Empirical Computational Complexity

ABSTRACT

Computational complexity is the standard language for talking about the asymptotic behavior of algorithms. We propose a similar notion, which we call *empirical computational complexity*, for describing the scalability of a program in practice. Constructing a model of empirical computational complexity involves running a program on workloads whose sizes span several orders of magnitude, measuring their performance, and constructing a model that predicts the program's performance as a function of input size.

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 function of input size. We show for several real-world programs (such as `gcc` and `gzip`) that the number of times that scalability-critical basic blocks execute is often well modeled by either a line ($y = a + bx$) or a power law ($y = ax^b$). We ran `trend-prof` on several real-world programs and found interesting facts about their scaling behavior including two developer-confirmed performance bugs. By using `trend-prof`, a programmer can find performance bugs before they happen.

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. A similar idea is useful for describing program behavior in practice. In particular, we propose measuring multiple runs of a program and constructing empirical models that predict the performance of the program as a function of the size of its inputs.

Consider the following code.

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

From a performance perspective, this code looks fishy; it is computing the last element in a list in time linear in the list's length.

The Trend Profile technique:

- *Many inputs spanning magnitudes*: Profile across inputs whose sizes span several orders of magnitude.
- *Separate source-line observations*: count the number of executions of each basic block separately.
- *Two dimensional model*: fit these counts versus some reasonable measure of input size.
- *Linear and power law fits*: fit to a line in linear-linear and log-log space, which tend to find linear and power law models respectively.

Adding a pointer directly to the last element in the list would admit an obvious constant time implementation of this function. Of course, if the list's size is a small constant, then the performance impact of this code is likely to be negligible, and adding the pointer might not be worth the cost in code complexity and memory footprint. On the other hand, if these lists tend to be long, and especially if their sizes increase with the size of the input to the program as a whole, then this code constitutes a performance bug.

The crucial piece of information is how this code is used in the context of the rest of the program. This code is from a C parser used in a program analysis system [11]. In practice the sizes of the lists increase as the square root of input size. On small- to medium-size inputs, this line is not particularly high on the list of what a typical profiler reports, but on large inputs the problem suddenly becomes apparent.

We describe a technique for constructing models of the *empirical computational complexity* of a program (Section 2). These models predict the asymptotic behavior of programs based on observation of multiple runs of the program. We describe our tool, Trend Profiler (`trend-prof`), that measures the number of times each basic block in a program is executed and automatically builds linear ($\hat{y} = a + bx$) and power law ($\hat{y} = ax^b$) models that predict how many times a basic block will be executed as a function of user-specified input size (Section 3).

With `trend-prof` we found the scalability problem described above and several other interesting characteristics of several large C and C++ programs, including `gcc` and `gzip` (Section 4). Running `trend-prof` reveals trends not only in the performance of programs we measured but also in the characteristics of the workloads on which we ran the programs. By extrapolating these trends, `trend-prof` can predict how many times each piece of the program will execute on inputs larger than any of those that it measured. In particular it can identify pieces of code that scale worse

Facts about the `trend-prof` technique:

- It is easy to apply to programs without detailed knowledge of their internals.
- Our models do not fit every line of code well, but those lines that execute frequently often fit well.
- Simply ranking the models in decreasing order by 1) exponent or 2) their prediction for a fixed large input size almost always points to interesting lines of code or shows that the whole program is linear.
- It reveals facts about 1) the program *and* 2) the data, as seen through the program.
- It can reveal a more accurate model of the actual behavior than a worst-case theoretical analysis.

than the program as a whole; such code constitutes a performance bug waiting to happen. We find that on the programs we measured, linear and powerlaw models generally fit the most-executed basic blocks of the programs well (Section 5). We propose that `trend-prof` or a similar technique be used to make quantitative statements about the performance and scalability of complex programs.

2. EMPIRICAL COMPUTATIONAL COMPLEXITY

A *model of empirical computational complexity* for a piece of code relates a workload's size to the code's performance on that workload. We use the term "size" rather loosely to refer to any measure of how hard the workload is. Thus, we expect "bigger" workloads to consume more units of performance. These models are quantitative; they predict a performance-relevant quantity as a function of a workload's size. For example, a model of a piece of code to compute the transitive closure of a graph might predict the number of machine instructions executed as a function of the number of nodes in the input graph. We propose empirical computational complexity as a measure of a piece of code's scalability.

In general, constructing a model of empirical computational complexity consists of the following steps.

- *Choose workloads* $\{w_1,\ldots,w_k\}$ on which to run the program.
- *Choose a measure of workload size* and assign a size to each workload $\{x_1,\ldots,x_k\}$. For a measure of workload size we might choose size in bytes of an input file or the number of nodes in a graph. We discuss measures of workload size further in Section 3.4.
- *Choose a measure of program performance*; run the program on each workload and record performance $\{y_1,\ldots,y_k\}$ for each line of code in the program. For a measure of program performance we might measure time in milliseconds or number of bytes read from disk. We discuss measures of performance further in Section 3.2.
- *Fit each of these sets of observations to a model,* $\hat{y} = f(x)$; these models predict the performance of each line of code as a function of workload size. For instance, one might conclude that a selection sort does $\frac{1}{2}(x^2-x)$ comparisons for an input size of x. We discuss construction of these models in Section 3.3.

3. Trend Profiler

In order to construct models of empirical computational complexity, we built Trend Profiler (`trend-prof`). Given a program and a set of workloads for it, `trend-prof` constructs a performance model for each basic block in the program. These models predict how many times the basic block will be executed as a function of the (user-supplied) input size.

3.1 An Example

Before exploring the choices we make in designing `trend-prof`, we illustrate the use of `trend-prof` with a toy example: selection sort. This example is purely for pedagogical purposes; `trend-prof` gives useful results in significantly more complex situations (see Section 4).

Our discussion uses the following implementation of selection sort.

```
// pre:  The memory at arr[0..size-1] is
// an array of ints.
// post:  The ints in arr[0..size-1] are
// sorted in place from least to greatest.
void selection_sort(int size, int *arr) {
    for(int i=0; i<size; ++i)
        for(int j=i+1; j<size; ++j)
            if (arr[i] > arr[j]) //compare
                swap(&arr[i], &arr[j]);
}
```

The size of a workload is the the number of integers in the input array, `arr`. As workloads, we generated three arrays of random integers with each of the following sizes (8, 32, 64, 256, 1024, 2048, 4096) for a total of 21 workloads. Notice that the workload sizes span several orders of magnitude.

