In conclusion, BB-TrendProf and CF-TrendProf have complementary strengths: CF-TrendProf is more precise in some situations, while BB-TrendProf better manages the complexity of large programs. CF-TrendProf creates precise models with multiple terms and multiple features when these models are justified by their low error and the program's control flow. In some cases, CF-TrendProf even produces exact or near-exact fits (for example, see Figures 3.13, 3.41, and 3.42). Because the relationship between performance and workload features is much harder to characterize for the core of large programs and complex algorithms, the added precision that CF-TrendProf brings is not as big a win over BB-TrendProf in these contexts. Large programs present a further problem for CF-TrendProf because its call-graph-centric view of performance, though useful, is not as succinct a summary of performance as BB-TrendProf's clusters. While CF-TrendProf makes important strides in modeling program performance and does much better than BB-TrendProf in some cases, its inability to manage the complexity of the performance of large programs makes it unwieldy where BB-TrendProf scales more gracefully. Fortunately, the features of BB-TrendProf that make it suitable for analyzing the performance of large programs are portable to CF-TrendProf. We discuss issues related to incorporating the best features of BB-TrendProf and CF-TrendProf in Section 3.10.1.

3.10 Future Work

Our work on BB-TrendProf and CF-TrendProf makes progress on characterizing and organizing the scalability of actual software implementations run on realistic workloads. Our results show that solutions to this problem must manage the difficult reality that performance is not always a clean function of workload features and that differing distributions of workloads can lead to different apparent scalability. Much of the future work we envision below has to do with refining the techniques we have investigated to better meet these challenges.

3.10.1 Combining Strengths of BB-TrendProf and CF-TrendProf

As we saw, BB-TrendProf and CF-TrendProf have complementary strengths. Providing both a call graph view of performance and a decomposition of locations into clusters would help manage the complexity of having many models for many locations while still enabling reasoning about how performance is distributed through the call graph. Forming clusters and cluster totals based on functions' total-self-costs seems right since these measurements are a partition of the program's total performance. Furthermore, coloring total-transitive-costs by the clusters to which they belong could aid in understanding how different call trees vary and which call trees scale worst.

We have argued (Section 3.9) that models of empirical computational complexity should minimize relative, rather than absolute error. More concretely, we must develop methods for linear and powerlaw (and perhaps other kinds of) regression that minimize some function of relative error ($\frac{y_i - \hat{y_i}}{y_i}$). With linear and powerlaw fits competing on even

ground — both minimizing the same measure of error and competing based on this error measure — curvy data should fit a powerlaw and linear data a line. Furthermore, the log-log scatter plot would seem to be more appropriate for understanding the relationship between performance and workload features since, unlike a linear-linear scatter plot, a constant relative error corresponds to constant distance.

3.10.2 What Is the Distribution of The Error Terms?

As we mention in Section 3.4.3, it is not clear what sort of distribution characterizes the error terms in our models. In particular, if we claim that a model, $\hat{y}(x)$, explains performance, then what distribution of error terms ($\frac{\hat{y}_i(x_i)-y_i}{y_i}$) is acceptable? Characterizing these error terms would allow for analytical reasoning about properties of our models such as confidence intervals (for regression parameters and predictions), and a more rigorous model selection criterion along the lines of Jaynes [JB03] or Brewer [Bre94].

Because of the difficulty of finding good models to predict program performance, we are willing to accept models that have systematic bias (for example, because they approximate a logarithmic factor as a powerlaw or miss a lower order term) as long as they are good approximations of actual performance; for example, models that approximate a logarithmic factor with a powerlaw or models that are missing a lower order term. To the extent that they are good approximations, these flawed models are still useful to a human — probably more useful than declining to fit any model. However, characterizing the distribution of the error terms for such models is challenging.

Empirical Bounds Versus Empirical Averages

Consider the performance behavior shown in Figure 3.24, Figure 3.31, and the bottom plot of Figure 3.57. For such situations, it might be profitable to formulate model fitting approaches that, rather than penalizing model overestimates (negative residuals) and model underestimates (positive residuals) uniformly, instead penalize underestimates more severely than overestimates. Such an approach should yield models that follow the upper line of points more closely than the lower line in noisy situations like those in the figures mentioned above.

In contrast to a big-O bound such an approach can provide no guarantee of performance. The utility of such an approach, though, is that it manages the reality of noisy relationships between performance and workload features by focusing on the trends in the more expensive workloads.

3.10.3 A More Robust Class of Models

CF-TrendProf's derived models are an all or nothing proposition: either the derived model with all of its terms wins or a direct model wins. Some of these derived models can have odd-looking terms like $3x^{1.67} + 4x^{1.24}$ (though recall that such terms are well motivated by the structure of the program). Two ways of generating additional candidate models from a more robust class of models are apparent. First, we could use the ceiling of the maximum degree term of each feature in our models to suggest the degree of a regularized polynomial model (e.g., 3 for $x^{2.3}$). Second, we could try dropping terms and re-computing coefficients for other terms. These approaches would smooth and simplify our

derived models when such simplification did not result in substantially less precision. An important question to consider, though, is whether these approaches are more likely to lead to over-fitting of training data.

3.10.4 Inferring Contexts

CF-TrendProf allows the user to mark function invocations with contexts based on the call graph or on arbitrary runtime data values. One cause of messy performance relationships is combining contexts; splitting them can yield cleaner, more precise models. There are situations where it might be possible to split contexts automatically.

Call Stack and Data Contexts

Some sort of clustering approach (something like k-nearest-neighbors, not to be confused BB-TrendProf's clustering) might identify situations where invocations of a function with different callers (or call stacks) caused the function to behave differently. More ambitiously, one might record data values, either user-provided or mined automatically from function parameters, and automatically determine if these values had any measurable correlation with performance.

Different Kinds of Workloads

Another step in the direction of identifying relevant contexts is for the user to (optionally) describe workloads with some sort of tag. For example, our first `dot` experiment might tag workloads with n edges, $1.1n$ edges, $1.2n$ edges, and $1.3n$ edges differently. These tags would be treated as contexts for all the measurements in the workload. Functions

whose performance seemed to be affected by the workload tag would be modeled separately for each tag. This sort of approach would offer a tool for exploring a larger portion of a program's input space and getting a handle on how different kinds of workloads affected (or did not affect) performance trends.

