Why Aren't More Users More Happy With Our VMs?

Laurence
Tratt

Warmup work in collaboration with:
Edd Barrett, Carl Friedrich Bolz, Rebecca Killick, and Sarah Mount

KING'S College LONDON

Software Development Team

2017-11-21

What to expect from this talk

What to expect from this talk

The gap

What we tell users to expect

The gap

What we tell users to expect

The gap

What users experience

The gap

The gap
is bigger than we think

A stroppy user? Or rightfully disappointed?

"You told me I'd get
a 10x speed-up,
but I only saw 1.2x"

Background

Dynamic ('JIT') compilation utilises information not known statically

Is this just about 'dynamic typing'?

Is this statically or dynamically typed?

```
fn f(a: Option<i64>) {
  match a {
    Some(i) => ...,
    None    => ...
  };
}
```

Is this just about ‘dynamic typing’?

Is this statically or dynamically typed? Both!

```
fn f(a: Option<i64>) {
  match a {
    Some(i) => ...,
    None    => ...
  };
}
```

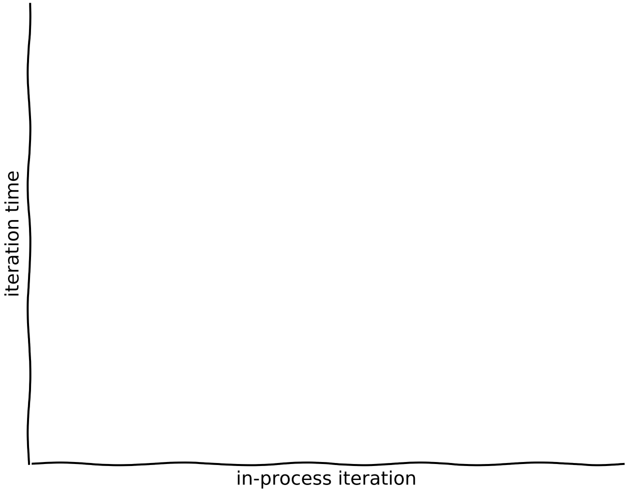
The best VMs are close in performance to,
and sometimes faster than,
gcc -O2

VM claims

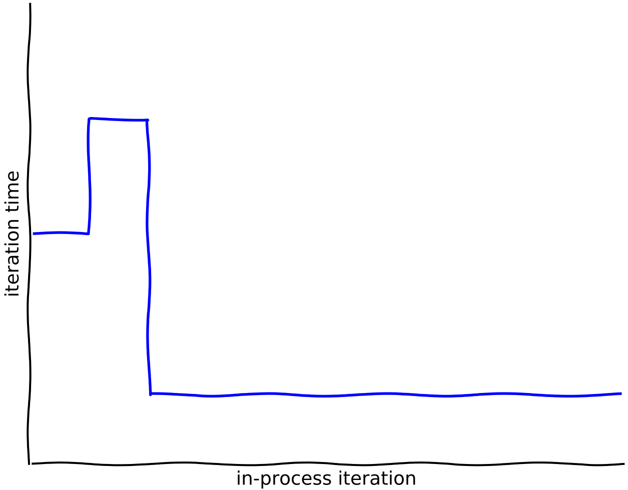
The best VMs are close in performance to,
and sometimes faster than,
gcc -O2

What’s being measured?

