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

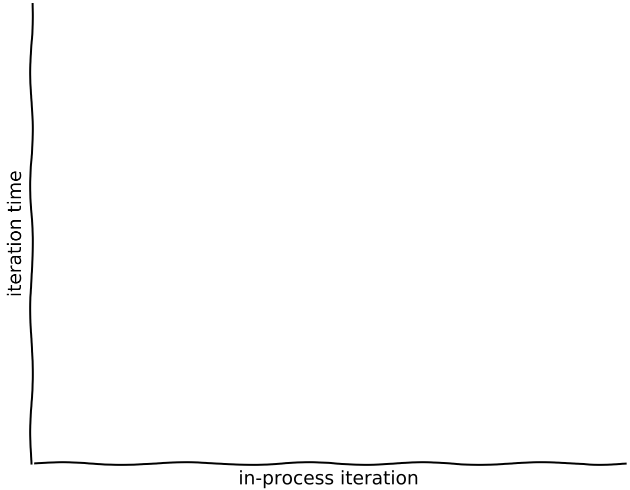
2018-05-10

What to expect from this talk

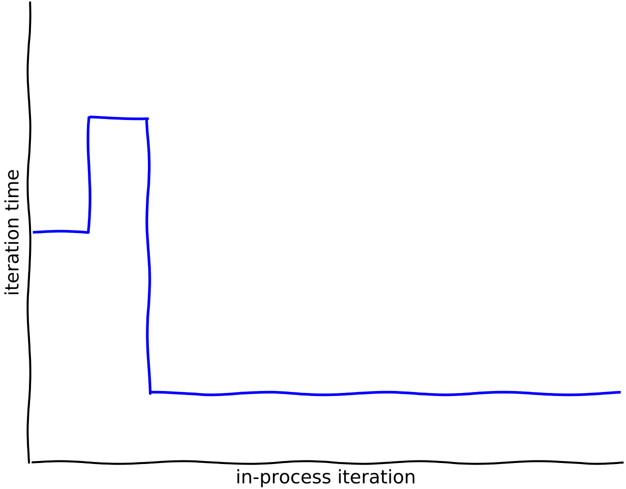
JVMs bring "gcc -O2" to the masses

–*Cliff Click*: A JVM does that?

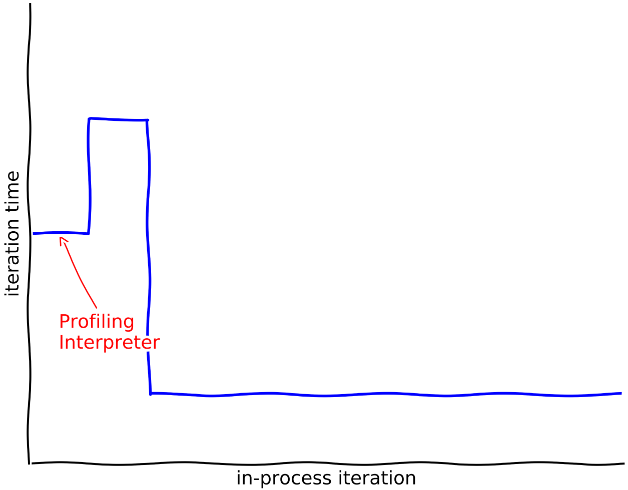
What do VM claims pertain to?



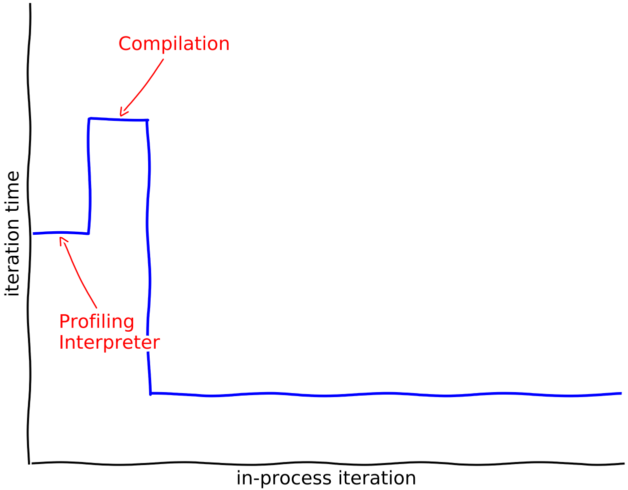
What do VM claims pertain to?



