Benchmarking: Are We Doing it Wrong?

Edd Barrett

Carl Friedrich Bolz-Tereick

Rebecca Killick (Lancaster)

Sarah Mount

Laurence Tratt

Software Development Team
2017-05-27
Virtual Machine Warmup Blows Hot and Cold

EDD BARRETT, King’s College London
CARL FRIEDRICH BOLZ-TEREICK, King’s College London
REBECCA KILICK, Lancaster University
SARAH MOUNT, King’s College London
LAURENCE TRATT, King’s College London

Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: the initial warmup phase determines which parts of a program would most benefit from dynamic compilation, before JIT compiling them into machine code; subsequently the program is said to be at a steady state of peak performance. Measurement methodologies almost always discard data collected during the warmup phase such that reported measurements focus entirely on peak performance. We introduce a fully automated statistical approach, based on changepoint analysis, which allows us to determine if a program has reached a steady state and, if so, whether that represents peak performance or not. Using this, we show that even when run in the most controlled of circumstances, small, deterministic, widely studied microbenchmarks often fail to reach a steady state of peak performance on a variety of common VMs. Repeating our experiment on 3 different machines, we found that at most 43.5% of (VM, benchmark) pairs consistently reach a steady state of peak performance.

Programming Language Benchmarking

- Used to decide if a language is performing well.

- Used by pretty much all language designers:
  - Google
  - Oracle
  - Facebook
  - Mozilla
  - ...

- Long established methods.
The Current State of the Art of Benchmarking
The Current State of the Art of Benchmarking

Constant performance over time

Iteration time vs. in-process iteration
The Current State of the Art of Benchmarking

JUST IN TIME (JIT) COMPILATION

ITERATION TIME

IN-PROCESS ITERATION
The Current State of the Art of Benchmarking

JUST IN TIME (JIT) COMPILATION

ITERATION TIME

IN-PROCESS ITERATION

PROFILING INTERPRETER
The Current State of the Art of Benchmarking

JUST IN TIME (JIT) COMPILATION

PROFILING INTERPRETER

IN-PROCESS ITERATION

ITERATION TIME
The Current State of the Art of Benchmarking

Just in Time (JIT) Compilation

Compilation

Profiling Interpreter

Peak Performance

Iteration Time

In-Process Iteration

http://soft-dev.org/
The Current State of the Art of Benchmarking

JUST IN TIME (JIT) COMPILATION

ITERATION TIME

← WARMUP →

IN-PROCESS ITERATION
The Current State of the Art of Benchmarking

MORE REALISTIC VM WARMUP

ITERATION TIME

IN-PROCESS ITERATION
The Current State of the Art of Benchmarking

More Realistic VM Warmup

Iteration Time

Some Noise

In-Process Iteration
The Current State of the Art of Benchmarking

More realistic VM warmup

Compilation tiers

Some noise

In-process iteration
The Current State of the Art of Benchmarking

MORE REALISTIC VM WARMUP

- Compilation Tiers
- GC Spikes
- Some Noise

Iteration Time vs. In-Process Iteration
Warmup Matters

Users hate noticeable warmup.

![Graph showing iteration time vs. in-process iteration with annotations: Frustrating and Happy Days!](http://soft-dev.org/)
Warmup Matters

VM AUTHORS HATE ALL WARMUP

ITERATION TIME

IN-PROCESS ITERATION
We should measure the warmup of modern language implementations
We should measure the warmup of modern language implementations

Hypothesis: Small, deterministic programs reach a steady state of peak performance.
Method 1: Which benchmarks?

The CLBG benchmarks are perfect for us (unusually)

http://benchmarksgame.alioth.debian.org/
Method 1: Which benchmarks?

The CLBG benchmarks are perfect for us (unusually)

http://benchmarksgame.alioth.debian.org/

We removed any CFG non-determinism
Method 1: Which benchmarks?

The CLBG benchmarks are perfect for us (unusually)

http://benchmarksgame.alioth.debian.org/

We removed any CFG non-determinism

We added checksums to all benchmarks
Method 2: How long to run?