3.10.5 Improved Handling of Recursion

Overall, CF-TrendProf's handling of recursion is inelegant. Modeling per-invocation cost of a recursive function entangles the initial call with the subsequent recursive calls and is unlikely to result in sensible models. Contexts (and our derived models, see Section 3.4.4) allow one to split out the initial call into a recursive cycle from the subsequent recursive calls, but this tedious process could be automated. One possible way forward is to treat loop iterations and function calls more uniformly: record each function entry or loop iteration in the trace, perform an interval analysis [ASU86] of the entire control flow of the program to find the loops, and model the self-cost, total-self-cost of each loop per entry and per invocation of each higher scope. This approach would be yet more data intensive than CF-TrendProf and would require more sophisticated compression of traces and presentation of data.

3.10.6 Toward Modeling Time

This work has demonstrated that modeling execution count as a function of workload features yields valuable insights into the scalability of programs. Given infrastructure for making the measurements, it is a triviality to substitute other measures of performance (machine instructions, cache misses, time) for execution counts in our methodology for

model selection and fitting — indeed, others [SY07] have done so. The important empirical question to investigate is whether the performance effects caused by caches and other micro-architectural features can be reasonably modeled with a simple statistical tool such as regression or whether more sophisticated techniques or other trade-offs are necessary.

3.10.7 Outliers and the Program as a Feature Detector for Workloads

One interesting aspect of TrendProf's models is that they, in the best case, establish a clear trend in performance — a baseline. Workloads that deviate from this trend (outliers in the scatter plots) are interesting because they violate this baseline behavior. Even though such workloads may not be particularly expensive, they invite one to ask why their performance deviated from the overall performance trend. Finding and characterizing these workloads (based on their performance behavior across the entire program) might point to neighborhoods of related workloads with properties that cause bad performance. A larger workload in such a neighborhood (that is, having whatever properties cause other workloads in that neighborhood to be expensive) might cause performance problems. Thus, identifying these neighborhoods has the potential to enhance TrendProf's ability to find scalability issues by extrapolating trends rather than observing problematic workloads.

Chapter 4

Threats to Validity

This chapter reviews concisely the circumstances under which TrendProf's models do not adequately describe program performance. Knowledge of these hazards has been a driving factor in the design of our techniques.

Some of the hazards we discuss illustrate the fundamental difficulties and trade-offs inherent in empirically modeling program performance as a function of workload features. For instance, while modeling performance based on measuring actual workloads focuses TrendProf's models on the empirical case, choosing atypical or insufficiently many workloads to train TrendProf causes its models to over-fit patterns particular to the chosen workloads (Section 4.1). Furthermore, obvious workload features may not be good predictors of performance (Section 4.2); put another way, performance may depend on properties of the workload that are difficult to measure (without running the program). TrendProf's best-fit and residuals plots and BB-TrendProf's bootstrapped confidence intervals seek to mitigate these hazards by allowing the user to recognize bad fits and outliers.

Other hazards are more specific to the implementation of BB-TrendProf or CF-TrendProf. Both systems intentionally limit the complexity of models that they use to fit performance data. While this limitation prevents over-fitting noisy performance data with a complex model, it also forces approximation of, say, polynomials with powerlaws or logarithmic terms with constants or powerlaws (Section 4.3).

4.1 The Importance of Workloads

The empiricism of TrendProf's models is both an advantage and a disadvantage. All of the models TrendProf builds are based on measuring a set of workloads the user provides. This set of workloads allows TrendProf to reason about scalability in the empirical case, often a difficult feat in theoretical settings. On the other hand, choosing atypical or insufficiently many workloads to train TrendProf causes its models to over-fit patterns particular to the chosen workloads.

When Workloads Reveal Empirical Truth

TrendProf does not distinguish correlations that are due to the structure of the program from those due to the distribution of workloads. This empiricism allows us to conclude that on typical C programs, an optimized implementation of Andersen's analysis scales much better than its worst-case bound of $O(n^3)$ in the size of the program (Section 2.4.4) and that a linked list append function that runs in linear time in the length of the list *is* a performance bug in `banshee`'s parser (Section 2.4.5), but the same idiom is *not* a bug in the context of `elsa`'s data structures for resolving name lookup (Section 2.4.6).

When Workloads Oversimplify

On the other hand, the user of TrendProf must choose workloads carefully or risk generating results that do not generalize. We illustrate this point further by considering four different kinds of workloads for our bubble sort example (for another example, see Section 3.5.5). Recall that the workloads we considered earlier (Section 2.2) were arrays of integers generated uniformly at random and that the locations break into 3 distinct clusters: compares, swaps, and size (Figure 2.1). Depending on the distribution of inputs, BB-TrendProf's classification of line 6 (swaps) changes: if our inputs consist respectively of arrays of integers (a) randomly permuted, (b) sorted from least to greatest, (c) sorted greatest to least, or (d) sorted from least to greatest but with $O(n)$ swaps of neighbors, then we observe respectively that line 6 (a) scales as $n^{1.93}$ and forms its own cluster (swaps), (b) never executes and thus does not appear in the output, (c) executes about $O(n^2)$ and thus falls into cluster compares, or (d) executes about $O(n)$ times and falls into cluster size.

In fact, line 6 may powerlaw-fit n quite poorly: any combination of these extremes is realizable for line 6 by picking suitable workloads. In contrast the cost of the other lines varies only with the size of the array, so their classification does not change.

Outliers in the best-fit scatter plots suggest the possibility of workloads that behave differently than the prevailing performance trend. Running more workloads, particularly workloads similar to the outlier, may increase the generality of TrendProf's results.

4.2 Performance Is Not Always a Function of Workload Features

It is rare that a function of workload features perfectly predicts the performance of a piece of code (though see Figure 3.13). Often though, there is a at least a trend in performance as some workload feature grows large (Figures 3.15, 3.30, and 2.11). Sometimes, however, the performance of a piece of code is simply not a function of any readily apparent feature of its input (Figures 2.12, and 3.24). For instance, depending on the distribution of inputs to the bubble sort example, SIZE may be a reasonable powerlaw predictor of SWAPS, but (as we discussed above) it may not be. There is no function that predicts SWAPS in terms of SIZE in general.

Similarly, AST does not adequately predict the points in Figure 2.12 nor the cost of the top cluster for `elsa` (not shown). It may be that some function of some readily available features of `elsa` workloads fit this data well, but we do not know. The performance curve for some programs may not even increase monotonically with workload size. In these situations, it is clear from the best-fit plot and residuals plot that TRENDPROF provides that its model is inadequate for the situation and that its predictions are not to be trusted.

For situations like these, BB-TRENDPROF allows the user to define features that depend on the runtime behavior of the program. One can designate the number of times a particular line of code executes as a feature for BB-TRENDPROF. Also, BB-TRENDPROF does not require workloads to be annotated with features until after they have run; the programmer may, for instance, modify the program to print the size of a data structure or the value of a counter and then use these as features. Furthermore, clustering identifies

groups of basic blocks that vary together, suggesting that they depend on the same subtle features on input — the exact property of the workload that causes performance to vary may not be clear, but locations that exhibit the same variations are grouped.

