Why VM Benchmarking is Probably Misleading You

Curry On 2018, Amsterdam


Edd Barrett


Carl Friedrich Bolz-Tereick


Rebecca Killick (Lancaster)


Sarah Mount


Laurence Tratt

King's College London

Software Development Team

2018-07-17

Context

Virtual Machine Warmup Blows Hot and Cold*

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

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 those parts 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**; *Interpreters*;

Additional Key Words and Phrases: Virtual machine, JIT, benchmarking, performance

ACM Reference Format:
Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Draft* 0, 0, Article 0 (October 2017), 40 pages. https://arxiv.org/abs/1602.00602v6

Benchmarking Concepts

Benchmarking Concepts

- The lifetime of a VM invocation

- The lifetime of a VM invocation
  - *“process execution”*

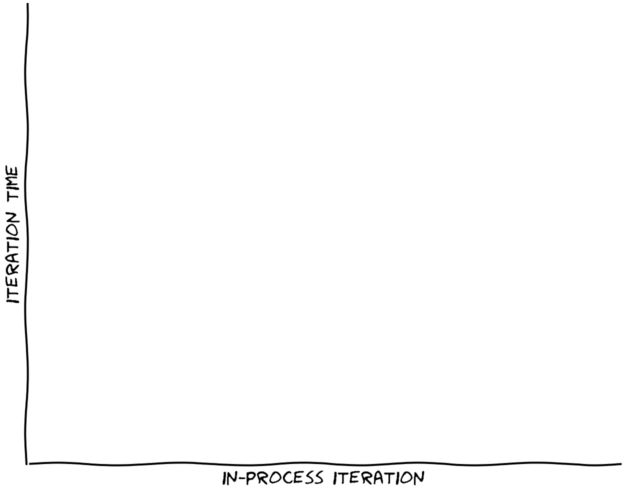
Benchmarking Concepts

- The lifetime of a VM invocation
  - *“process execution”*
- Run benchmark in a loop

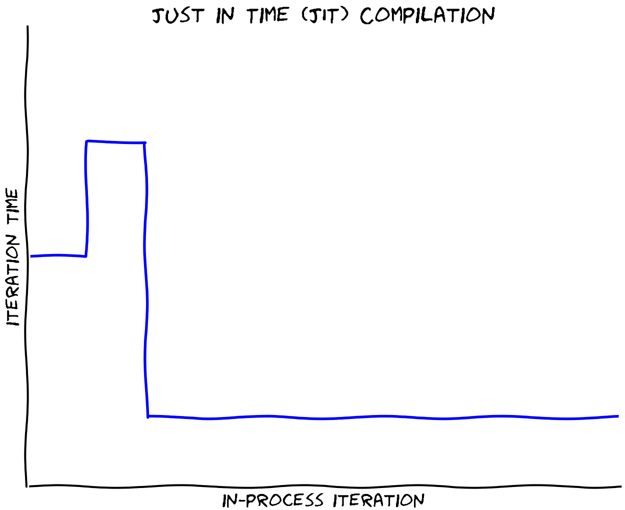
Benchmarking Concepts

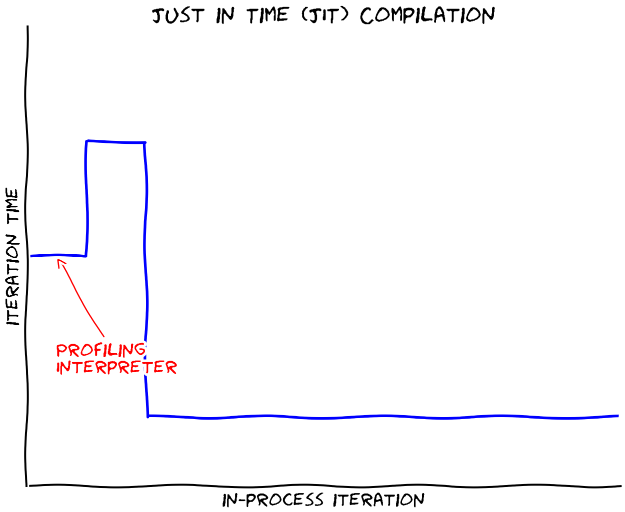
- The lifetime of a VM invocation
  - *"process execution"*
- Run benchmark in a loop
  - *"in-process iterations"*

