Benchmarking: Are We Doing it Wrong?

Edd Barrett

Carl Friedrich Bolz-Tereick

Rebecca Killick (Lancaster)

Sarah Mount

Laurence Tratt

KING'S College LONDON

Software Development Team

2017-05-27

To appear: OOPSLA 2017, Vancouver.

https://arxiv.org/abs/1602.00602

1

Virtual Machine Warmup Blows Hot and Cold

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

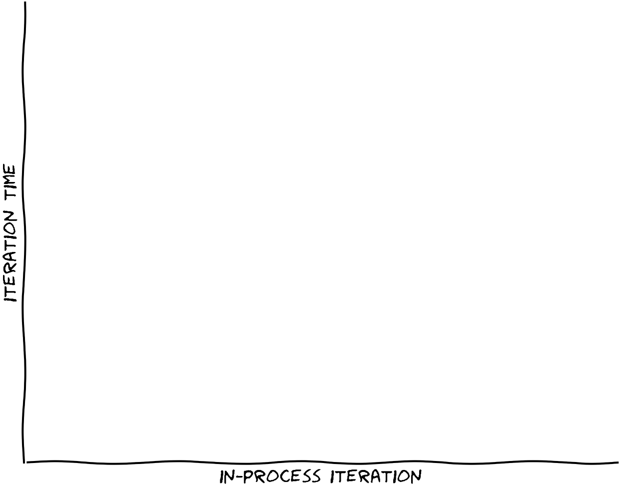
Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: the initial warmup phase determines which parts of a program would most benefit from dynamic compilation, before JIT compiling them into machine code; subsequently the program is said 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.

CCS Concepts: •**Software and its engineering → Software performance; Just-in-time compilers;** *In-*

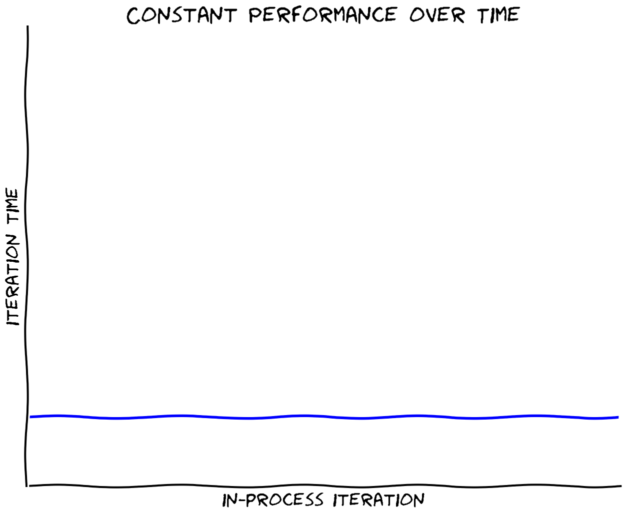
Programming Language Benchmarking

- Used to decide if a language is performing well.
- Used by pretty much all language designers:
  - Google
  - Oracle
  - Facebook
  - Mozilla
  - ...
- Long established methods.

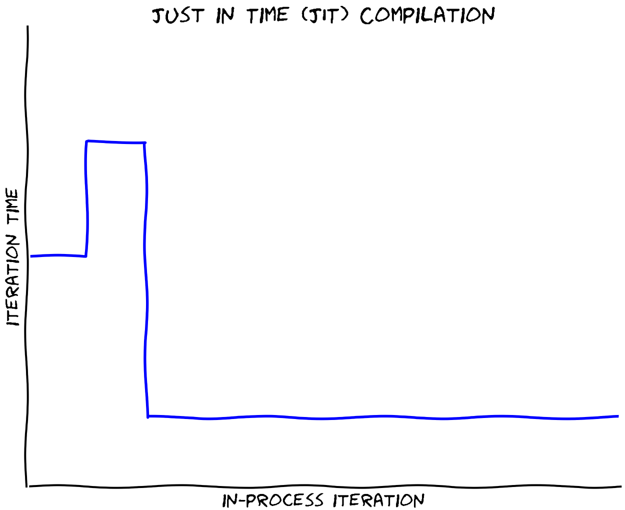
The Current State of the Art of Benchmarking



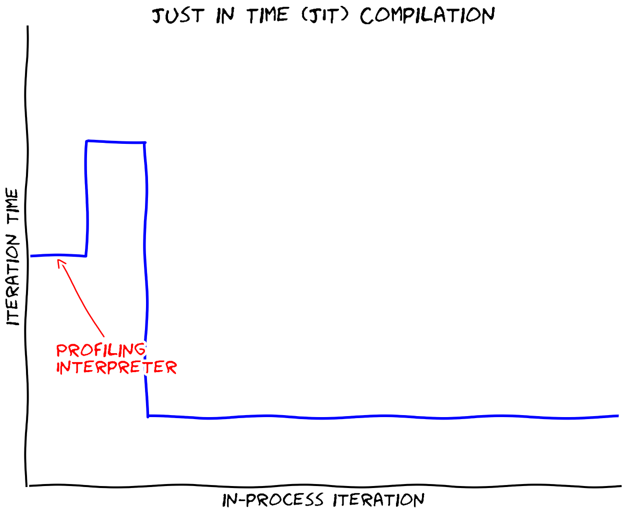
The Current State of the Art of Benchmarking