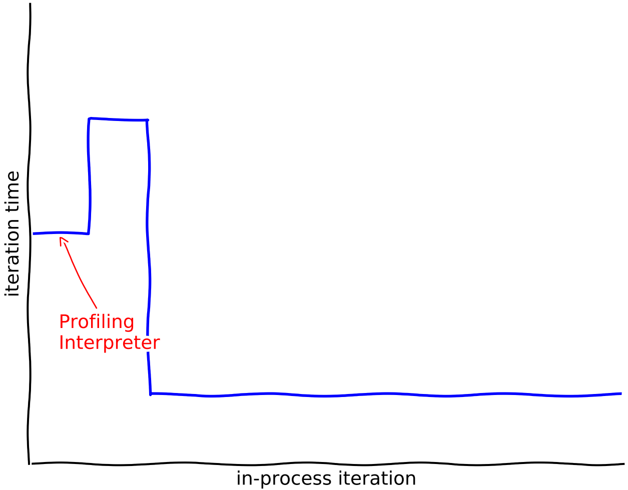
What our claims pertain to



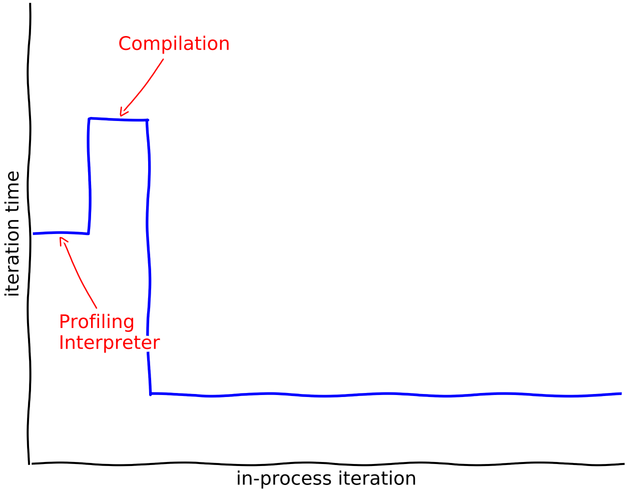
What our claims pertain to



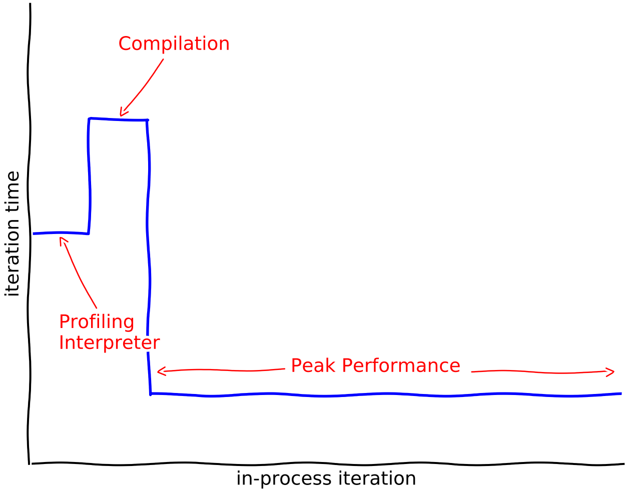
What our claims pertain to



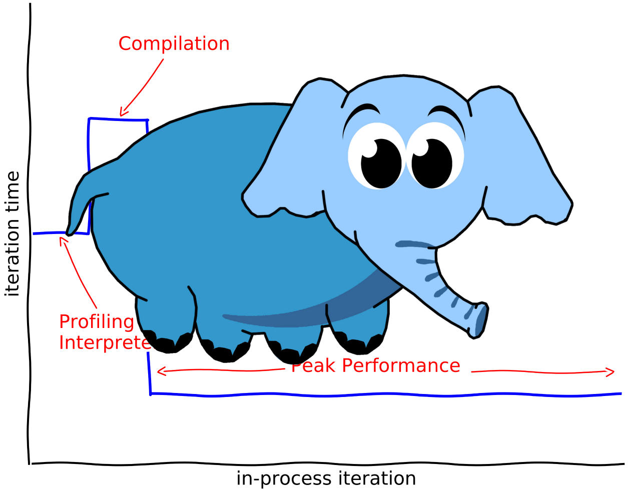
What our claims pertain to

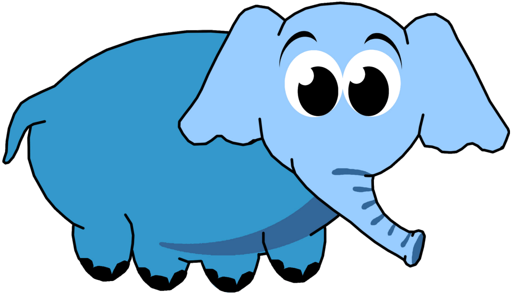


What our claims pertain to



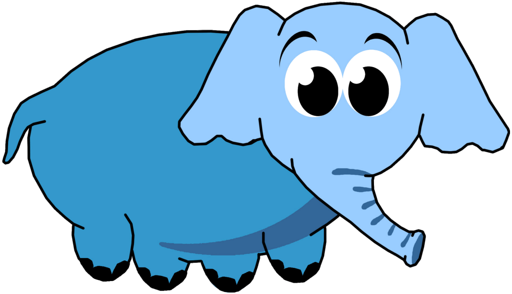
What our claims pertain to





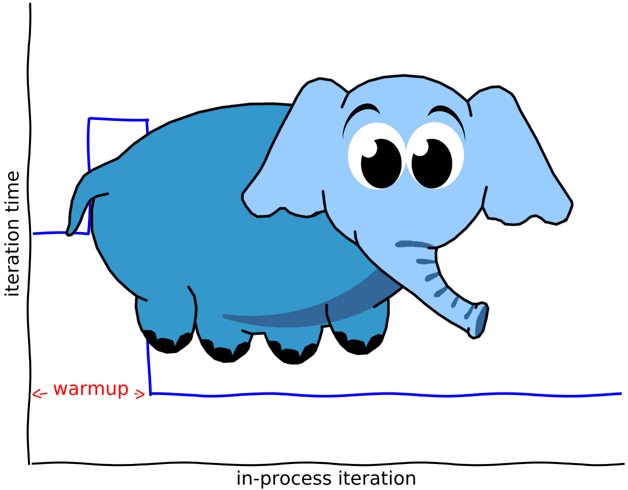
This is Barry

What our claims pertain to



This is Barry: the benchmarking elephant in the room

What our claims pertain to



Warmup

Users *always* perceive warmup

Users *always* perceive warmup

Maybe we should know how long it is?

The Warmup Experiment

Measure warmup of modern language implementations

The Warmup Experiment

Measure warmup of modern language implementations

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

Method 1: Which benchmarks?

The language benchmark games are perfect for us (unusually)

Method 1: Which benchmarks?

The language benchmark games are perfect for us (unusually)

We removed any CFG non-determinism

Method 1: Which benchmarks?

The language benchmark games are perfect for us (unusually)

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 (cron, smtpd)

Method 5: Krun

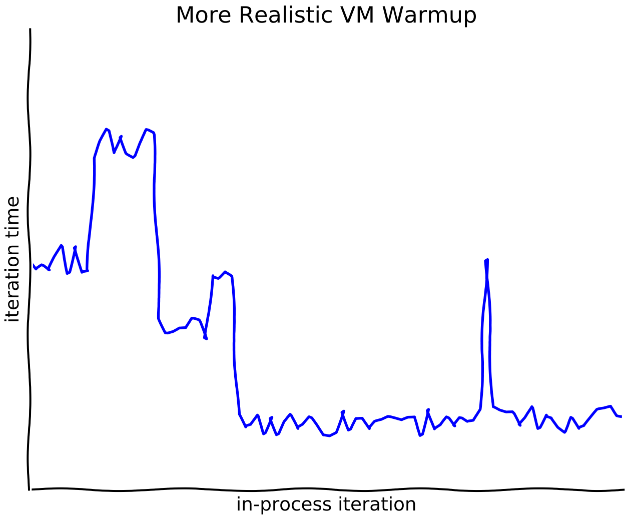
Benchmark runner: tries to control as many confounding variables as possible

Method 5: Krun

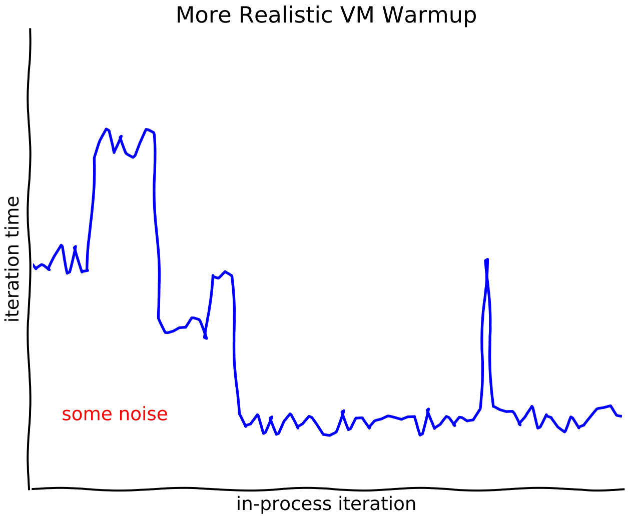
Benchmark runner: tries to control as many confounding variables as possible e.g.:

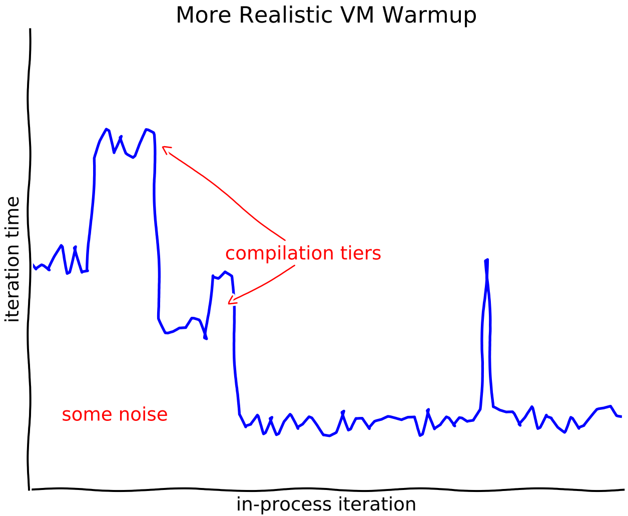
- Minimises I/O
- Sets fixed heap and stack ulimits
- Drops privileges to a ‘clean’ user account
- Automatically reboots the system prior to each proc. exec
- Checks `dmesg` for changes after each proc. exec
- Checks system at (roughly) same temperature for proc. execs
- Enforces kernel settings (tickless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

