Measuring Empirical Computational Complexity Using \textit{trend-prof}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{plot.png}
\caption{Cluster with highest exponent: \(6.3191e-09 \times x^{1.98130}\).}
\end{figure}

Simon Fredrick Vicente Goldsmith
Daniel Shawcross Wilkerson
Alex Aiken
Algorithmic Scalability is a Timeless Concern

- No matter your resources, an unnecessary super-linearity can eat them all.
  - That is, never send a quadratic algorithm to do a linear algorithm's job.
- **We want** an understanding of performance that has
  - the **concreteness of empiricism** on a realistic set of workloads for a real program,
  - and the **generality of a trend** without the difficulty of theoretical analysis.
Empirical Asymptotic: Combining the Strength of Two Approaches

Consider performance of Insertion Sort

• NOT: Theoretical Asymptotic: analysis
  - worst case $\Theta(n^2)$
  - best case $\Theta(n)$
  - expected case depends on input distribution

• NOT: Empirical Pointwise: gprof
  - e.g., 2% of total time

• BUT: Empirical Asymptotic: trend-prof
  - empirically scales as, e.g., $n^{1.2}$
Core Idea

• For each line of the program
• Relate cost to input size
Our Method

• **Measure** performance, making a **matrix**:
  - one row per line of code,
  - one column per input workload.

• **Cluster** rows based on **linear correlation**.

• **Fit** rows to a **power law**.
Running Example: \texttt{bsort}

\begin{verbatim}
void bsort(int n, int *arr) {
    int i=0;
    while (i<n) {
        int j=i+1;
        while (j<n) {
            if (arr[i] > arr[j])
                swap(&arr[i], &arr[j]);
            j++;
        }
        ++i; } }
\end{verbatim}
Measure Performance, Making a Matrix

- Run workloads and measure performance.
- Record the results for each workload as a column of the matrix.

<table>
<thead>
<tr>
<th>Cost</th>
<th>Workload_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line_1</td>
<td>1</td>
</tr>
<tr>
<td>Line_2</td>
<td>61</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Line_5</td>
<td>1770</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Measure Performance, Making a Matrix

- Run workloads and measure performance.
- Record the results for each workload as a column of the matrix.

<table>
<thead>
<tr>
<th>Cost</th>
<th>Work (load)₁</th>
<th>Work₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line₁</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Line₂</td>
<td>61</td>
<td>201</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Line₅</td>
<td>1770</td>
<td>19900</td>
</tr>
</tbody>
</table>
Measure Performance, Making a Matrix

- Run workloads and measure performance.
- Record the results for each workload as a column of the matrix.

<table>
<thead>
<tr>
<th>Cost</th>
<th>Work₁</th>
<th>Work₂</th>
<th>...</th>
<th>Work₆₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line₁</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>Line₂</td>
<td>61</td>
<td>201</td>
<td>...</td>
<td>60001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Line₅</td>
<td>1770</td>
<td>19900</td>
<td>...</td>
<td>1.79997e9</td>
</tr>
</tbody>
</table>
Problem 1: Too Many Lines of Code

<table>
<thead>
<tr>
<th>Program</th>
<th>Basic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>1032</td>
</tr>
<tr>
<td>maximus</td>
<td>1220</td>
</tr>
<tr>
<td>elsa</td>
<td>33647</td>
</tr>
<tr>
<td>banshee</td>
<td>13308</td>
</tr>
</tbody>
</table>

- Leads to too many results to look at
  - Observation: Many lines vary together
Solution 1: *Clusters*, Subsets of Correlated Lines of Code

- Greedily assign lines of code to all clusters whose *cluster rep* they fit with $R^2 > 0.98$
- Lines of code that don't fit any cluster rep become new cluster reps
- Initial cluster rep is the input size
Empirical Fact: Clustering Works

<table>
<thead>
<tr>
<th>Program</th>
<th>Basic Blocks</th>
<th>Clusters</th>
<th>Costly Clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>1032</td>
<td>23</td>
<td>10</td>
</tr>
<tr>
<td>maximus</td>
<td>1220</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>elsa</td>
<td>33647</td>
<td>1489</td>
<td>30</td>
</tr>
<tr>
<td>banshee</td>
<td>13308</td>
<td>859</td>
<td>26</td>
</tr>
</tbody>
</table>

- Order of magnitude less clusters than blocks
- Furthermore there are few “costly” clusters
  - a cluster is “costly” if it accounts for more than 2% of total performance on any workload
Running Example: Clusters for bsort

```c
void bsort(int n, int *arr) {
    int i=0;
    while (i<n) {
        int j=i+1;
        while (j<n) {
            if (arr[i] > arr[j])
                swap(&arr[i], &arr[j]);
            j++;
        }
        ++i;
    }
}
```
From Now on, Think Clusters Rather Than Lines of Code

• Use the clusters as “abstract lines of code”
  - from now on, we just call them lines of code
Problem 2: How Do We Get a Trend From a Bunch O’Dots?