Consideration of this issue led to several features of CF-TrendProf. Its decomposition of performance allows for multiple views onto performance, essentially allowing more chances for finding a meaningful chunk of a program's execution that has some relationship to workload features — one call to a function might not bear any relation to workload features, but all the calls from a particular caller or all the calls in a workload might; the transitive-cost of a handful of callees might not follow any discernible trend, but their sum, as captured in the transitive-cost of their caller, might. Furthermore, by annotating their program with invocation features and context annotations, the user can help CF-TrendProf identify clear relationships between values of program variables at runtime and program performance.

4.3 Inability to Find the Right Model To Fit

There may be some relationship between performance and user-provided workload features, but TrendProf may not choose the right model to capture this relationship — most likely because this model is not in TrendProf's vocabulary. This situation is different from the one we discuss above where there is no relationship between user-provided features and input size. When there is no relationship between provided features and input size, the best one can hope for is that there is yet one more feature that will explain performance where others have not or that considering performance from a different vantage point

(transitive-cost versus self-cost or total-transitive-cost versus transitive-cost or transitive-cost of a caller versus transitive-cost of a callee) is more enlightening. When the data (particularly the best fit scatter plot) shows a clear relationship between performance and a feature, we must ask instead what sort of model might TrendProf fit to this data and is including this model to fit true instances of it worth the "false positives" of over-fitting noise using this model.

4.3.1 Limitations of the Powerlaw Fit

Our first technique, BB-TrendProf, considers only linear and powerlaw models. In our experience the simple, two-parameter powerlaw fit works amazingly well. However, there are situations where a powerlaw fit does not precisely capture the variation of a cluster's cost across workloads. These situations are quite clear when we examine the scatter plots and residuals plots that TrendProf generates. Wide confidence intervals for the coefficient and exponent or a low R^2 are also warnings that the powerlaw may not be a suitable model. The converse does not hold: these statistics may still be quite good for data that a powerlaw does not adequately describe.

The Logarithmic Factor Although a powerlaw cannot fit functions such as $n \log n$, such logarithmic factors are not a major problem in practice. For example, the number of compares that quicksort performs grows as $O(n \log n)$ where n is the size of the array being sorted. The left part of Figure 4.1 shows a scatter plot of the number of compares a Quicksort performs (y axis) versus the number of elements in the array to be sorted (x axis). The line is a powerlaw fit to the diamond shaped points ($\hat{y} = 1.5x^{1.16}$). The fit closely

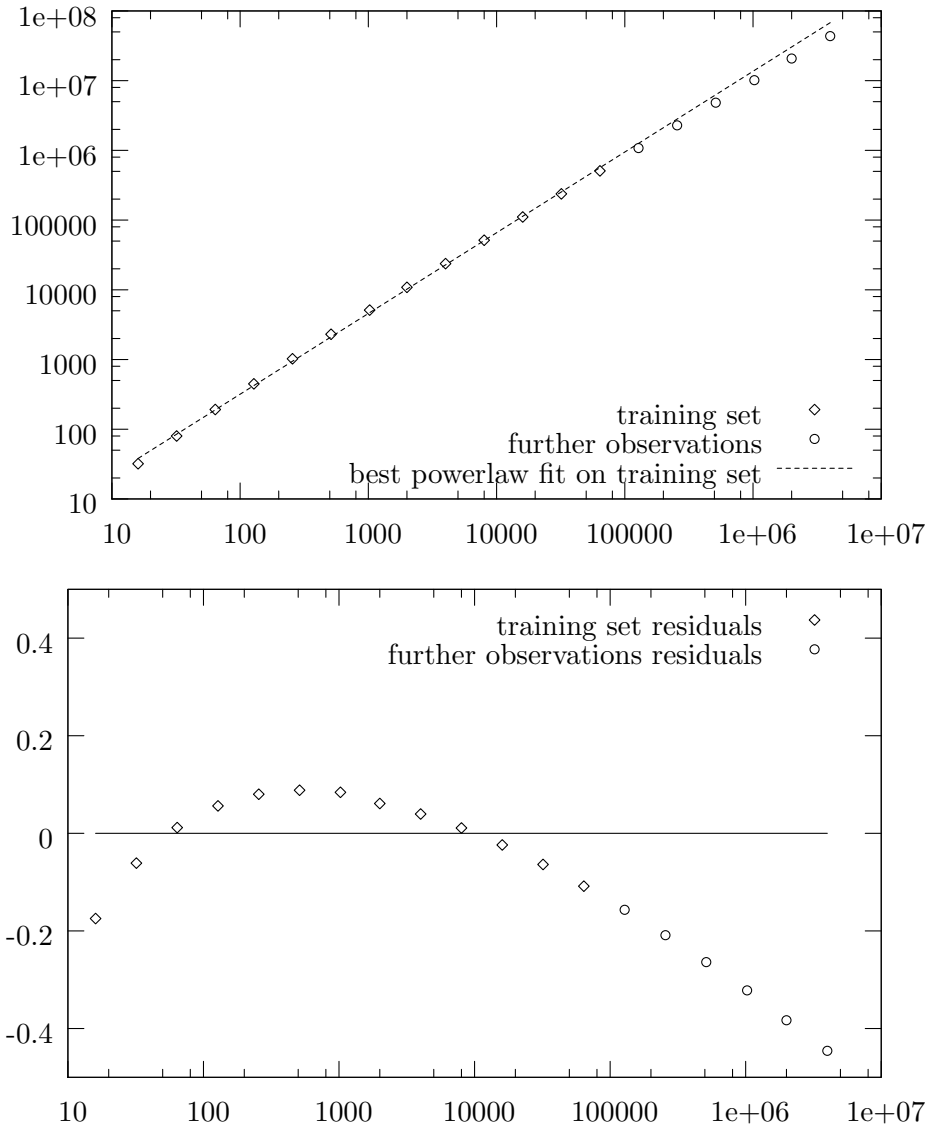

Figure 4.1: On the top is a log-log plot of number of comparisons done in a call to `qsort` (y axis) versus the size of the array (x axis). On the same plot, we show the best powerlaw fit to the diamond shaped points ($\hat{y} = 1.5n^{1.16}$, $R^2 > 0.99$). On the bottom is the residuals plot for the powerlaw fit. Note that the residuals are clearly not randomly distributed.

tracks the data, but it is clear from the residuals plot that there is more going on. The hump shaped residuals plots suggests that the data grows more slowly than the powerlaw; such a curve suggests a logarithmic factor.