The Current State of the Art of Benchmarking

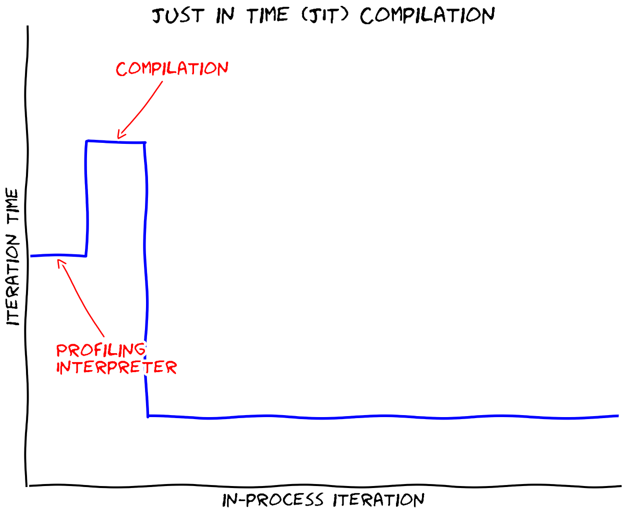


The Current State of the Art of Benchmarking

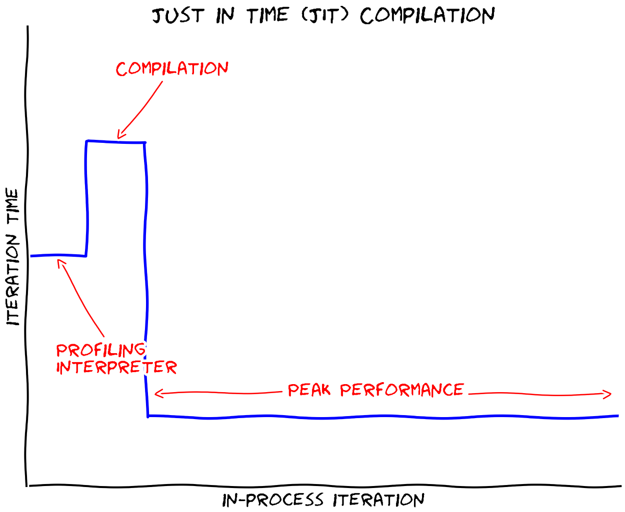


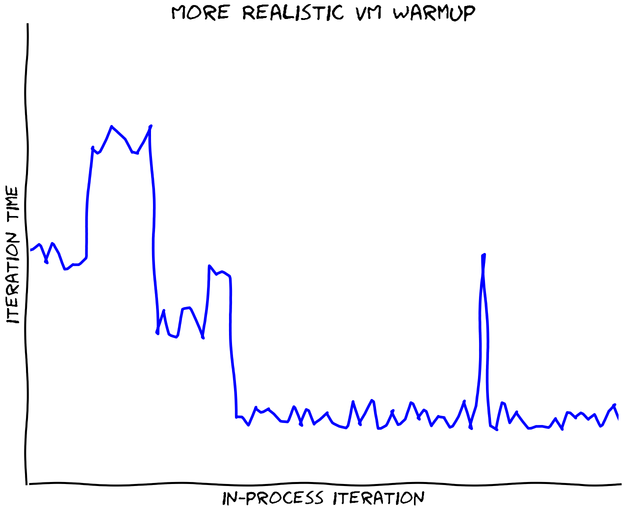

JUST IN TIME (JIT) COMPILATION
ITERATION TIME
PROFILING INTERPRETER
IN-PROCESS ITERATION

The Current State of the Art of Benchmarking

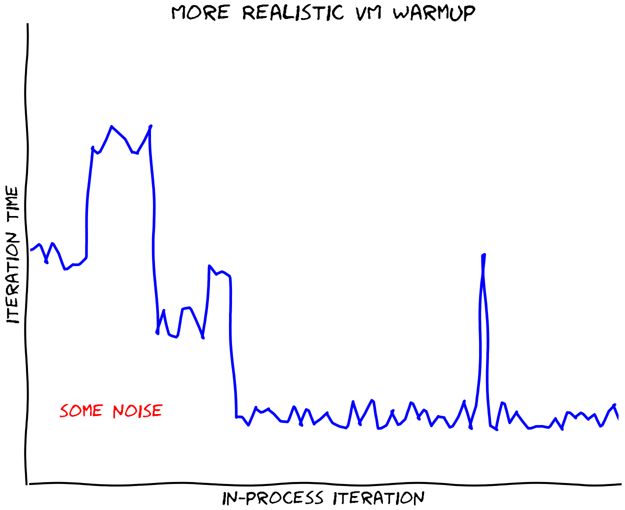


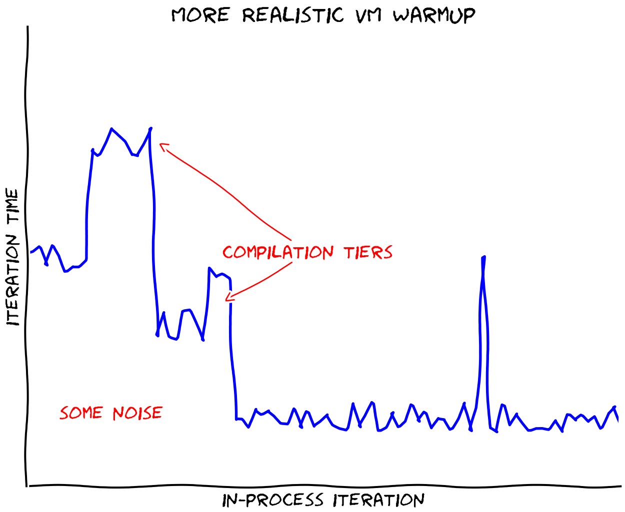
The Current State of the Art of Benchmarking