Method 6: Expectations

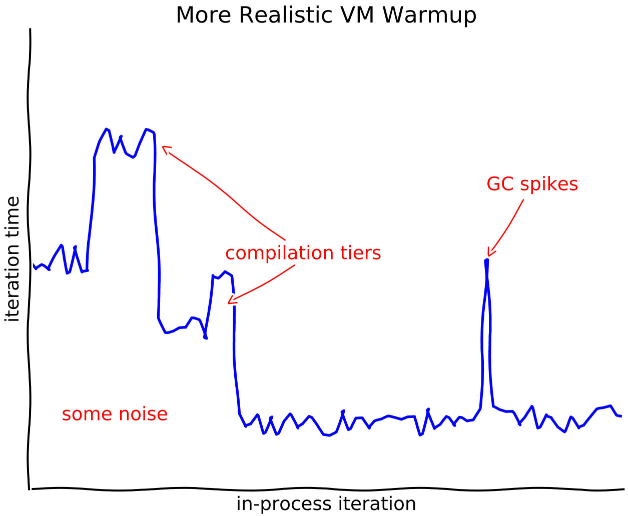


Method 6: Expectations

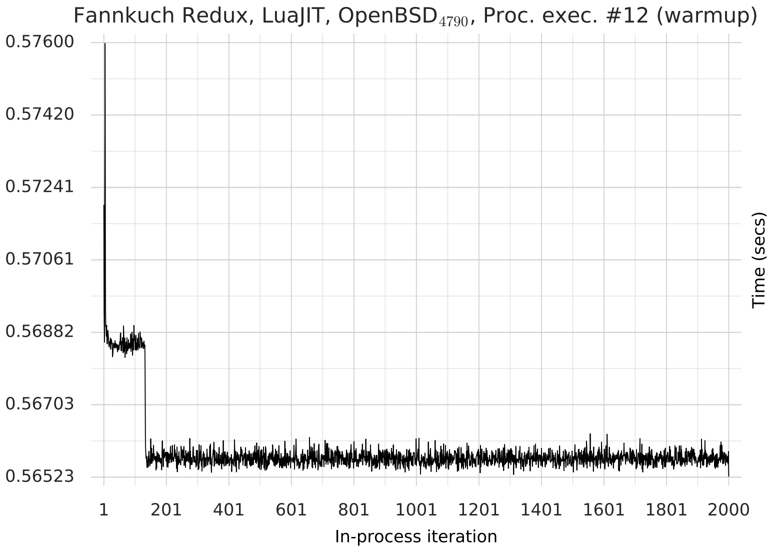



More Realistic VM Warmup
iteration time
compilation tiers
some noise
in-process iteration

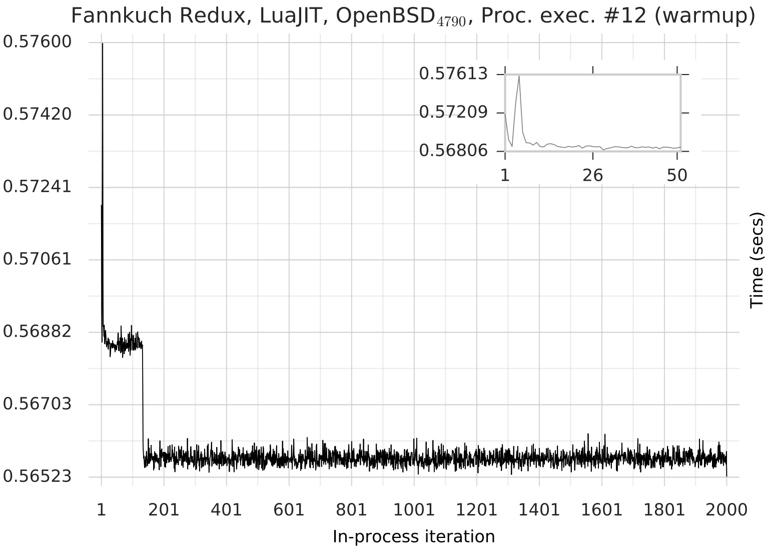
Method 6: Expectations



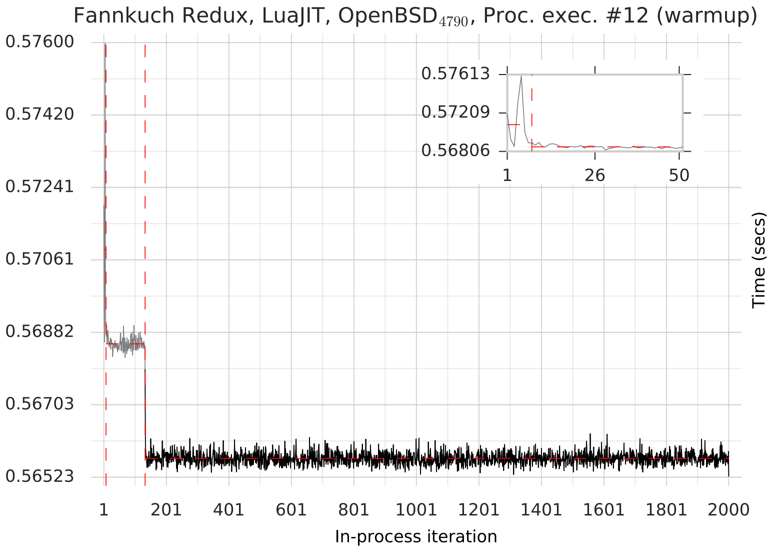
Warmup & flat (1)



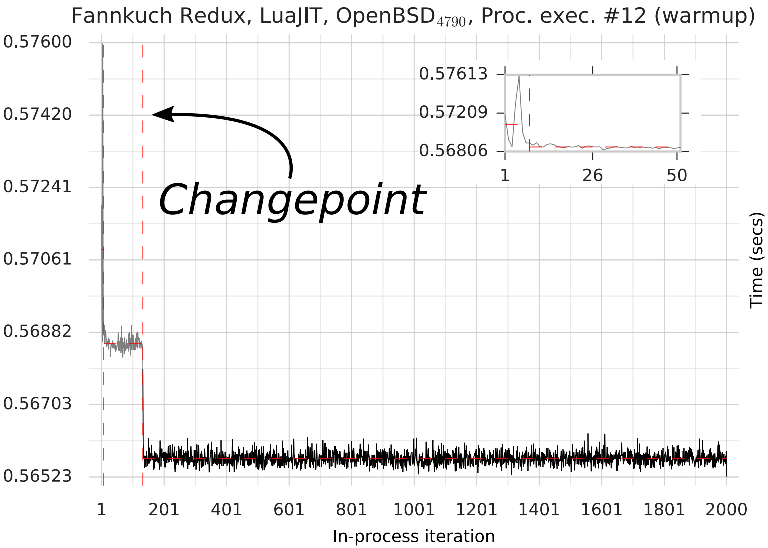
Warmup & flat (1)