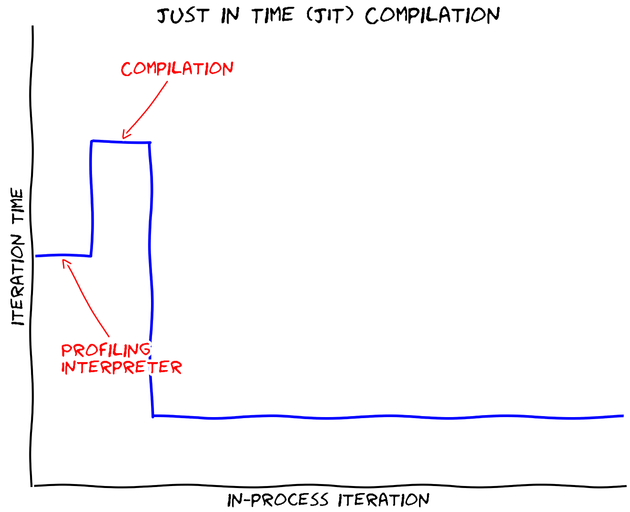
The Current State of the Art of Benchmarking



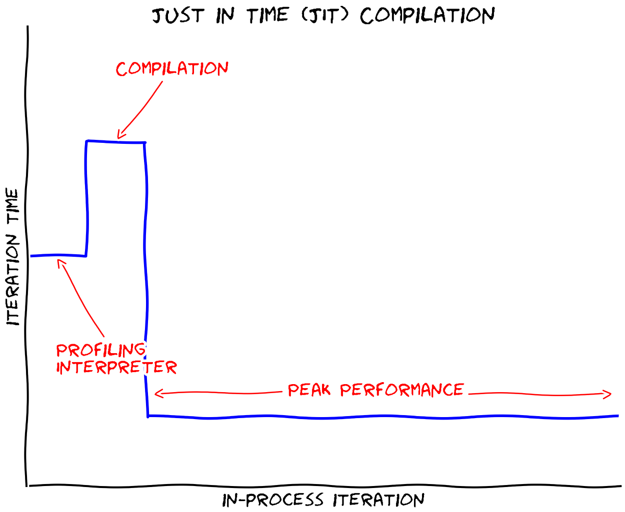
The Current State of the Art of Benchmarking



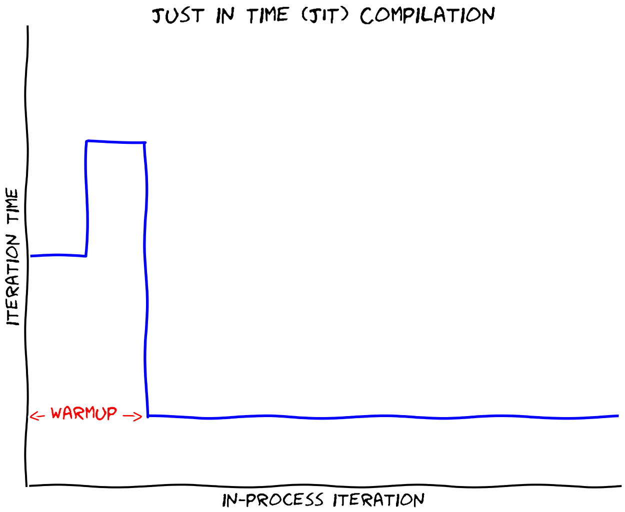
The Current State of the Art of Benchmarking



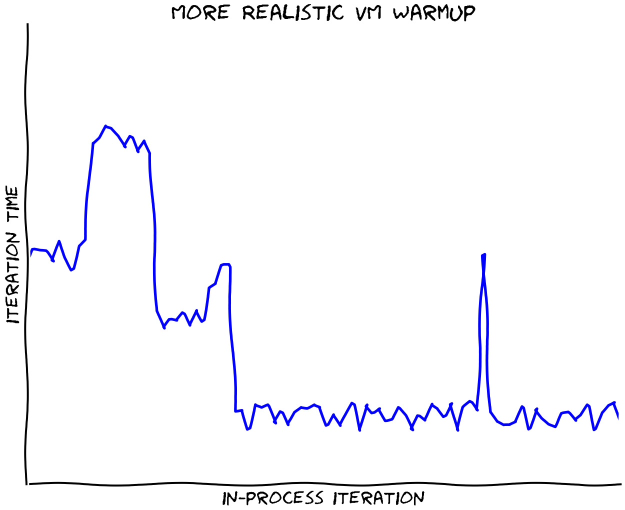
The Current State of the Art of Benchmarking



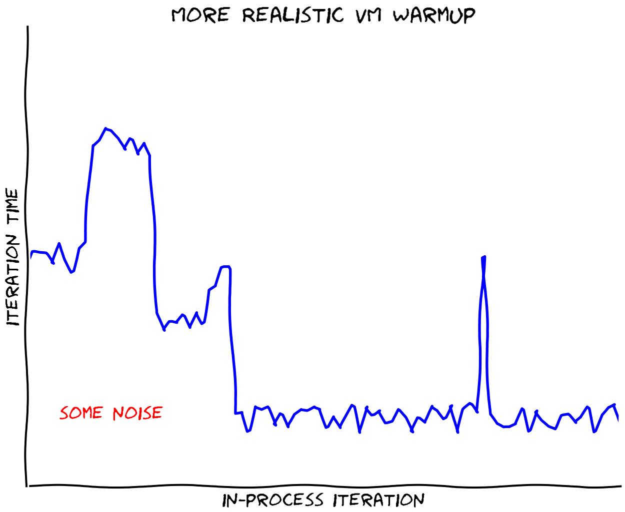
The Current State of the Art of Benchmarking