The circular points show further observations of compares versus array size. Even for arrays 60 times larger than any BB-TrendProf used to fit the initial powerlaw, the fit's prediction (68 million compares) is less than a factor of two from the observed value (43 million compares).

The Lower Order Term Our bubble sort example illustrates the effect of a lower order term on a powerlaw fit. Line 4 executes exactly $0.5n^2 + 0.5n$ times while lines 5 and 7 execute exactly $0.5n^2 - 0.5n$ times each; the cluster as a whole costs $1.5n^2 - 0.5n$ basic block executions. The powerlaw fit converges to the highest order term: given large enough workloads, BB-TrendProf predicts the cost of this cluster as $1.5n^2$. That is, for smaller workloads the lower order terms distort the powerlaw fit; however, for large enough n, the quadratic term dominates the linear one. To the extent that one term dominates the others, BB-TrendProf's powerlaw fit is a reasonable, low-dimensional approximation. This distorting effect that lower order terms can have on powerlaw fits led to our desire to explore more precise fits with CF-TrendProf.

4.3.2 Limitations of CF-TrendProf's Model Selection

In choosing a model for performance, CF-TrendProf has a wider range of possibilities than BB-TrendProf. Nonetheless, the number of terms in any given model is bounded by the control flow of the program. If a single loop has performance that scales

quadratically or cubicly with a workload feature, CF-TrendProf is forced to model it with a powerlaw instead of a more precise polynomial. Similarly, CF-TrendProf only combines models (into derived models) according to control flow; for instance if an inner loop's performance depended on $x_1 \times x_2$, CF-TrendProf will not find this model unless the outer loop's performance depends on either x_1 or x_2 since it will not try the model $x_1 \times x_2$ directly nor will it be able to construct a derived model.

These limitations arise from a trade-off in the design of CF-TrendProf. Since there is no limit to the complexity of code's performance and since there need not be any relationship between performance and workload features, CF-TrendProf prefers to choose simple models over complex ones unless there is evidence in the program's control flow to support the choice of a complex model. Alternative approaches might try more models, but these approaches would have to decide which models to try (and which to skip) and this choice leads to trading off model simplicity and interpretability for more precision. The danger of complex models is that they tend to over-fit noise: situations where there is no good model would have a complex, difficult to understand model instead of a simple line or powerlaw that averages the noise.

The ultimate arbiter of the goodness of a CF-TrendProf model is the human user. Complex models, especially those with two or more variables, impede the comprehensibility of a model. Therefore, CF-TrendProf errs on the side of choosing simple models in the absence of any reason to do otherwise; any complex models are derived by composing simpler models of subcomponents, each of which can be evaluated separately.

Chapter 5

Related Work

The main branches of related work are other profilers and other techniques that construct models of program performance based on simulation, measurement, or reasoning about source code.

5.1 Profilers

Gprof [GKM82] and many profilers like it periodically sample the program counter during a single run of a program. A post-processing step propagates these samples through the call graph to estimate how much of the program's running time was spent in each function. Such profilers are the standard way to find opportunities to improve a program's performance.

Ammons et al. [ACGS04] describe a framework for finding bottlenecks in large programs based on execution profiles. Using their framework, they develop two tools for digesting profile information and finding performance bottlenecks. One tool finds expensive

call sequences (e.g., `F` is expensive when called from `G`, but not when called from `H`). Another tool compares two runs of the same program with the same workload but a different program configuration (for instance with runtime security checks enabled versus disabled) to find the paths responsible for making one configuration fast while the other is slow. They emphasize the importance of extensibility in how their interface summarizes and displays the costs of program paths — in our terms, they enable a tool designer to define new notions of location and context to focus on very specific paths through the program. In this regard, their work is complementary to ours; their system computes the cost of a path for a workload or two, while TrendProf builds models to describe how the cost of a location increases with workload features. An exciting piece of future work might combine their fine grained control over profiles with a TrendProf style analysis to examine how the costs of particular paths through the program grow with workload features.

Jinsight EX [SdPK01] is another tool for managing the vast sea of data that comes out of profiling a large program. Jinsight exhaustively traces the execution of a Java program, recording the number of objects of a particular type that are allocated, the number a times a method is called, etc. To aid in exploring the sequence and resource usage of the program, Jinsight allows the user to organize subsets of program activity (thread creation, method invocation, object allocation) into *execution slices* based on static and dynamic properties of the trace. For example, the user can specify a set of methods whose invocations (perhaps with their callees) constitute an execution slice; furthermore, dynamic properties like object lifetime or data values can also define an execution slice. The user can then filter the call tree or resource usage histograms based on these execution slices. Like

Bottlenecks, Jinsight is complementary to TrendProf: it defines a rich, dynamic notion of location and enables exploration of the call tree and resource usage of a single workload; in contrast, TrendProf finds performance trends across many workloads.

We built TrendProf to answer questions that these traditional profilers do not address: traditional profilers present information about one run of the program, whereas TrendProf presents a view across many runs with an eye toward finding trends and predicting performance on workloads that have not been run.

5.2 Empirical Performance Models

Kluge et al. [KKN05] focus specifically on how the time a parallel program spends communicating scales with the number of processors on which it is run. In our terms, they construct an empirical model of computational complexity where their measure of performance, y, is MPI communication time and their measure of workload size, x, is number of processors. They fit these observations to a degree-two polynomial, finding a, b, and c to fit ($\hat{y} = a + bx + cx^2$). Their goal is to find programs that do not parallelize well; that is, programs whose amount of communication scales super-linearly with the number of processors. Any part of the program with a large value for c is said to parallelize badly. The goal of TrendProf is more general; we aim to characterize the scalability of a program in terms of a user-specified notion of input size.

Su and Yelick [SY07] adapted the BB-TrendProf methodology and much of the prototype code to build `ti-trend-prof`, a tool for debugging communication-performance for Partitioned Global Address Space (PGAS) languages like Titanium [YSP+98]. In PGAS

languages, remote reads and writes look exactly the same as local ones; this sameness makes code easier to write, but communication performance bugs harder to spot. Indeed, some communication bugs are not apparent until a program is run at scale on hundreds of nodes or at large problem sizes. Fortunately, TrendProf-style performance models help a great deal. The user runs their code with a fixed number of processors and several problem sizes and then again on a fixed problem size with varying numbers of processors. For every program point that does remote memory accesses, `ti-trend-prof` measures the number of communication calls and builds models that describe how communication scales with problem size or number of processors. These models point to performance bugs: places where communication grows faster than it ought to. The authors report that using `ti-trend-prof`, which uses the methodology in Chapter 2, they found performance bugs in hours that would take days to find manually. Furthermore, using `ti-trend-prof` allows them to do meaningful performance debugging on a laptop instead of a super-computer and earlier in the development cycle instead of later. Their work is a triumph for the methodology we describe in this thesis.