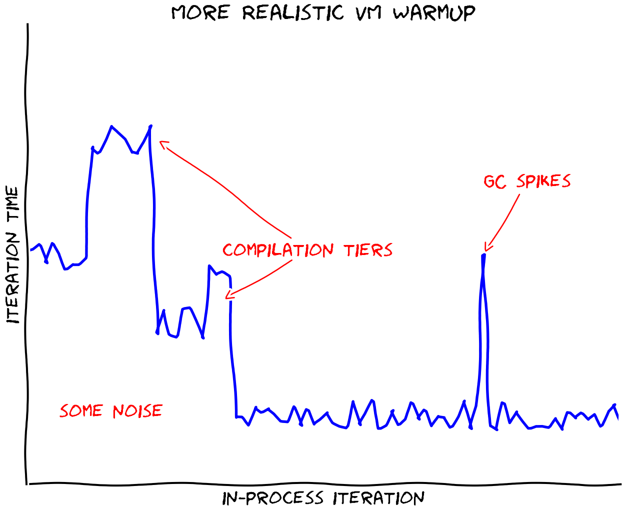



MORE REALISTIC VM WARMUP
ITERATION TIME
IN-PROCESS ITERATION

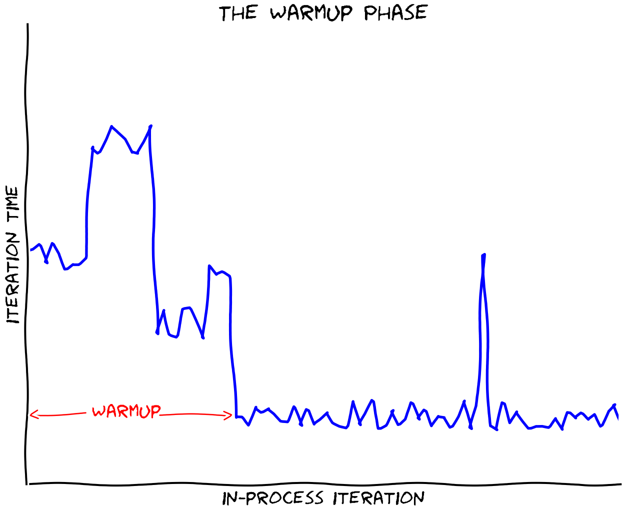
The Current State of the Art of Benchmarking




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


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

The Current State of the Art of Benchmarking



Warmup Matters

Warmup Matters

1. Users dislike poor warmup.

Warmup Matters

1. Users dislike poor warmup.

2. VM authors dislike poor warmup.

Warmup Matters

1. Users dislike poor warmup.

2. VM authors dislike poor warmup.

Warmup is important!

The Warmup Experiment

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?

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?

Method 2: How Long to Run?

2000 *in-process iterations*

30 *process executions*

Method 3: VMs

- Graal-0.22
- HHVM-3.19.1
- TruffleRuby-20170502
- 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 off.
- Hyper-threading off.

Method 5: Benchmark Harness

KRUN

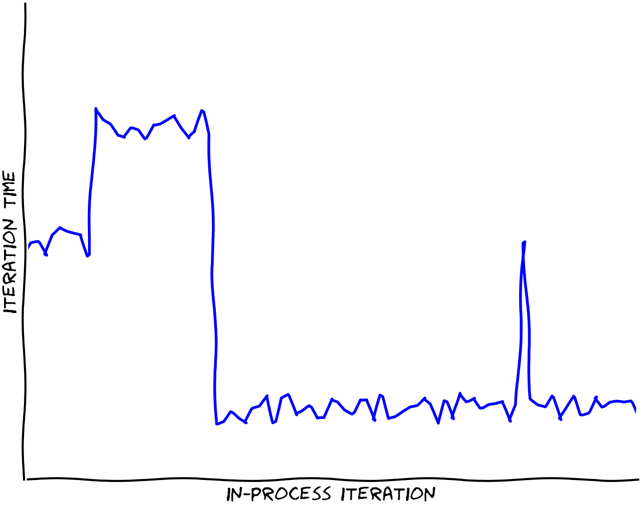
Control as many confounding variables as possible

KRUN

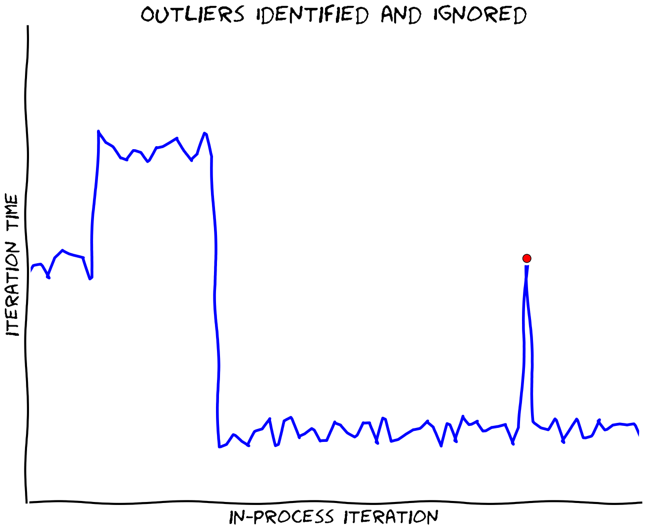
Control as many confounding variables as possible

- 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, ...)

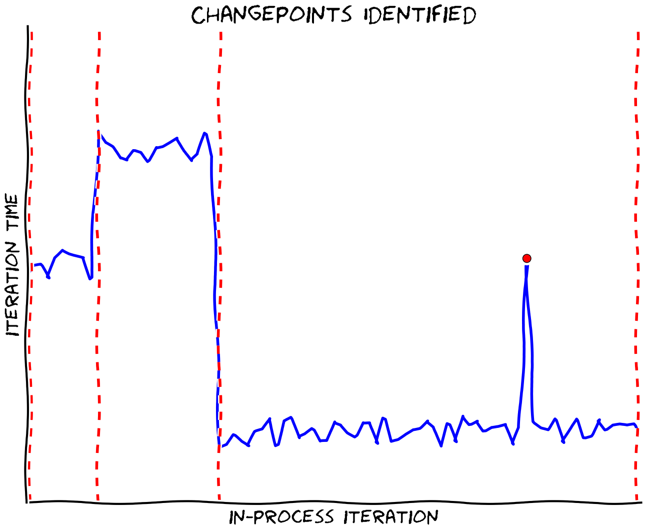
Method 6: Changepoint Analysis and Classification