The measure of performance for `trend-prof` is the number of times each basic block is executed on each workload. Thus, `trend-prof` runs the selection sort program on each workload and measures the number of times each basic block is executed. For each basic block, `trend-prof` fits the observations to a power law model that predicts number of times the basic block will be executed as a function of number of `ints` in the array.

A little thought reveals that the comparison in the code above runs exactly $\frac{1}{2}n(n-1)$ times when `size` = n. Figure 1 shows the scatter plot of observations and the power law fit for the comparison line in the selection sort code. The best power law fit to the observations predicts $0.45\ n^{2.02}$ comparisons for an input array of size n. On the plot in Figure 1, this model is visually indistinguishable from the true answer of $0.5\ n^2\ -\ 0.5\ n$, but it does not match it perfectly because the lower order term adds some noise at smaller input sizes. If we instead run `trend-prof` with workloads of sizes (256, 1024, 2048, 4096, 8000, 16000, 32000, 64000, 128000), `trend-prof`'s model predicts $\frac{1}{2}n^2$ compares.

3.2 Measurement

Our focus in designing `trend-prof` is modeling scalability rather than exact running time. This view led to our choice of basic block counts as a measure of performance and of power law models to describe each basic block. If an algorithm scales quadratically with input size, it will execute some block a quadratic number of times.

The amount of time (or number of clock cycles) each basic block takes is another measure of performance, but we chose basic block counts because of the following advantages:

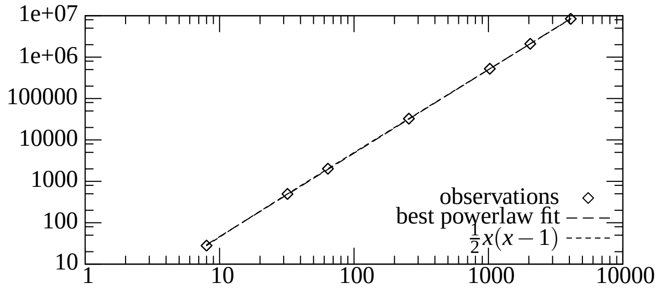


Figure 1: Number of times the comparison in the selection sort code executes as a function of input size. Shows observations and the best powerlaw fit ($\hat{y} = 0.45\ n^{2.02}$, $R^2 = 1.00$, $MRE = 0.4\%$) on a log-log scale.

- ACCURACY: Our measurements are *accurate*; we count exactly how many times each basic block executes. Issues of insufficient timer resolution do not apply to counting basic blocks.
- REPEATABILITY: Our measurements are as *repeatable* as the program's control flow is deterministic. If the control flow of a program is deterministic, we measure the same basic block execution counts on all runs of the program on a particular workload. Our measurements do not depend the operating system or architecture if the program's control flow does not.
- LACK OF BIAS: Our measurements are *unbiased*. Unlike measuring time, counting basic blocks does not affect the count of any basic block. We do not sample, so there is no possibility of sampling bias.
- PORTABILITY: Our solution is *portable*; we rely only on `gcc`'s coverage mechanism and not on platform-specific performance registers.

3.3 Model Construction

Running the client program and measuring basic block counts yields a set of (input size, count) pairs for each basic block $\{(x_i, y_i)\}_{i=1}^{k}$. For each basic block, `trend-prof` constructs a performance model that predicts how many times that block will be executed, $\hat{y}$, as a function of the (user-specified) workload size, x.

There is no single right answer to what model fits `trend-prof`'s observations. Statistical regression and other machine learning techniques are essentially optimization problems; one does not compute the "right" model, rather one selects the model that minimizes some measure of error.

Power Law Models. Our interest in measuring the scalability of a program led us toward power law models. A power law predicts y as a constant times a power of x, that is $y = ax^b$. To fit observations to a power law, `trend-prof` uses linear regression on $(\log x_i, \log y_i)$. In other words, `trend-prof` finds a and b that minimize the following quantity.

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

On log-log axes, the plot of a power law is a straight line. The power law fit has a number of advantages.

- It is *low-dimensional*; it estimates only two parameters. Models that estimate many parameters are subject to overfitting and require a lot more data.
- It focuses attention on the *highest order term*.
- It minimizes *relative error*; that is, it tolerates larger errors on larger input sizes. For instance, the model's predicting a basic block executes 100 times when it actually executes 1000 times is much more serious than predicting that it runs 1000 times when it actually runs 2000 times.
- It produces *simple models* that are easy to understand.
- It is usually *close enough*. We recognize bad fits and report them as such. Inspection of the residuals plot often reveals when there is an important lower order term.

Since the logarithm of 0 is not defined, `trend-prof` ignores points where the observed execution count is 0 when fitting to a power law. Thus, the models produced predict how many times a basic block is executed if it is executed at all. The output of `trend-prof` reports how many points were 0, and thus ignored, for each model. Some models may thus be fit from very few data points. After looking at some of these fits of few data points, we chose to have `trend-prof` not display models constructed with fewer than 10 data points. Models constructed with so few points do not convey much information and are unlikely to make accurate predictions.

Linear Models. In addition to fitting to a power law, `trend-prof` also fits the data for each basic block to a linear model, that is $\hat{y} = a + bx$ via linear regression (x_i, y_i). Linear regression finds a and b that minimize the following quantity.

$$\sum_{i=1}^{k} (y_i - (a + bx_i))^2$$

The quantity $r_i \equiv_{def} y_i - (a + bx_i)$ is called the *residual* of the fit at (x_i, y_i). Notice that linear regression minimizes the sum of the squared residuals.