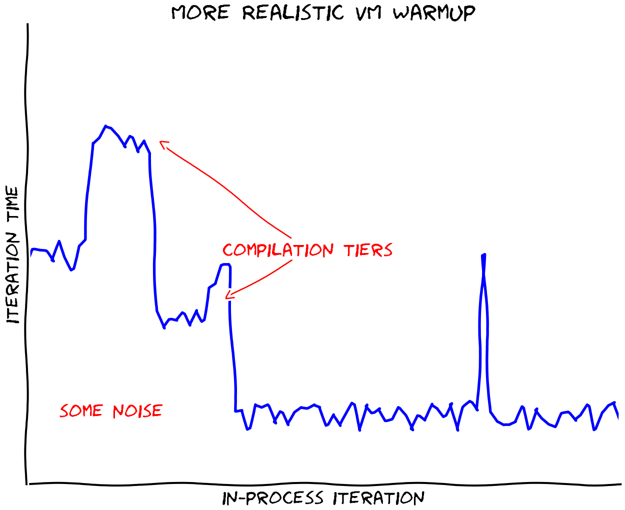
The Current State of the Art of Benchmarking

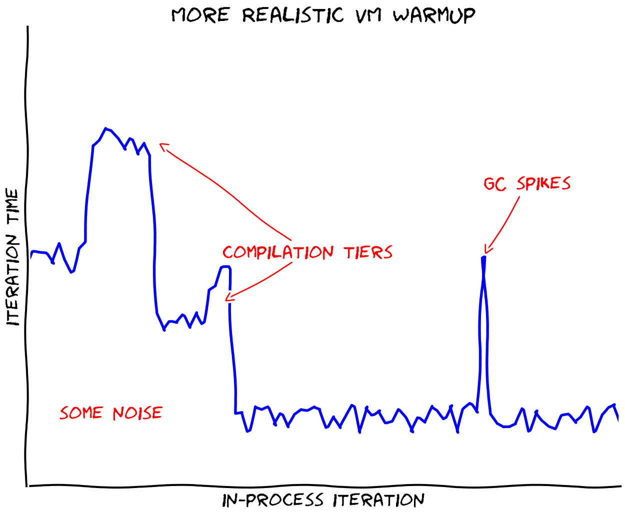


The Current State of the Art of Benchmarking

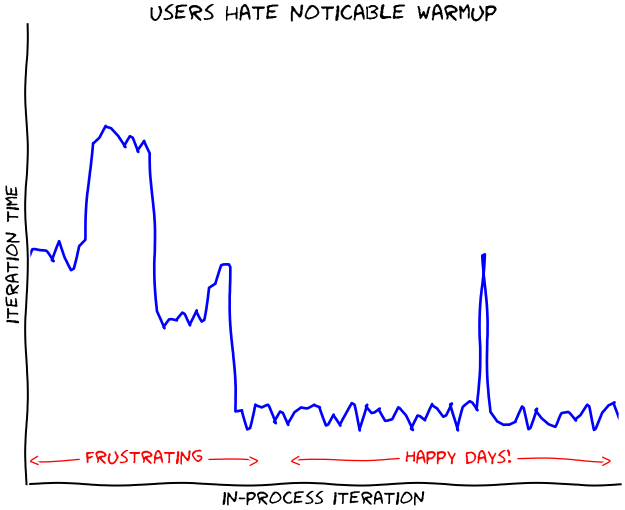


The Current State of the Art of Benchmarking

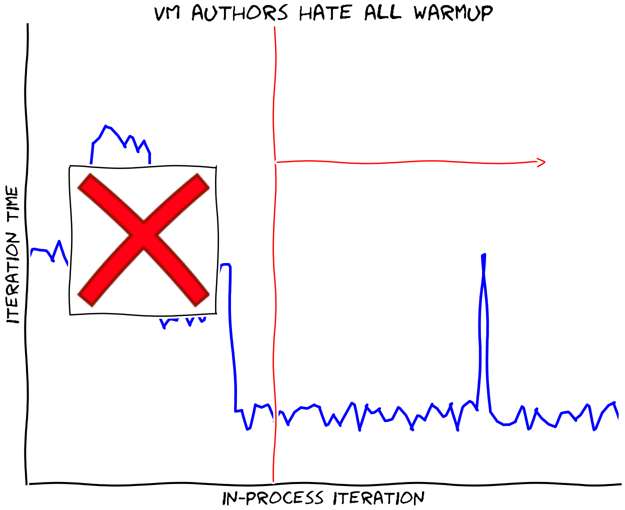



MORE REALISTIC VM WARMUP
ITERATION TIME
IN-PROCESS ITERATION
COMPILATION TIERS
GC SPIKES
SOME NOISE

Warmup Matters



Warmup Matters



We should measure the warmup of modern language implementations

The Warmup Experiment

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*

Method 2: How long to run?

2000 *in-process iterations*

30 *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_{4790}, Debian 8, 24GiB RAM
- $\text{Linux}_{\text{E3-1240v5}}$, Debian 8, 32GiB RAM
- OpenBSD_{4790}, OpenBSD 6.0, 32GiB RAM

Method 4: Machines

- Linux_{4790}, Debian 8, 24GiB RAM
- $\text{Linux}_{\text{E3-1240v5}}$, Debian 8, 32GiB RAM
- OpenBSD_{4790}, OpenBSD 6.0, 32GiB RAM