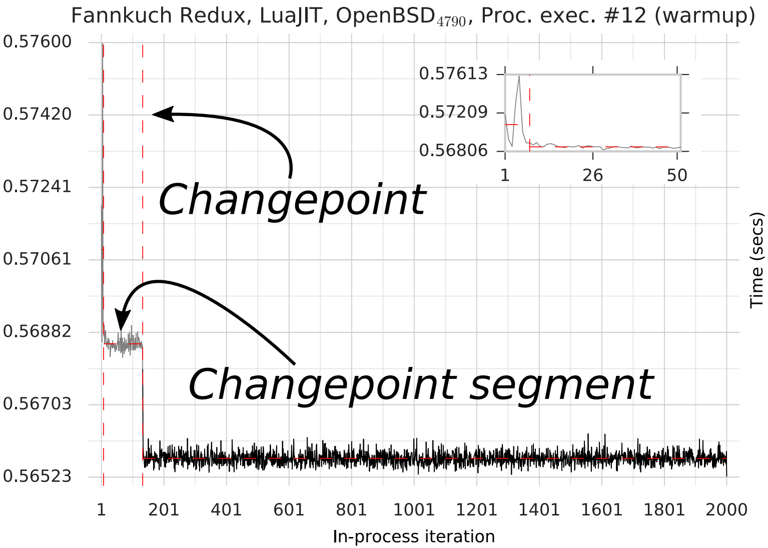
Warmup & flat (1)

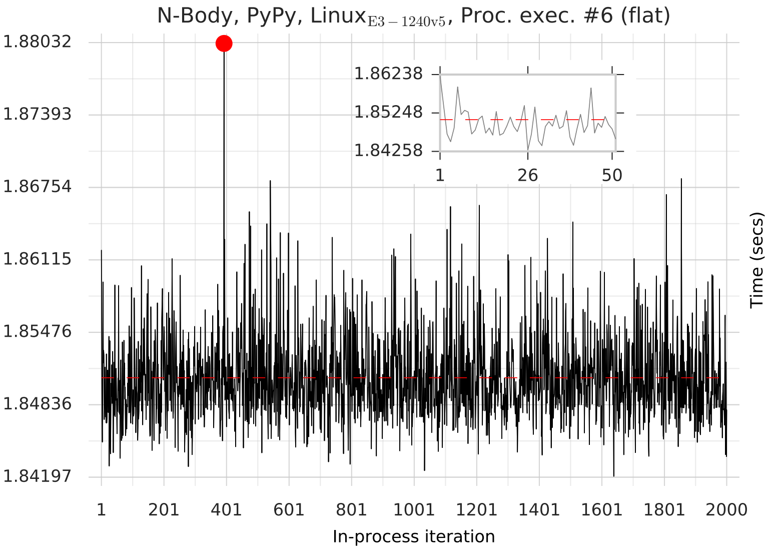


Warmup & flat (1)



Warmup & flat (1)




N-Body, PyPy, Linux$_{\mathrm{E3-1240v5}}$, Proc. exec. #6 (flat)
1.88032
1.87393
1.86754
1.86115
1.85476
1.84836
1.84197
1.86238
1.85248
1.84258
1
26
50
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)

Method 7: Classification

Classification algorithm (steps in order):

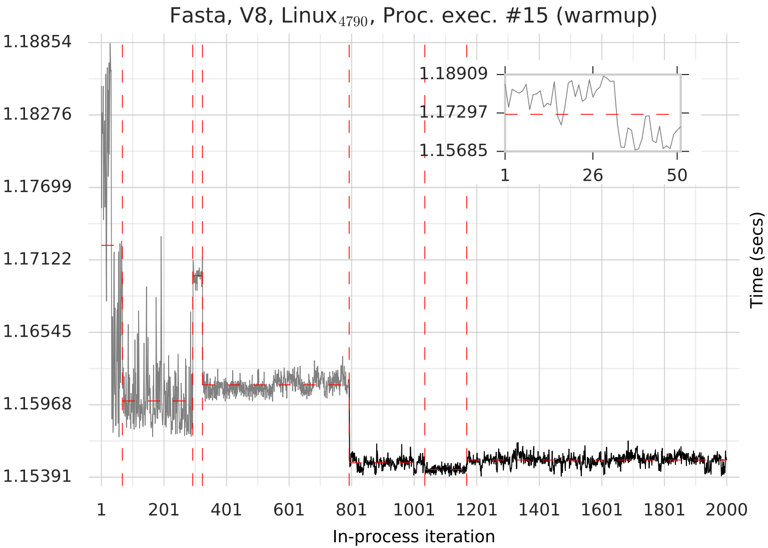
- All segs are equivalent: *flat*

Method 7: Classification

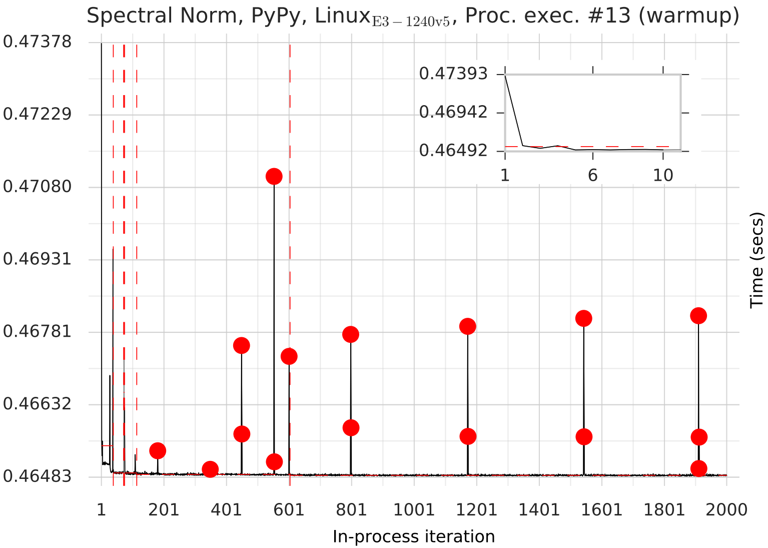
Classification algorithm (steps in order):

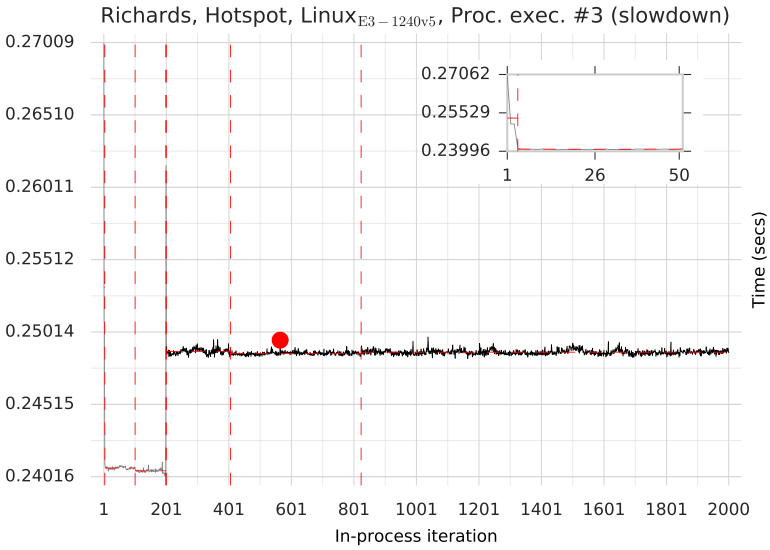
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*

Warmup & flat (2)



Warmup & flat (2)





Richards, Hotspot, Linux$_{E3-1240v5}$, Proc. exec. #3 (slowdown)
0.27009
0.26510
0.26011
0.25512
0.25014
0.24515
0.24016
0.27062
0.25529
0.23996
1
26
50
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)

