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
- *Measuring empirical computational complexity*, Goldsmith et al., ESEC/FSE 2007

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

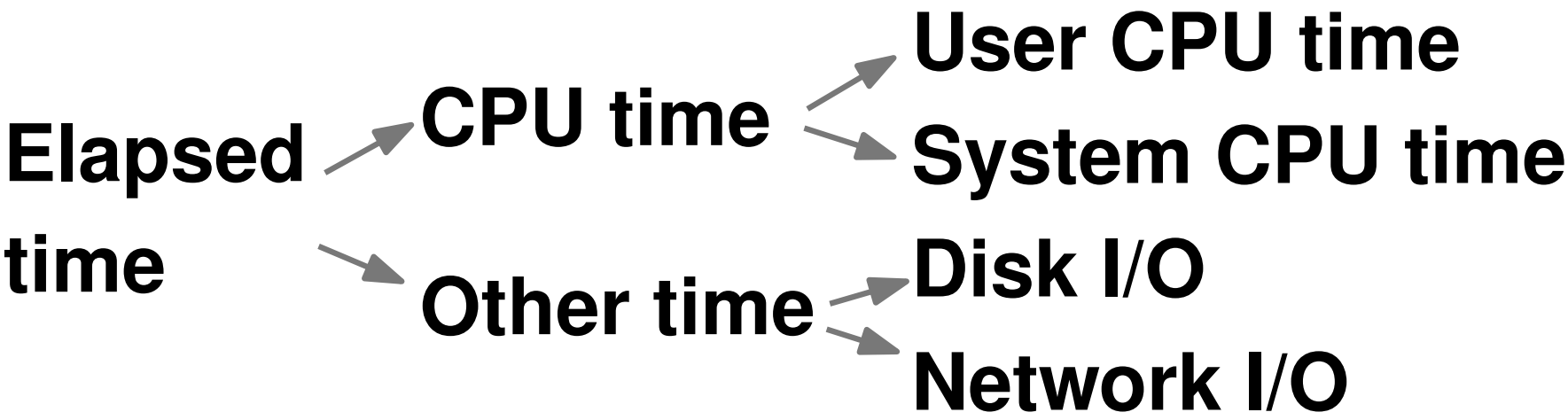
- Speedup of A over B is $15/10 = 1.5$
  (A is 1.5x faster than B)
- A improves over B by $\frac{15 - 10}{15} = 33\%$

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”?



(See Unix ”time” command)

What do we want to measure the execution time of?