3.3.1 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. For each model, `trend-prof` shows the data points and the line of best fit. Inspecting the plot is good way to decide if `trend-prof`'s model is plausible. Plots are not very compact, however, so on its summary page, `trend-prof` reports both the R^2 statistic for the fit, and the mean relative error (MRE) (proposed by Brewer [2] as a measure of goodness of fit for models of program performance). Values for R^2 range from 0 (bad) to 1 (excellent). Values for MRE range from 1 (excellent) upwards (worse). We follow Brewer's example and report MRE as a percentage, $100 \times (MRE - 1)\,\%$; so an MRE of 1.00 means 0% error, an MRE of 1.035 is 3.5% error, and an MRE of 3.00 is 200% error.

We review the formulae for R^2 and MRE below. The same formulae apply to power law fits, but with x replaced by $\log x$ and y

replaced by $\log y$.

$$\hat{y}_i =_{def} a + bx_i$$

$$\bar{y} =_{def} \frac{1}{k}\sum_{i=1}^{k} y_i$$

$$R^2 =_{def} \frac{\sum_{i=1}^{k}(\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{k}(y_i - \bar{y})^2}$$

$$\text{MRE} =_{def} \sqrt[k]{\prod_{i=1}^{k} \frac{y_i + |y_i - \hat{y}_i|}{y_i}}$$

The plot of the residuals versus input size $(x_i,\ r_i)$ is another important tool for determining whether a model captures all of the variation in a data set. The residuals plot should look like random noise. Any pattern in the residuals plot is an indication that there is more going on than the model describes – essentially a lower-order term is missing from the model. One case where `trend-prof`'s power law model misses such a term is with logarithmic factors. Consider calling the `qsort` function in C and measuring how many times (user supplied) comparison function is called. One would expect the number of calls to scale as $O(n \log n)$ in the size of the array being sorted.

Figure 2 shows `trend-prof`'s powerlaw model for this scenario and the residuals plot. The diamond shaped points show the observations used to construct the model; they range in size between 16 and 64,000. The line shows the best power law fit to these points $\hat{y} = 1.5x^{1.16}$. The fit closely tracks the data, but it's clear from the residuals plot that there is more going on. The hump shaped residuals suggest that the data grows more slowly than the power law suggests; such a shape suggests a logarithmic factor.

The circular points show further observations of the comparison function on larger input sizes that those used to construct the model. Even at input sizes several orders of magnitude larger than the largest training point, the model is within a factor of two from the measurement. At array size four million the model estimates about 68 million compares; we observed about 43 million (about 37% error).

In such cases, a power law model does not exactly describe the data. For our purposes, however, it is good enough. It shows the performance trend of the comparison function and gives a way to compare its empirical computational complexity to other parts of the program. Essentially, we have made a trade-off: our models are simple at the cost of not modeling some lower order terms.

3.4 Input Size

It is important for the user to pick a useful measure of input size; to `trend-prof` these input sizes are just numbers that it dutifully fits to models. To the extent that the input size is a good predictor of the execution count, the observations will be tightly clustered and a suitable model will fit the data well. To the extent that the input size is not a good predictor, the observations will look like random noise and no model will be able to say much about the program.

For many programs file size in bytes is a reasonable notion of input size. See Figure 4 for examples of programs on which we ran `trend-prof` and the notion of input size we chose. For file compression programs such as `bzip2` and `gzip`, file size is the only sensible measure of input size. For a graph processing program like `dot`, however, we conjecture that the number of edges (or nodes) in the input graph is a better predictor of performance than file size. Measuring the size of an input to programming language tools is harder. For our inputs to `banshee` we had data available about how many abstract syntax tree nodes each input file yielded; for `gcc` and `elsa` files size was all we had.

Choosing the right measure of input size can be rather subtle. In constructing workloads for our selection sort example, we wrote a script to output sequences of random integers as text; when generating n random integers, we would generate integers between 1 and n. This feature is benign except when combined with using the size of these files as the input size. Since larger numbers require more digits, the file size grew as $O(n \log n)$ with the number of integers. The resulting models concluded that selection sort executed $0.19\ n^{1.78}$ comparisons on a file of size n, about what one might expect for a power law approximation of $\frac{n^2}{\log n}$. We found the bug because the residuals plot suggested the presence of a logarithmic factor.

In addition to picking a good measure of input size, it is also important that a user of `trend-prof` pick a set of workloads that span the space of possible inputs to the program. The models `trend-prof` constructs will be representative of the chosen workloads; the model will be a good predictor of a new workload only to the extent that the new workload is like the ones on which `trend-prof` was initially run.

3.5 Implementation Details

Our tool, `trend-prof`, takes the following as input:

- A C or C++ program compiled and instrumented with `gcc -ftest-coverage -fprofile-arcs` to measure how many times each basic block is executed.
- A set of workloads annotated with input sizes.

In addition to the performance models for each basic block, `trend-prof` outputs an annotated copy of each source file with the model for a basic block next to the source lines corresponding to that block. Finally, `trend-prof` outputs a summary page that shows any user-specified combination of the following.

- For some user-specified input size, evaluate the model for each basic block at that input size and rank the basic blocks by predicted execution count at that input size.
- The models for each basic block, sorted by exponent.
- A model that predicts how many *total* basic blocks the program executes as a function of input size.
- An index of all the annotated source files.

4. EXPERIMENTS

We ran `trend-prof` on the programs listed in Figure 3 with workloads as described in Figure 4. In this section we report interesting facts and trends that `trend-prof` found in both the programs and the inputs.

The output of `trend-prof` is configurable as described in Section 3.5. For these experiments, we looked at models for basic blocks ranked in the following ways:

- highest exponent first
- highest predicted number of times the basic block executes at an input size near the largest measured input size
- highest predicted number of times the basic block executes at an input size about ten times larger than the largest measured input size

The tables and plots in this section give a feel for `trend-prof`'s output. In its output `trend-prof` often presents the same model for several consecutive lines of code. In presenting results, we filter out this extraneous information.