Brewer [Bre95] constructs models that predict the performance of a library routine as a function of problem parameters; for instance the performance of a radix sort might be modelled by the number of keys per node, radix width in bits, and key width in bits. Given a problem instance and settings of the parameters, the model predicts how several implementations of the same algorithm perform. Based on the prediction, the library chooses an implementation of the algorithm to run for an instance of the problem. The user must choose the terms for a model; powers of the terms are not considered in building the model,

but cross terms are. For instance, for problem parameters l, w, and h, the model is in terms of

$$\hat{y} = c_0 + c_1 l + c_2 w + c_3 h + c_4 lw + c_5 lh + c_6 wh + c_7 lwh$$

The requirement that the user provide the terms for the model, particularly the powers of those terms, assumes a deeper level of understanding of the code's performance than TrendProf does: while the resulting models can be more descriptive and precise, each implementation of each algorithm must be considered separately and terms chosen carefully. However, in the larger context of the program, the features on which a code's performance depends may not be readily apparent; furthermore, due to bugs, gaps in the user's understanding, or fortuitous configurations of inputs, the scalability of program may not be what the user expects. Therefore, TrendProf seeks to describe the performance of each of the many locations in a large program and focus the user's attention on those with unanticipated performance or scalability problems. Crudely put, TrendProf is concerned with finding the right exponents of the right terms to describe performance for each location in an entire program while Brewer's work is concerned with finding the right coefficients of the right terms for smaller pieces of code. Our differing goals lead us to different assumptions and trade-offs.

Sarkar [Sar89] predicts the mean and variance of loop execution times using counter-based profiles. His system measures the execution frequency of each basic block, carefully optimizing placement of counters based on interval structure and control dependences. After collecting these basic block frequencies for a workload, he uses them together with a static estimate of how much time each basic block takes to run on the target ar-

chitecture to estimate the mean and variance of the run time of each loop. Rather than predicting run time for a workload on a particular architecture, TrendProf predicts the number of operations a piece of code will perform as a function of workload features.

In as yet unpublished work, Ganapathi et al. [GKD+08] consider the problem of predicting the performance, measured in elapsed time, CPU time, disk IO operations, and network traffic, of a database query before it starts executing. Like TrendProf, they seek to predict performance from workload features and make their performance predictions based on measurements of other workloads; though, their notion of performance is richer than TrendProf's. In order to predict the performance of a novel query, they use a statistical machine learning technique called Kernel Canonical Correlation Analysis (KCCA) to essentially interpolate an estimate of the novel query's performance based on the similarity of its feature vector with that of training examples. Compared to TrendProf, their technique trades off the interpretability of its models for precision in predicting performance; indeed, the authors note that dissecting the workings of KCCA is computationally difficult. Furthermore, whereas TrendProf models performance for each location of a general program, they consider whole-program performance for a constrained set of programs (database queries) with a rich set of features, including data from the query optimizer's cardinality estimates for joins and other relational operators. Fundamentally, their work makes different trade-offs than TrendProf in order to solve a more constrained problem.

5.2.1 Modeling Micro-architecture Parameters

Vaswani et al. [VTSJ07] build regression models that relate a benchmark's performance to micro-architectural parameters, compiler optimization flags, and associated

compiler optimization heuristic parameters (for instance maximum loop unrolling). They use these models to (a) predict performance at arbitrary compiler and micro-architecture settings, (b) identify micro-architectural features that interact (both beneficially and detrimentally) with compiler optimization settings, and finally (c) find optimal settings for a particular program. They use three different regression techniques to find models and in one case give up on interpretability in favor of precision.

Along similar lines, Lee and Brooks [LB06] build regression models to predict (a) performance and (b) power consumption for varying micro-architectural parameters. They find that these two modeling problems benefit from different statistical techniques.

These systems (Vaswani et al., Lee and Brooks) explore a vast space of design trade-offs. Their focus is on choosing good designs or understanding interactions of design decisions. In contrast, TrendProf focuses on modeling program cost as workloads change.

5.3 Performance Models by Simulation

There is a long history of predicting the running time of complex systems, such as distributed systems and embedded systems (including those with real-time performance constraints), via simulation. These simulations are often geared towards making system design decisions, tuning system parameters, or deciding how much capacity a system needs to sustain the desired level of throughput. The literature is too vast to adequately discuss here, but we consider some examples.

5.3.1 Simulation of Distributed System Performance

Rugina and Schauser [RS98] simulate the computation and communication of parallel programs to predict their worst-case running time. Their simulation takes as input (a) a parallel program whose communication does not depend on its data, (b) parameters for the program such as size of data blocks and a communication pattern, and (c) LogGP [AISS95] parameters for the target machine; their simulation outputs a time. Their focus is on tuning the performance of a constrained class of program (for a fixed workload size) by choosing the best data block size and communication pattern from among those they simulated. Their work solves a substantially different problem than TrendProf.

Avritzer and Weyuker [AW04] describe a case study where they test, tune, and simulate the performance of an e-commerce application. They build a simulation aimed at reproducing, diagnosing, and fixing an infrequent, but serious performance slowdown. Based on their experience with the system, they built their simulation to model the effects of the following factors on system performance: the dynamics of their particular Java Virtual Machine's garbage collector (including the fact that it stops all threads for a full garbage collection), the heap size of the garbage collector, the memory requirements of each thread, quality of service algorithms used to throttle or refuse connections, number of threads, arrival rate for work, etc. Based on varying the parameters of their simulation, they diagnosed the problem as happening due to the large delay imposed by garbage collecting a 3GB heap combined with the kernel overhead caused by a large number of threads. By further simulation, they found that setting the heap size to 1GB, using a quality of service enforcement algorithm, or running several instances of the application server on the

same multi-processor node fixed the issue. Thus, they found a specific problem in a specific system by constructing and querying a performance model at the right level of abstraction.

5.3.2 Simulation of Embedded System Performance

Thiele and Wandeler [Thi07] survey some techniques for simulating embedded system performance for deciding issues such as which functions should be implemented in software and which in hardware, which hardware components should be used, which buses or processors are likely to be bottlenecks, etc. They mention that simulation is insufficient to establish solid worst case execution time (WCET) bounds for schedulability of real-time systems and that typically, analytic methods must be used.

5.3.3 Statistical Models Versus Simulation

These systems are not particularly similar to TrendProf, but there is one piece of common ground. Like TrendProf, the systems above seek to isolate parameters that affect program performance and predict performance as these parameters change. The examples above solve a myriad of problems related to how the parameters of complex systems and the interactions of their components affect their performance; exploration of the parameter space varies from fully manual to fully automatic depending on the nature and structure of the problem.

