Program Testing and Analysis: Performance Profiling

Dr. Michael Pradel
Software Lab, TU Darmstadt
Outline

1. Introduction
2. CPU Time Profiling
3. Empirical Complexity

Partially based on these papers:

- *Evaluating the accuracy of Java profilers*, Mytkowicz et al., PLDI 2010
Motivation

- **Performance**: Non-functional property

- **Important because**:
  - Users dislike slow applications
  - Related to monetary cost (e.g., in data centers or automated trading)
  - Related to energy consumption

- **Simple changes may yield huge improvements**
Performance Profiling

- **Profiling**: Dynamic analysis to measure performance

- Observe runtime behavior to
  - Measure performance of code
  - Understand performance bottlenecks

- **Ultimate goal**: Provide insights that help developer address bottlenecks
Performance

- Various quantities to measure:
  - Time (focus of this lecture), memory usage, network bandwidth

- Absolute performance
  - E.g., milliseconds (time) or megabyte (memory usage)

- Relative performance
  - Compare versions of same program
  - Compare different programs
  - Compare ways to execute the same program
**Speedup vs. Improvement**

Two representations of relative performance:

**Speedup**

\[ s = \frac{t_{baseline}}{t_{measured}} \]

**Improvement**

\[ i = \frac{t_{baseline} - t_{measured}}{t_{baseline}} \]

Example: A takes 10 seconds, B takes 15 seconds

- Speedup of A over B is \( \frac{15}{10} = 1.5 \)
  (A is 1.5x faster than B)

- A improves over B by \( \frac{15 - 10}{15} = 33\% \)
Scalability

- Related to performance, but not the same

- Scalability: How does performance change w.r.t. to some parameter

- Typical parameters:
  - Input size
  - Number of CPU cores
  - Available memory
Execution Time

What is "time"?

Elapsed time → CPU time

User CPU time ← System CPU time

Other time ← Disk I/O

Network I/O

(See Unix "time" command)

What do we want to measure the execution time of?

- Entire program
- Code segment of interest
<table>
<thead>
<tr>
<th>Program</th>
<th>Elapsed time</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4s</td>
<td>3s</td>
</tr>
<tr>
<td>B</td>
<td>7s</td>
<td>4s</td>
</tr>
</tbody>
</table>

Which of the following is true?

- A is 1.43x faster than B
- B has a speedup of 0.57x over A
- A has a speedup of 1.75x over B
- A improves the CPU-time consumption by 25%
- A improves the CPU-time consumption by 33%
Quiz

<table>
<thead>
<tr>
<th>Program</th>
<th>Elapsed time</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4s</td>
<td>3s</td>
</tr>
<tr>
<td>B</td>
<td>7s</td>
<td>4s</td>
</tr>
</tbody>
</table>

Which of the following is true?

- A is 1.43x faster than B \[1.75x\]
- B has a speedup of 0.57x over A ✔
- A has a speedup of 1.75x over B ✔
- A improves the CPU-time consumption by 25% ✔
- A improves the CPU-time consumption by 33% 25%
Outline

1. Introduction
2. CPU Time Profiling
3. Empirical Complexity

Partially based on these papers:

- Evaluating the accuracy of Java profilers, Mytkowicz et al., PLDI 2010
- Measuring empirical computational complexity, Goldsmith et al., ESEC/FSE 2007
CPU Time Profiling

- Most widely used profiling technique

- Goal:
  - Measure how much time is spent in different parts of the program
  - Identify "hot" functions

- Result of profiling
  - Relative time spent in each function
  - Dynamic call tree: Time spent in caller vs. callee

- Implementation
  - Sampling-based vs. instrumentation-based
## CPU Time Profiling

