Measuring Empirical Computational Complexity

Simon F. Goldsmith*
Computer Science Dept.
UC Berkeley
California, USA
sfg@eecs.berkeley.edu

Alex S. Aiken
Computer Science Dept.
Stanford University
California, USA
aiken@cs.stanford.edu

Daniel S. Wilkerson
California, USA
daniel.wilkerson@gmail.com

ABSTRACT

The standard language for describing the asymptotic behavior of algorithms is theoretical computational complexity. 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. Grouping and ranking program locations based on these models focuses attention on scalability-critical code. We describe our tool, the *Trend Profiler* (`trend-prof`), for constructing models of empirical computational complexity that predict how many times each basic block in a program runs as a linear ($y = a + bx$) or a powerlaw ($y = ax^b$) function of user-specified features of the program's workloads. We ran `trend-prof` on several large programs and report cases where a program scaled as expected, beat its worst-case theoretical complexity bound, or had a performance bug.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing Tools, Debugging Aids; D.2.8 [**Metrics**]: Performance Measures

General Terms

Performance, Measurement

Keywords

trend-prof, empirical computational complexity

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-0234689. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

ESEC/FSE'07, September 3–7, 2007, Cavat near Dubrovnik, Croatia.

1. INTRODUCTION

Computer scientists talk about the scalability of algorithms in terms of computational complexity: Quicksort is $O(n \log n)$ in the size of the array; depth-first search is $O(e)$ in the number of edges in the graph. Big-O bounds describe the worst case performance of an algorithm as its inputs become large, but large programs contain many interconnected data structures and algorithms whose empirical performance depends on the particulars of the program's workloads. In contrast, by measuring a run of a program, profilers like `gprof` [8] estimate the fraction of the program's run time for which each *location* (e.g., each basic block) in the program is responsible. Profilers give a detailed picture of a program's performance on a single workload, but say nothing about its performance on workloads on which the program was not run. In this work, we seek to combine the empiricism of `gprof` with the generality of a big-O bound by measuring and statistically modelling the performance of each location across many workloads. The following code illustrates the need for our technique.

```
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. 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 the list length increases with the size of the program input, then use of this idiom constitutes a performance bug.

The crucial piece of information is how this list is used in the context of the rest of the program. The code above is from a C parser used in a program analysis system [10] and is called from a list append function. In practice the sizes of the lists increase as inputs grow larger. On small- to medium-size inputs, `last_node` is not particularly high on the list of what a typical, `gprof`-style profiler reports, but on large inputs the problem 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 [11] that turned out to be benign: the lists are so small in practice that use of this idiom does not substantially contribute to the overall performance of the system. Our technique automatically distinguishes these two different situations.

Our technique for measuring empirical computational complexity is as follows:

- *Choose a program* to profile.
- *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 cost of each each basic block, ℓ, as a k-vector: $(y_{\ell,1}, \ldots, y_{\ell,k})$.
- `Trend-prof` *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, $y = af^b$.

These *models of empirical computational complexity* describe the scalability of a piece of code in practice. By comparing the empirically-determined model to our expectations, we can determine whether code performs as expected or if it has a performance bug. Our contributions are as follows:

- We describe our models of empirical computational complexity and explore some implementation design choices (Section 2).
- We present `trend-prof`, a tool that describes program performance using linear and powerlaw models: predicting basic block executions in terms of user-specified features of program workloads (Section 4).
- We show that `trend-prof` reports simple results for programs with simple performance behavior (Section 5.2), confirm that desired performance behavior is realized in practice (Section 5.3), measure the empirical performance of a complex algorithm (Section 5.4), and find a scalability bug (Section 5.5).
- We argue that `trend-prof` reports the empirical computational complexity of a program succinctly (Section 5.1) and that it helps focus attention on performance and scalability critical code (Section 5.6).
- We consider threats to the validity of `trend-prof`'s results and discuss features of `trend-prof` that mitigate these threats (Section 6).

2. 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.