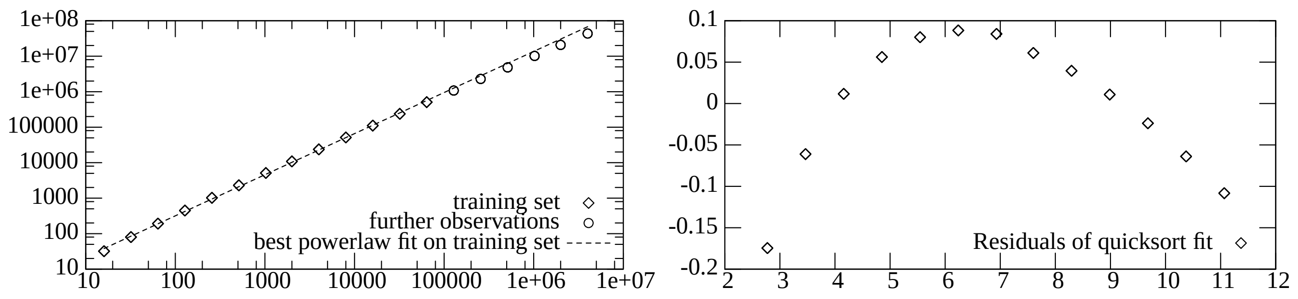


Figure 2: On the left is a log-log plot of number of comparisons done in a call to `qsort` (y axis) versus the size of the array (x axis). On the same plot, we show the best powerlaw fit to the diamond shaped points ($\hat{y} = 1.5\,n^{1.16}$, $R^2 = 0.999$, $MRE = 1.0\%$). On the right is the residuals plot for the power law fit.

Program	Description	Workloads
`gzip 1.2.4` [8]	Compresses files	Tarballs of preprocessed source code
`bzip2 1.0.3` [3]	Compresses files	Tarballs of preprocessed source code
`gcc 4.0.2` [6]	C and C++ compiler	Preprocessed C and C++ programs
`banshee 2005.10.07` [11]	Computes Andersen's alias analysis [1] on a C program	Preprocessed C programs
`elsa 2005.10.14` [5]	Parses, typechecks, and elaborates C and C++ code	Preprocessed C and C++ programs
`maximus`	Ukkonen's suffix tree algorithm [14] for finding common substrings	C source code
`dot`	Part of Graphviz [4] 2.6; renders a graph as postscript	Dataflow graphs of C and C++ programs
`jpgToPng`	Part of ImageMagick [9] 6.2.5.1; converts JPEG images to PNG images	JPEGs downloaded from an image search

Figure 3: Programs for which we present performance models and descriptions of the workloads for each program.

Program	Smallest workload	Largest Workload	Number of Workloads
`gzip`, `bzip2`	1.02×10^5 B	1.47×10^9 B	4137
`jpgToPng`	6×10^3 B	3.5×10^6 B	304
`dot`	3 edges	5.78×10^4 edges	146
`banshee`	7.0×10^2 AST nodes	2.1×10^6 AST nodes	31
`gcc`, `elsa`	1.0×10^2 B	4.59×10^6 B	253
`maximus`	7.3×10^2 B	6.5×10^5 B	36

Figure 4: Smallest workload, largest workload, and number of workloads for each program.

File	Line	Model	R^2	MRE	Prediction at $n = 10^8$
deflate.c	517	$2.5\,n^{1.03}$	0.99	0.3%	4.06×10^8
deflate.c	542	$0.62\,n^{1.03}$	0.99	0.4%	1.01×10^8
util.c	74	n	1.00	0.02%	1.00×10^8

Figure 5: Some top ranked basic blocks for `gzip` when sorted by predicted number of executions at input size $n = 10^8$ bytes.

4.1 gzip: Just Plain Linear

We ran `trend-prof` on `gzip`, a mature program for compressing and decompressing files. A workload for `gzip` consists of compressing a tarball full of preprocessed source code. The size of the workload is the size of the input file. We ran 4137 workloads.

Figure 5 shows the top ranked basic blocks for `gzip`. The results show that `gzip` scales very nearly linearly with the size of its input; the highest exponent is 1.03. For reference, at the astronomical input size of 10^{20} bytes, the model predicts $2.5(10^{20})^{1.03} \approx 10^{21}$, roughly a factor of 4 more than $2.5(10^{20})$. Despite the fact that most basic blocks in `gzip` scale linearly, a power law is nonetheless a good fit.

4.2 gcc: The Program as a Feature Detector for Trends in the Data

We exercised `gcc` in its capacity as a C and C++ compiler. A workload for `gcc` consists of compiling a preprocessed source file

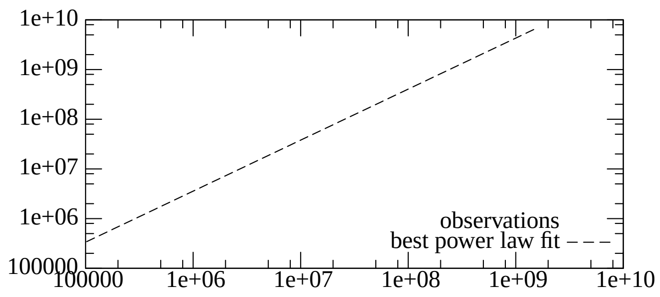


Figure 6: Log-log plot of observations and best powerlaw fit ($\hat{y} = 2.5\times n^{1.03}$, $R^2 = 0.999$, $MRE = 0.3\%$) for top ranked result for gzip: deflate.c, line 517.

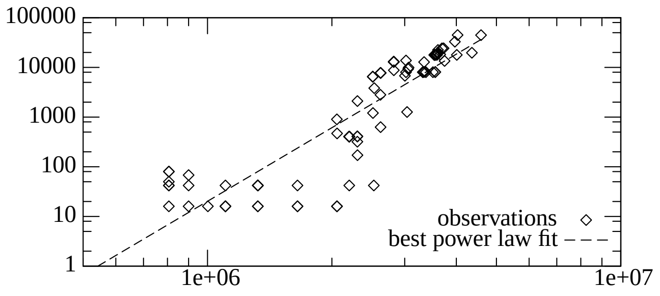


Figure 7: Observed basic block counts and best power law fit ($\hat{y} = 6.1 \times 10^{-29} n^{4.92}$, $R^2 = 0.77$, $MRE = 21\%$) showing a trend in the workloads for `gcc` (`search.c`, line 1655).

with most optimizations enabled (`gcc -O3`). The size of a workload is its size in bytes. As we discussed in Section 3.4, we conjecture that a more application-specific notion of input size is would yield tighter fits, but size in bytes is a reasonable first approximation.