In contrast, TrendProf takes aim at a specific problem: how the number of operations a program performs grows with input size. This focus on a single aspect of scalability allows us to use an interpretable formula (e.g., $10x^2$) where other systems use an opaque simulation; this formula serves as an automatic way explore a program's parameter

space — to predict how a program will behave on different workloads. While it by no means solves all performance problems, TrendProf models an aspect of a program's scalability that is relevant to a large class of programs.

5.4 Performance Models from Static Analysis

Wegbreit [Weg75] describes a static analysis for computing closed form expressions that describe the minimum, maximum, and "average" performance cost of simple LISP programs in terms of the size of their input. Le Métayer [Mét88] focuses on statically analyzing maximum performance cost for FP (a functional programming language) programs. Rosendahl [Ros89] describes an abstract interpretation for transforming a LISP program into code that computes the worst case running time of the program. Such systems produce precise models of performance, but it is unclear how to adapt such approaches to large imperative programs.

Furthermore, a fundamental problem these techniques would encounter in analyzing even medium-sized programs is the sheer size of the parameter space: each loop, each conditional, and each input from the environment adds another dimension to the space and potentially another parameter to statically-derived models of performance. Analyzing performance of all possible workloads, as static analysis is forced to do, requires one to consider the possibility that all of these parameters vary independently and thus leads one to complex models in terms of obscure parameters. However, as we have shown in this dissertation, considering a realistic set of workloads, which tend to occupy a tiny subregion of the space of possible inputs, causes many of the dimensions of this parameter space to

collapse: a few workloads features do a reasonable job of explaining the performance of a large number of locations.

Gulavani and Gulwani [GG08] describe a precise numerical abstract domain that allows computation (via abstract interpretation) of upper bounds on the number of steps required to evaluate small C programs that involve mostly integer expressions. Their technique can deduce polynomial, logarithmic, exponential, and disjunctive (using a max operator) bounds.

Gulwani et al. [GMC09] describe a technique, SPEED, that makes enormous strides in computing worst case execution times of imperative programs via static analysis. Their technique involves instrumenting back-edges in a program with counters and using a linear invariant generator, like that described in [GG08] extended with a theory of uninterpreted functions, to bound these counters in terms of inputs to a procedure. They make an effort to ensure that they use few counters with few dependencies among them to ensure precise bounds. They deal with abstract data structures by having the user provide *quantitative functions*, like the length of a list or the height of a tree, and annotations on data structure operations that show how the operation updates the quantitative function.

The most striking difference between SPEED and TrendProf is that SPEED produces worst-case (over all inputs) bounds on procedure execution time while TrendProf produces average-case (over given workloads) bounds. Both techniques require some help from the user: SPEED requires extensive annotations to data structures while TrendProf requires the user to provide workloads, features, and (optionally) contexts. While TrendProf produces a performance model for any code that is covered by the user's work-

loads, SPEED's analysis can fail to produce any bound. On the other hand, when SPEED produces a bound, that bound comes with a proof; in contrast, TrendProf's models are statistical and may be utter nonsense in the worst case.

Combining SPEED and TrendProf offers some fascinating possibilities. SPEED and the quantitative functions it requires are a rich source of invocation features for TrendProf. Furthermore, SPEED's bounds provide hints as to the shape of average case models and constrain the possible models that TrendProf might consider (for example, if SPEED proves a worst-case bound that is quadratic in n, TrendProf need not consider models that are cubic in n). Also, TrendProf could measure the extent to which SPEED's worst case bounds are realized in practice.

5.4.1 Analyzing Data Structures

Danielsson [Dan08] outlines a library, Thunk, for analyzing the amortized complexity of purely functional data structures that use laziness. The mechanism is to annotate each function with types that describe how many steps the function takes to compute its result and use a dependent type system to verify these bounds. Thus, assuming the user uses the annotations correctly and adheres to a few syntactic restrictions, the type of an expression (for instance, a function call) includes how many steps it takes to compute the expression. In general, this type can include parameters like the size of a data structure.

Along similar lines, Krone et al. [KOS06] develop a system for specifying and verifying performance contracts of software components. These contracts specify the duration of a function call (perhaps based on parameters or other data) and how much memory it consumes. This performance verification requires functional specification and verification

of the code as well. They use their system to specify several data structures including a spanning forest component built out of smaller components. For larger components, they report an annotation to code ratio of about one to three; smaller, simpler components require relatively less code and may have a ratio closer to one to one.

These systems model performance of data structures and components based on parameters relating to their size and other runtime concerns. In contrast, TrendProf provides an automatic analysis that provides a more whole-program view of performance: TrendProf models the performance of data structures, but these models reflect how actual workloads to the program exercise these data structures.

A tool like TrendProf could benefit from the existence of such performance contracts: these contracts suggest variables and functions in terms of which to model performance. As we saw in Section 3.4.3, the model selection problem is difficult: it is not clear what functional relationship, if any, there is between performance and workload size. Thus, having the user specify the form of the performance model as a performance contract would be a benefit to TrendProf. To a system that verified performance contracts, TrendProf could add inference: rather than specify an entire performance contract with all its terms and coefficients, the user could give TrendProf a hint (e.g., performance is roughly $O\left(n^2\right)$); based on runs of the program, TrendProf could derive coefficients or perhaps even extra terms (e.g., $5n^2+10n$) to performance contracts. Furthermore, if the variables in the performance contracts are very local (for instance, some list's size), a TrendProf-like tool could attempt to model these local variables in terms of workload features.

Chapter 6

Conclusion

TrendProf's models of empirical computational complexity allow developers to compare the empirical reality of how their code is scaling on realistic workloads to their expectations.

We advocate the use of empirical computational complexity for understanding program performance and scalability. We have presented two techniques, BB-TrendProf and CF-TrendProf, that, given a program and workloads for it, build models of execution count in terms of user-specified workload features. Although these models are not always accurate, we may assess their plausibility using the scatter plots and residuals plots that TrendProf provides. Both BB-TrendProf's clustering and ranking and CF-TrendProf's call-graph oriented summary of program performance focuses attention on scalability-critical code. These models allow us to predict the performance of programs on novel workloads, including workloads bigger than any measured. The trends that TrendProf finds can point to potential scalability problems with a piece of code even if that

piece of code is not a huge percentage of any workload's performance.

By using clues from the program's control flow, CF-TrendProf is able to consider a rich family of models to precisely explain program performance. With help from the user, CF-TrendProf can distinguish different performance contexts based on control flow or data. CF-TrendProf provides per-workload and per-function-invocation views onto performance. These improvements over BB-TrendProf are sufficient to model, to within a logarithmic factor, the performance of several algorithms and data structures, some of which have very complex formal analyses.

