Program Analysis – Lecture 12
Performance Profiling

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

true  false  Something else
Warm-up Quiz

What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

true  false  Something else
Warm-up Quiz

What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

Floating point numbers are represented with finite precision (not only in JavaScript)

true  false  Something else
Warm-up Quiz

What does the following code print?

```javascript
var a = (0.1 + 0.2) + 0.3;
var b = 0.1 + (0.2 + 0.3);
console.log(a === b);
```

0.30000000000000004 (due to rounding)

true  false  Something else
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:**

\[ S = \frac{t_{\text{baseline}}}{t_{\text{measure}}} \]

**Example:** A takes 10 seconds, B takes 15 seconds

**Speedup of A over B:**

\[ S = \frac{15}{10} = 1.5x \]

(i.e. A is 1.5x faster than B)

**Improvement:**

\[ i = \frac{t_{\text{baseline}} - t_{\text{measure}}}{t_{\text{baseline}}} \]

**A improves over B by:**

\[ i = \frac{15 - 10}{15} = 33\% \]
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"?

(See Unix "time" command)

What do we want to measure the execution time of?

- Entire program
- Code segment of interest
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
- 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>Function</th>
<th>Profile 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(idle)</td>
<td>4447.3 ms</td>
<td>4447.3 ms</td>
</tr>
<tr>
<td>montReduce</td>
<td>2162.6 ms</td>
<td>2165.4 ms</td>
</tr>
<tr>
<td>(garbage collector)</td>
<td>1951.8 ms</td>
<td>1951.8 ms</td>
</tr>
<tr>
<td>lin_solve</td>
<td>1643.9 ms</td>
<td>1652.8 ms</td>
</tr>
<tr>
<td>Scheduler.schedule</td>
<td>1476.7 ms</td>
<td>1964.1 ms</td>
</tr>
<tr>
<td>(program)</td>
<td>1271.8 ms</td>
<td>1271.8 ms</td>
</tr>
<tr>
<td>bnpSquareTo</td>
<td>1170.8 ms</td>
<td>1172.0 ms</td>
</tr>
<tr>
<td>GeneratePayloadTree</td>
<td>987.9 ms</td>
<td>1081.7 ms</td>
</tr>
<tr>
<td>a8</td>
<td>884.5 ms</td>
<td>2269.5 ms</td>
</tr>
<tr>
<td>one_way_unify1_nboyer</td>
<td>763.5 ms</td>
<td>837.0 ms</td>
</tr>
<tr>
<td>a6</td>
<td>720.7 ms</td>
<td>720.7 ms</td>
</tr>
<tr>
<td>rewrite_nboyer</td>
<td>682.6 ms</td>
<td>1577.2 ms</td>
</tr>
<tr>
<td>SplayTree.splay</td>
<td>624.5 ms</td>
<td>624.5 ms</td>
</tr>
<tr>
<td>Exec</td>
<td>619.2 ms</td>
<td>846.0 ms</td>
</tr>
<tr>
<td>(anonymous function)</td>
<td>558.0 ms</td>
<td>795.0 ms</td>
</tr>
<tr>
<td>Plan.execute</td>
<td>540.0 ms</td>
<td>540.4 ms</td>
</tr>
<tr>
<td>(anonymous function)</td>
<td>517.8 ms</td>
<td>799.7 ms</td>
</tr>
<tr>
<td>loop2</td>
<td>458.2 ms</td>
<td>1348.3 ms</td>
</tr>
<tr>
<td>HandlerTask.run</td>
<td>402.6 ms</td>
<td>402.8 ms</td>
</tr>
<tr>
<td>Constraint.satisfy</td>
<td>320.8 ms</td>
<td>582.2 ms</td>
</tr>
<tr>
<td>loop3</td>
<td>312.6 ms</td>
<td>1333.4 ms</td>
</tr>
<tr>
<td>sc_loop1_98</td>
<td>301.8 ms</td>
<td>1348.7 ms</td>
</tr>
<tr>
<td>deriv_trees</td>
<td>274.2 ms</td>
<td>1349.6 ms</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:

At time $t_1$: exp(), bar(), foo()
At time $t_2$: baz(), goo()
At time $t_3$: exp(), bar(), hoo()
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
Example

```c
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**:

  \[
  \begin{array}{c|cccc}
    \text{locations} & w_1 & w_2 & \ldots & w_k \\
    \hline
    \ell_1 & y_{1,1} & y_{1,2} & \ldots & y_{1,k} \\
    \ell_2 & y_{2,1} & y_{2,2} & \ldots & y_{2,k} \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    \ell_n & y_{n,1} & y_{n,2} & \ldots & y_{n,k} \\
  \end{array}
  \]

  \[
  \begin{array}{c|cccc}
    \text{features} & f & f_1 & f_2 & \ldots & f_k \\
    g & g_1 & g_2 & \ldots & g_k \\
  \end{array}
  \]

  **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
Quiz

```javascript
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 `elem=23`
  Predicted performance: \( y(x) = 1 \)
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 elem=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