What The Heck Is This Trend ?!
Solution 2: Fit the Matrix Rows to a Power Law

Look for performance trends:
- each row records work done by each line of code

<table>
<thead>
<tr>
<th>Cost</th>
<th>( \text{Work}_{1} )</th>
<th>( \text{Work}_{2} )</th>
<th>( \ldots )</th>
<th>( \text{Work}_{60} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line_1</td>
<td>1</td>
<td>1</td>
<td>( \ldots )</td>
<td>1</td>
</tr>
<tr>
<td>Line_2</td>
<td>61</td>
<td>201</td>
<td>( \ldots )</td>
<td>60001</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>Line_5</td>
<td>1770</td>
<td>19900</td>
<td>( \ldots )</td>
<td>( 1.79997e9 )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
...Versus a User-Defined Notion of Input Size

Look for performance trends:
- each row records work done by each block
- with respect to user-specified input size

<table>
<thead>
<tr>
<th>Cost</th>
<th>Work(_1)</th>
<th>Work(_2)</th>
<th>...</th>
<th>Work(_{60})</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputSize</td>
<td>60</td>
<td>200</td>
<td>...</td>
<td>60000</td>
</tr>
<tr>
<td>Line(_1)</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>Line(_2)</td>
<td>61</td>
<td>201</td>
<td>...</td>
<td>60001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line(_5)</td>
<td>1770</td>
<td>19900</td>
<td>...</td>
<td>1.79997e9</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Again, Model Performance as a *Powerlaw* of Input Size

\[ \text{(Cost)} = a \times (\text{Input Size})^b \]

- Low dimensional
  - gives us high confidence for less data
- Easy to interpret
- Captures the high-order term
  - logarithmic factors are quite small in practice
  - polynomials converge to high order term
Handy Fact:
Powerlaws are Lines on Log-Log Axes

\[ \log (\text{Cost}) = \log a + b \cdot \log (\text{Input Size}) \]
Demo
### Running Example:

**trend-prof** results for `bsort`

| Max Cost in billions of basic block executions | Cluster Name | Cluster Total as a function of Input Size | $R^2$  
0=bad 1=good |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11 Compares</td>
<td>$3.1n^{2.00}$</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>2.5 Swaps</td>
<td>$3.0n^{1.93}$</td>
<td>0.996</td>
<td></td>
</tr>
<tr>
<td>&lt; 1 Size</td>
<td>$22n^{1.00}$</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>
bsort: Plots

- **Best-fit Plot**
  - $x$: log(size)
  - $y$: log(swap cluster)
  - line slope = 1.93

- **Residuals Plot**
  - $y$: residual
  - lack of randomness means the model missed something
trend-prof flow chart

- workloads
- input size
- plots of trends

run workloads
- matrix
- cluster
- matrix of cluster totals
- powerlaw fit

user trend-prof
Results
Confirmed Linear Scaling

- **Ukkonen's Algorithm (maximus)**
  - Theoretical Complexity: $\Theta(n)$
  - Empirical Complexity: $\sim n$
Empirical Complexity: Andersen's

- Andersen's points-to analysis (banshee)
  - Theoretical Complexity: $\Theta(n^3)$
  - Empirical Complexity: $\sim n^{1.98}$

Slope = 1.98
Empirical Complexity: GLR

• GLR C++ parser (elkhound / elsa)
  - Theoretical Complexity: $\mathcal{O}(n^3)$
  - Empirical Complexity: $\sim n^{1.13}$
How well do you know your code?

- Output routines (maximus)
  - Theoretical Complexity: $\theta(n)$?
  - Empirical Complexity: $\sim n^{1.30}$

Slope = 1.30
Algorithms in context

- The linear-time list append in banshee's parser is a bug

Slope = 1.21
R^2 = 0.95
The linear-time list append in Elsa's name lookup code is not a bug.
Results Recap

- Confirmed linear scaling (maximus)
- Empirical scalability (Andersen's, GLR)
- Unexpected behavior (maximus)
- Algorithms in context (elsa, banshee)
  - found a performance bug in banshee's parser
  - found similar situation, but no bug in elsa
Technical Contributions of Part 1

• Built `trend-prof`
  - a tool to measure empirical computational complexity
• Discovered the following empirical facts
  - programs have few clusters, fewer costly ones
  - powerlaw fits work well
• Showed that powerlaw fits of basic block counts reveal general trends
  - low-dimension but still nice precision
  - plots reveal the subtleties of actual computation...
Conclusion

• Semi-automatically empirically modeling performance trends as a function of input size works
  - examining the matrix rows instead of columns yields insight into scalability
  - control flow suggests precise models that sometimes improve upon direct models

• Comparing these models to expectation finds bugs or finds properties of the data

• Trend-prof belongs in the toolbox for performance / scalability testing
Thanks

- To Alex Aiken for all the stuff advisors do
- To Daniel Wilkerson for wonderful suggestions for improving this and other talks as well as our collaboration
Questions?

Code
trend-prof.tigris.org

Publications