After running and measuring k workloads, we have a k-vector of costs for each location (one measurement per workload) and 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` [8] 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	$\ldots$	w_k
locations	ℓ_1	$y_{1,1}$	$y_{1,2}$	$\ldots$	$y_{1,k}$
	ℓ_2	$y_{2,1}$	$y_{2,2}$	$\ldots$	$y_{2,k}$
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	ℓ_n	$y_{n,1}$	$y_{n,2}$	$\ldots$	$y_{n,k}$
features	f	f_1	f_2	$\ldots$	f_k
	g	g_1	g_2	$\ldots$	g_k

In general, this matrix contains a great deal of data. `Trend-prof` summarizes it with succinct, interpretable models as follows.

Clusters. The costs of many locations vary together. If a matrix row x is sufficiently linearly-correlated with the matrix row y, that is, if there are a, b such that $a+bx$ is a good prediction of y, then we place x and y into the same *cluster* (Section 4.1). We do not necessarily know what determines the performance of the locations in a cluster or what asymptotic complexity they have, but whatever it is, it is the same for all the locations in the cluster — if one is $\Theta(n^2 \log n)$, the others are too. Section 4.1 describes our clustering algorithm in detail. We show in Section 5.1 that empirically programs have many fewer clusters than locations.

Performance Models. The ultimate goal of `trend-prof` is to explain the performance of the program as a function of features of the input. In order to keep output succinct, `trend-prof` models performance one cluster at a time. For each cluster, `trend-prof` powerlaw-fits the cost of the cluster y with each feature x. If the model fits well, that is if we find a, b such that ax^b is a good predictor of y, then we have succeeded: we have a model of the cluster's cost as the input exhibits more of this feature (e.g., cost increases quadratically in the number of nodes in the input). Even if the model is not a perfect fit, it is often a good low-dimensional approximation of a more complex relationship (Section 6.2). We may further use these models to predict the cluster's cost — even for inputs larger than any we measured (Section 4.2).

2.1 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 [4] is cheap (Section 5).
- PORTABILITY: We rely only on `gcc`'s coverage mechanism [7] and not on platform-specific performance registers.

2.2 Other notions of location

Our notion of basic blocks as locations is useful, but is not the only sort of location we might measure. We could extend the notion of location by aggregating existing location counts into larger locations. For instance, `gprof` estimates the amount of work of a function in its own code and the transitive work that it and its callees do. Also, the work of Ammons et al. [2] (see discussion in Section 7.1) measures the work of a sequence of nested function calls. With some information about the control flow and a call graph, `trend-prof` could compute and operate on counts for these "locations". Alternately, one might regard each memory reference as a location and measure cost in terms of real time; this approach more closely models bottom line performance, but requires more effort to obtain useful measurements.

While the program runs, `trend-prof` keeps a histogram where each bin records the execution count of a single basic block. Other dimensions of bins are possible as follows. Bins could count each *(call stack, basic block)* combination; in this scenario we might limit the significant depth of the call stack to keep the number of bins small (e.g., `gprof` counts caller-callee pairs, a significant depth of two). Also, bins could count each *(object, basic block)* combination; for example, we could separate the counts for a hash table class by instance of the hash table.

We have not tried these approaches. They are sensible extensions to this work, but they involve even larger amounts of data.

2.3 Model Construction

In constructing models to predict performance and put locations into clusters, `trend-prof` 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 suitability of these models, but `trend-prof` provides diagnostics that allow the user to asses their plausibility (Section 2.3.1).

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:

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

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, `trend-prof` uses linear regression on $(\log x_i, \log y_i)$ to find a and b that minimize the following quantity:

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

2.3.1 How good is a model?

There are a number of ways for the user of `trend-prof` to evaluate the usefulness of a particular model. The primary output of `trend-prof` is a web page showing a list of clusters with their models. Following a link for a model leads to 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 `trend-prof`'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 6.2).

Plots are not very compact, however, so for each of its fits `trend-prof` 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 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:

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

3. AN EXAMPLE

Before exploring our methodology in detail, we illustrate the use of `trend-prof` with the following implementation of bubble sort. This example is purely for pedagogical purposes; `trend-prof` gives useful results in significantly more complex situations (see Section 5).