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

Method 5: Benchmark Harness

KRUN

Control as many confounding variables as possible

Method 5: Benchmark Harness

KRUN

Control as many confounding variables as possible

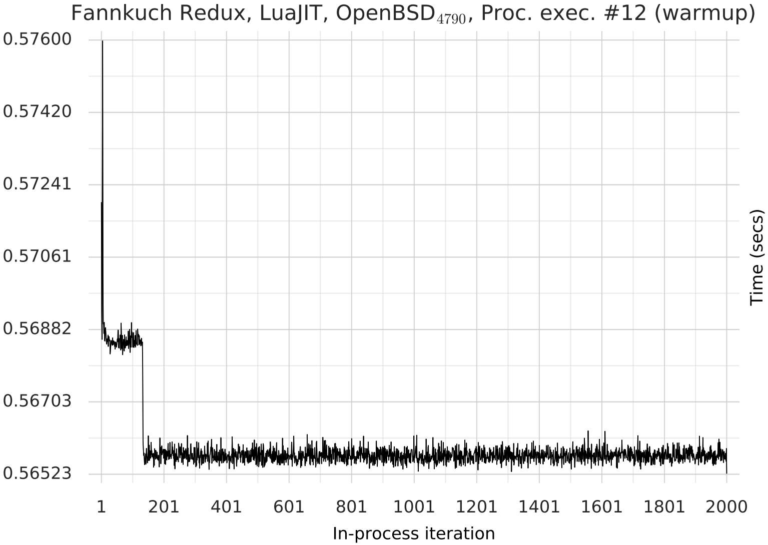
- Minimises I/O
- Sets fixed heap and stack ulimits
- Drops privileges to a ‘clean’ user account
- Automatically reboots the system prior to each proc. exec
- Checks `dmesg` for changes after each proc. exec
- Checks system at (roughly) same temperature for proc. execs
- Enforces kernel settings (tickless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

Let’s look at some plots for our results.

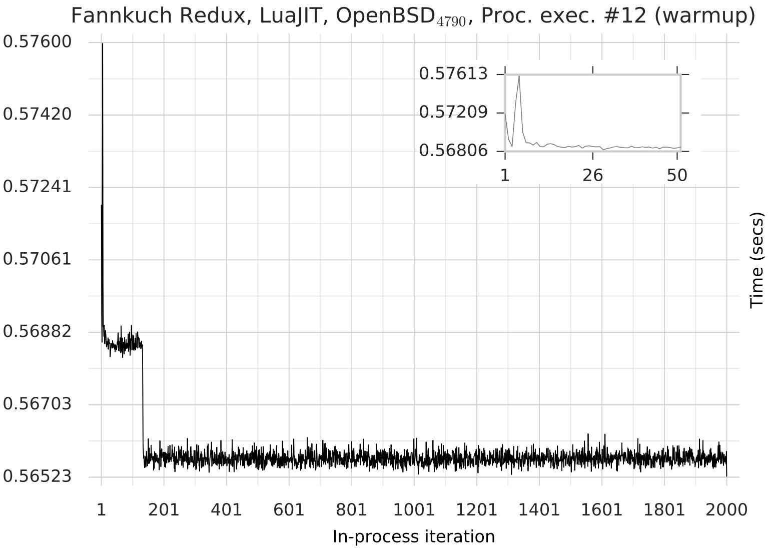
We also classify each process execution.

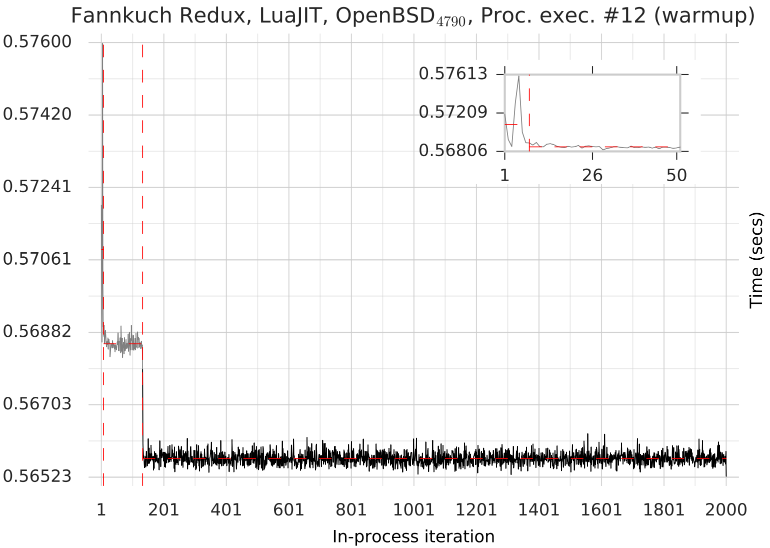
Classification uses *changepoint analysis*.