2000 in-process iterations
2000 \textit{in-process iterations}

30 \textit{process executions}
Method 3: VMs

- Graal-0.22
- HHVM-3.19.1
- JRuby/Truffle (git #6e9d5d381777)
- Hotspot-8u121b13
- LuaJit-2.0.4
- PyPy-5.7.1
- V8-5.8.283.32
- GCC-4.9.4

Note: same GCC (4.9.4) used for all compilation
Method 4: Machines

- Linux\textsubscript{4790}, Debian 8, 24GiB RAM
- Linux\textsubscript{E3-1240v5}, Debian 8, 32GiB RAM
- OpenBSD\textsubscript{4790}, OpenBSD 6.0, 32GiB RAM
Method 4: Machines

- Linux\textsubscript{4790}, Debian 8, 24GiB RAM
- Linux\textsubscript{E3-1240v5}, Debian 8, 32GiB RAM
- OpenBSD\textsubscript{4790}, OpenBSD 6.0, 32GiB RAM

- Turbo boost and hyper-threading disabled
- Network card turned off.
- Daemons disabled (crond, smtpd, sshd, atd, ...)

http://soft-dev.org/
Method 5: Benchmark Harness

KRUN

Control as many confounding variables as possible
Method 5: Benchmark Harness

KRUN

Control as many confounding variables as possible

- Minimises I/O
- Sets fixed heap and stack ulimits
- Drops privileges to a ‘clean’ user account
- Automatically reboots the system prior to each proc. exec
- Checks `dmesg` for changes after each proc. exec
- Checks system at (roughly) same temperature for proc. execs
- Enforces kernel settings (tickless mode, CPU governors, ...)
Results and Classification

Let’s look at some plots for our results.

We also classify each process execution.

Classification uses *changepoint analysis*. 
In-process iteration

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.56523</td>
<td>0.56703</td>
<td>0.56882</td>
<td>0.57061</td>
<td>0.57241</td>
<td>0.57420</td>
<td>0.57600</td>
<td></td>
</tr>
</tbody>
</table>

Time (secs)

Fannkuch Redux, LuaJIT, OpenBSD, Proc. exec. #12 (warmup)

http://soft-dev.org/
Changepoint Analysis 101

Fannkuch Redux, LuaJIT, OpenBSD, Proc. exec. #12 (warmup)

In-process iteration

Time (secs)

0.56523
0.56703
0.56882
0.57061
0.57241
0.57420
0.57600

In-process iteration

Time (secs)

Fannkuch Redux, LuaJIT, OpenBSD 4790, Proc. exec. #12 (warmup)

In-process iteration

Time (secs)
Changepoint Analysis 101

Fannkuch Redux, LuaJIT, OpenBSD<sup>4790</sup>, Proc. exec. #12 (warmup)

Changepoint
Results and Classification

- All segs are equivalent:
  \text{FLAT}
Results and Classification: Flat

N-Body, PyPy, Linux\textsubscript{E3 -1240v5}, Proc. exec. #6 (flat)

In-process iteration

Time (secs)
Results and Classification

- All segs are equivalent: Flat
- Final equivalent segs $\geq 500$ iters:
  - Final seg is equivalent to the fastest seg: Warmup
Results and Classification: Warmup

Fasta, V8, Linux, Proc. exec. #15 (warmup)
Results and Classification: Warmup

Spectral Norm, PyPy, Linux, Proc. exec. #13 (warmup)

In-process iteration

Time (secs)

0.46483
0.46632
0.46781
0.46931
0.47080
0.47229
0.47378

0.46492
0.46942
0.47393

http://soft-dev.org/
Results and Classification

- All segs are equivalent: **Flat**
- Final equivalent segs $\geq 500$ iters:
  - Final seg is equivalent to the fastest seg: **Warmup**
  - Final seg is not equivalent to the fastest seg: **Slowdown**
Results and Slowdown

Richards, Hotspot, Linux<sub>E3−1240v5</sub>, Proc. exec. #3 (slowdown)

In-process iteration

Time (secs)
Results and Slowdown

Fasta, V8, Linux, Proc. exec. #14 (slowdown)
Results and Classification

- All segs are equivalent: **FLAT**
- Final equivalent segs >= 500 iters:
  - Final seg is equivalent to the fastest seg: **WARMUP**
  - Final seg is not equivalent to the fastest seg: **SLOWDOWN**
- Otherwise: **NO STEADY STATE**
Results and Classification: No steady state

Binary Trees, V8, Linux, Proc. exec. #24 (no steady state)