```
    // 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 bubble_sort(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 `bubble_sort` consists of an array of `n` integers. The size, `n`, is a *feature* of the workload. We ran `bubble_sort` 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 `trend-prof` on this example.

4. Trend Profiler

We describe how `trend-prof` builds and ranks clusters and how it models the performance of these clusters.

4.1 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 `bubble_sort` 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 $R^2 > 1 - \alpha$, for some small constant $0 < \alpha < 0.5$. Every location belongs to at least one, and possibly multiple, clusters.

`Trend-prof` 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 tradeoff between how many clusters `trend-prof` 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 Section 4.1.1, 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, a location whose cost is always between 100 and 120 might be so discarded.

4.1.1 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: Without loss of generality, assume that x, y, and p are normalized to have mean 0 and variance 1, since 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

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

Cluster	Max	Fit with n	R^2
COMPARES	$1.1\ e10$	$3.0\ n^{2.00}$	1.00
SWAPS	$2.6\ e9$	$3.1\ n^{1.93}$	0.99
SIZE	$1.3\ e6$	$22\ n^{1.00}$	1.00

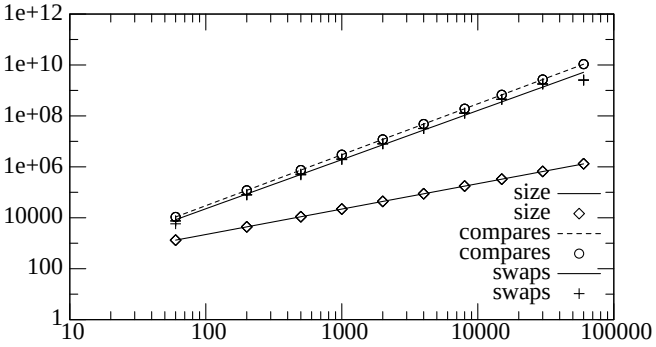

Figure 1: The table (top) shows powerlaw models predicting cluster costs for the Bubble Sort example. The graph (bottom) shows three powerlaw best-fit plots showing observed cluster costs for COMPARES, SWAPS, and SIZE (y axis) versus n (x axis) with their lines of best fit.

4.1.2 Example

In the `bubble_sort` example, `trend-prof` 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 `trend-prof` 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 the values of n in our workloads, the quadratic term is the only important one for describing scalability.

4.2 Powerlaw Fits Measure Scalability

We define the *cost* of a cluster as the sum of the costs of all the locations in the cluster. `Trend-prof` 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, `trend-prof` 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 `trend-prof` 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. Currently, 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* [13]. A detailed discussion of the bootstrap is beyond the scope of this paper. 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) then computing the function in question on each example data set and collecting those results into a "function value" set then 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 `trend-prof` 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, `trend-prof` 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 `trend-prof` 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 `trend-prof`'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, `trend-prof` 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. `Trend-prof` may be configured to suppress the display of models constructed with few data points as such models are unlikely to make accurate predictions.

4.2.1 Example

In the `bubble_sort` example we have only one feature, n, but it powerlaw-fits all cluster totals well. Figure 1 shows the scatter plot and lines of best fit for these powerlaws.

5. RESULTS

We ran `trend-prof` on the programs listed in Figure 2 with workloads as described in Figure 3. Figure 3 also mentions the average (geometric mean) overhead of running a workload with edge profiling enabled versus having it disabled (Overhead) and the total time in hours that our straightforward Perl implementation of `trend-prof` 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 `trend-prof`'s post-processing including clustering, model-fitting, and generation of plots and results pages. Once `trend-prof` generates its results, they are browseable interactively.