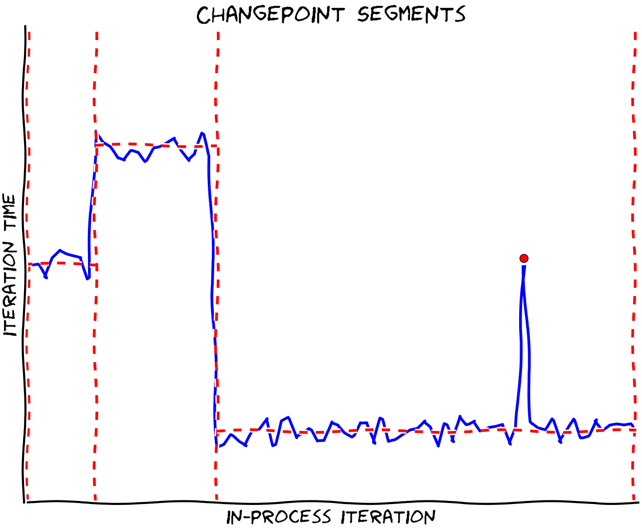
Method 6: Changepoint Analysis and Classification



Method 6: Changepoint Analysis and Classification



Method 6: Changepoint Analysis and Classification



- One changepoint segment:
  FLAT ($-$)

Method 6: Changepoint Analysis and Classification

- One changepoint segment:
  FLAT (—)
- No changepoint after 1500 in-process iterations:
  - The last seg is the fastest:
    WARMUP (⅂)

Method 6: Changepoint Analysis and Classification

- One changepoint segment:
  FLAT (—)
- No changepoint after 1500 in-process iterations:
  - The last seg is the fastest:
    WARMUP (¯⌊)
  - not the fastest seg:
    SLOWDOWN (_⌈)

Method 6: Changepoint Analysis and Classification

- One changepoint segment:
  FLAT (—)
- No changepoint after 1500 in-process iterations:
  - The last seg is the fastest:
    WARMUP ( )
  - not the fastest seg:
    SLOWDOWN ( )
- Otherwise:
  NO STEADY STATE ( )

Method 6: Changepoint Analysis and Classification

Very close segments are considered equivalent

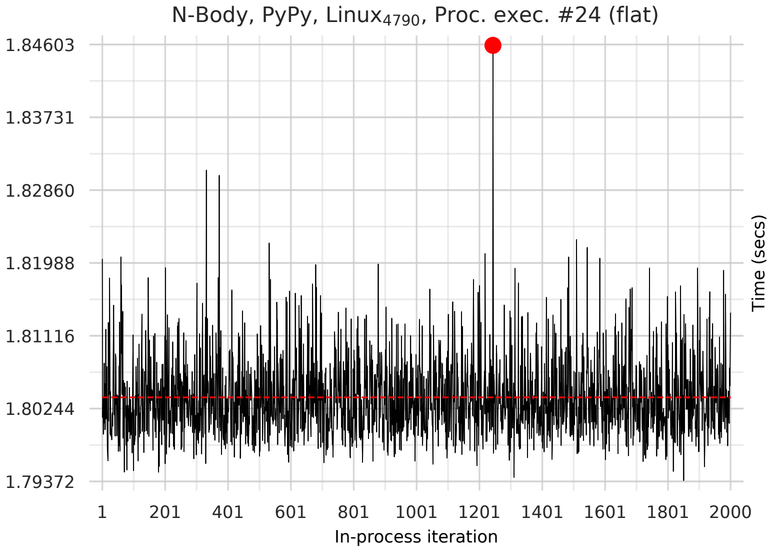
Results

Results

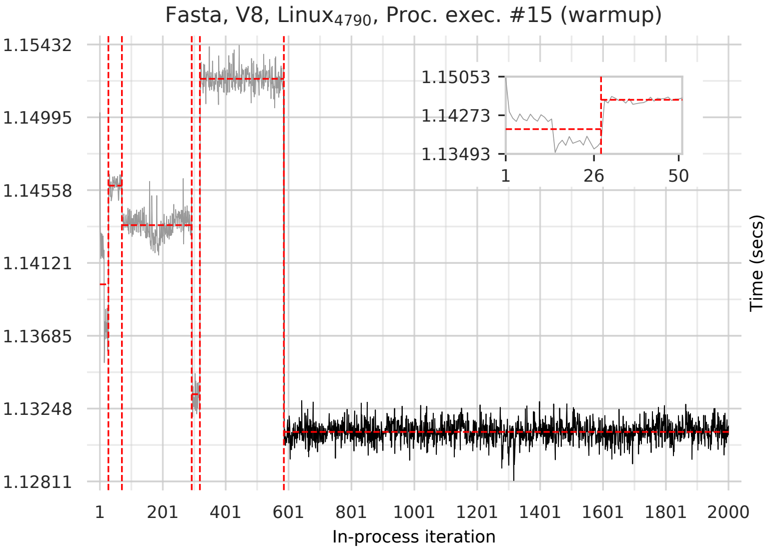
- The results I'm showing are from v1.5 of the data.
  `https://archive.org/download/softdev_warmup_experiment_artefacts/v1.5/`