Changepoint Analysis 101

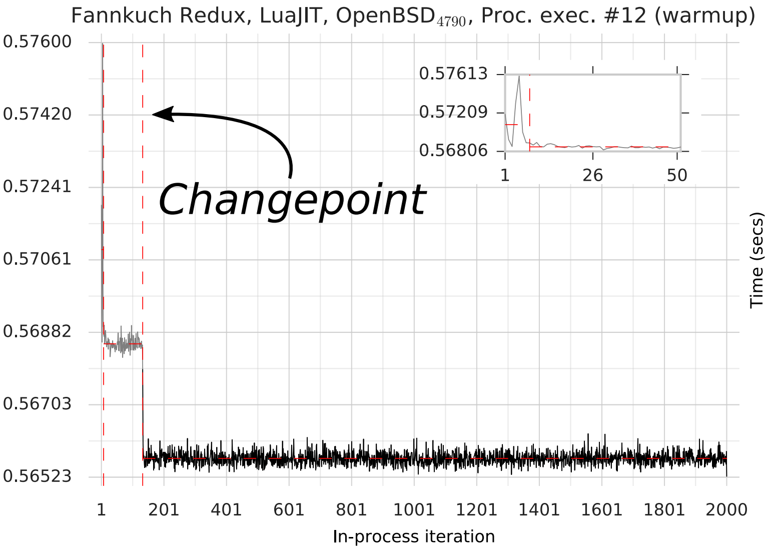


Changepoint Analysis 101

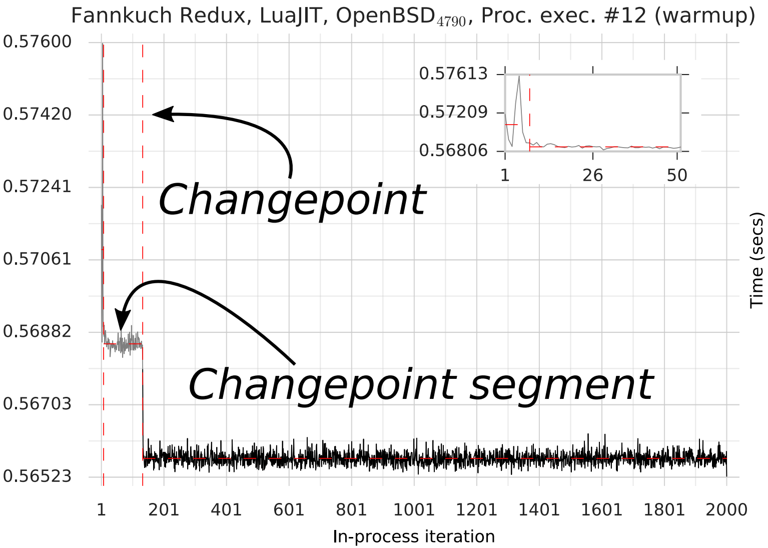



Fannkuch Redux, LuaJIT, OpenBSD$_{4790}$, Proc. exec. #12 (warmup)
0.57600
0.57420
0.57241
0.57061
0.56882
0.56703
0.56523
0.57613
0.57209
0.56806
1
26
50
Time (secs)
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration

Changepoint Analysis 101

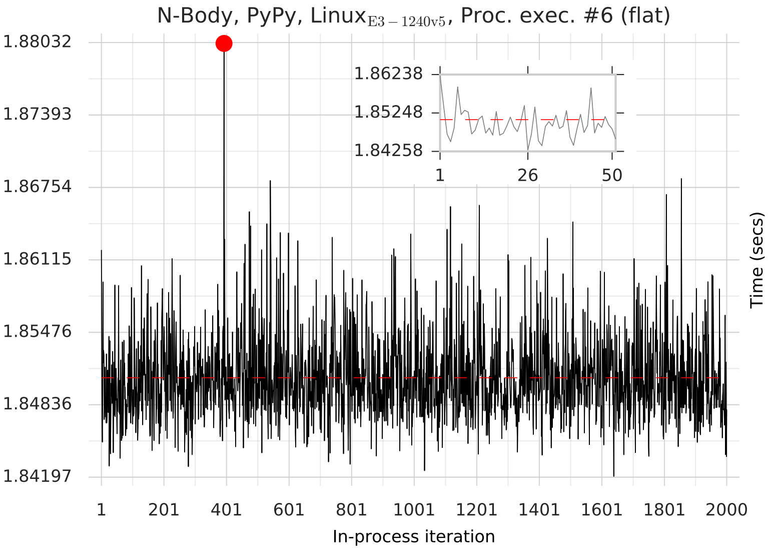


Changepoint Analysis 101



- All segs are equivalent:
  FLAT

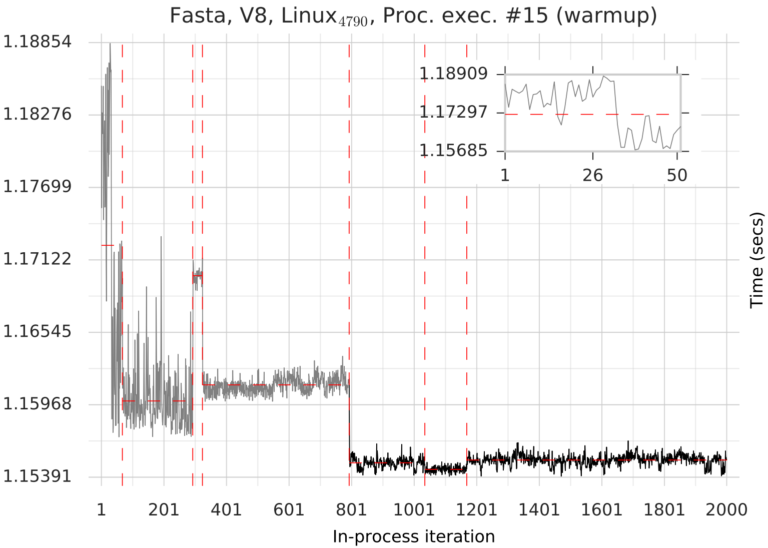
Results and Classification: Flat



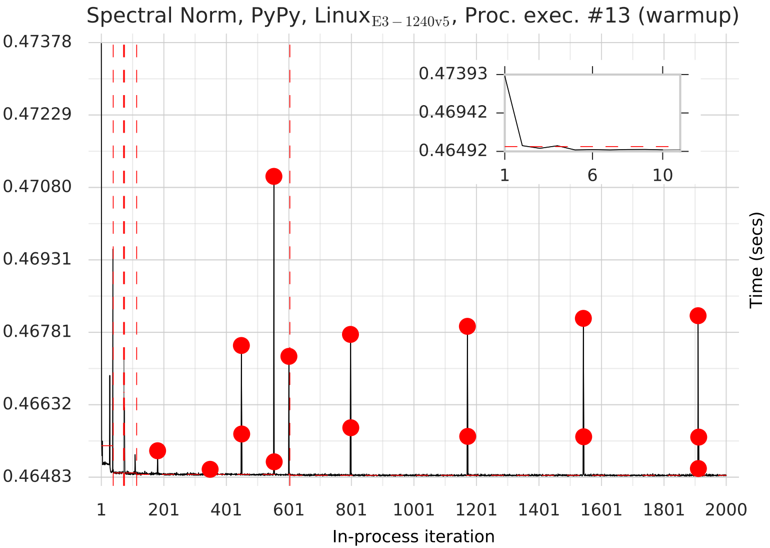
Results and Classification

- All segs are equivalent:
  FLAT
- Final equivalent segs >= 500 iters:
  - Final seg is equivalent to the fastest seg:
    WARMUP