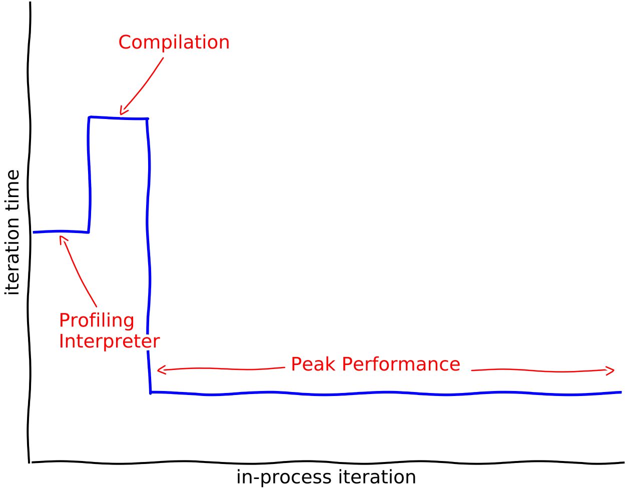
What do VM claims pertain to?



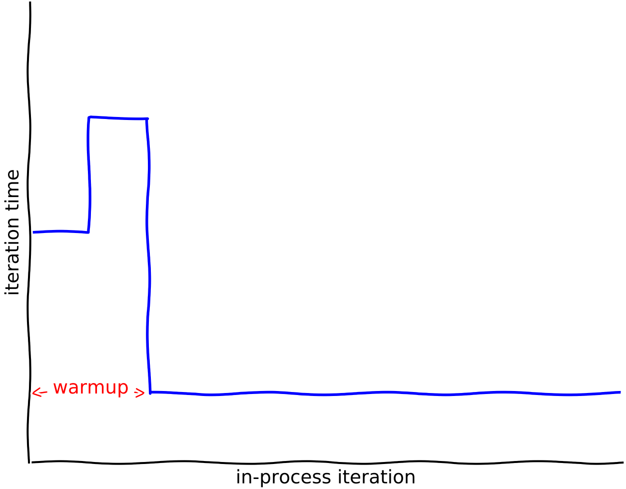
What do VM claims pertain to?



What do VM claims pertain to?



What do VM claims pertain to?



Warmup

Users *always* perceive warmup

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
- 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 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
- Reruns any proc. exec where the CPU was throttled
- Checks `dmesg` for changes after each proc. exec
- Checks system at (roughly) same temperature for proc. execs
- Enforces kernel settings (tickless mode, CPU governors, ...)

And now for data...

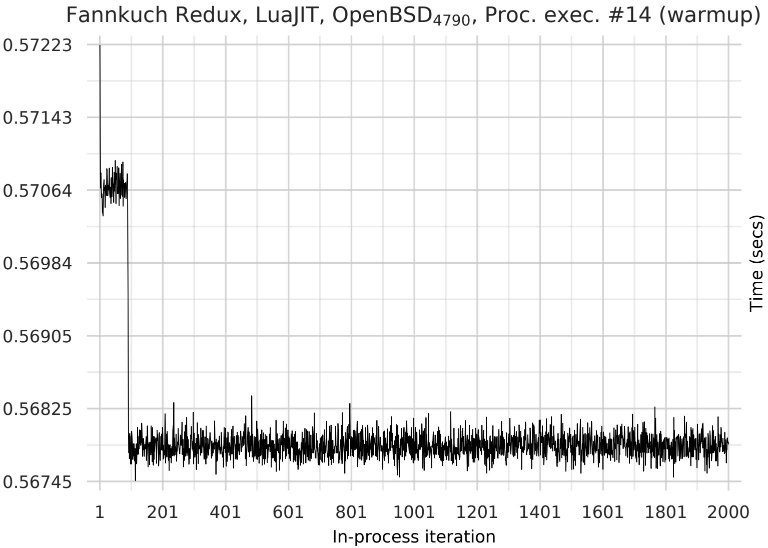
The experiment has gone through many versions.