At input size 1 MB, the lexer and parser dominate. At input size 10 MB, a new tend becomes apparent. There is a basic block that `trend-prof` predicts will execute $(1.5 \times 10^{-36}) n^{6.18}$ on an input of n bytes (about 27 million times on a 10 MB input). Some investigation shows that this code is concerned with clearing the visited flag for a depth first search of a class's superclasses and that it is invoked from the following code (which we present as pseudocode) from `search.c`. If no two of a `node`'s superclasses inherit directly from the same class (that is, if there are "diamond" patterns in the superclasses), then the DFS is guaranteed to never visit a node more than once. In this case, `gcc` invokes a faster version of DFS that does not need to mark visited nodes nor clear these marks when it is done.

```
if (no_diamonds(node))
    // DFS cannot encounter the same
    // node via multiple paths.
    rval = dfs_without_marking (node);
else
    rval = dfs_with_marking(node);
```

According to `trend-prof`'s model, the `if` statement is executed $0.0045 n^{1.10}$ times on an input of size n. The `then` branch is executed $0.0061 n^{1.07}$ times, but the `else` branch scales as $6.1 \times 10^{-29} n^{4.92}$ (Figure 7). Clearly the `if` statement is not adding a factor of $n^{3.8}$. What `trend-prof` has found is a trend in the *workloads*: as workloads get larger, a sharply increasing number of them have diamond inheritance patterns. *In addition to finding trends in the program,* `trend-prof` *also uses the program as a feature detector for finding trends in the input data.*

Examination of the scatter plots for this code confirms that the optimized (`true`) branch of this `if` statement is indeed the common case for the inputs on which we ran `gcc`. The models, however, predict that for input sizes somewhere around 10^7 bytes the non-optimized case will be more common. *So,* `trend-prof` *can find optimizations that yield diminishing returns with larger inputs.*

4.3 Elsa: A Quadratic Performance Surprise

`Elsa` is a C and C++ front end; it parses and typechecks C and C++ code and elaborates implicit syntax in C++. We ran it on the same workloads as `gcc`.

Source File	Line	Model	R^2	MRE	Prediction at $n = 10^7$
tree.h	166	$0.00092\ n^{1.50}$	0.94	6.5%	2.80×10^7
search.c	1667	$1.5 \times 10^{-36}\ n^{6.18}$	0.77	24%	2.77×10^7
search.c	456	$0.0014\ n^{1.44}$	0.96	5.9%	1.58×10^7
search.c	2491	$7.0 \times 10^{-30}\ n^{5.19}$	0.78	16%	1.53×10^7
parser.c	428	$0.99\ n^{1.02}$	0.97	3.2%	1.34×10^7

Figure 8: Some top ranked basic blocks for `gcc` when sorted by predicted number of executions at input size $n = 10^7$ bytes.

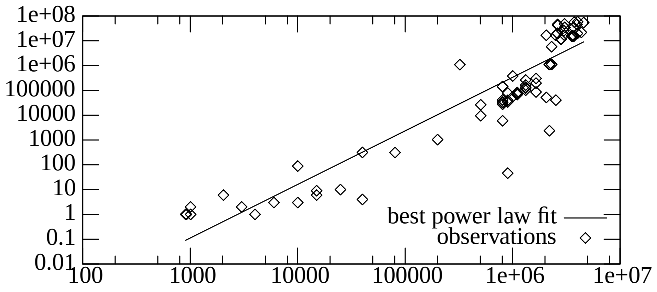


Figure 9: Observed basic block counts and best power law fit ($\hat{y} = 3.7 \times 10^{-8}\ n^{2.16}$, $R^2 = 0.8534$, $MRE = 28\%$) for the loop body of the performance bug in `elsa` (line 146, lookupset.cc).

One of the top ranked models for `elsa` is a quadratic insert into a set (that is, a linear time insert that is called a linear number of times). In the code below, `trend-prof` predicts that `LookupSet::add` is called $0.002\ n^{1.26}$ times on an input of size n. The loop body is predicted to be executed $3.7 \times 10^{-8}\ n^{2.16}$ times, approximately a linear factor more. Figure 9 shows the observations and the power law model for the loop body.

```
void LookupSet::add(Variable *v) {
    foreach elem in this
        if (sameEntity(v, elem)) return;
    add(v);
}
```

The developer points out that the sets are small, and indeed they are for the input sizes we measured. The model predicts, however, that at input sizes of 10^8 bytes, the body of this loop will execute more than 10^9 times. The developer agrees that the trend constitutes a low priority performance bug.

Another of the top ranked trends for `elsa` is an optimization of traversal of the inheritance graph similar to the one we found in `gcc`. Again, `trend-prof`'s output showed a trend in the data; there are more diamond shaped inheritance patterns in larger source files.

4.4 Ukkonen's algorithm: Linear Data Structure, but Super-Linear Output

Ukkonen's algorithm [14] finds common substrings in a string by constructing a data structure called a suffix tree. We ran `trend-prof` on an implementation of Ukkonen's algorithm in a tool called `maximus`. A workload for `maximus` consists of a string; the size of a workload is the number of characters in the string. We used source files for `pine` [12], a popular email client, as workloads.

We looked at `trend-prof`'s models for the constructors for suffix tree edges ($\hat{y} = 1.1\ n^{1.00}$, $R^2 = 0.99$, $MRE = 3\%$) and nodes

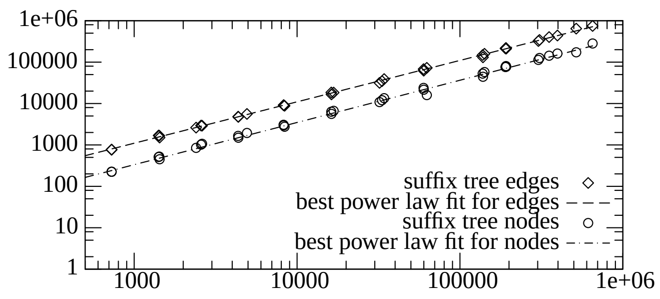


Figure 10: Observed execution counts and and power law fits for the suffix tree node and suffix tree edge constructors in `maximus`.

