Measuring Empirical Computational Complexity with trend-prof

Simon Goldsmith
Alex Aiken
Daniel Wilkerson

FSE 2007
September 7, 2007
Understanding Performance

- **Existing tools**
  - theoretical asymptotic complexity
    - e.g., $\mathcal{O}$ bounds, $\Theta$ bounds
  - empirical profiling
    - e.g., gprof

- **We propose an “empirical asymptotic” tool**
  - trend-prof
How does my code scale?

- Consider insertion sort
- Theoretical Asymptotic Complexity
  - worst case $\Theta(n^2)$
  - best case $\Theta(n)$
  - expected case depends on input distribution
- Empirical Profiling
  - e.g., 2% of total time
- trend-prof
  - empirically scales as, e.g., $n^{1.2}$
trend-prof measures workloads

- Run workloads and measure performance

<table>
<thead>
<tr>
<th>Workloads:</th>
<th>$w_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1:</td>
<td>1</td>
</tr>
<tr>
<td>Block 2:</td>
<td>61</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Block 5:</td>
<td>1770</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
**trend-prof**

- Run workloads and measure performance

<table>
<thead>
<tr>
<th>Workloads:</th>
<th>$w_1$</th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1:</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Block 2:</td>
<td>61</td>
<td>201</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block 5:</td>
<td>1770</td>
<td>19900</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
run workloads and measure performance
trend-prof

- Look for performance trends in each block

<table>
<thead>
<tr>
<th>Workloads:</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$\cdots$</th>
<th>$w_{60}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1:</td>
<td>1</td>
<td>1</td>
<td>$\cdots$</td>
<td>1</td>
</tr>
<tr>
<td>Block 2:</td>
<td>61</td>
<td>201</td>
<td>$\cdots$</td>
<td>60001</td>
</tr>
<tr>
<td>Block 5:</td>
<td>1770</td>
<td>19900</td>
<td>$\cdots$</td>
<td>$1.79997e9$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
trend-prof: Input Size

• Look for performance trends in each block
  – with respect to user-specified input size

<table>
<thead>
<tr>
<th>Workloads:</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>...</th>
<th>$w_{60}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Size:</td>
<td>60</td>
<td>200</td>
<td>...</td>
<td>60000</td>
</tr>
<tr>
<td>Block 1:</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>Block 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>Block 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>
Core Idea

- Relate **performance** of each basic block to **input size**
Uses of trend-prof

• Measure the performance trend an implementation exhibits on realistic workloads
  – and compare that to your expectations

• Identify locations that scale badly
  – may perform ok on smaller workloads, but dominate larger workloads
Example: `bsort`

```c
void bsort(int n, int *arr) {

1:  int i=0;
2:  while (i<n) {
3:    int j=i+1;
4:    while (j<n) {
5:      if (arr[i] > arr[j])
6:        swap(&arr[i], &arr[j]);
7:      j++;
8:    } ++i;

// O(n^2)
// O(n^2)
```

Challenges

• How to relate performance to input size?
• How to summarize a large amount of data?
Problem: Too Many Basic Blocks

<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 basic blocks vary together
Summarize with Clusters

- Group basic blocks with similar performance into the same *cluster*
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>

- Furthermore most clusters are small and cheap
  - a cluster is “costly” if it accounts for more than 2% of total performance on any workload
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; }
}
```
Cluster Total as Matrix Row

- Relate total executions of each cluster to input size
Relate Performance to Input Size

- Powerlaw regression is great
- \((\text{Cost}) = a (\text{Input Size})^b\)
  - Linear regression on \((\log \text{Input Size}, \log \text{Cost})\)
- Captures the high-order term
  - logarithmic factors don't matter in practice
  - polynomials converge to high order term
Powerlaw fit

\[ 2 \times \log(x) \]
\[ 3 \times x^2 - 10x \]
\[ 4 \times x^2 \]
\[ 5 \times x^3 \]
Output: `bsort`

<table>
<thead>
<tr>
<th>max cost (billions of basic block executions)</th>
<th>Cluster</th>
<th>Cluster Total as a function of input size</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Compares</td>
<td>3.1 $n^{2.00}$</td>
<td>1.00</td>
</tr>
<tr>
<td>2.5</td>
<td>Swaps</td>
<td>3.0 $n^{1.93}$</td>
<td>0.996</td>
</tr>
<tr>
<td>&lt; 1</td>
<td>Size</td>
<td>22 $n^{1.00}$</td>
<td>1.00</td>
</tr>
</tbody>
</table>
bsort: Plots

- log(size) vs log(swaps cluster)
- slope = 1.93

- residuals plot
  - they are small
  - they are not random
Results
Confirmed Linear Scaling

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

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

Slope = 1.98
Empirical Complexity: GLR

- GLR C++ parser (elkhound / elsa)
  - Theoretical Complexity: $O(n^3)$
  - Empirical Complexity: $\sim n^{1.13}$

Slope = 1.13
How well do you know your code?

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

Slope = 1.30
• 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

$R^2 = 0.65$
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
Technical Contributions

- **trend-prof**
  - a tool to measure empirical computational complexity

- Discovery of the following empirical facts
  - programs have few costly clusters
  - powerlaw fits work well
Conclusion

- **trend-prof** models total basic block count of a cluster as a powerlaw function \( y = ax^b \) of user-specified input size
  - enables thorough comparison of your expectations about scalability to empirical reality
  - finds locations that scale badly
download trend-prof at http://trend-prof.tigris.org