Program	Description	Workloads
`bzip2 1.0.3` [6]	Compresses files	Tarballs of preprocessed source code
`banshee 2005.10.07` [10]	Computes Andersen's alias analysis [3] on a C program	Preprocessed C programs
`elsa` [11]	Parses, type-checks, and elaborates C and C++ programs	Preprocessed C++ programs
`maximus`	Ukkonen's suffix tree algorithm [18] for finding common substrings	C source code

Figure 2: We ran `trend-prof` on these programs with workloads as described above.

Program	Workloads	Min – Max	Overhead	Time (h)
`bzip`	1000	3 e7 – 2 e11	22%	19 + 0.1
`banshee`	277	4 e6 – 1 e10	18%	0.7 + 1.1
`maximus`	910	3 e4 – 8 e09	10%	3.7 + 0.1
`elsa`	785	9 e5 – 4 e09	103%	3.3 + 7.4

Figure 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 `trend-prof`'s time in hours to run workloads post-process data (Time).

5.1 Programs Have Few Clusters

For each of our benchmark programs, Figure 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 `trend-prof` 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.

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 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 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.

5.2 Simple Programs Have Simple Profiles

Running `trend-prof` on `bzip` reveals that it scales linearly in the size of its input and that most of the locations vary together. Figure 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 `trend-prof` a program with simple performance has a simple profile.

5.3 Confirming Expected Performance of the Implementation of a Complex Algorithm

Measuring the empirical computational complexity of a program using `trend-prof` can verify that it scales as expected. Ukkonen's algorithm [18] 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 `trend-prof` 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 `trend-prof` incorporates these features into its calculations. As expected, NODES and EDGES both linearly fit CHARS and thus wind up in its cluster. Figure 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 `trend-prof`'s output. The top ranked cluster scales as $11\text{CHARS}^{1.29} (R^2 = 0.99)$ and includes the output routines; Figure 7 shows the relevant fit.

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 `trend-prof` was still required to find it.

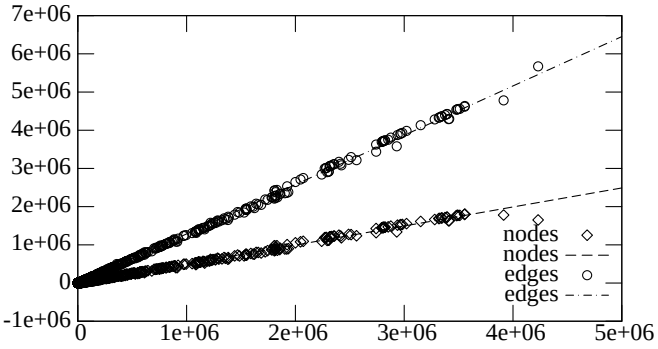

Figure 6: These two linear best-fit plots for `maximus` show that the number of suffix tree nodes and edges (y axis) grows linearly with the number of characters (x axis) in the workload.

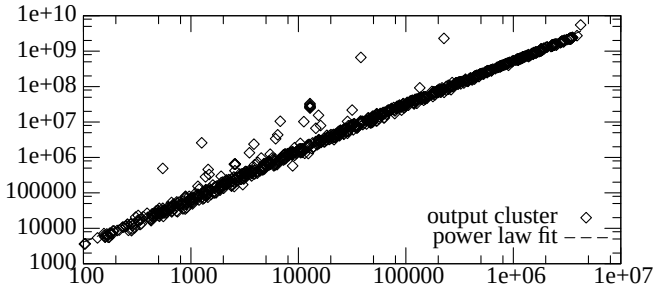

Figure 7: The crisp powerlaw fit in this best-fit plot for `maximus`'s output routines shows that their cost grows super-linearly in the number of characters in the input ($\hat{y} = 11\text{CHARS}^{1.29}$).