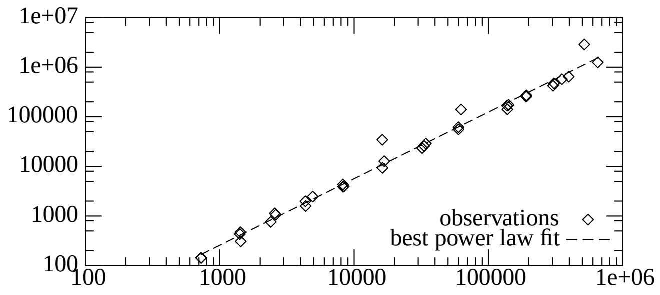


Figure 11: The `trend-prof` power law model showing the super-linearity in `maximus`'s output routines.

($\hat{y} = 0.29\ n^{1.02}$, $R^2 = 0.99$, $MRE = 4\%$) to confirm that the size of the suffix tree scaled linearly in the size of the input (shown in Figure 10). This knowledge is a comfort since Ukkonen's algorithm is difficult to implement correctly and bugs tend to make it quadratic or worse. Although the suffix tree code scales linearly, `trend-prof` found a super-linearity ($\hat{y} = 0.024\ n^{1.34}$, $R^2 = 0.99$, $MRE = 5\%$) in the code to render output (Figure 11 shows the observations and line of best fit for this code). The developer was initially surprised by this super-linearity, but after some reflection he understood its cause. The suffix tree representation of common substrings in a string is too compact to be comprehensible to a human, so `maximus` expands it. 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.

4.5 Banshee: Empirical Complexity Differs from Theoretical Worst-Case Complexity

We ran `trend-prof` on `banshee`, a program analysis framework [11]. Specifically, we ran `trend-prof` on `banshee`'s implementation of Andersen's pointer analysis [1]. We used as workloads preprocessed C programs. The size of a workload is the number of abstract syntax tree nodes in the workload.

We found a scalability bug in the C parser that `banshee` uses for its Andersen's analysis. The model for the `last_node` function below predicts that it is called $0.076\ n^{1.24}$ times and that the loop body executes $0.028\ n^{1.71}$ times on an input of size n. These predictions suggest that the average size of these lists grows as $n^{0.47}$ and that at workload sizes of 10^7 AST nodes, this line of code will be executed more than any other line in the program. Clearly, a pointer to the last node in the list is called for.

Source File	Line	Model	R^2	MRE	Prediction at $n = 10^7$
AST.c	34	$0.028\ n^{1.71}$	0.94	5.4%	132×10^{10}
hashset.c	119	$0.000021\ n^{1.95}$	0.83	25%	8.28×10^{10}
hash.c	299	$0.0084\ n^{1.58}$	0.96	4.9%	3.88×10^{10}
env.c	54	$7.8\ n^{1.18}$	0.99	1.8%	2.12×10^{10}
hashset.c	98	$0.000098\ n^{1.79}$	0.84	17%	1.94×10^{10}
jcollection.c	265	$0.000018\ n^{1.86}$	0.85	24%	1.33×10^{10}
hash.c	301	$0.0029\ n^{1.58}$	0.96	5.3%	1.31×10^{10}

Figure 12: Some top ranked basic blocks for `banshee` when sorted by predicted number of executions at input size 10^7 AST nodes.

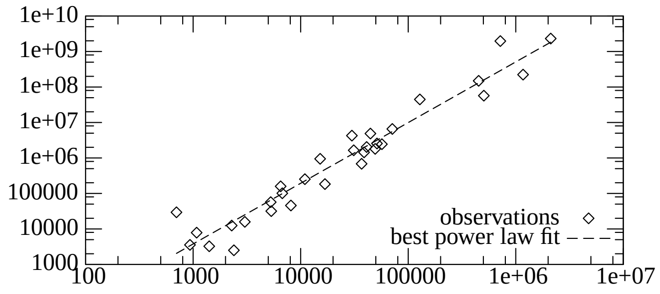


Figure 13: Observed basic block counts and best power law fit ($\hat{y} = 0.028\ n^{1.71}$, $R^2 = 0.9386$, $MRE = 5.4\%$) for the loop body of the performance bug in `banshee` (AST.c, line 34).

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

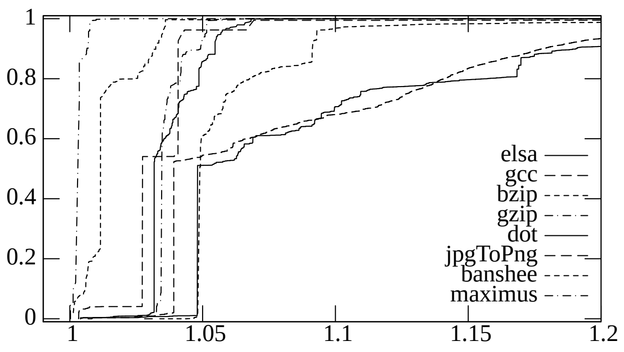
The interesting thing about the `banshee` results is that Andersen's analysis is a worst-case cubic algorithm. In practice though, the workloads we measured scaled much better than that. All of the top ranked fits at input size 10^8 AST nodes (see Figure 12) had scalability better than n^2. The reason `banshee` has sub-cubic behavior is its extensive optimizations that dramatically improve the common case; `trend-prof` allows us to quantify that improvement.

5. PROGRAMS MOSTLY SCALE AS A POWER LAW

The primary goal of `trend-prof` is to model empirically the scalability of programs. The basic blocks that are least important to scalability are those that are executed a constant number of times; that is, those whose execution count has no relationship to the size of the input. That is not to say that the most important basic blocks are those that execute the most – a basic block that looks innocent for medium size inputs could explode into super-linear behavior on larger inputs; however, the quality of `trend-prof`'s models for basic blocks that are barely executed is of little importance.

In many cases, `trend-prof` constructs power law or linear models that are a good fit for the observed data. There are, however, many models that do not fit as well. What we argue in this section is that `trend-prof`'s models are adequate for most of the *interesting* basic blocks in the program.