Results and Classification: Warmup

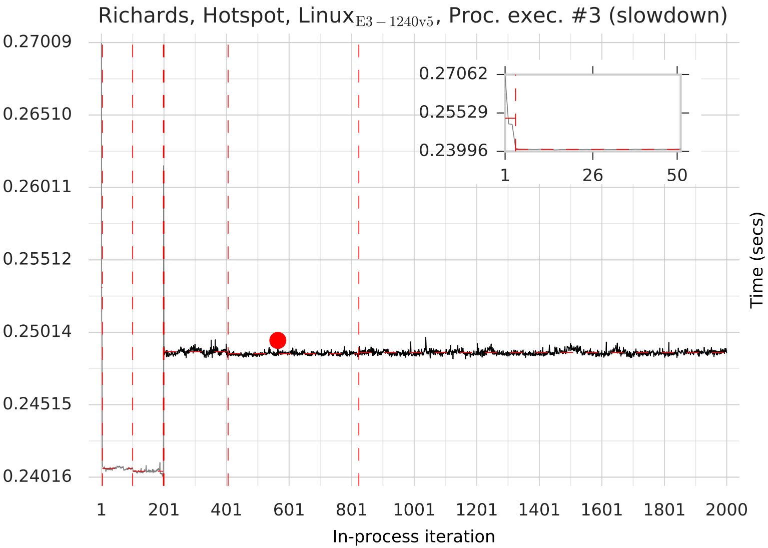


Results and Classification: Warmup

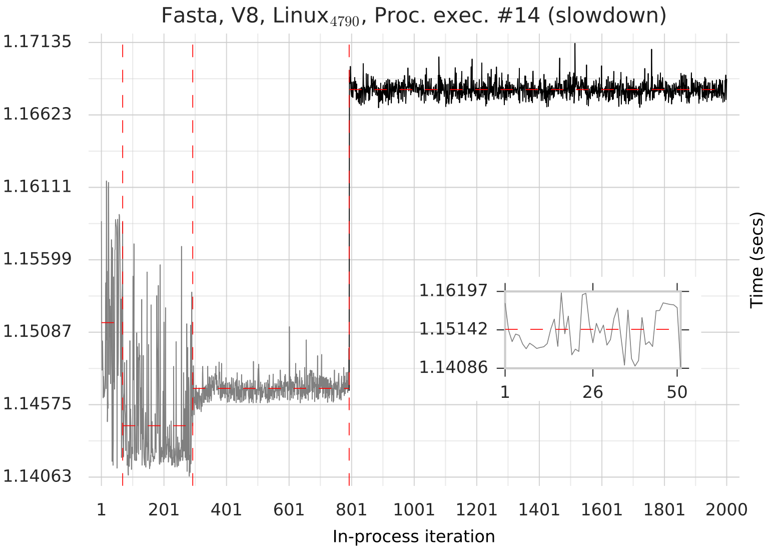


- All segs are equivalent:
  FLAT
- Final equivalent segs >= 500 iters:
  - Final seg is equivalent to the fastest seg:
    WARMUP
  - Final seg is not equivalent to the fastest seg:
    SLOWDOWN