5.4 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 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.

5.5 This List Traversal is a Bug

As mentioned in Section 1, we found a scalability bug in the C parser used by `banshee`. `Trend-prof` predicts that the `last_node` function (see Section 1) 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 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.

5.6 Focusing on Scalability-Critical Code

We look now at the results of running `trend-prof` on another large, well optimized program with complex inputs. The `elsa` benchmark is a parser, type-checker, and elaborator for C and C++ code. Running `trend-prof` on `elsa` with C++ programs as input divides the roughly 33,000 basic blocks of `elsa` into fewer than

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 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.

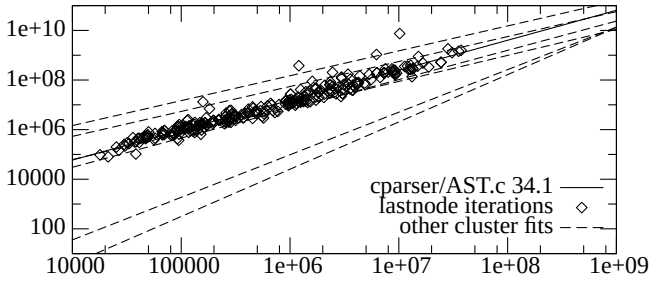

Figure 9: Powerlaw best-fit plot for the loop body of the performance bug in `banshee` ($\hat{y} = 0.87$ BYTES$^{1.21}$). We show lines of best fit for other cluster costs for reference.

1500 clusters. Figure 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.

5.6.1 An Empirical Measure of GLR Performance

The top several clusters contain code that is critical to the performance and scalability of `elsa` for large workloads. Figure 11 shows the powerlaw fit and residuals plot for one such cluster whose representative is 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 [11], 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.

5.6.2 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 12 shows the scatter plot and powerlaw fit for this cluster's cost; it also shows the powerlaw fit for another top cluster

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

Figure 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.

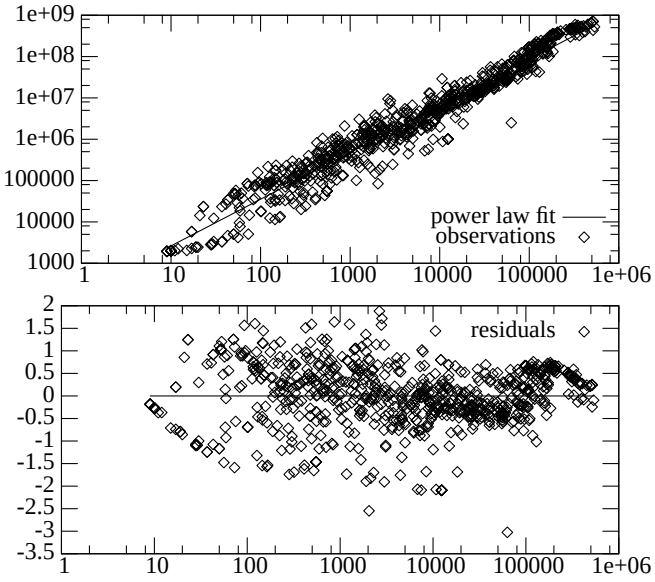

Figure 11: A nice powerlaw best-fit plot showing the slight super-linearity of `elsa`'s GLR parsing ($\hat{y} = 195$ AST$^{1.13}$, $R^2 = 0.95$) (top) and the corresponding residuals plot (bottom).