- The OOPSLA paper uses v0.8 data.
  `https://archive.org/download/softdev_warmup_experiment_artefacts/v0.8/`

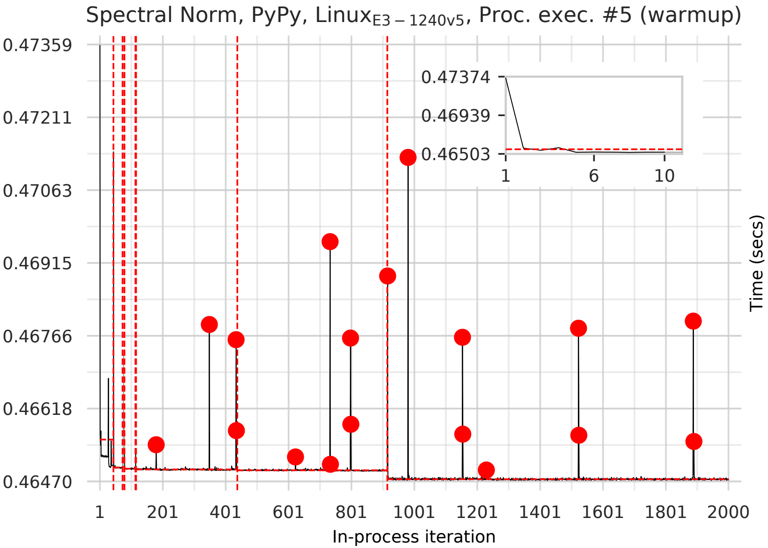
Results: Flat



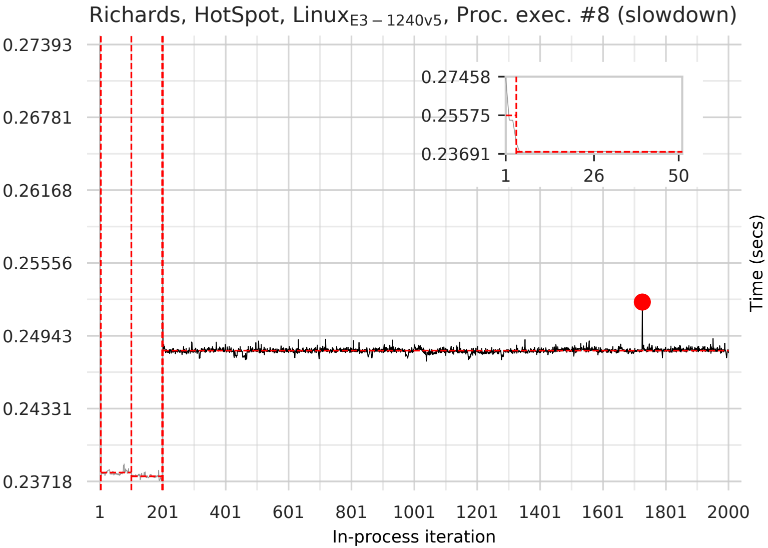
Results: Warmup



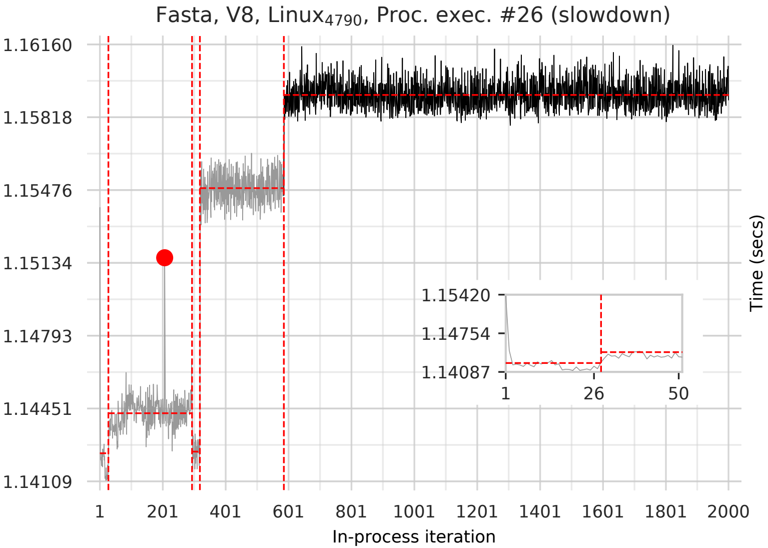
Results: Warmup



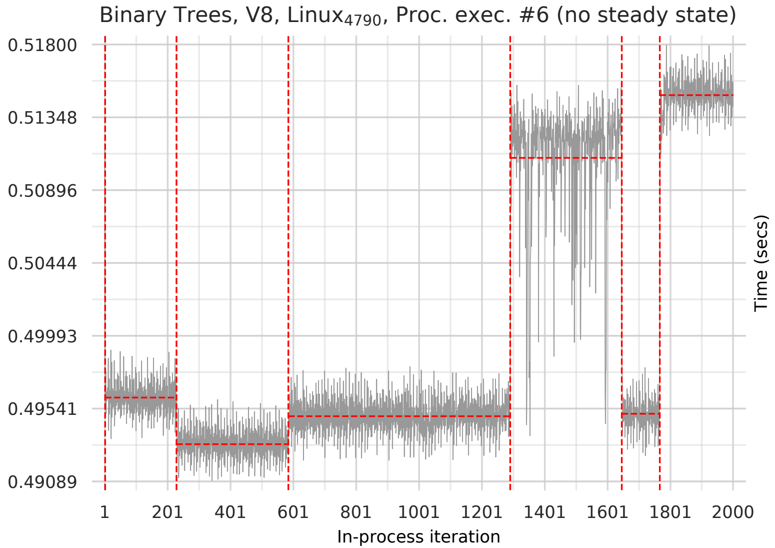
Results: Slowdown



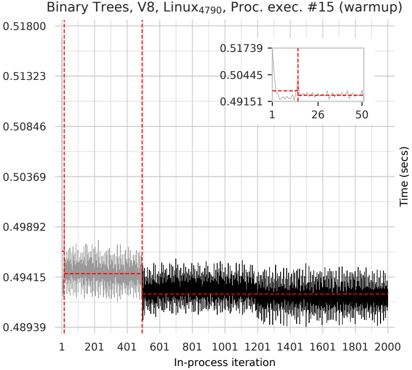
Results: Slowdown

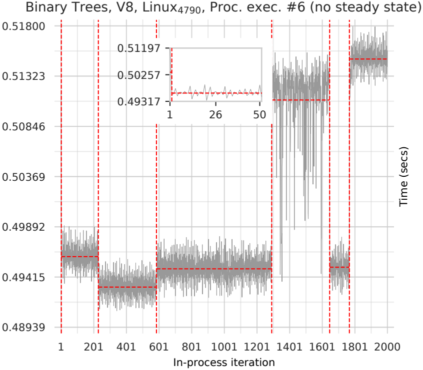


Results: No Steady State



Results: Inconsistent Process-executions





(Same machine)

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	*binary trees*	≍ (27⅂, 3⅃)	32.0 (17.0, 193.8)	6.60 (3.729, 36.608)	0.18594 ±0.000315	*n-body*	⅂	8.0 (7.5, 8.5)	1.22 (1.126, 1.358)	0.13334 ±0.000045
HHVM	*binary trees*	≍ (24⅂, 4⅃, 2~)				*n-body*	≍ (16⅂, 11⅃, 3~)			
HotSpot	*binary trees*	≍ (25⅂, 5⅃)	7.0 (7.0, 53.5)	1.19 (1.182, 9.703)	0.18279 ±0.000116	*n-body*	⅂	2.0 (2.0, 2.0)	0.14 (0.141, 0.143)	0.13699 ±0.000032
JRuby+Truffle	*binary trees*	⅃	1082.0 (999.0, 1232.5)	2219.59 (2039.304, 2516.021)	2.05150 ±0.017738	*n-body*	⅂	69.0 (69.0, 70.0)	17.95 (17.716, 18.127)	0.20644 ±0.001568
LuaJIT	*binary trees*	≍ (23⅂, 4⅃, 2—, 1~)				*n-body*	—			0.25399 ±0.004471
PyPy	*binary trees*	≍ (27⅃, 3~)				*n-body*	—			1.85835 ±0.012893
V8	*binary trees*	≍ (15—, 9⅂, 6⅃)	1.5 (1.0, 794.0)	0.25 (0.000, 391.026)	0.49237 ±0.003198	*n-body*	= (25—, 5⅂)	1.0 (1.0, 361.6)	0.00 (0.000, 87.367)	0.24138 ±0.000389
C	*fannkuch redux*	≍ (21—, 6⅂, 2⅃, 1~)				*Richards*	≍ (19—, 5⅃, 4~, 2⅂)			
Graal	*fannkuch redux*	≍ (28⅂, 1~, 1⅃)				*Richards*	≍ (28⅂, 1~, 1⅃)			
HHVM	*fannkuch redux*	⅂	10.0 (10.0, 10.0)	52.66 (52.660, 52.708)	1.35779 ±0.011948	*Richards*				
HotSpot	*fannkuch redux*	⅂	390.0 (2.0, 390.0)	153.70 (0.407, 155.254)	0.36202 ±0.022767	*Richards*	≍ (26⅃, 2~, 2⅂)			