Method 7: Classification

Classification algorithm (steps in order):

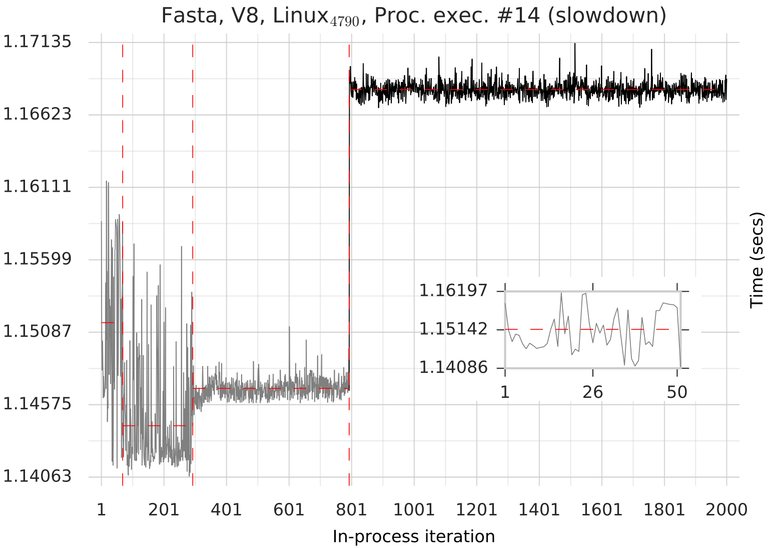
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*

Method 7: Classification

Classification algorithm (steps in order):

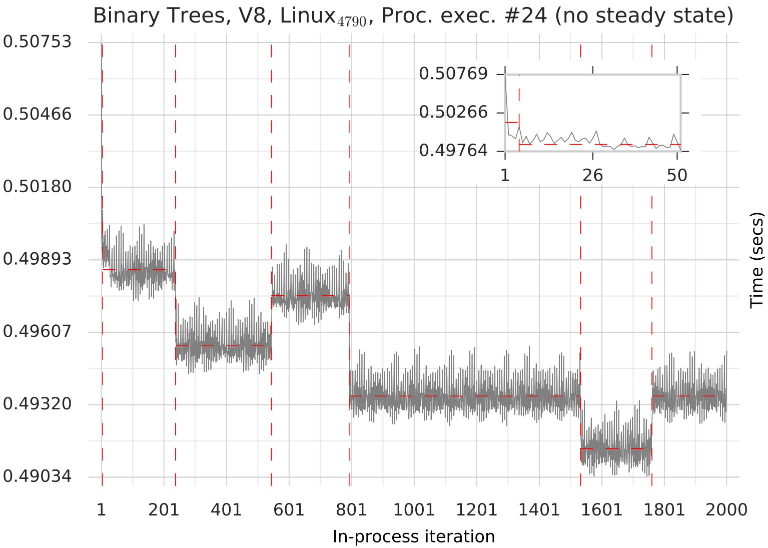
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*

Slowdown (2)



Fasta, V8, Linux$_{4790}$, Proc. exec. #14 (slowdown)
1.17135
1.16623
1.16111
1.15599
1.15087
1.14575
1.14063
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)
1.16197
1.15142
1.14086
1
26
50

No steady state (1)



Classification algorithm (steps in order):

- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*

Classification (3)

Classification algorithm (steps in order):

- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*
- Else: *no steady state*

Classification (3)

Classification algorithm, in order:

- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*
- Else: *no steady state*

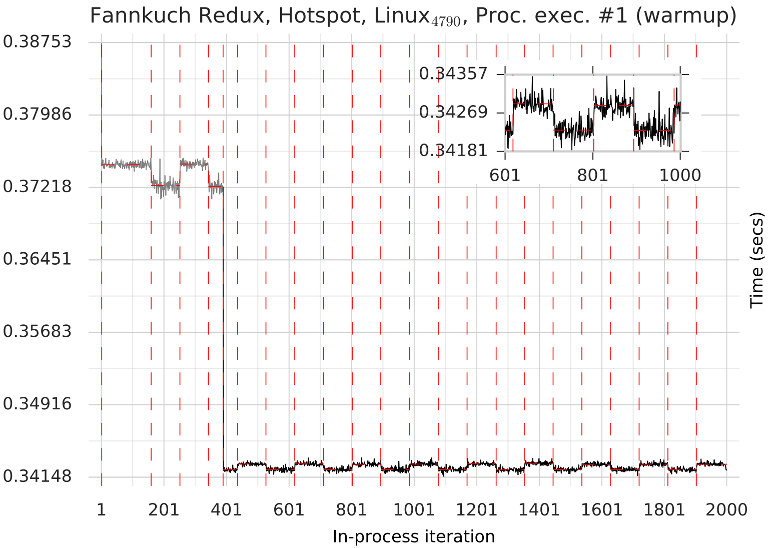
Good

Classification algorithm, in order:

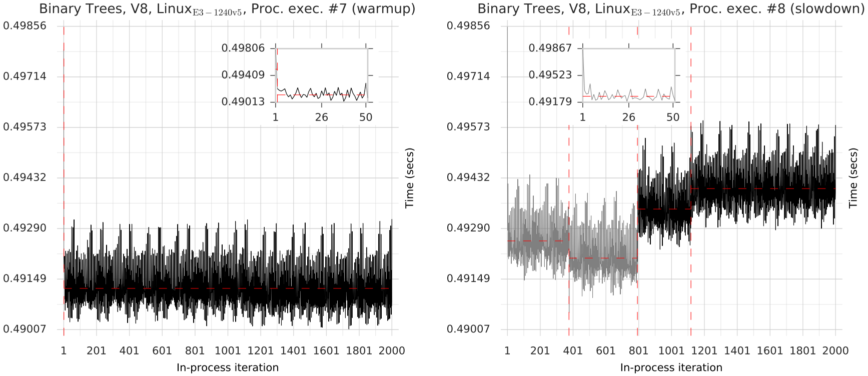
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*
- Else: *no steady state*

Bad

Warmup or no steady state?

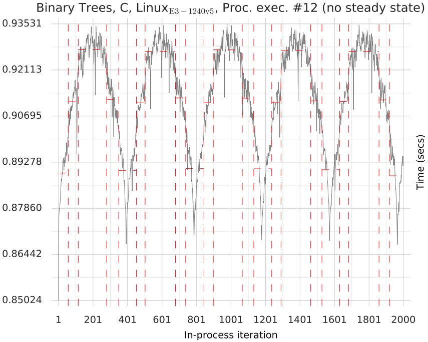


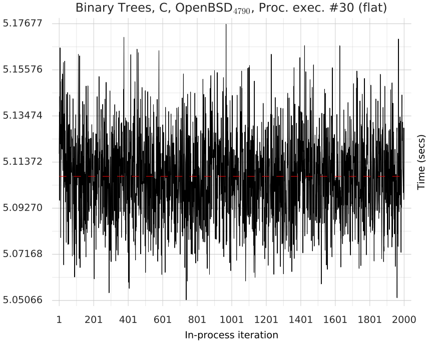
Inconsistent Process-executions



(Same machine)

Inconsistent Process-executions





(Different machines. Bouncing ball Linux-specific)