In-process iteration

Time (secs)
Inconsistent Process-executions

(Same machine)
Inconsistent Process-executions

(Different machines. Bouncing ball Linux-specific)
## Quantitative Results

<table>
<thead>
<tr>
<th>Class.</th>
<th>Steady</th>
<th>Steady</th>
<th>Steady</th>
<th>Class.</th>
<th>Steady</th>
<th>Steady</th>
<th>Steady</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iter (#)</td>
<td>iter (s)</td>
<td>perf (s)</td>
<td></td>
<td>iter (#)</td>
<td>iter (s)</td>
<td>perf (s)</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HHVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotSpot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LuaJIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyPy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| C      |        |        |        |        |        |        |        |
| Graal  |        |        |        |        |        |        |        |
| HHVM   |        |        |        |        |        |        |        |
| HotSpot |        |        |        |        |        |        |        |
| JRuby+Truffle |        |        |        |        |        |        |        |
| LuaJIT |        |        |        |        |        |        |        |
| PyPy   |        |        |        |        |        |        |        |
| V8     |        |        |        |        |        |        |        |

| C      |        |        |        |        |        |        |        |
| Graal  |        |        |        |        |        |        |        |
| HHVM   |        |        |        |        |        |        |        |
| HotSpot |        |        |        |        |        |        |        |
| JRuby+Truffle |        |        |        |        |        |        |        |
| LuaJIT |        |        |        |        |        |        |        |
| PyPy   |        |        |        |        |        |        |        |
| V8     |        |        |        |        |        |        |        |

Software Development Team

http://soft-dev.org/
<table>
<thead>
<tr>
<th>Class.</th>
<th>Steady iter (#)</th>
<th>Steady iter (s)</th>
<th>Steady perf (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graal</td>
<td>× (27lasses, 3lasses)</td>
<td>32.0 (17.0, 193.8)</td>
<td>6.60 (3.729, 36.608)</td>
</tr>
<tr>
<td>HHVM</td>
<td>× (24lasses, 4lasses, 2w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotSpot</td>
<td>× (25lasses, 5lasses)</td>
<td>7.0 (7.0, 53.5)</td>
<td>1.19 (1.182, 9.703)</td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td>_allocator</td>
<td>1082.0 (999.0, 1232.5)</td>
<td>2219.59 (2039.304, 2516.021)</td>
</tr>
<tr>
<td>LuaJIT</td>
<td>× (23lasses, 4lasses, 2−, 1w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyPy</td>
<td>× (27lasses, 3w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td>× (15−lasses, 9lasses, 6lasses)</td>
<td>1.5 (1.0, 794.0)</td>
<td>0.25 (0.000, 391.026)</td>
</tr>
</tbody>
</table>
## Results: Summary

<table>
<thead>
<tr>
<th>Class.</th>
<th>Linux$_{4790}$</th>
<th>Linux$_{1240v5}$</th>
<th>OpenBSD$_{4790}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(VM, benchmark) pairs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−</td>
<td>8.7%</td>
<td>13.0%</td>
<td>6.7%</td>
</tr>
<tr>
<td>⊣</td>
<td>28.3%</td>
<td>23.9%</td>
<td>10.0%</td>
</tr>
<tr>
<td>⊥</td>
<td>6.5%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>≈</td>
<td>4.3%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>=</td>
<td>6.5%</td>
<td>6.5%</td>
<td>13.3%</td>
</tr>
<tr>
<td>≿</td>
<td>45.7%</td>
<td>43.5%</td>
<td>70.0%</td>
</tr>
</tbody>
</table>

| Process executions |                  |                   |                   |
|-------------------|------------------|-------------------|
| −                 | 26.4%            | 20.9%             | 34.0%             |
| ⊣                 | 48.3%            | 51.5%             | 52.1%             |
| ⊥                 | 16.7%            | 17.9%             | 11.1%             |
| ≈                 | 8.7%             | 9.6%              | 2.8%              |
## Results: Summary

<table>
<thead>
<tr>
<th>Class.</th>
<th>Linux$_{4790}$</th>
<th>Linux$_{1240v5}$</th>
<th>OpenBSD$_{4790}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg$</td>
<td>8.7%</td>
<td>13.0%</td>
<td>6.7%</td>
</tr>
<tr>
<td>$\ll$</td>
<td>28.3%</td>
<td>23.9%</td>
<td>10.0%</td>
</tr>
<tr>
<td>$\lhd$</td>
<td>6.5%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>$\approx$</td>
<td>4.3%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>$=$</td>
<td>6.5%</td>
<td>6.5%</td>
<td>13.3%</td>
</tr>
<tr>
<td>$\nless$</td>
<td>45.7%</td>
<td>43.5%</td>
<td>70.0%</td>
</tr>
</tbody>
</table>