And now for data...

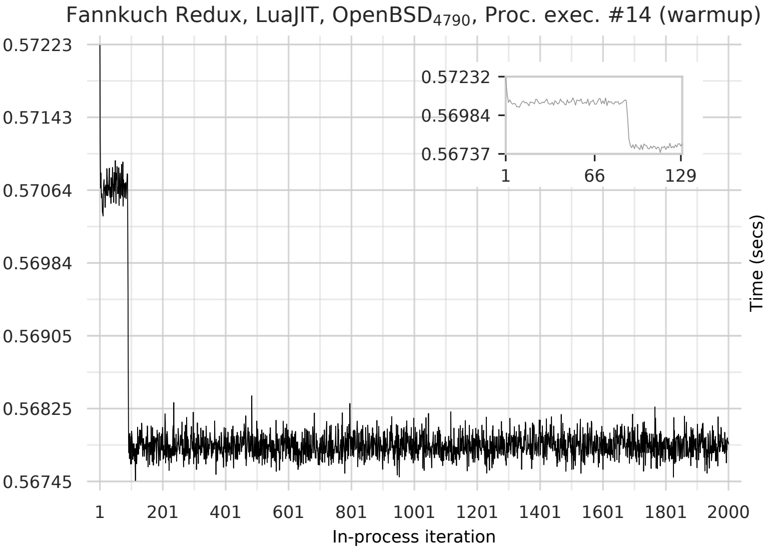
The experiment has gone through many versions.

The following data is from the 1.5 run.

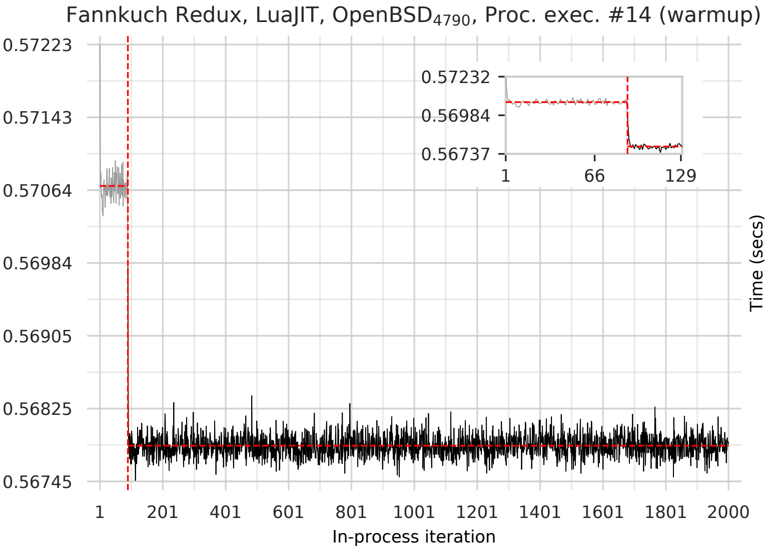
Warmup & flat (1)