Individual benchmark stats

Benchmark	VM	Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
binary trees	C	⁓			
	Graal	⤫ (27ʅ, 3ʃ)	32.0 (17.0, 193.8)	6.60 (3.729, 36.608)	0.18594 ±0.000315
	HHVM	⤫ (24ʅ, 4ʃ, 2⁓)			
	HotSpot	⤫ (25ʅ, 5ʃ)	7.0 (7.0, 53.5)	1.19 (1.182, 9.703)	0.18279 ±0.000116
	JRuby+Truffle	ʃ	1082.0 (999.0, 1232.5)	2219.59 (2039.304, 2516.021)	2.05150 ±0.017738
	LuaJIT	⤫ (23ʅ, 4ʃ, 2–, 1⁓)			
	PyPy	⤫ (27ʃ, 3⁓)			
	V8	⤫ (15–, 9ʅ, 6ʃ)	1.5 (1.0, 794.0)	0.25 (0.000, 391.026)	0.49237 ±0.003198
fannkuch redux	C	⤫ (21–, 6ʅ, 2ʃ, 1⁓)			
	Graal	⤫ (28ʅ, 1⁓, 1ʃ)			
	HHVM	ʅ	10.0 (10.0, 10.0)	52.66 (52.660, 52.708)	1.35779 ±0.011948
	HotSpot	ʅ	390.0 (2.0, 390.0)	153.70 (0.407, 155.254)	0.36202 ±0.022767
	JRuby+Truffle	ʃ	1016.5 (999.0, 1023.1)	1039.04 (1014.290, 1059.967)	1.08833 ±0.038580
	LuaJIT	–			0.56285 ±0.000097
	PyPy	⤫ (15ʅ, 13–, 2ʃ)	2.0 (1.0, 28.9)	1.57 (0.000, 43.483)	1.55442 ±0.020549
	V8	= (19ʅ, 11–)	2.0 (1.0, 25.5)	0.31 (0.000, 7.525)	0.30401 ±0.000154
fasta	C	–			0.07048 ±0.000210
	Graal	⤫ (29ʅ, 1⁓)			
	HHVM	⤫ (27ʅ, 2⁓, 1ʃ)			
	HotSpot	⤫ (18ʅ, 12ʃ)	261.0 (6.0, 595.0)	30.73 (0.614, 70.021)	0.11744 ±0.001723
	JRuby+Truffle				
	LuaJIT	⁓			
	PyPy	⁓			
	V8	⤫ (19ʅ, 10ʃ, 1⁓)			

Benchmark	VM	Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
n-body	C	–			0.40555 ±0.005110
	Graal	ʅ	8.0 (7.5, 8.5)	1.22 (1.126, 1.358)	0.13334 ±0.000045
	HHVM	⤫ (16ʅ, 11ʃ, 3⁓)			
	HotSpot	ʅ	2.0 (2.0, 2.0)	0.14 (0.141, 0.143)	0.13699 ±0.000032
	JRuby+Truffle	ʅ	69.0 (69.0, 70.0)	17.95 (17.716, 18.127)	0.20644 ±0.001568
	LuaJIT	–			0.25399 ±0.004471
	PyPy	–			1.85835 ±0.012893
	V8	= (25–, 5ʅ)	1.0 (1.0, 361.6)	0.00 (0.000, 87.367)	0.24138 ±0.000389
Richards	C	⤫ (19–, 5ʃ, 4⁓, 2ʅ)			
	Graal	⤫ (28ʅ, 1⁓, 1ʃ)			
	HHVM				
	HotSpot	⤫ (26ʃ, 2⁓, 2ʅ)			
	JRuby+Truffle	ʃ	1021.0 (1014.9, 1027.0)	917.30 (901.708, 946.683)	0.89509 ±0.027590
	LuaJIT	⤫ (21⁓, 7ʃ, 2ʅ)			
	PyPy	ʅ	2.0 (2.0, 5.0)	1.12 (1.114, 4.054)	0.96809 ±0.010950
	V8	= (29ʅ, 1–)	4.0 (4.0, 16.0)	1.44 (1.434, 7.135)	0.47421 ±0.001218
spectralnorm	C	⤫ (28ʅ, 1–, 1ʃ)	997.0 (2.0, 1001.0)	546.38 (0.546, 547.717)	0.54547 ±0.002562
	Graal	⤫ (29ʃ, 1–)	15.0 (2.0, 21.0)	12.40 (0.812, 17.804)	0.89293 ±0.000087
	HHVM	ʅ	35.0 (34.0, 41.0)	139.18 (136.909, 147.626)	1.40690 ±0.011133
	HotSpot	ʅ	7.0 (7.0, 8.6)	1.90 (1.901, 2.425)	0.31470 ±0.000029
	JRuby+Truffle	ʅ	1011.0 (1007.5, 1014.0)	893.38 (888.985, 896.562)	0.83633 ±0.014403
	LuaJIT	–			0.22435 ±0.000020
	PyPy	ʅ	75.0 (75.0, 75.0)	34.43 (34.429, 34.437)	0.46489 ±0.000046
	V8	ʅ	3.0 (3.0, 3.0)	0.55 (0.554, 0.554)	0.24963 ±0.000039

Individual benchmark stats

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

Overall benchmark stats

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%

Overall benchmark stats

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%

Classical warmup occurs for only:

Summary

Classical warmup occurs for only:

72.4%–74.7% of process executions

Classical warmup occurs for only:

72.4%–74.7% of process executions

43.4%–43.5% of (VM, benchmark) pairs

Summary

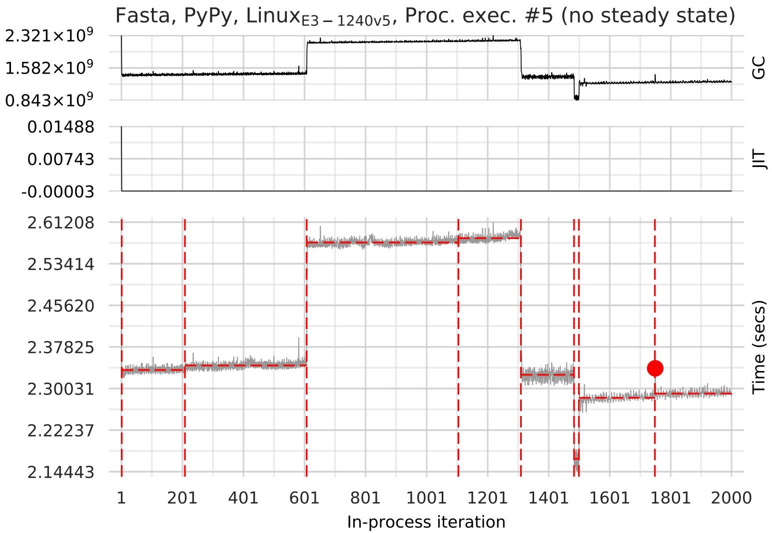
Classical warmup occurs for only:

72.4%–74.7% of process executions

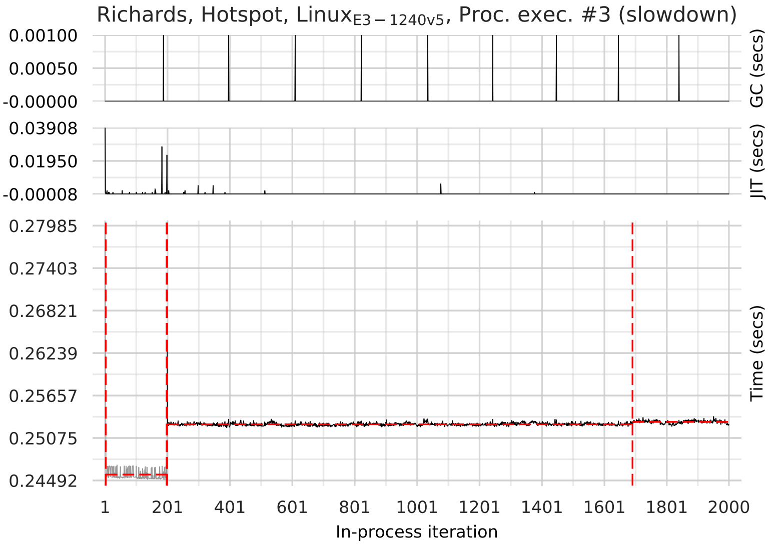
43.4%–43.5% of (VM, benchmark) pairs

0% of benchmarks for (VM, benchmark, machine) triples

Are odd effects correlated with compilation and GC?



Are odd effects correlated with compilation and GC?

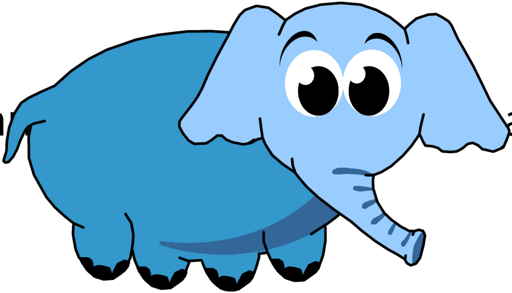


Benchmark suites

Benchmark suites

Benchmarks guide our optimisations

Benchmark suites

Bench  ations

Benchmarks guide our optimisations

Are they complete guides?

A war story

A war story

Symptom: poor performance of a Pyston benchmark on PyPy

A war story

Symptom: poor performance of a Pyston benchmark on PyPy

Cause: RPython traces recursion

A war story

Symptom: poor performance of a Pyston benchmark on PyPy

Cause: RPython traces recursion

Fix: Check for recursion before tracing

A war story: the basis of a fix

```
diff --git a/rpython/jit/metainterp/pyjitpl.py b/rpython/jit/metainterp/pyjitpl.py
--- a/rpython/jit/metainterp/pyjitpl.py
+++ b/rpython/jit/metainterp/pyjitpl.py
@@ -951,9 +951,31 @@
          if warmrunnerstate.inlining:
              if warmrunnerstate.can_inline_callable(greenboxes):
+                 # We’ve found a potentially inlinable function; now we need to
+                 # see if it’s already on the stack. In other words: are we about
+                 # to enter recursion? If so, we don’t want to inline the
+                 # recursion, which would be equivalent to unrolling a while
+                 # loop.
                  portal_code = targetjitdriver_sd.mainjitcode
-                 return self.metainterp.perform_call(portal_code, allboxes,
-                                                     greenkey=greenboxes)
+                 inline = True
+                 if self.metainterp.is_main_jitcode(portal_code):
+                     for gk, _ in self.metainterp.portal_trace_positions:
+                         if gk is None:
+                             continue
+                         assert len(gk) == len(greenboxes)
+                         i = 0
+                         for i in range(len(gk)):
+                             if not gk[i].same_constant(greenboxes[i]):
+                                 break
+                         else:
+                             # The greenkey of a trace position on the stack
+                             # matches what we have, which means we’re definitely
+                             # about to recurse.
+                             inline = False
+                             break
+                 if inline:
+                     return self.metainterp.perform_call(portal_code, allboxes,
+                                 greenkey=greenboxes)
```

A war story: mixed fortunes

Success: slow benchmark now 13.5x faster

A war story: mixed fortunes

Success: slow benchmark now 13.5x faster

Failure: some PyPy benchmarks slow down

A war story: mixed fortunes

Success: slow benchmark now 13.5x faster

Failure: some PyPy benchmarks slow down

Solution: allow *some* tracing into recursion

A war story: data

#unrollings	1	2	3	5	7	10
hexiom2	1.3	1.4	1.1	1.0	1.0	1.0
raytrace-simple	3.3	3.1	2.8	1.4	1.0	1.0
spectral-norm	3.3	1.0	1.0	1.0	1.0	1.0
sympy_str	1.5	1.0	1.0	1.0	1.0	1.0
telco	4	2.5	2.0	1.0	1.0	1.0
polymorphism	0.07	0.07	0.07	0.07	0.08	0.09

http://marc.info/?l=pypy-dev&m=141587744128967&w=2

A war story: conclusion

The benchmark suite said 7 levels, so that’s what I suggested

A war story: conclusion

The benchmark suite said 7 levels, so that's what I suggested

Even though I doubted it was the right global value

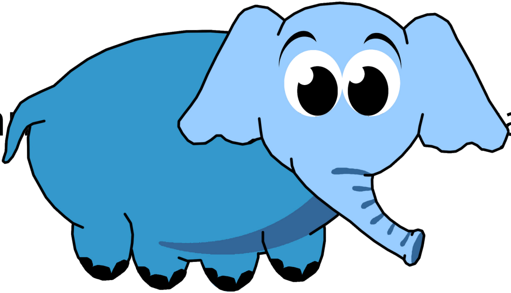
Benchmark suites (2)

Benchmark suites (2)

Benchmarks guide our optimisations

Benchmark suites (2)

Bench



ations

Benchmark suites (2)

Benchmarks guide our optimisations

Are they correct guides?