JRuby+Truffle	*fannkuch redux*	⅃	1016.5 (999.0, 1023.1)	1039.04 (1014.290, 1059.967)	1.08833 ±0.038580	*Richards*	⅃	1021.0 (1014.9, 1027.0)	917.30 (901.708, 946.683)	0.89509 ±0.027590
LuaJIT	*fannkuch redux*	—			0.56285 ±0.000097	*Richards*	≍ (21~, 7⅃, 2⅂)			
PyPy	*fannkuch redux*	≍ (15⅂, 13—, 2⅃)	2.0 (1.0, 28.9)	1.57 (0.000, 43.483)	1.55442 ±0.020549	*Richards*	⅂	2.0 (2.0, 5.0)	1.12 (1.114, 4.054)	0.96809 ±0.010950
V8	*fannkuch redux*	= (19⅂, 11—)	2.0 (1.0, 25.5)	0.31 (0.000, 7.525)	0.30401 ±0.000154	*Richards*	= (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—, 1⅃)	997.0 (2.0, 1001.0)	546.38 (0.546, 547.717)	0.54547 ±0.002562
Graal	*fasta*	≍ (29⅂, 1~)				*spectralnorm*	≍ (29⅃, 1—)	15.0 (2.0, 21.0)	12.40 (0.812, 17.804)	0.89293 ±0.000087
HHVM	*fasta*	≍ (27⅂, 2~, 1⅃)				*spectralnorm*	⅂	35.0 (34.0, 41.0)	139.18 (136.909, 147.626)	1.40690 ±0.011133
HotSpot	*fasta*	≍ (18⅂, 12⅃)	261.0 (6.0, 595.0)	30.73 (0.614, 70.021)	0.11744 ±0.001723	*spectralnorm*	⅂	7.0 (7.0, 8.6)	1.90 (1.901, 2.425)	0.31470 ±0.000029
JRuby+Truffle	*fasta*					*spectralnorm*	⅂	1011.0 (1007.5, 1014.0)	893.38 (888.985, 896.562)	0.83633 ±0.014403
LuaJIT	*fasta*	~				*spectralnorm*	—			0.22435 ±0.000020
PyPy	*fasta*	~				*spectralnorm*	⅂	75.0 (75.0, 75.0)	34.43 (34.429, 34.437)	0.46489 ±0.000046
V8	*fasta*	≍ (19⅂, 10⅃, 1~)				*spectralnorm*	⅂	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	*spectralnorm*	⨯ (27⅂, 2—, 1⅃)	775.0 (1.5,780.0)	425.16 (0.246,426.809)	0.54581 ±0.033116
Graal		⅃	14.0 (2.0,94.6)	13.60 (0.830,98.737)	1.05685 ±0.000126
HHVM		⨯ (29⅂, 1〰)			
HotSpot		⅂	7.0 (7.0,7.5)	1.91 (1.902,3.645)	0.31472 ±0.169143
LuaJIT		—			0.22181 ±0.000039
PyPy		= (27—, 3⅂)	1.0 (1.0,45.2)	0.00 (0.000,20.597)	0.46480 ±0.000085
TruffleRuby		⨯ (25⅃, 5〰)			
V8		⅂	3.0 (3.0,3.0)	0.52 (0.523,0.526)	0.25362 ±0.000034