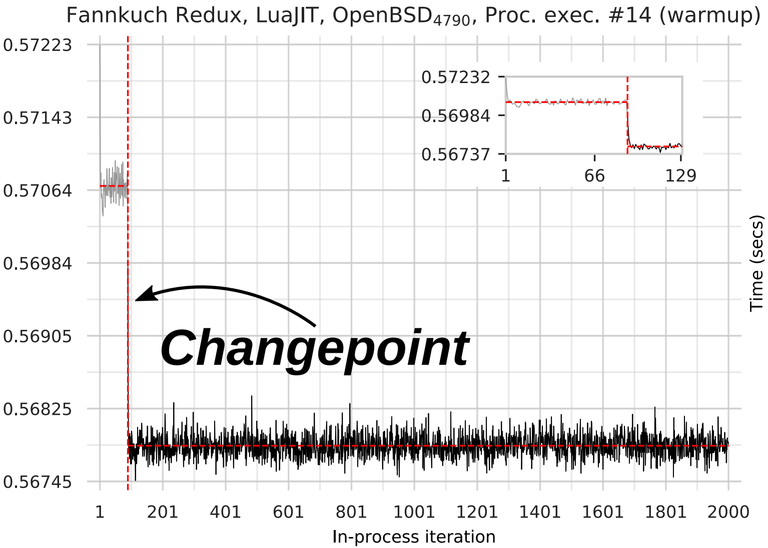
Warmup & flat (1)



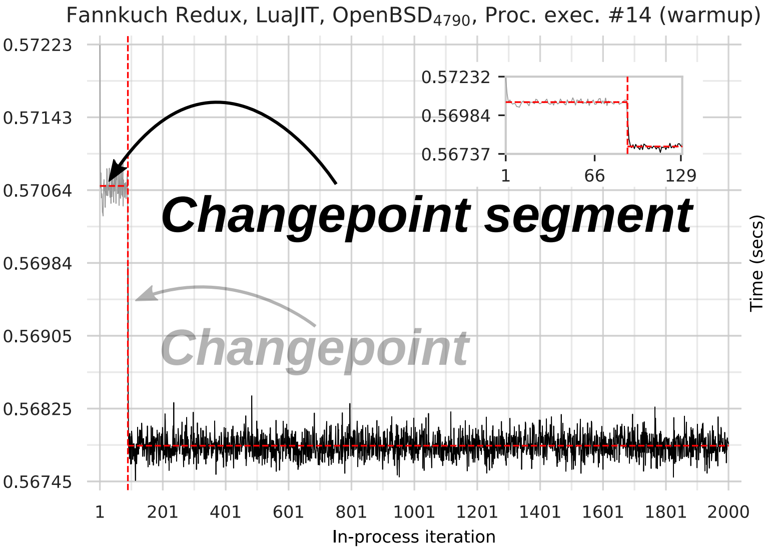
Warmup & flat (1)



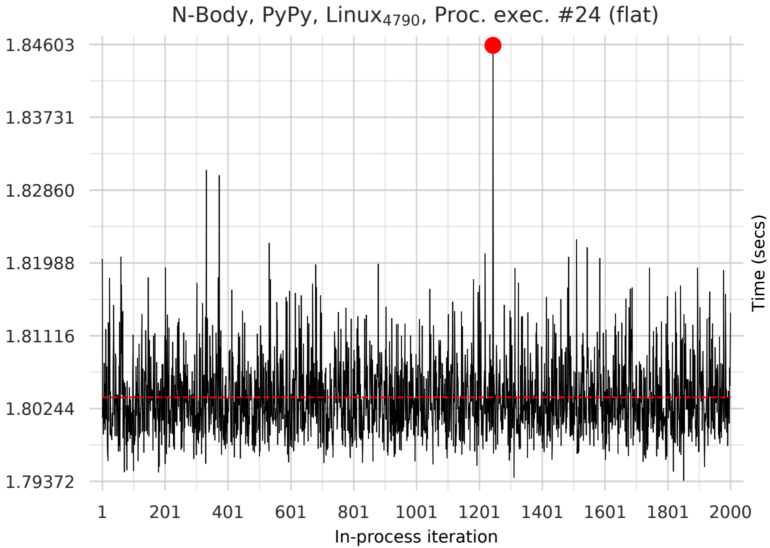
Warmup & flat (1)



Warmup & flat (1)



Warmup & flat (1)



Classification algorithm (steps in order):