The best way to decide if a model fits observed data is for a human to look at the scatter plot and a line of best fit. If the regular



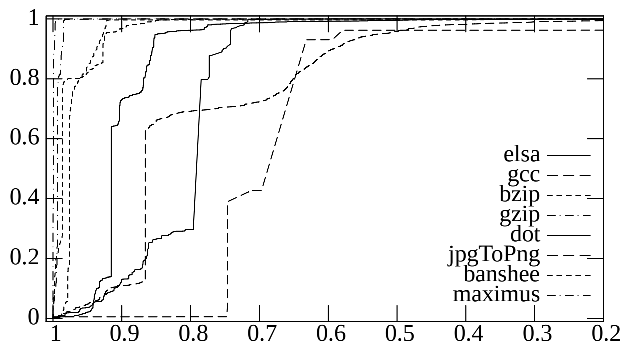


Figure 14: Goodness of fit (*x*) versus percentage of total basic block executions whose models fit better than *x*. The left plot shows MRE; the right plot shows R^2. For each basic block, we take the better MRE or R^2 of the linear and power law models. Graphically, lots of area under a line is good.

scatter plot looks like a line, then the code probably scales linearly. If, the log-log scatter plot looks like a line, then the code probably scales as a power law. Since `trend-prof` generates many models for a program, we must also rely on the MRE and R^2 as measures of goodness of fit.

5.1 Automatically Measuring Goodness of Fit

Figure 14 and Figure 15 show the relationship between how much a basic block executes and how good, according to the MRE and R^2 statistics, `trend-prof`'s model for that basic block is. For each goodness of fit *x* (MRE on the left, R^2 on the right), Figure 14 shows what percentage of total basic block executions have a model whose goodness of fit is *x* or better. In other words, if you move your finger up from $R^2 = 0.80$, the *y* coordinate where you hit the line shows what percentage of the total execution count of all basic blocks has a model that fits better than $R^2 = 0.80$. Graphically, a line with a large amount of area underneath it shows that `trend-prof`'s models do well for many of the basic blocks that execute the most times.

Figure 15 gives another view onto the goodness of fit versus execution count issue. Each plot in this figure is actually two plots compressed into one. The *x* axis (log scale) is the total number of times a basic block was executed. The *y* axis above the line at 1 shows the MRE (lower MRE is better) versus the total count. The *y* axis below the line shows R^2 (higher R^2 is better) versus the total count. Both `gzip` and `maximus` fit important basic blocks well according to both MRE and R^2. According to the MRE, `dot` also fits important basic blocks well and even `gcc` is mostly pretty good. Our other benchmark programs lie somewhere in the middle.

Although `trend-prof` does not perfectly predict the execution count of every basic block in the program, it does pretty well. For some of our programs `trend-prof` constructs good models for almost all basic blocks of any importance. For more complex programs like `gcc`, `elsa`, and to a lesser extent `dot` and `jpgToPng`, `trend-prof` constructs good models for many important basic blocks, but mediocre or bad models for others. The MRE, R^2, and residuals plots give users enough information to disregard bad models. These plots show that in general, the bad fits are concentrated in the basic blocks with low execution counts. In particular, *there are few basic blocks that execute many times and have a bad fit.*

5.2 Fits that are "Good Enough"

Figure 16 shows a fit is not perfect, but is "good enough" . There is some sort of trend and the model shows it, but the fit is noisy. The MRE and R^2 agree that the fit is middling.

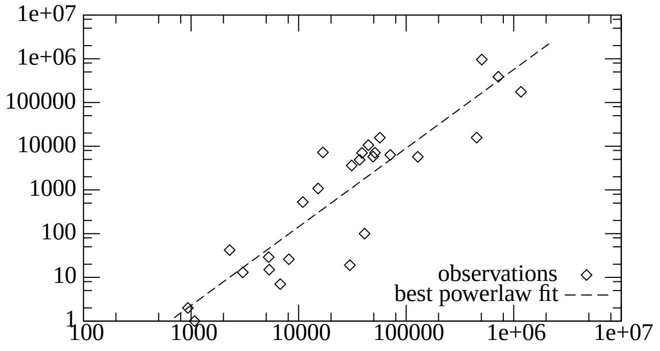


Figure 16: Observations and best power law fit ($\hat{y} = 9.2 \times 10^{-6}\ n^{1.8}$, $R^2 = 0.81$, $MRE = 30\%$) for `banshee`'s `engine/jcollection.c`, line 227.

5.3 Bad Fits

We present a few examples of bad fits and discuss the implications to our technique.

Figure 17 shows a situation where there is a fairly clear trend in a dense cloud of points, but with a few extreme outliers. The R^2 statistic penalizes the fit severely, but the MRE is not so bad. In more extreme cases, we have observed a data set where every observation but one or two was the same; again the R^2 was terrible, but the MRE was fine. Such fits are actually quite good, but blindly relying on R^2 would cause one to reject them.

Figure 18 shows a situation where there is no clear trend or pattern. We doubt that any simple model describes the behavior of this basic block with respect to the input image size. It is possible that some other notion of input size would lead to a tight model.

6. RELATED WORK

The main branches of related work are other profilers and other techniques that use empirical models of program performance.

6.1 Profilers

Gprof [7] 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 of the program 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.