Our technique is useful for understanding program performance: TrendProf's models allow us to compare the empirical computational complexity on typical workloads to our expectations. Such comparisons can either confirm the expected performance or reveal a difference from it: even on our few examples, we have discovered several surprises which the usual testing process could easily miss and furthermore demonstrated CF-TrendProf's ability to find still more performance surprises. By modelling the performance of the program on workloads that we have not actually measured we add a new dimension of generality to traditional profilers. Further, the complexity of algorithms on realistic workloads can easily differ from their theoretical worst-case behavior. Our `banshee` and `elsa` experiments illustrate both of these points: that is, no current profiler would have discovered that Andersen's analysis actually scales quadratically in practice or that `elsa`'s GLR C++ parser is only mildly super-linear, in contrast to their cubic theoretical worst-case bounds. Our analysis therefore gives engineers a more accurate working performance model.

While anyone could attempt a performance-trend analysis of their program most

engineers do not; a generic and convenient tool for automatically computing a comprehensive performance-trend analysis belongs in every programmer's toolbox.

Appendix A

Regression

A.1 Model Construction with Regression

In constructing models to predict performance and put locations into clusters, TrendProf makes use of least-squares linear regression and powerlaw regression. Regression selects model parameters (a and b below) that minimize some measure of error. Regression does not evaluate the applicability of these models, but TrendProf provides diagnostics that allow the user to asses their plausibility (Section A.1.5).

A.1.1 Linear Models

Given a set of points (x_i, y_i), least-squares linear regression constructs a model that predicts y as $\hat{y}(x) \stackrel{\text{def}}{=} a + bx$, an affine function of x. Given a data point, (x_i, y_i), define $\hat{y_i} \stackrel{\text{def}}{=} \hat{y}(x_i) = a + bx_i$. The quantity $r_i \stackrel{\text{def}}{=} y_i - \hat{y_i}$ is called the *residual* of the fit at (x_i, y_i).

Linear regression chooses a and b to minimize the sum of the squared residuals:

$$\sum_{i=1}^{k} r_i^2 = \sum_{i=1}^{k} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{k} (y_i - (a + bx_i))^2 .$$

A.1.2 Constant Models

We use the sample mean $\hat{y} = \frac{1}{k}\sum_{i=1}^{k} y_i$ to approximate a set of points (y_i) with a constant. This model, $\hat{y}$, minimizes the sum of the squared residuals: $\sum_{i=1}^{k} (y_i - \hat{y})^2$. A constant model is clearly just a degenerate case of a linear model ($b = 0$); sometimes this trading off of precision for simplicity is worthwhile.

A.1.3 Powerlaw Models

Our interest in measuring the scalability of a program led us toward powerlaw models. A powerlaw predicts y as $\hat{y}(x) = ax^b$. On log-log axes, the plot of a powerlaw is a straight line. Thus to fit observations to a powerlaw, TrendProf uses linear regression on $(\log x_i, \log y_i)$ to find a and b that minimize the following quantity:

$$\sum_{i=1}^{k} (\log y_i - (\log a + b \log x_i))^2 = \sum_{i=1}^{k} \left(\log \frac{y_i}{ax_i^b} \right)^2 = \sum_{i=1}^{k} \left(\log \frac{ax_i^b}{y_i} \right)^2 .$$

Notice that here our residuals are expressed in logarithmic space as $\log \frac{y_i}{ax_i^b}$.

A.1.4 Numerical Stability

It is important to compute regression coefficients in a numerically stable fashion. We adapt an algorithm from Higham's book [Hig02] to compute regression coefficients efficiently from our run-length encoded data points.

A.1.5 How good is a model?

There are a number of ways for the user of TrendProf to evaluate the usefulness of a particular model. For each model, TrendProf presents two scatter plots: one with the data points $(x_i,\ y_i)$ and the line of best fit $(x,\ \hat{y}(x))$ and another with the residuals $(x_i,\ r_i)$. Inspecting these plots is a good way to decide if TrendProf's model is plausible. To the extent that a model captures the variation in a data set, the data points in the best-fit scatter plot closely track the line of best fit and the residuals scatter plot looks like random noise. Therefore, any pattern in the residuals plot or systematic deviation from the line of best fit is an indication that there is more going on than the model describes (Section 4.3).

Plots are not very compact, however, so for each of its fits BB-TrendProf reports the R^2 statistic, a measure of the model's goodness-of-fit that quantifies the fraction of the variance in y accounted for by a least-squares linear regression on x:

$$R^2 \stackrel{\text{def}}{=} \frac{\sum_{i=1}^{k} (\hat{y}_i - \overline{y})^2}{\sum_{i=1}^{k} (y_i - \overline{y})^2} = \frac{\left(\sum_{i=1}^{k} (x_i - \overline{x})(y_i - \overline{y})\right)^2}{\left(\sum_{i=1}^{k} (x_i - \overline{x})^2\right)\left(\sum_{i=1}^{k} (y_i - \overline{y})^2\right)}.$$

The equality on the right assumes least squares linear regression. The formula for R^2 applies to powerlaw fits, but with x replaced by $\log x$ and y replaced by $\log y$. Values for R^2 range from 0 (bad) to 1 (excellent). Note that $\overline{y}$ denotes the sample mean of a k-vector y and σ_y^2 denotes its bias-corrected sample variance:

$$\overline{y} \stackrel{\text{def}}{=} \frac{1}{k} \sum_{i=1}^{k} y_i \qquad \sigma_y^2 \stackrel{\text{def}}{=} \frac{1}{k-1} \sum_{i=1}^{k} (y_i - \overline{y})^2 .$$

Instead of R^2, CF-TrendProf reports the standard error for each of its models, a measure akin to the standard deviation of a data set that instead measures the deviation

of the data from the model as follows.

$$S \stackrel{\text{def}}{=} \sqrt{\frac{\sum_{i=1}^{k} (\hat{y}_i - y_i)^2}{k-2}}$$

A standard error of 0 indicates a perfect fit, while higher standard errors indicate worse fits. In order to make a meaningful comparison of powerlaw fits to linear fits, CF-TrendProf uses the residuals $(\hat{y}_i - y_i)$ suggested by the formula above to evaluate the standard error of powerlaw fits.