- All segs are equivalent: *flat*

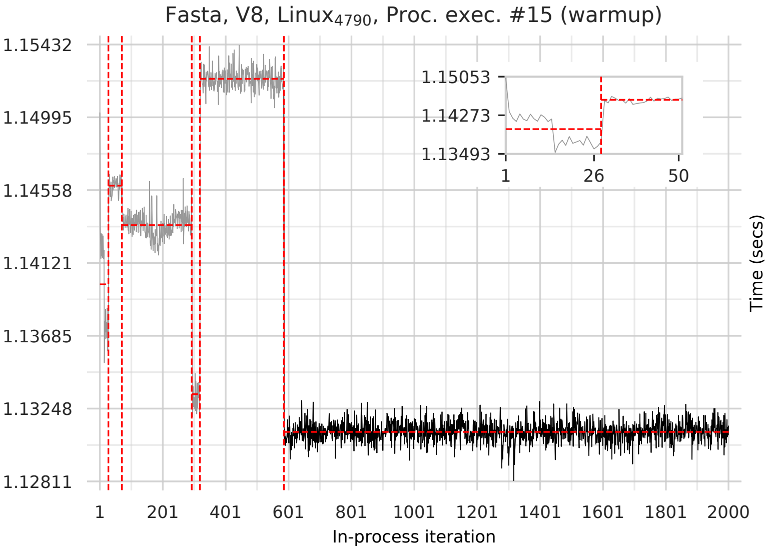
Method 7: Classification

Classification algorithm (steps in order):

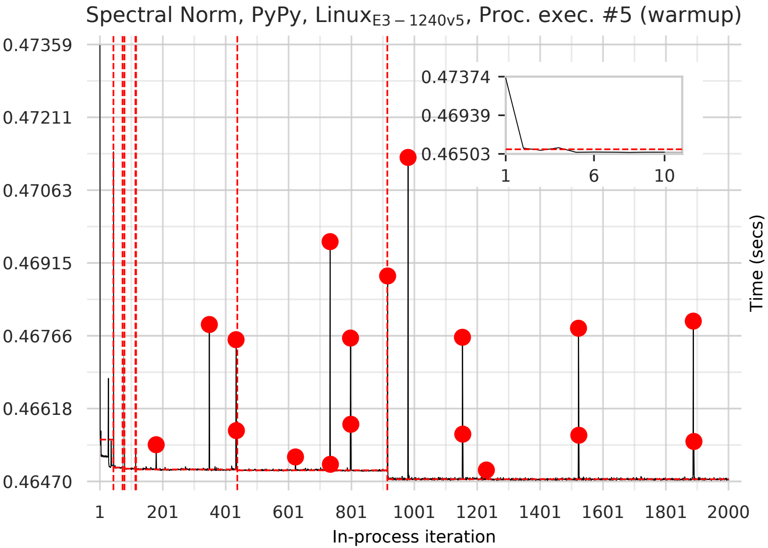
All segs are equivalent: *flat*

Final seg is in fastest set: *warmup*

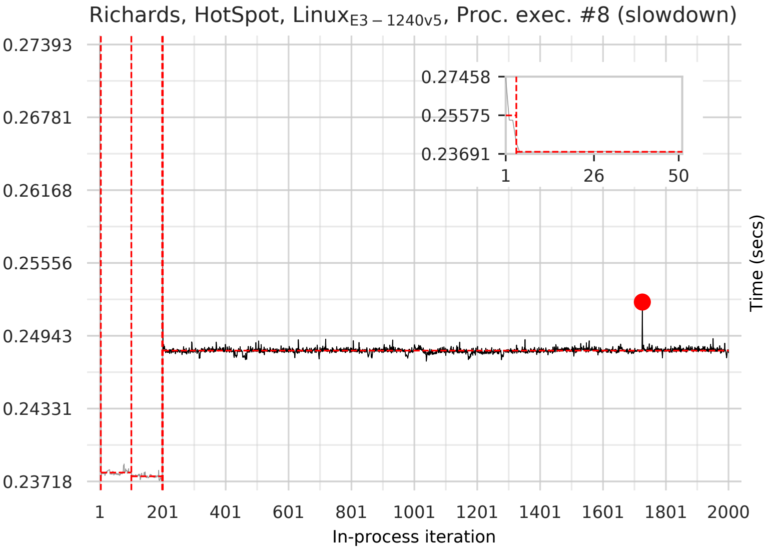
Warmup & flat (2)