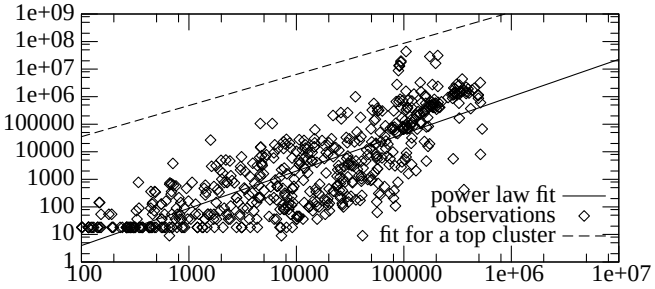

Figure 12: There is no clear relationship between feature AST and these points. Thus, the line of best fit (solid) is dubious. Nonetheless, comparing to the powerlaw fit from Figure 11 (dotted line) suggests that this cluster is not a scalability problem.

whose representative is `elkhound/glr.cc` line 362 (also shown in Figure 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.

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

This pattern is exactly the sort of code that was a performance bug in the `banshee` benchmark, but here `trend-prof` provided us with enough information to conclude that it is not a serious scalability problem.

6. THREATS TO VALIDITY

There are situations where `trend-prof`'s models do not adequately describe the program's performance on novel workloads. Choosing atypical or insufficiently many workloads to train `trend-prof` causes its models to over-fit patterns in the data (Section 6.1). There are cases where the powerlaw fit `trend-prof` uses to model cluster cost does not adequately capture important performance variations (Section 6.2).

6.1 The Importance of Workloads

The empirical aspect of `trend-prof`'s models of empirical computational complexity is both an advantage and a disadvantage.

6.1.1 When Workloads Reveal Empirical Truth

Using `trend-prof` we do not distinguish correlations among the costs of various locations 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 5.4) 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 5.5), but the same idiom is *not* a bug in the context of `elsa`'s data structures for resolving name lookup (Section 5.6).

6.1.2 When Workloads Oversimplify

On the other hand, the user of `trend-prof` 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. Recall that the workloads we considered earlier (Section 3) were arrays of integers generated uniformly at random and that the locations break into 3 distinct clusters: COMPARES, SWAPS, and SIZE (Figure 1). Depending on the distribution of inputs, `trend-prof`'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 convex 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.

6.2 Limitations of the Powerlaw Fit

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 `trend-prof` 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.

6.2.1 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 13 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 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 `trend-prof` 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).

6.2.2 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, `trend-prof` 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, `trend-prof`'s powerlaw fit is a reasonable, low-dimensional approximation.

6.2.3 The Missing Feature

There are clusters whose cost is not well approximated by a powerlaw of any feature. 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 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, which `trend-prof` provides,

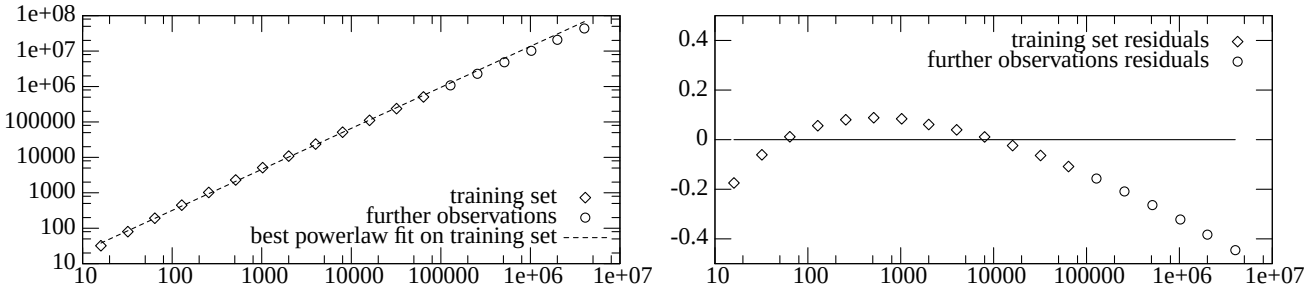

Figure 13: On the left is a log-log plot of number of comparisons done in a call to `qsort` (y axis) versus the size of the array (x axis). On the same plot, we show the best powerlaw fit to the diamond shaped points ($\hat{y} = 1.5n^{1.16}$, $R^2 > 0.99$). On the right is the residuals plot for the powerlaw fit. Note that the residuals are clearly not randomly distributed.

that `trend-prof`'s powerlaw model is inadequate for the situation and that its predictions are not to be trusted.