Appendix B

Proof of Cluster Theorem

THEOREM: *If x, y, and p are vectors of length k such that x and y both fit p with $R^2 > 1-\alpha$ and $0 < \alpha < 0.5$, then x fits y with $R^2 > 1 - 4\alpha(1-\alpha)$.*

PROOF: Without loss of generality, assume that x, y, and p are normalized to have mean 0 and variance 1 and that $x \cdot p \geq 0$ and $y \cdot p \geq 0$; they can be made so with an affine transformation and such transformations preserve R^2. We denote the R^2 statistic for the fit of x to p by $R^2_{x,p}$ and the angle (in $\mathbb{R}^k$) between x and p by $\phi_{x,p}$. We have

$$1 - \alpha < R^2_{x,p} = (x \cdot p)^2 = \cos^2 \phi_{x,p}$$

Rearranging terms yields $\phi_{x,p} < \arcsin \sqrt{\alpha}$ and similarly for $\phi_{y,p}$. By the triangle inequality on the surface of the k-sphere and substitution,

$$\phi_{x,y} \leq \phi_{x,p} + \phi_{y,p} < 2 \arcsin \sqrt{\alpha}$$

and so

$$R^2_{x,y} = \cos^2 \phi_{x,y} > 1 - 4\alpha(1-\alpha) \quad \blacksquare$$

Bibliography

[ACGS04] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP 2004*. Springer Berlin / Heidelberg, 2004.

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model — One step closer towards a realistic model for parallel computation. In *SPAA 1995: Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, New York, NY, USA, 1995. ACM Press.

[AKLW02] Alberto Avritzer, Joe Kondek, Danielle Liu, and Elaine J. Weyuker. Software performance testing based on workload characterization. In *WOSP 2002: Proceedings of the 3rd international Workshop On Software and Performance*, pages 17–24, New York, NY, USA, 2002. ACM.

[AKM06] Tobias Achterberg, Thorsten Koch, and Alexander Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006. See `http://miplib.zib.de`.

[And94] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.d. thesis, DIKU, Unversity of Copenhagen, 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AW04] Alberto Avritzer and Elaine J. Weyuker. The role of modeling in the performance testing of e-commerce applications. *IEEE Trans. Softw. Eng.*, 30(12):1072–1083, 2004.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[Bre94] Eric Allen Brewer. *Portable High Performance Supercomputing: High-Level Platform Dependent Optimization*. Ph.d. thesis, Massachusetts Institute of Technology, 1994.

[Bre95] Eric A. Brewer. High-level optimization via automated statistical modeling. In *PPOPP 1995: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, New York, NY, USA, 1995. ACM Press.

[BZ2] `bzip2` project homepage. `http://www.bzip.org/`.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Mc-Graw Hill, 1990.

[Dan08] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for

purely functional data structures. In *POPL 2008: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 133–144, New York, NY, USA, 2008. ACM.

[Deb] Debian project homepage. `http://www.debian.org/`.

[GAW07] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *ESEC-FSE 2007: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, pages 395–404, New York, NY, USA, 2007. ACM.

[GCO] `gcov` documentation. `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[GG08] Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV 2008: Proceedings of the 20th international conference on Computer Aided Verification*, pages 370–384, Berlin, Heidelberg, 2008. Springer-Verlag.

[GKD+08] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple performance metrics for queries: Better decisions enabled by machine learning. `http://radlab.cs.berkeley.edu/people/fox/wp/wp-content/uploads/perf_prediction_vldb_submitted.pdf` accessed on June 23, 2008, 2008.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN 1982: Proceedings of the 1982 SIGPLAN*

Symposium on Compiler Construction, pages 120–126, New York, NY, USA, 1982. ACM Press.

[GMC09] Sumit Gulwani, Krishna Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL 2009: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, New York, NY, USA, 2009. ACM.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[Gra] `graphviz` project homepage. `http://www.graphviz.org/`.

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[JB03] Edwin T. Jaynes and G. Larry Bretthorst. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

[KA05] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS 2005: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[KKN05] Michael Kluge, Andreas Knüpfer, and Wolfgang E. Nagel. Knowledge based automatic scalability analysis and extrapolation for MPI programs. In *Euro-*

Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[KOS06] Joan Krone, William F. Ogden, and Murali Sitaraman. Performance analysis based upon complete profiles. In *SAVCBS 2006: Proceedings of the 2006 conference on Specification and Verification of Component-Based Systems*, pages 3–10, New York, NY, USA, 2006. ACM.

[Lam] Michael Lamont. Source code for quicksort. `http://linux.wku.edu/~lamonml/algor/sort/quick.html` accessed around November 15, 2007.

[LB06] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 185–194, New York, NY, USA, 2006. ACM.

[LPS] `lp_solve` project homepage. `http://tech.groups.yahoo.com/group/lp_solve/`.

[Més] Csaba Mészáros. Csaba Mészáros's collection of linear programs. `http://www.sztaki.hu/~meszaros/bpmpd/` accessed around October 25, 2006.

[Mét88] Daniel Le Métayer. ACE: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[Mit] Hans Mittelmann. Hans Mittelmann's collection of linear programs. `http://plato.asu.edu/ftp/lptestset/` accessed around October 25, 2006.

[MN04] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Conference on Compiler Construction (CC04)*, 2004.

[Ric06] John A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 2006.

[Ros89] M. Rosendahl. Automatic complexity analysis. *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156, 1989.

[RS98] Radu Rugina and Klause Schauser. Predicting the running times of parallel programs by simulation. In *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, 1998.

[Sar89] V. Sarkar. Determining average program execution times and their variance. In *PLDI 1989: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 298–312, New York, NY, USA, 1989. ACM Press.

[Sau] Shane Saunders. Source code for Dijkstra's algorithm and a Fibonacci heap. `http://www.cosc.canterbury.ac.nz/tad.takaoka/alg/spalgs/spalgs.html` accessed around November 15, 2007.

[SdPK01] Gary Sevitsky, Wim de Pauw, and Ravi Konuru. An information exploration tool for performance analysis of Java programs. In *TOOLS 2001: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 85, Washington, DC, USA, 2001. IEEE Computer Society.

[SY07] Jimmy Su and Katherine Yelick. Automatic communication performance debugging in PGAS languages. In *20th International Workshop on Languages and Compilers for Parallel Computing*, 2007.

[Thi07] Lothar Thiele. Performance analysis of distributed embedded systems. In *EMSOFT 2007: Proceedings of the 7th ACM & IEEE international conference on Embedded Software*, pages 10–10, New York, NY, USA, 2007. ACM.

[Ukk90] Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. In *Algorithmica*, volume 5, pages 313–323, 1990.

[VTSJ07] Kapil Vaswani, Matthew J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *CGO 2007: Proceedings of the International Symposium on Code Generation and Optimization*, pages 131–143, Washington, DC, USA, 2007. IEEE Computer Society.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18(9):528–539, 1975.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.