Warmup & flat (2)



Slowdown (1)



Richards, HotSpot, Linux$_{E3-1240v5}$, Proc. exec. #8 (slowdown)
0.27393
0.26781
0.26168
0.25556
0.24943
0.24331
0.23718
0.27458
0.25575
0.23691
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*

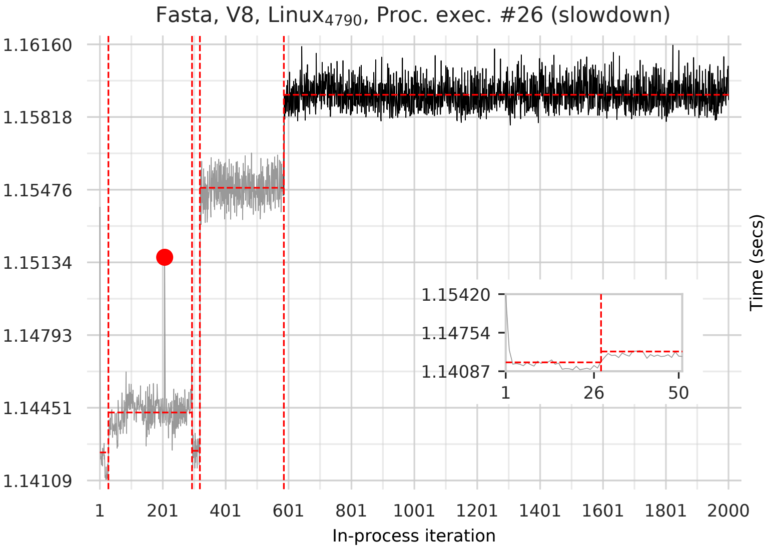
Final seg is in fastest set: *warmup*

Method 7: Classification

Classification algorithm (steps in order):

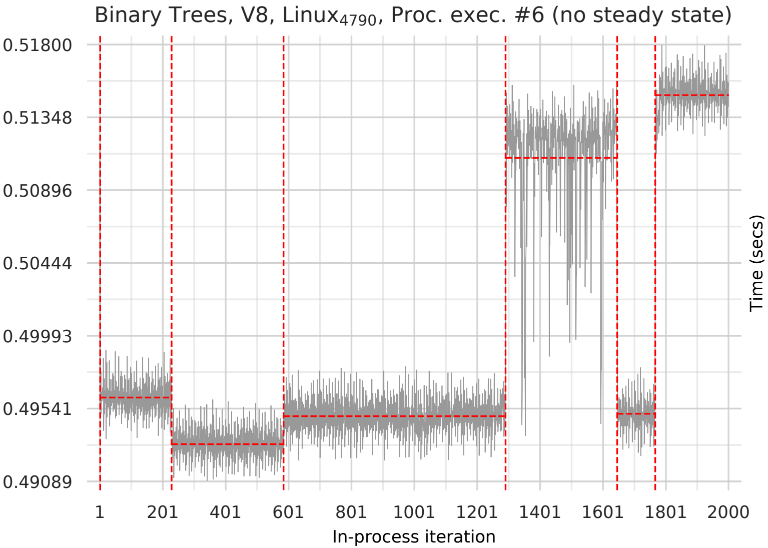
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*

Slowdown (2)



Fasta, V8, Linux$_{4790}$, Proc. exec. #26 (slowdown)
1.16160
1.15818
1.15476
1.15134
1.14793
1.14451
1.14109
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)
1.15420
1.14754
1.14087
1
26
50

No steady state (1)