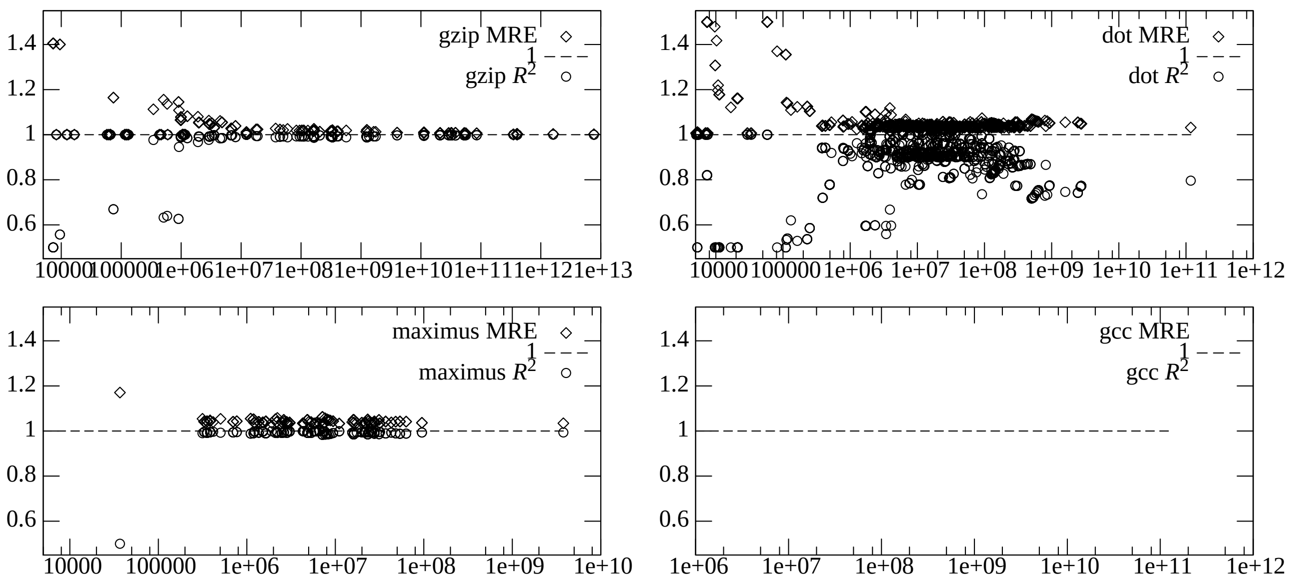


Figure 15: Goodness of fit (*y* axis) versus total number of executions of a basic block (*x* axis; log scale) for `gzip`, `dot`, `maximus`, and `gcc`. Above the line shows MRE versus total count. Below the line shows R^2 versus total count. We would like the basic blocks that are executed the most to have good fits. Graphically, we would like the upper right (MRE) and lower right (R^2) areas of the graph to be empty.

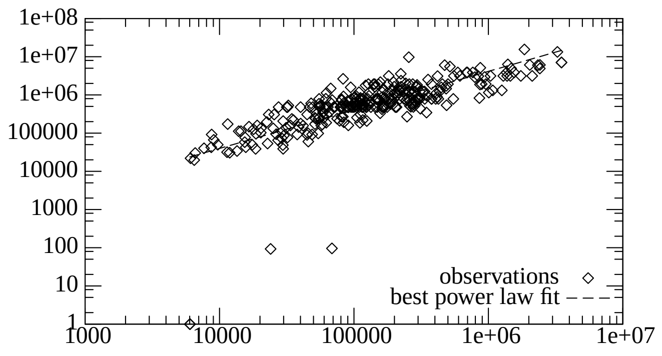


Figure 17: Observations and best power law fit ($\hat{y} = 3.7\ n^{1.01}$, $R^2 = 0.53$, *MRE* $= 6.6\%$) for `jpgToPng`'s `coders/jpg.c`, line 646.

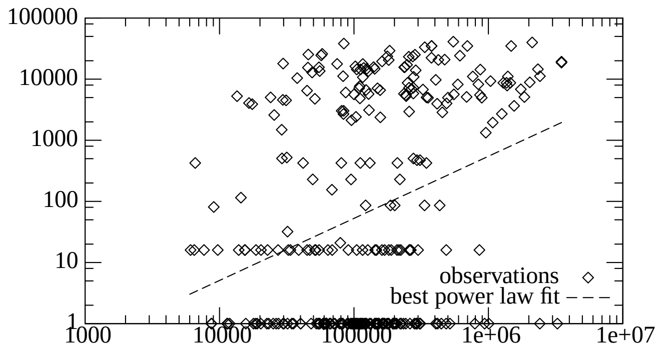


Figure 18: Observations and best power law fit ($R^2 = 0.097$, *MRE* $= 140\%$) for `jpgToPng`'s `coders/png.c`, line 5958.

Jinsight EX [13] 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 interactively browse this data to explore the performance of the program.

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

6.2 Building Models of Program Performance

Kluge et al. [10] 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 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 [2] constructs models that predict the performance of a library routine as a function of problem parameters; for instance he might model the performance of a radix sort in terms of 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 would perform. Based on the prediction, the library chooses an implementation of the algorithm to run the 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

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

Brewer's goal is to choose among several fairly well-understood implementations of an algorithm. The models serve to predict how an implementation will fare on particular problem instance on a particular architecture. The requirement that the user provide the terms for the model assumes a deeper level of understanding of the code's performance than `trend-prof` does. The resulting models can be more descriptive and precise, but each implementation of each algorithm must be considered separately and terms chosen carefully. In contrast, `trend-prof` seeks to describe the scalability of each of the many basic blocks in a large program and focus the user's attention on those with performance or scalability problems. *Our simpler, more automatic modeling is more appropriate for our goals.*

7. CONCLUSION

We propose models of empirical computational complexity for describing the performance of programs in practice. Our tool, `trend-prof`, measures runs of a program on many workloads and constructs linear and power law models of empirical computational complexity that predict how many times each basic block in a program runs as a function of input size. Linear and power law models fit observed basic block counts well for frequently executed basic blocks. Using `trend-prof`, we found some performance bugs and some characteristics of our workloads.

Many research papers present performance results for complex systems as a chart or a scatter plot of performance measurements. In contrast, our approach allows more precise, quantitative claims as well as revealing components that have interesting performance behavior not visible in the overall performance observations. In particular, one can report both the highest exponents that `trend-prof` predicts and `trend-prof`'s predictions at input sizes an order of magnitude or two higher than those measured.

8. ACKNOWLEDGMENTS

Blank because of double blind submission.

9. REFERENCES

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

[2] E. A. Brewer. High-level optimization via automated statistical modeling. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 80–91, New York, NY, USA, 1995. ACM Press.

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

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

[5] `elsa` project homepage. `http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/`.

[6] `gcc` project homepage. `http://gcc.gnu.org/`.

[7] 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.

[8] `gzip` project homepage. `http://www.gzip.org/`.

[9] Imagemagick project homepage. `http://www.imagemagick.org/`.

[10] 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.

[11] 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.

[12] `pine` project homepage. `http://www.washington.edu/pine/`.

[13] 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.

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