- Entire program
- Code segment of interest

Quiz

Program	Elapsed time	CPU time
A	4s	3s
B	7s	4s

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

Program	Elapsed time	CPU time
A	4s	3s
B	7s	4s

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**

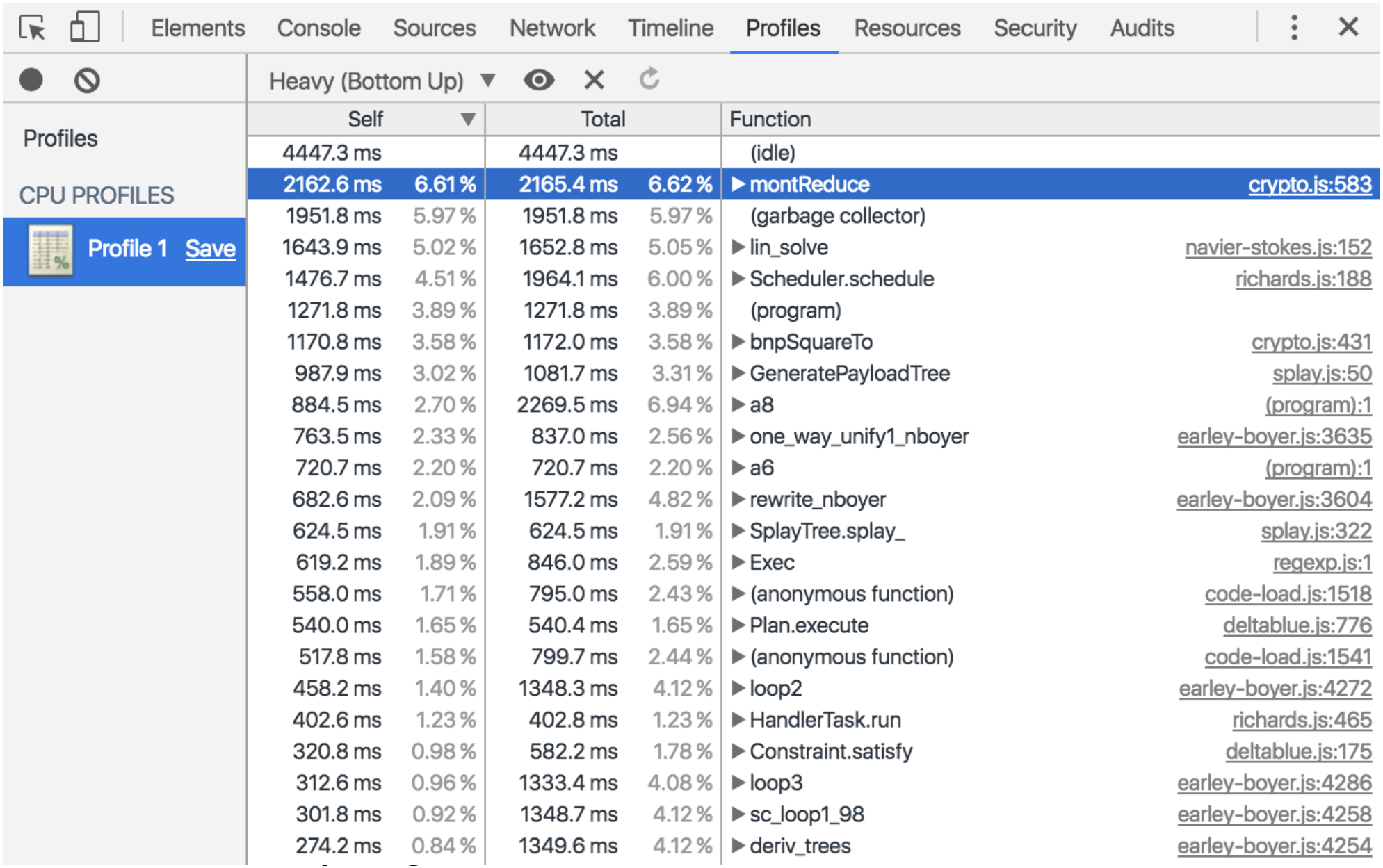
Partially based on these papers:

- *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

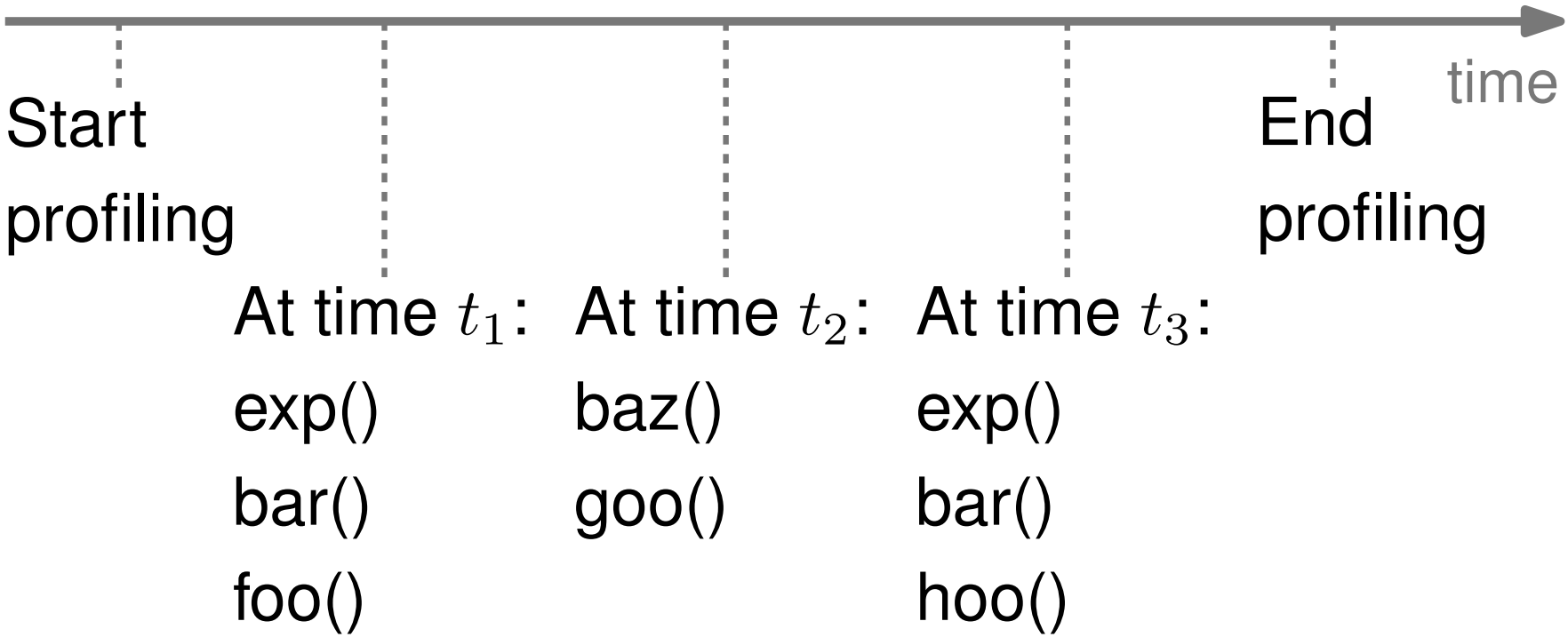


Self		Total		Function	
4447.3 ms		4447.3 ms		(idle)	
2162.6 ms	6.61 %	2165.4 ms	6.62 %	▶ montReduce	crypto.js:583
1951.8 ms	5.97 %	1951.8 ms	5.97 %	(garbage collector)	
1643.9 ms	5.02 %	1652.8 ms	5.05 %	▶ lin_solve	navier-stokes.js:152
1476.7 ms	4.51 %	1964.1 ms	6.00 %	▶ Scheduler.schedule	richards.js:188
1271.8 ms	3.89 %	1271.8 ms	3.89 %	(program)	
1170.8 ms	3.58 %	1172.0 ms	3.58 %	▶ bnpSquareTo	crypto.js:431
987.9 ms	3.02 %	1081.7 ms	3.31 %	▶ GeneratePayloadTree	splay.js:50
884.5 ms	2.70 %	2269.5 ms	6.94 %	▶ a8	(program):1
763.5 ms	2.33 %	837.0 ms	2.56 %	▶ one_way_unify1_nboyer	earley-boyer.js:3635
720.7 ms	2.20 %	720.7 ms	2.20 %	▶ a6	(program):1
682.6 ms	2.09 %	1577.2 ms	4.82 %	▶ rewrite_nboyer	earley-boyer.js:3604
624.5 ms	1.91 %	624.5 ms	1.91 %	▶ SplayTree.splay_	splay.js:322
619.2 ms	1.89 %	846.0 ms	2.59 %	▶ Exec	regexp.js:1
558.0 ms	1.71 %	795.0 ms	2.43 %	▶ (anonymous function)	code-load.js:1518
540.0 ms	1.65 %	540.4 ms	1.65 %	▶ Plan.execute	deltablue.js:776
517.8 ms	1.58 %	799.7 ms	2.44 %	▶ (anonymous function)	code-load.js:1541
458.2 ms	1.40 %	1348.3 ms	4.12 %	▶ loop2	earley-boyer.js:4272
402.6 ms	1.23 %	402.8 ms	1.23 %	▶ HandlerTask.run	richards.js:465
320.8 ms	0.98 %	582.2 ms	1.78 %	▶ Constraint.satisfy	deltablue.js:175
312.6 ms	0.96 %	1333.4 ms	4.08 %	▶ loop3	earley-boyer.js:4286
301.8 ms	0.92 %	1348.7 ms	4.12 %	▶ sc_loop1_98	earley-boyer.js:4258
274.2 ms	0.84 %	1349.6 ms	4.12 %	▶ deriv_trees	earley-boyer.js:4254

Sampling-based Profiling

- Probe the target program's call stack at regular intervals
- Implemented through OS interrups or VM hooks

Example:



Instrumentation-based Profiling

Add instructions to the target program

- **Time measurements**

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

- **Counters**

```
                totalCalls++;
foo();  ->      foo();
```

Comparison

Sampling-based

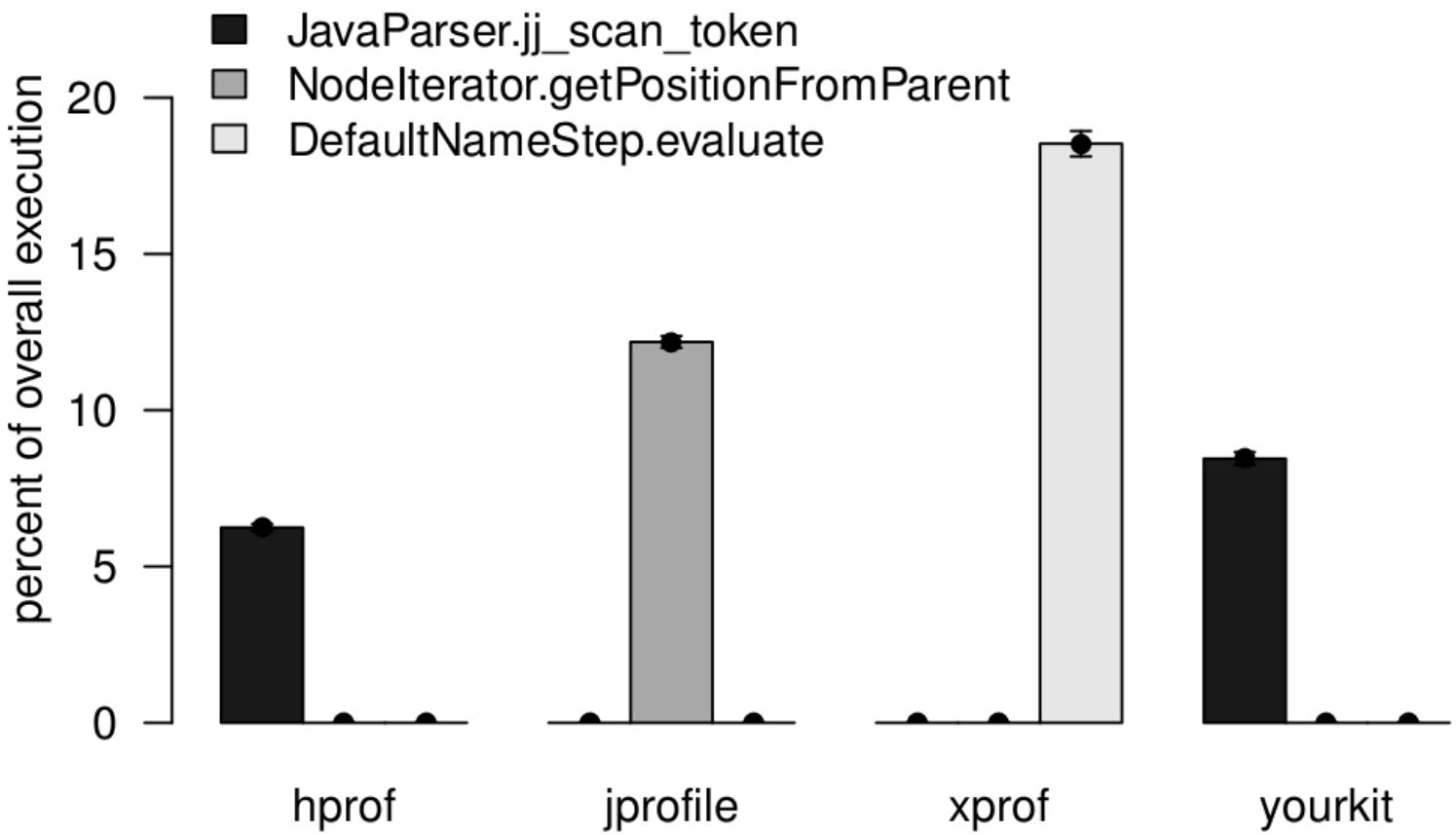
- Little impact on performance of target program
- Program runs relatively fast
- Lower resolution
- Sampling may be biased

Instrumentation-based

- Impact on performance of target program may cause observer effect
- Significant slowdown of program
- Higher resolution
- No sampling

Mytkowicz' Paper (1)

- How accurate are sampling-based profilers?
- Compare different profilers with same benchmark