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*

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

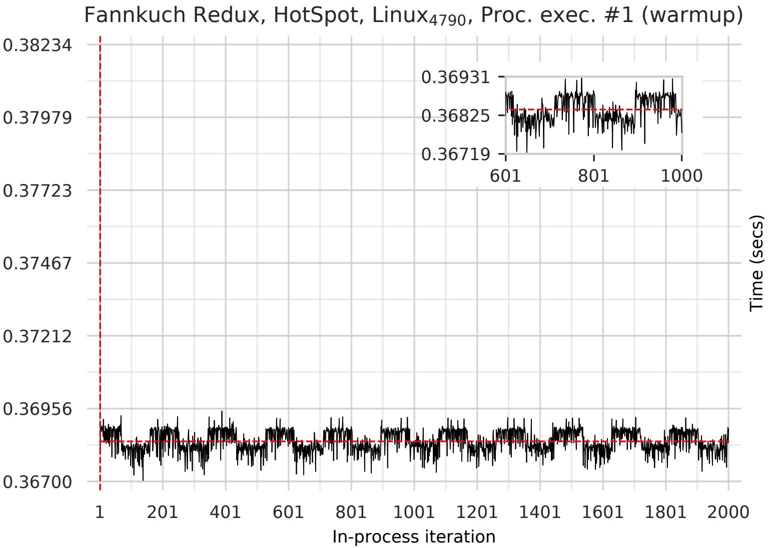
Classification (3)

Classification algorithm, in order:

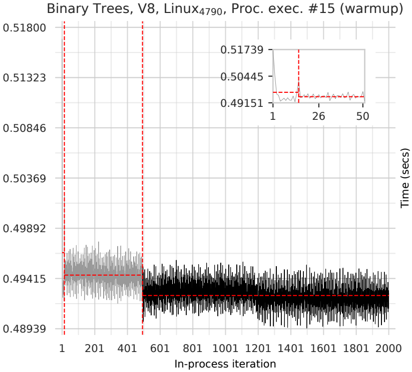
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*
- Else: *no steady state*

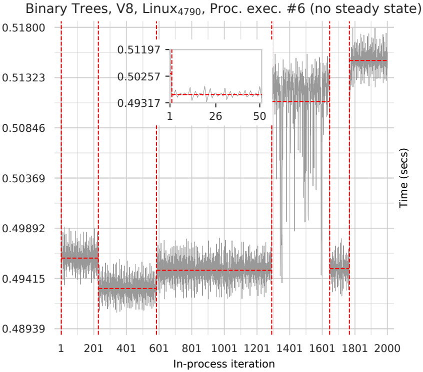
Bad

Warmup or no steady state?



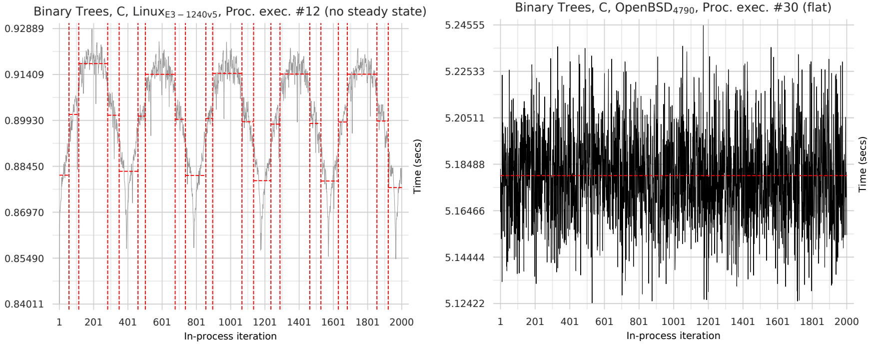
Inconsistent Process-executions





(Same machine)

Inconsistent Process-executions



(Different machines. Bouncing ball Linux-specific)