For situations like these, `trend-prof` 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. Also, `trend-prof` 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.

7. 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.

7.1 Profilers

Gprof [8] 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.

Jinsight EX [17] exhaustively traces the execution of a program, recording the number of objects of a particular type that are allocated, the number a times a method is called, etc. The user may browse this data to explore the performance of the program.

Ammons et al. [2] describe a profiler for finding expensive paths through a program and for computing how the cost of a path differs between two similar runs of a program. The essence of their technique computes the cost of a sequence of nested function calls from a call-tree profile. In our terms, they compute performance data for a more general notion of location. In this regard, their work is complementary to ours; their system computes the cost of a path for one workload, while `trend-prof` builds models to describe how the cost of a location increases with workload features.

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

7.2 Empirical Performance Models

Kluge et al. [9] 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 `trend-prof` is more general; we aim to characterize the scalability of a program in terms of a user-specified notion of input size.

Brewer [5] 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 `trend-prof` 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 parameters on which a basic block's performance depends may not be readily apparent; therefore, `trend-prof` 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. Our goal of making interpretable models that predict scalability drive us towards simpler, lower-dimensional models that are more appropriate for our goals.

Sarkar [16] predicts the mean and variance of loop execution times using counter-based profiles. In contrast, `trend-prof` models performance as a function of workload features.

7.3 Performance Models by Simulation

Rugina and Schauser [15] 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 [1] parameters for the target machine; their simulation outputs a time. Their focus is on tuning a program with 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 `trend-prof`.

7.4 Static Performance Models

Wegbreit [19] 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 [12] focuses on statically analyzing maximum performance cost for FP (a functional programming language) programs. Rosendahl [14] describes an abstract interpretation transforms 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.

8. CONCLUSION

We advocate the use of empirical computational complexity for understanding program performance and scalability. We have presented our tool, `trend-prof`, that, given a program and workloads for it, 1) builds models of basic block execution frequency in terms of user-specified workload features and 2) provides means to assess the plausibility and applicability of these models. By grouping related locations into clusters and modelling the performance of these clusters, `trend-prof` 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.

Ranking these clusters based on their cost and predicting their scalability by powerlaw-fitting their cost to user-specified features focuses attention on scalability-critical code. Although these models are not always accurate, we may assess their plausibility using the scatter plots and residuals plots that `trend-prof` provides. These models allow us to predict the performance of programs on novel workloads, including workloads bigger than any measured.

Our technique is useful for understanding program performance: `trend-prof`'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. 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, in contrast to its cubic theoretical worst-case bound. 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.

9. ACKNOWLEDGMENTS

The authors would like to thank Johnathon Jamison and Armando Solar-Lezama for trying `trend-prof`; Karl Chen for trying and helping debug `trend-prof`, 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 the paper; Jimmy Su, Jonathan Traupman, and Joseph Dale for useful discussions.

10. REFERENCES

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

[2] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP 2004*. Springer Berlin / Heidelberg.

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

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

[5] E. A. Brewer. High-level optimization via automated statistical modeling. In *PPOPP '95: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, New York, NY, USA, 1995. ACM Press.

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

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

[8] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.

[9] M. Kluge, A. Knüpfer, and W. 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.

[10] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

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

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

[13] J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 2006.

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

[15] R. Rugina and K. 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.

[16] V. Sarkar. Determining average program execution times and their variance. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 298–312, New York, NY, USA, 1989. ACM Press.

[17] G. Sevitsky, W. de Pauw, and R. Konuru. An information exploration tool for performance analysis of Java programs. In *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 85, Washington, DC, USA, 2001. IEEE Computer Society.

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

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