### $\langle$ VM, benchmark $\rangle$ pairs

<table>
<thead>
<tr>
<th>Class.</th>
<th>Linux$_{4790}$</th>
<th>Linux$_{1240v5}$</th>
<th>OpenBSD$_{4790}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg$</td>
<td>26.4%</td>
<td>20.9%</td>
<td>34.0%</td>
</tr>
<tr>
<td>$\ll$</td>
<td>48.3%</td>
<td>51.5%</td>
<td>52.1%</td>
</tr>
<tr>
<td>$\lhd$</td>
<td>16.7%</td>
<td>17.9%</td>
<td>11.1%</td>
</tr>
<tr>
<td>$\approx$</td>
<td>8.7%</td>
<td>9.6%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

### Process executions

<table>
<thead>
<tr>
<th>Class.</th>
<th>Linux$_{4790}$</th>
<th>Linux$_{1240v5}$</th>
<th>OpenBSD$_{4790}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg$</td>
<td>26.4%</td>
<td>20.9%</td>
<td>34.0%</td>
</tr>
<tr>
<td>$\ll$</td>
<td>48.3%</td>
<td>51.5%</td>
<td>52.1%</td>
</tr>
<tr>
<td>$\lhd$</td>
<td>16.7%</td>
<td>17.9%</td>
<td>11.1%</td>
</tr>
<tr>
<td>$\approx$</td>
<td>8.7%</td>
<td>9.6%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

Best case no. good process executions: 86.1%
## Results: Summary

### ⟨VM, benchmark⟩ pairs

<table>
<thead>
<tr>
<th>Class</th>
<th>$\text{Linux}_{4790}$</th>
<th>$\text{Linux}_{1240v5}$</th>
<th>OpenBSD$_{4790}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\equiv$</td>
<td>8.7%</td>
<td>13.0%</td>
<td>6.7%</td>
</tr>
<tr>
<td>$\leq$</td>
<td>28.3%</td>
<td>23.9%</td>
<td>10.0%</td>
</tr>
<tr>
<td>$\geq$</td>
<td>6.5%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>$\approx$</td>
<td>4.3%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>$\cong$</td>
<td>6.5%</td>
<td>6.5%</td>
<td>13.3%</td>
</tr>
<tr>
<td>$\neq$</td>
<td>45.7%</td>
<td>43.5%</td>
<td>70.0%</td>
</tr>
</tbody>
</table>

### Process executions

<table>
<thead>
<tr>
<th>Class</th>
<th>$\text{Linux}_{4790}$</th>
<th>$\text{Linux}_{1240v5}$</th>
<th>OpenBSD$_{4790}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\equiv$</td>
<td>26.4%</td>
<td>20.9%</td>
<td>34.0%</td>
</tr>
<tr>
<td>$\leq$</td>
<td>48.3%</td>
<td>51.5%</td>
<td>52.1%</td>
</tr>
<tr>
<td>$\geq$</td>
<td>16.7%</td>
<td>17.9%</td>
<td>11.1%</td>
</tr>
<tr>
<td>$\approx$</td>
<td>8.7%</td>
<td>9.6%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

Best case good ⟨VM, benchmark⟩ pairings: 43.5%
Are the effects due to JIT and GC?

Fasta, PyPy, Linux\textsubscript{E3 1240v5}, Proc. exec. #5 (no steady state)
Are the effects due to JIT and GC?

Richards, Hotspot, Linux\textsuperscript{E3-1240v5}, Proc. exec. #3 (slowdown)

In-process iteration

Time (secs)

JIT (secs)

GC (secs)
Are the effects due to JIT and GC?

However

In many cases, the JIT/GC can’t explain oddness
What Have We Learned?

- Benchmarks often don’t warmup as we expect.

- Repeating a benchmark often gives a different warmup characteristic.

- Invalid benchmarking assumptions may have misled us!
  - Ineffectual or bad optimisations?
What Can We Do?

1. Unbenchmark for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. To avoid over-training on small benchmark suites.
5. The more benchmarks, the better.
6. Focus on predictable performance.
1. Run benchmarks for longer to uncover issues.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.
5. The more benchmarks, the better.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.
5. The more benchmarks, the better.
6. Focus on predictable performance.
References

VM Warmup Blows Hot and Cold
E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount and L. Tratt.

Rigorous Benchmarking in Reasonable Time
T. Kalibera and R. Jones

Specialising Dynamic Techniques for Implementing the Ruby Programming Language
C. Seaton (Chapter 4)

Quantifying performance changes with effect size confidence intervals
T. Kalibera and R. Jones