Octane

17 JavaScript benchmarks from V8

Octane

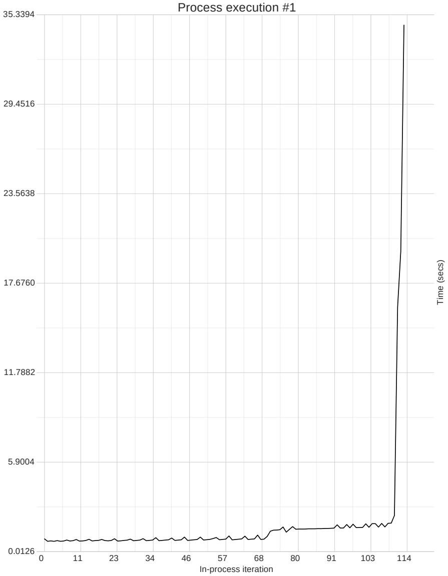
17 JavaScript benchmarks from V8

Let's make each benchmark run for 2000 iterations

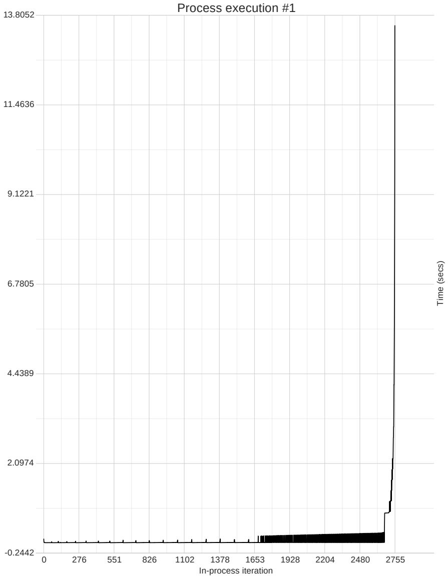
Octane: pdf.js explodes

```
$ d8 run.js
Richards
DeltaBlue
Encrypt
Decrypt
RayTrace
Earley
Boyer
RegExp
Splay
NavierStokes
PdfJS

<--- Last few GCs --->

14907865 ms: Mark-sweep 1093.9 (1434.4) -> 1093.4 (1434.4) MB, 274.8 / 0.0 ms [allocation failure] [GC in old space
14908140 ms: Mark-sweep 1093.4 (1434.4) -> 1093.3 (1434.4) MB, 274.4 / 0.0 ms [allocation failure] [GC in old space
14908421 ms: Mark-sweep 1093.3 (1434.4) -> 1100.5 (1418.4) MB, 280.9 / 0.0 ms [last resort gc].
14908703 ms: Mark-sweep 1100.5 (1418.4) -> 1107.8 (1418.4) MB, 282.1 / 0.0 ms [last resort gc].


<--- JS stacktrace --->

==== JS stack trace =========================================

Security context: 0x20d333ad3ba9 <JS Object>
    2: extractFontProgram(aka Type1Parser_extractFontProgram) [pdfjs.js:17004] [pc=0x3a13b275421b] (this=0x3de358283
    3: new Type1Font [pdfjs.js:17216] [pc=0x3a13b2752078] (this=0x4603fbdaea9 <a Type1Font with map 0x1f822134f7e1>,


#
# Fatal error in CALL_AND_RETRY_LAST
# Allocation failed - process out of memory
#

zsh: illegal hardware instruction  d8 run.js
```

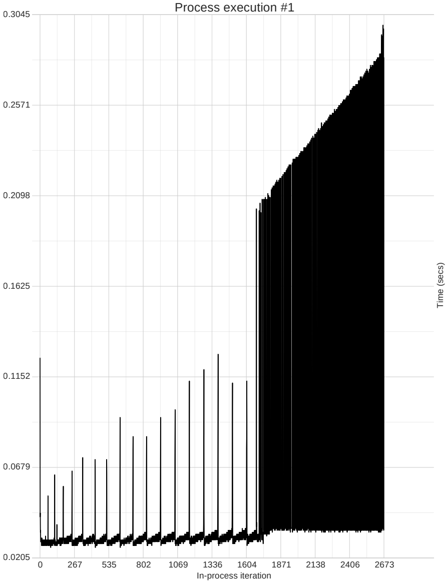
Octane: analysing pdf.js



Octane: analysing pdf.js



Octane: analysing pdf.js



Octane: debugging

pdfjs.js (/tmp/octane) - GVIM2

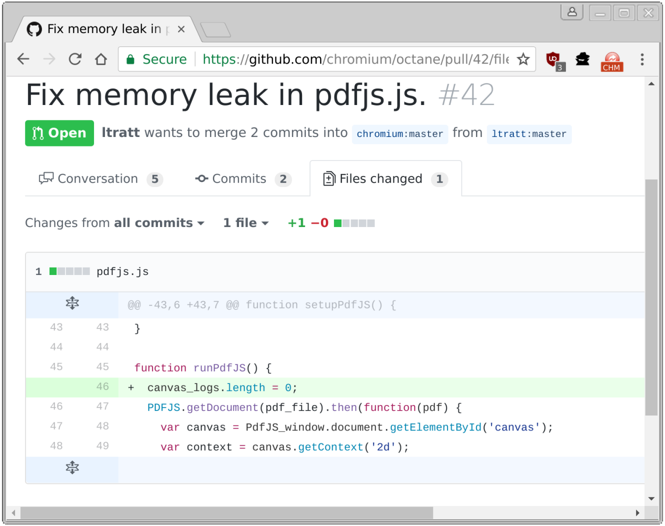
```
//////// Benchmark set-up. (c) by Google. ////////

var pdf_file = "test.pdf";
var canvas_logs = [];

var PdfJS = new BenchmarkSuite("PdfJS", [10124921], [
  new Benchmark("PdfJS", false, false, 24,
    runPdfJS, setupPdfJS, tearDownPdfJS, null, 4)
]);
```

```
NORMAL  master  pdfjs.js   jav…  utf-8[unix]  0% ¶  28/33053 :  1
```

```
function runPdfJS() {
  PDFJS.getDocument(pdf_file).then(function(pdf) {
    var canvas = PdfJS_window.document.getElementById('canvas');
    var context = canvas.getContext('2d');
    var renderContext = {canvasContext: context};
    canvas_logs.push(context.__log__);

    // Cycle through all pages.
    function renderPages(i, j) {
      if (i > j) return;
      context.clearRect(0, 0, canvas.width, canvas.height);
      pdf.getPage(i).then(function(page) {
        renderContext.viewport = page.getViewport(1);
        canvas.height = renderContext.viewport.height;
        canvas.width = renderContext.viewport.width;
        page.render(renderContext).then(renderPages.bind(null, i + 1, j));
      });
    }
    renderPages(1, pdf.numPages);
  });

  // Wait for everything to complete.
  PdfJS_window.flushTimeouts();
}
```

```
pdfjs.js   jav…  utf-8[unix]  0% ¶  50/33053 : 38
```

Octane: fixing



Octane: other issues

pdfjs isn't the only problem

Octane: other issues

pdfjs isn’t the only problem

CodeLoadClosure also has a memory leak

Octane: other issues

pdfjs isn't the only problem

CodeLoadClosure also has a memory leak

zlib complains that `Cannot enlarge memory arrays in asm.js` (a memory leak? I don't know)

Octane: other issues

pdfjs isn't the only problem

CodeLoadClosure also has a memory leak

zlib complains that `Cannot enlarge memory arrays in asm.js` (a memory leak? I don't know)

Timings are made with a non-monotonic microsecond timer

Summary

Summary

Why aren't more users more happy with our VMs?

Why aren’t more users more happy with our VMs?

My thesis: our benchmarking *and* our benchmarks have misled us

How to benchmark a bit better

How to benchmark a bit better

1. Run benchmarks for longer to uncover issues.

How to benchmark a bit better

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.

How to benchmark a bit better

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

How to benchmark a bit better

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.

How to benchmark a bit better

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.

How to benchmark a bit better

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.

What we're doing next

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

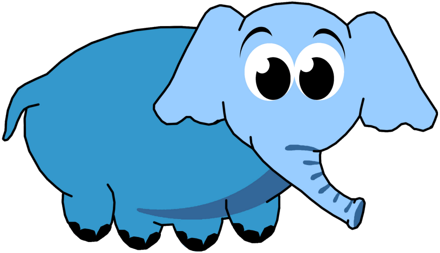
Thanks

- EPSRC: *COOLER* and *Lecture*.
- Oracle: various.

K Whiteford for Barry, the Benchmarking Elephant in the Room

Thanks for listening

Thanks for listening



And don't forget Barry