Most widely used profiling technique

**Goal:**
- Measure how much time is spent in different parts of the program
- Identify "hot" functions

**Result of profiling**
- Relative time spent in each function
- Dynamic call tree: Time spent in caller vs. callee

### Implementation
- Sampling-based vs. instrumentation-based

### CPU Profiles

<table>
<thead>
<tr>
<th>Profiles</th>
<th>Self</th>
<th>Total</th>
<th>Function</th>
<th>Source File</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heavy (Bottom Up)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profiles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU PROFILES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profile 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Self</th>
<th>Total</th>
<th>Function</th>
<th>Source File</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>4447.3 ms</td>
<td>4447.3 ms</td>
<td>(idle)</td>
<td>crypto.js:583</td>
<td></td>
</tr>
<tr>
<td>2162.6 ms</td>
<td>2165.4 ms</td>
<td>montReduce</td>
<td>navier-stokes.js:152</td>
<td></td>
</tr>
<tr>
<td>1951.8 ms</td>
<td>1951.8 ms</td>
<td>lin_solve</td>
<td>richards.js:188</td>
<td></td>
</tr>
<tr>
<td>1643.9 ms</td>
<td>1652.8 ms</td>
<td>Scheduler.schedule</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>1476.7 ms</td>
<td>1964.1 ms</td>
<td>GeneratePayloadTree</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>1271.8 ms</td>
<td>1271.8 ms</td>
<td>bnpSquareTo</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>1170.8 ms</td>
<td>1172.0 ms</td>
<td>a8</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>987.9 ms</td>
<td>1081.7 ms</td>
<td>one_way_unify1_nboyer</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>884.5 ms</td>
<td>2269.5 ms</td>
<td>GeneratePayloadTree</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>763.5 ms</td>
<td>837.0 ms</td>
<td>a6</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>720.7 ms</td>
<td>720.7 ms</td>
<td>Exec</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>682.6 ms</td>
<td>1577.2 ms</td>
<td>rewrite_nboyer</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>624.5 ms</td>
<td>624.5 ms</td>
<td>SplayTree.splay_</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>619.2 ms</td>
<td>846.0 ms</td>
<td>Exec</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>558.0 ms</td>
<td>795.0 ms</td>
<td>(anonymous function)</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>540.0 ms</td>
<td>540.4 ms</td>
<td>Plan.execute</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>517.8 ms</td>
<td>799.7 ms</td>
<td>(anonymous function)</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>458.2 ms</td>
<td>1348.3 ms</td>
<td>(anonymous function)</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>402.6 ms</td>
<td>402.8 ms</td>
<td>HandlerTask.run</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>320.8 ms</td>
<td>582.2 ms</td>
<td>Constraint.satisfy</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>312.6 ms</td>
<td>1333.4 ms</td>
<td>loop3</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>301.8 ms</td>
<td>1348.7 ms</td>
<td>sc_loop1_98</td>
<td>crypto.js:431</td>
<td></td>
</tr>
<tr>
<td>274.2 ms</td>
<td>1349.6 ms</td>
<td>deriv_trees</td>
<td>crypto.js:431</td>
<td></td>
</tr>
</tbody>
</table>
**Sampling-based Profiling**

- **Probe** the target program’s call stack at regular intervals
- **Implemented** through OS interrupts or VM hooks

**Example:**

- **Start profiling**
- At time $t_1$: `exp()`, `bar()`, `foo()`
- At time $t_2$: `baz()`, `goo()`
- At time $t_3$: `exp()`, `bar()`, `hoo()`
- **End profiling**
Instrumentation-based Profiling

Add instructions to the target program

- **Time measurements**

  ```javascript
  var start = performance.now();
  foo();
  var total = performance.now() - start;
  ```

- **Counters**

  ```javascript
  totalCalls++;
  foo();
  ```
Comparison

**Sampling-based**

- Little impact on performance of target program
- Program runs relatively fast
- Lower resolution
- Sampling may be biased

**Instrumentation-based**

- Impact on performance of target program may cause observer effect
- Significant **slowdown** of program
- Higher resolution
- No sampling
How accurate are sampling-based profilers?

Compare different profilers with same benchmark

Hottest method according to 4 profilers (Mytkowicz et al., 2010)
Causality analysis to find which profiler is correct

- Profiler is "actionable" if making hot code faster speeds up the program
- Slow down a method (by adding some code) and check if profiler attributes slowdown to the method
- Result: None of the profilers produces actionable results
Mytkowicz’ Paper (3)

- Reason: **Samples are not taken randomly** but at yield points
- New profiler with **time-based random sampling**
  - Sample every $t \pm r$ milliseconds (where $t$ is constant and $r$ random within some range)
- Found new hot methods
- Optimized some and got 50% performance improvement with simple changes
Lessons Learned