Results and Slowdown



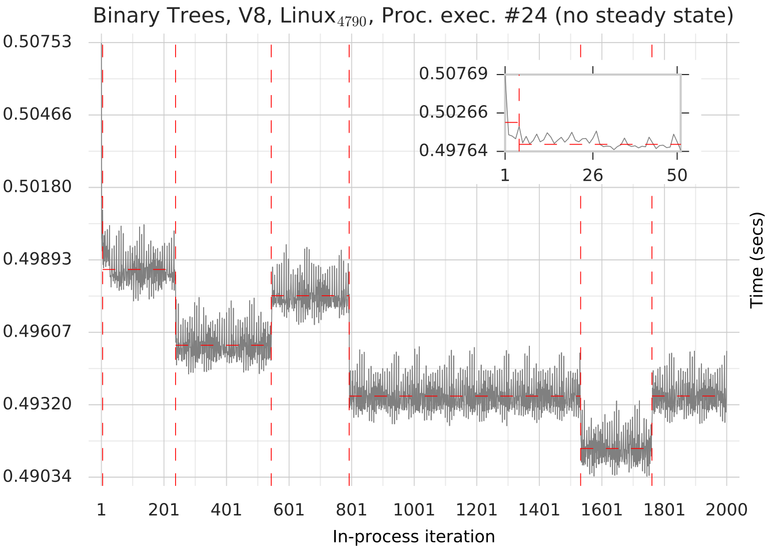
Results and Slowdown



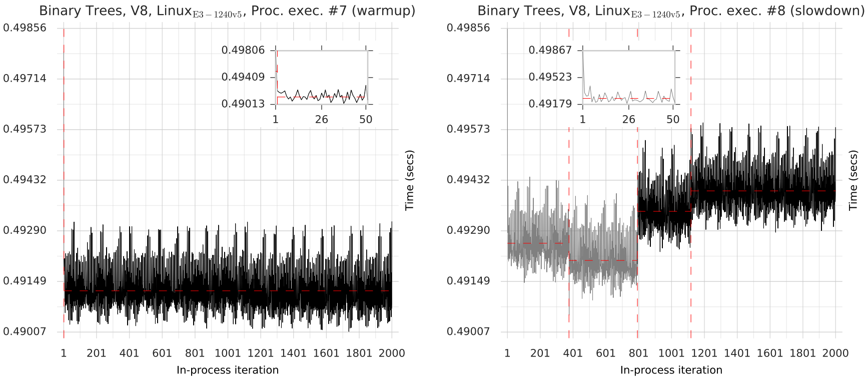
Results and Classification

- 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

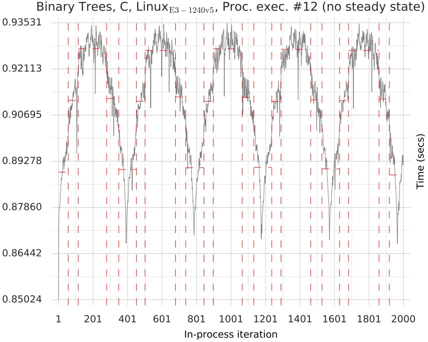


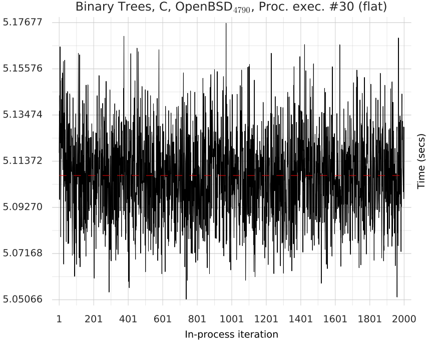
Inconsistent Process-executions



(Same machine)

Inconsistent Process-executions





(Different machines. Bouncing ball Linux-specific)

Quantitative Results

	Benchmark	Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)	Benchmark	Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*binary trees*	⏦				*n-body*	–			0.40555 ±0.005110
Graal		≍ (27⅂, 3ʃ)	32.0 (17.0, 193.8)	6.60 (3.729, 36.608)	0.18594 ±0.000315		⅂	8.0 (7.5, 8.5)	1.22 (1.126, 1.358)	0.13334 ±0.000045
HHVM		≍ (24⅂, 4ʃ, 2⏦)					≍ (16⅂, 11ʃ, 3⏦)			
HotSpot		≍ (25⅂, 5ʃ)	7.0 (7.0, 53.5)	1.19 (1.182, 9.703)	0.18279 ±0.000116		⅂	2.0 (2.0, 2.0)	0.14 (0.141, 0.143)	0.13699 ±0.000032
JRuby+Truffle		ʃ	1082.0 (999.0, 1232.5)	2219.59 (2039.304, 2516.021)	2.05150 ±0.017738		⅂	69.0 (69.0, 70.0)	17.95 (17.716, 18.127)	0.20644 ±0.001568
LuaJIT		≍ (23⅂, 4ʃ, 2–, 1⏦)					–			0.25399 ±0.004471
PyPy		≍ (27ʃ, 3⏦)					–			1.85835 ±0.012893
V8		≍ (15–, 9⅂, 6ʃ)	1.5 (1.0, 794.0)	0.25 (0.000, 391.026)	0.49237 ±0.003198		= (25–, 5⅂)	1.0 (1.0, 361.6)	0.00 (0.000, 87.367)	0.24138 ±0.000389
C	*fannkuch redux*	≍ (21–, 6⅂, 2ʃ, 1⏦)				*Richards*	≍ (19–, 5ʃ, 4⏦, 2⅂)			
Graal		≍ (28⅂, 1⏦, 1ʃ)					≍ (28⅂, 1⏦, 1ʃ)			
HHVM		⅂	10.0 (10.0, 10.0)	52.66 (52.660, 52.708)	1.35779 ±0.011948					
HotSpot		⅂	390.0 (2.0, 390.0)	153.70 (0.407, 155.254)	0.36202 ±0.022767		≍ (26ʃ, 2⏦, 2⅂)			
JRuby+Truffle		ʃ	1016.5 (999.0, 1023.1)	1039.04 (1014.290, 1059.967)	1.08833 ±0.038580		ʃ	1021.0 (1014.9, 1027.0)	917.30 (901.708, 946.683)	0.89509 ±0.027590
LuaJIT		–			0.56285 ±0.000097		≍ (21⏦, 7ʃ, 2⅂)			
PyPy		≍ (15⅂, 13–, 2ʃ)	2.0 (1.0, 28.9)	1.57 (0.000, 43.483)	1.55442 ±0.020549		⅂	2.0 (2.0, 5.0)	1.12 (1.114, 4.054)	0.96809 ±0.010950
V8		= (19⅂, 11–)	2.0 (1.0, 25.5)	0.31 (0.000, 7.525)	0.30401 ±0.000154		= (29⅂, 1–)	4.0 (4.0, 16.0)	1.44 (1.434, 7.135)	0.47421 ±0.001218
C	*fasta*	–			0.07048 ±0.000210	*spectralnorm*	≍ (28⅂, 1–, 1ʃ)	997.0 (2.0, 1001.0)	546.38 (0.546, 547.717)	0.54547 ±0.002562
Graal		≍ (29⅂, 1⏦)					≍ (29ʃ, 1–)	15.0 (2.0, 21.0)	12.40 (0.812, 17.804)	0.89293 ±0.000087
HHVM		≍ (27⅂, 2⏦, 1ʃ)					⅂	35.0 (34.0, 41.0)	139.18 (136.909, 147.626)	1.40690 ±0.011133
HotSpot		≍ (18⅂, 12ʃ)	261.0 (6.0, 595.0)	30.73 (0.614, 70.021)	0.11744 ±0.001723		⅂	7.0 (7.0, 8.6)	1.90 (1.901, 2.425)	0.31470 ±0.000029
JRuby+Truffle							⅂	1011.0 (1007.5, 1014.0)	893.38 (888.985, 896.562)	0.83633 ±0.014403
LuaJIT		⏦					–			0.22435 ±0.000020
PyPy		⏦					⅂	75.0 (75.0, 75.0)	34.43 (34.429, 34.437)	0.46489 ±0.000046
V8		≍ (19⅂, 10ʃ, 1⏦)					⅂	3.0 (3.0, 3.0)	0.55 (0.554, 0.554)	0.24963 ±0.000039