Hottest method according to 4 profilers (Mytkowicz et al., 2010)

Mytkowicz' Paper (2)

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** 

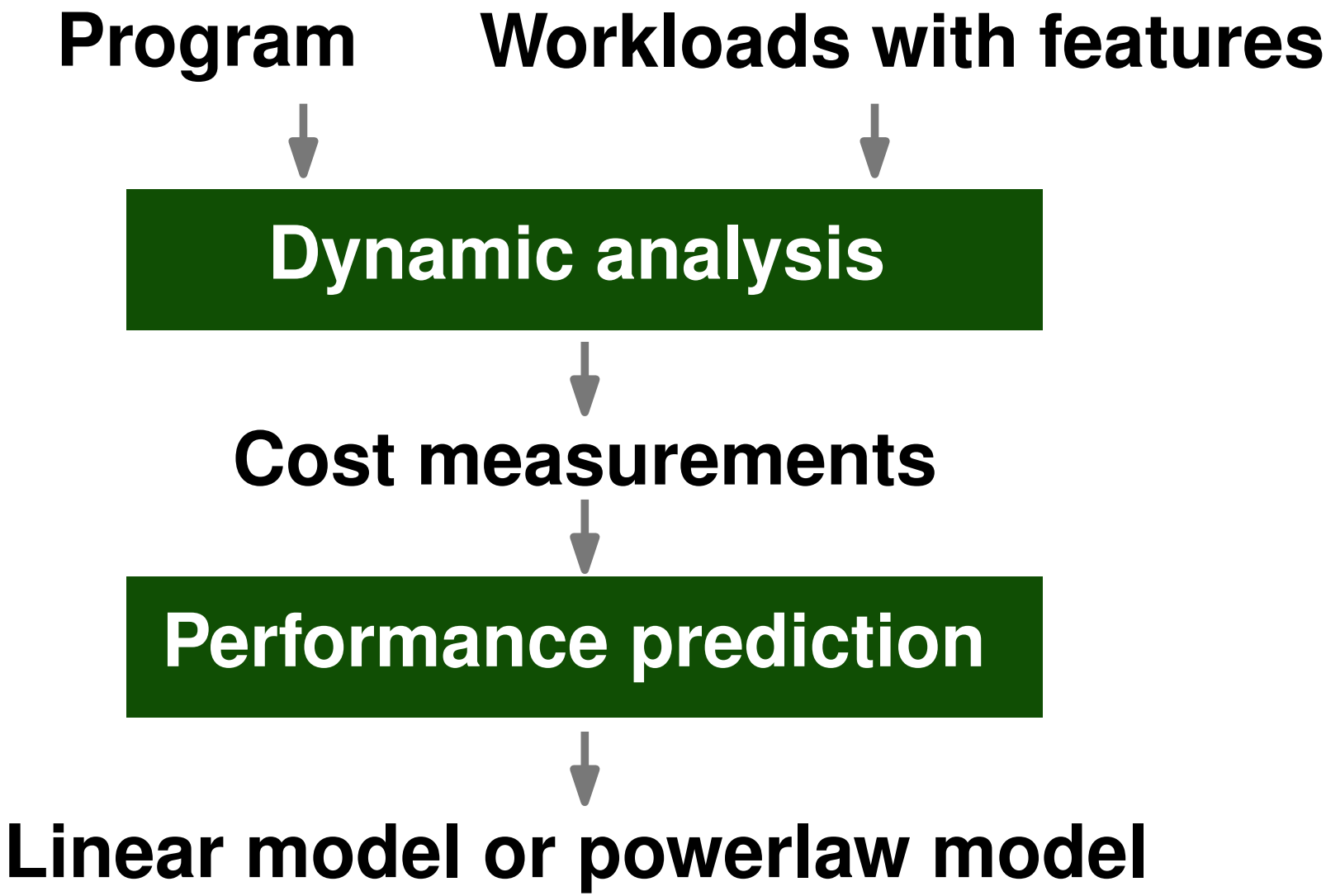
Partially based on these papers:

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

Empirical Complexity

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



Example

```
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

```
   void bubble_sort(int n, int *arr) {
1    int i = 0;
2    while (i < n) {
3      int j = i + 1;
4      while (j < n) {
5        if (arr[j] < arr[i]) //compare
6          swap(&arr[i], &arr[j]);
7        j++;
       }
8      i++;
     }
   }
```

Measuring Execution Cost

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

		workloads			
		w_1	w_2	$\cdots$	w_k
locations	ℓ_1	$y_{1,1}$	$y_{1,2}$	$\cdots$	$y_{1,k}$
	ℓ_2	$y_{2,1}$	$y_{2,2}$	$\cdots$	$y_{2,k}$
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	ℓ_n	$y_{n,1}$	$y_{n,2}$	$\cdots$	$y_{n,k}$
features	f	f_1	f_2	$\cdots$	f_k
	g	g_1	g_2	$\cdots$	g_k

Source: Goldsmith et al., 2007

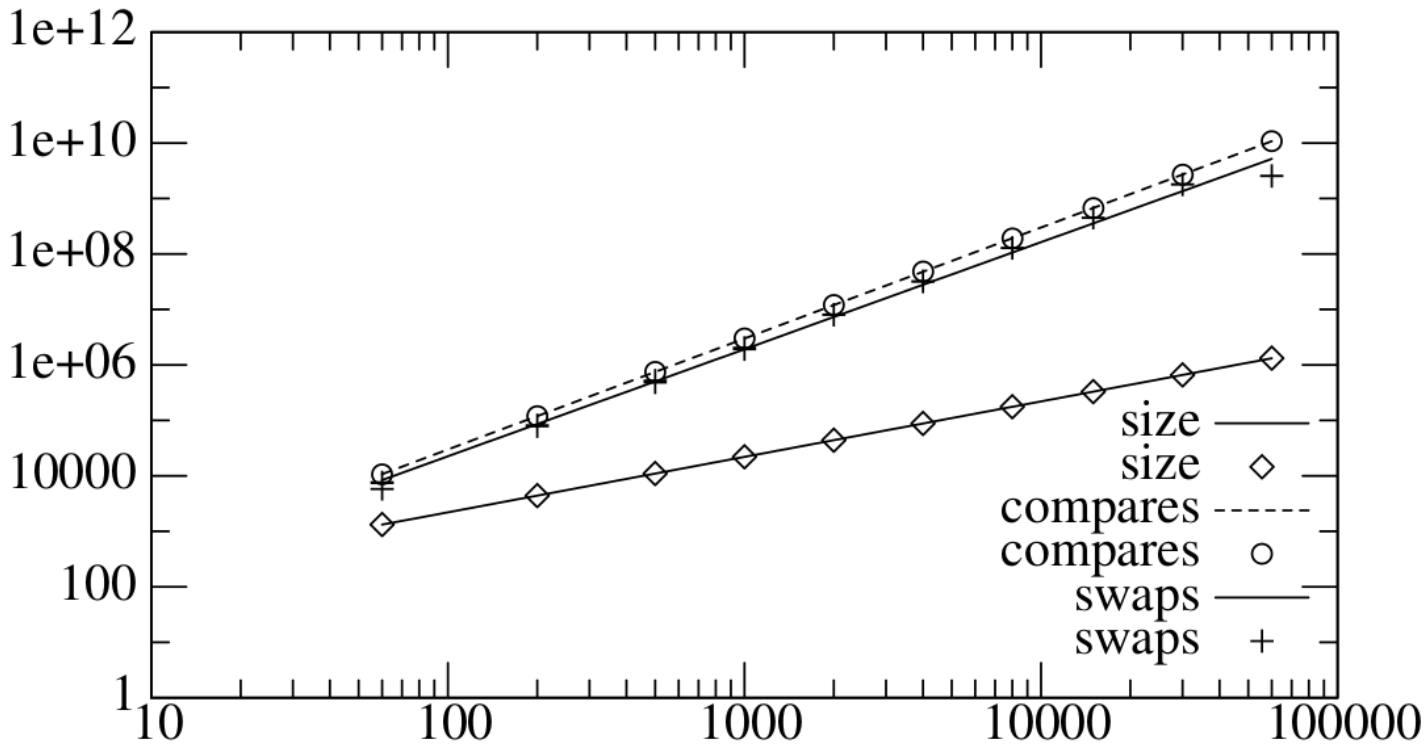
Predicting Performance

- **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



Quiz

```
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

Quiz

```
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