- Performance measurements are non-trivial
- Even widely used tools may be wrong
- Knowing where to optimize is key to performance improvement
Outline

1. Introduction
2. CPU Time Profiling
3. Empirical Complexity

Partially based on these papers:

- *Evaluating the accuracy of Java profilers*, Mytkowicz et al., PLDI 2010
Empirical Complexity

- **Worst case complexity**: Commonly considered to choose algorithm
  - How does execution time vary with input size?
  - E.g., $O(n^2)$

- **But**: What is the **complexity of the actual implementation**?
  - Maybe better than expected for most inputs
  - Maybe worse than expected because of a bug

- **Idea**: Measure execution times and fit them to a model
Trendprof: Overview

Program

Workloads with features

Dynamic analysis

Cost measurements

Performance prediction

Linear model or powerlaw model
void bubble_sort(int n, int *arr) {
    
}

- Workloads: Arrays of $n$ integers
- Feature of a workload: Size $n$
- Example: 3 arrays of random integers of sizes 60, 200, 500, 1000, 2000, 4000, 8000, 15000, 30000, and 60000
Example

Locations in the code: Basic blocks

```c
void bubble_sort(int n, int *arr) {
    int i = 0;
    while (i < n) {
        int j = i + 1;
        while (j < n) {
            if (arr[j] < arr[i]) // compare
                swap(&arr[i], &arr[j]);
            j++;
        }
        i++;
    }
}
```
Measuring Execution Cost

- Execute and measure **number of executions** of each location
- Number of executions: **Proxy metric** for cost of locations
- Result matrix:

<table>
<thead>
<tr>
<th>locations</th>
<th>( \ell_1 )</th>
<th>( \ell_2 )</th>
<th>( \ddots )</th>
<th>( \ell_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_{1,1} )</td>
<td>( y_{1,2} )</td>
<td>( \ldots )</td>
<td>( y_{1,k} )</td>
<td></td>
</tr>
<tr>
<td>( y_{2,1} )</td>
<td>( y_{2,2} )</td>
<td>( \ldots )</td>
<td>( y_{2,k} )</td>
<td></td>
</tr>
<tr>
<td>( \ddots )</td>
<td>( \ddots )</td>
<td>( \ddots )</td>
<td>( \ddots )</td>
<td></td>
</tr>
<tr>
<td>( y_{n,1} )</td>
<td>( y_{n,2} )</td>
<td>( \ldots )</td>
<td>( y_{n,k} )</td>
<td></td>
</tr>
</tbody>
</table>

| features | \( f \) | \( f_1 \) | \( f_2 \) | \( \ldots \) | \( f_k \) |
|-----------|---------------|---------------|---------------|---------------|
| \( g \)  | \( g_1 \)    | \( g_2 \)    | \( \ldots \) | \( g_k \)    |

**Source:** Goldsmith et al., 2007
Predicting Performance

- Model execution cost as a function of features
- Divide locations into clusters (based on similar cost)
- Fit the (feature, cost) pairs of a cluster to a function
  - Linear function: $y(x) = a + bx$
  - Powerlaw function: $y(x) = ax^b$
Prediction for Example

Powerlaw functions for three clusters of locations

- **x axis**: Input size
- **y axis**: Frequency of execution
function findElement(arr, elem) {
    for (var i=0; i < arr.length; i++) {
        if (arr[i] == elem) return i; // location X
    }
}

Question: What function does trendprof predict for location X?

- Feature = Size of arr
- Workload 1: Arrays of random numbers and a random number contained in the array
- Workload 2: Arrays filled with 23 and 23
function findElement(arr, elem) { 
    for (var i=0; i < arr.length; i++) {
        if (arr[i] == elem) return i; // location X
    }
}

Question: What function does trendprof predict for location X?

- Feature = Size of arr
- Workload 1: Arrays of random numbers and a random number contained in the array
  Predicted performance: $y(x) = 0.5 \cdot x$
- Workload 2: Arrays filled with 23 and 23
  Predicted performance: $y(x) = 1$
Conclusion

Performance profiling: Dynamic analysis to measure and understand performance

- CPU time profiling: Identify hot functions
- Empirical complexity: Validate assumptions about complexity
- Profiling results heavily depend on inputs

Open challenges

- Generate inputs for profiling
- Suggest optimizations based on profiling results