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*binary trees*	ʍ			
Graal		⤫ (27⅂, 3⅃)	32.0 (17.0, 193.8)	6.60 (3.729, 36.608)	0.18594 ±0.000316
HHVM		⤫ (24⅂, 4⅃, 2ʍ)			
HotSpot		⤫ (25⅂, 5⅃)	7.0 (7.0, 53.5)	1.19 (1.182, 9.703)	0.18279 ±0.000116
JRuby+Truffle		⅃	1082.0 (999.0, 1232.5)	2219.59 (2039.304, 2516.021)	2.05150 ±0.017737
LuaJIT		⤫ (23⅂, 4⅃, 2—, 1ʍ)			
PyPy		⤫ (27⅃, 3ʍ)			
V8		⤫ (15—, 9⅂, 6⅃)	1.5 (1.0, 794.0)	0.25 (0.000, 391.026)	0.49237 ±0.003198

Results: Summary

Class.	Linux$_{4790}$	Linux$_{1240v5}$	OpenBSD$_{4790}$
	⟨VM, benchmark⟩ pairs		
—	8.7%	13.0%	6.7%
ʅ	28.3%	23.9%	10.0%
ʃ	6.5%	6.5%	0.0%
⋀⋁	4.3%	6.5%	0.0%
=	6.5%	6.5%	13.3%
⨯	45.7%	43.5%	70.0%
	Process executions		
—	26.4%	20.9%	34.0%
ʅ	48.3%	51.5%	52.1%
ʃ	16.7%	17.9%	11.1%
⋀⋁	8.7%	9.6%	2.8%

Results: Summary

Class.	Linux_{4790}	Linux_{1240v5}	OpenBSD_{4790}
	⟨VM, benchmark⟩ pairs		
—	8.7%	13.0%	6.7%
⌊	28.3%	23.9%	10.0%
⌈	6.5%	6.5%	0.0%
∿	4.3%	6.5%	0.0%
=	6.5%	6.5%	13.3%
⨯	45.7%	43.5%	70.0%
	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%

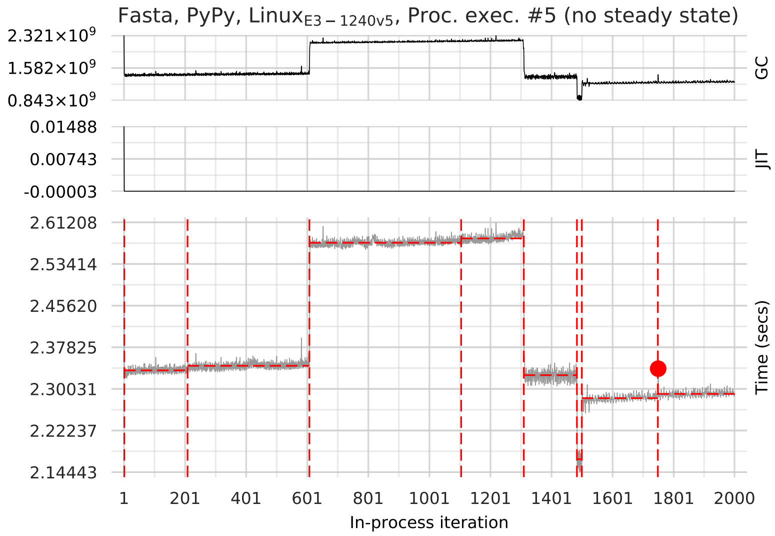
Best case no. good process executions: 86.1%

Results: Summary

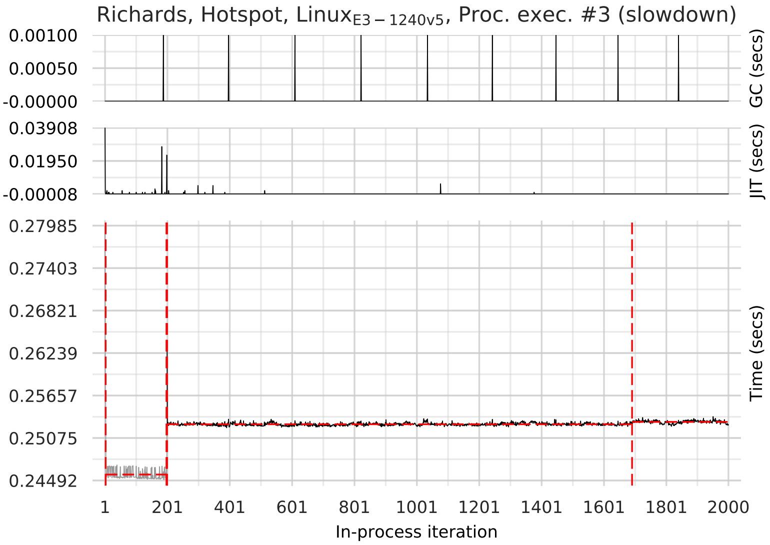
Class.	Linux_{4790}	Linux_{1240v5}	OpenBSD_{4790}
	⟨VM, benchmark⟩ pairs		
—	8.7%	13.0%	6.7%
˥	28.3%	23.9%	10.0%
ʃ	6.5%	6.5%	0.0%
ʌʌ	4.3%	6.5%	0.0%
=	6.5%	6.5%	13.3%
✕	45.7%	43.5%	70.0%
	Process executions		
—	26.4%	20.9%	34.0%
˥	48.3%	51.5%	52.1%
ʃ	16.7%	17.9%	11.1%
ʌʌ	8.7%	9.6%	2.8%

Best case good ⟨VM, benchmark⟩ pairings: 43.5%

Are the effects due to JIT and GC?



Are the effects due to JIT and GC?



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?

What Can We Do?

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

Thanks for Listening