Results: Summary

Class.	Linux_{4790}	Linux_{1240v5}	OpenBSD_{4790} †
	⟨VM, benchmark⟩ pairs		
—	8.9%	11.1%	13.3%
⌊	20.0%	17.8%	20.0%
⌈	4.4%	4.4%	3.3%
∿	4.4%	4.4%	0.0%
=	11.1%	8.9%	13.3%
≭	51.1%	53.3%	50.0%
	Process executions		
—	22.0%	23.3%	37.7%
⌊	48.3%	43.9%	35.2%
⌈	20.1%	22.1%	12.1%
∿	9.6%	10.8%	15.0%

Results: Summary

Class.	Linux$_{4790}$	Linux$_{1240v5}$	OpenBSD$_{4790}$ †
	⟨VM, benchmark⟩ pairs		
—	8.9%	11.1%	13.3%
⌊	20.0%	17.8%	20.0%
⌈	4.4%	4.4%	3.3%
∿	4.4%	4.4%	0.0%
=	11.1%	8.9%	13.3%
≠	51.1%	53.3%	50.0%
	Process executions		
—	22.0%	23.3%	37.7%
⌊	48.3%	43.9%	35.2%
⌈	20.1%	22.1%	12.1%
∿	9.6%	10.8%	15.0%

Summary

“Good” warmup occurs for only:

Summary

"Good" warmup occurs for only:

67.2%–70.3% of process executions

"Good" warmup occurs for only:

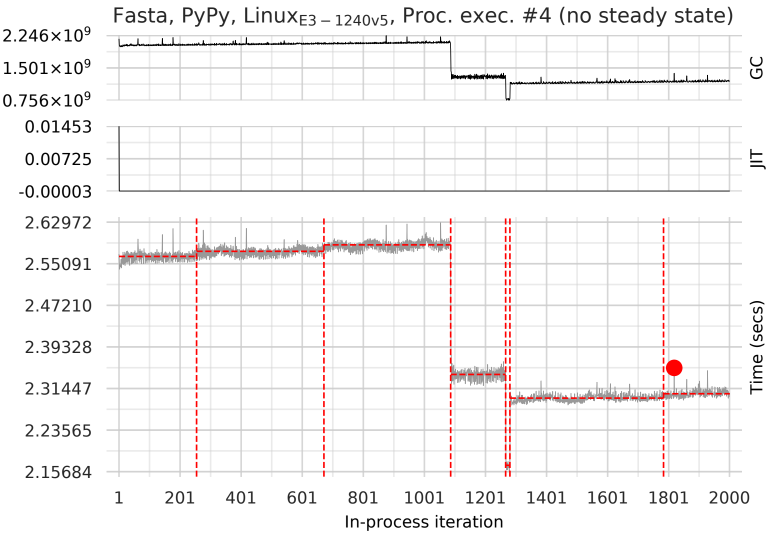
67.2%–70.3% of process executions

37.8%–40.0% of ⟨VM, benchmark⟩ pairs

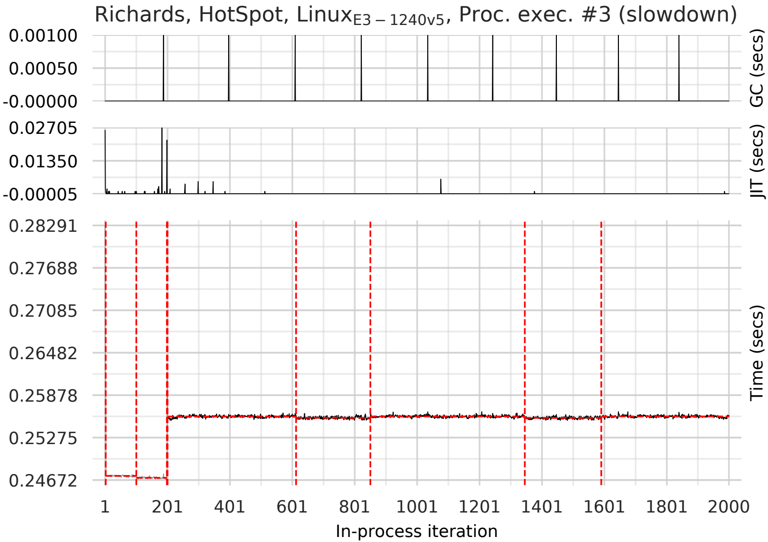
Hypothesis Invalid!

~~Hypothesis: Small, deterministic programs reach a steady state of peak performance.~~

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.
- Have we been misled?
  - 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. We can’t always blame GC or JIT compilation.

What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. We can’t always blame GC or JIT compilation.
4. 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. We can’t always blame GC or JIT compilation.
4. Always report warmup time.
5. Engineer VMs for predictable performance?

Links

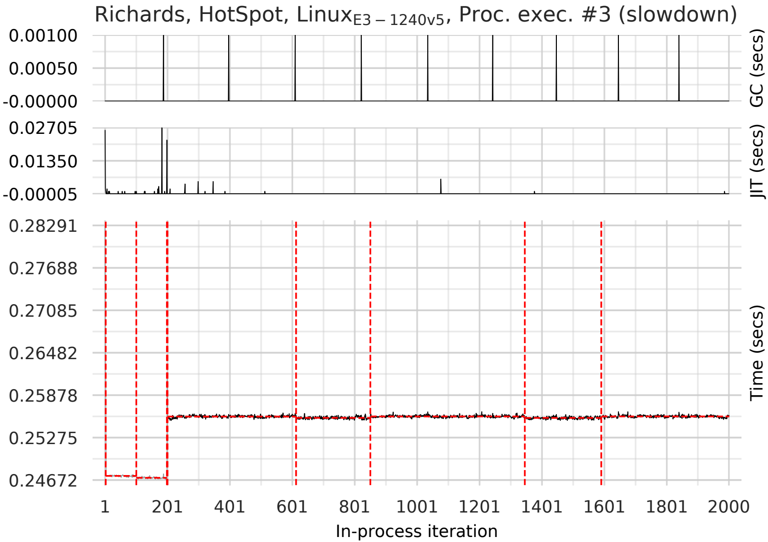
OOPSLA paper:
`https://arxiv.org/abs/1602.00602v6/`

Main experiment repo:
`https://github.com/softdevteam/warmup_experiment/`

Krun Benchmark Runner:
`https://github.com/softdevteam/krun/`

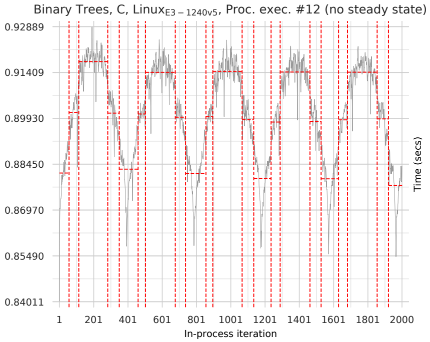
Archived data sets:
`https://archive.org/details/softdev_warmup_experiment_artefacts/`

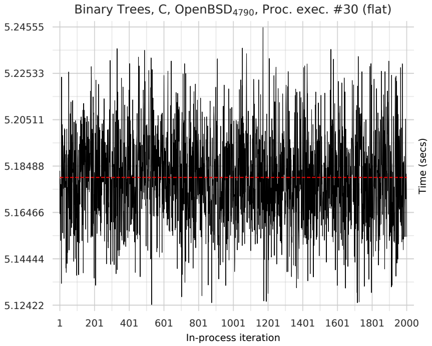
Thanks for Listening



Backup Slides

The Bouncing Ball Pattern





Method 7: Summary Statistics

For each machine, we summarise each $\langle$VM, Benchmark$\rangle$ pairing.

Method 7: Summary Statistics

For each machine, we summarise each ⟨VM, Benchmark⟩ pairing.

Consistent

All 30 process executions were classified the same.

Method 7: Summary Statistics

For each machine, we summarise each ⟨VM, Benchmark⟩ pairing.

Consistent

All 30 process executions were classified the same.

— ⎿ ⎾ ʌʌʌ

Inconsistent

A mix of classifications arose within the 30 process executions. E.g.:

- Good inconsistent: $=$ (25—, 5⎿)
- Bad inconsistent: $\neq$ (20—, 3⎾, 7ʌʌʌ)

Method 7: Summary Statistics

For each machine, we summarise each $\langle$VM, Benchmark$\rangle$ pairing.

Consistent

All 30 process executions were classified the same.

$-$ $\rceil\!\lfloor$ $\rfloor\!\lceil$ $\wedge\!\!\vee$

Inconsistent

A mix of classifications arose within the 30 process executions. E.g.:

- Good inconsistent: $=$ (25$-$, 5$\rceil\!\lfloor$)
- Bad inconsistent: $\not\asymp$ (20$-$, 3$\rfloor\!\lceil$, 7$\wedge\!\!\vee$)

If possible report: steady state performance, time until steady state, etc.