Individual benchmark stats

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*binary trees*	∿				*n-body*	–			0.40557 ±0.006155
Graal		⋈ (27⅂, 2⅃, 1∿)					⅂	6.0 (5.0,7.0)	0.86 (0.704,1.090)	0.13677 ±0.000035
HHVM		⋈ (19⅂, 7⅃, 4∿)								
HotSpot		⋈ (15⅂, 13⅃, 2∿)					⅂	2.0 (2.0,2.0)	0.14 (0.142,0.151)	0.13699 ±0.000029
LuaJIT		⋈ (25⅂, 3∿, 2⅃)					= (29–, 1⅂)	1.0 (1.0,1.0)	0.00 (0.000,0.000)	0.25401 ±0.002040
PyPy		⋈ (27∿, 3⅃)					–			1.83314 ±0.024563
TruffleRuby		⋈ (19⅃, 11⅂)	1003.0 (283.8,1009.2)	2621.96 (749.093,2639.554)	2.61136 ±0.036511		⋈ (28⅂, 2⅃)	68.0 (68.0,1003.0)	40.45 (40.056,481.916)	0.46998 ±0.001516
V8		⋈ (13–, 9⅃, 8⅂)	228.0 (1.0,588.0)	106.34 (0.000,275.806)	0.46927 ±0.003135		= (20–, 10⅂)	1.0 (1.0,363.0)	0.00 (0.000,87.733)	0.24148 ±0.000304
C	*fannkuch redux*	⋈ (25–, 5⅃)	1.0 (1.0,538.9)	0.00 (0.000,224.243)	0.41742 ±0.011343	*Richards*	⋈ (18–, 8⅃, 4∿)			
Graal		⋈ (26⅂, 4∿)					⅂	2.0 (2.0,35.3)	0.95 (0.879,9.745)	0.26465 ±0.007761
HHVM		⋈ (28⅂, 2∿)								
HotSpot		⋈ (17⅃, 10⅂, 3∿)					⋈ (28⅃, 1∿, 1⅂)			
LuaJIT		–			0.56287 ±0.000113		⋈ (18∿, 7⅃, 5⅂)			
PyPy		⋈ (19–, 9⅂, 2⅃)	1.0 (1.0,1120.3)	0.00 (0.000,1802.797)	1.60962 ±0.022821		⅂	2.0 (2.0,2.0)	1.04 (1.023,1.102)	0.88940 ±0.035495
TruffleRuby		⋈ (26⅃, 3∿, 1⅂)					⅃	1021.0 (334.6,1068.9)	1521.33 (499.847,1585.296)	1.48585 ±0.011040
V8		= (28⅂, 2–)	2.0 (1.5,25.0)	0.32 (0.142,7.583)	0.30634 ±0.000160		⅂	3.0 (3.0,189.8)	1.04 (0.961,97.303)	0.47681 ±0.020468
C	*fasta*	–			0.06848 ±0.000167	*spectralnorm*	⋈ (27⅂, 2–, 1⅃)	775.0 (1.5,780.0)	425.16 (0.246,426.809)	0.54581 ±0.033116
Graal		⅂	4.0 (3.0,34.4)	0.75 (0.535,5.873)	0.16188 ±0.000738		⅃	14.0 (2.0,94.6)	13.60 (0.830,98.737)	1.05685 ±0.000126
HHVM		⋈ (13⅃, 10⅂, 7∿)					⋈ (29⅂, 1∿)			
HotSpot		⋈ (27⅂, 3⅃)	6.0 (5.0,595.0)	0.62 (0.500,70.062)	0.11691 ±0.001949		⅂	7.0 (7.0,7.5)	1.91 (1.902,3.645)	0.31472 ±0.169143
LuaJIT		⋈ (19⅃, 9–, 1∿, 1⅂)					–			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		⋈ (27⅃, 3⅂)	586.0 (318.0,587.6)	612.12 (330.270,622.328)	1.04904 ±0.020905		⅂	3.0 (3.0,3.0)	0.52 (0.523,0.526)	0.25362 ±0.000034

Individual benchmark stats

		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

Overall benchmark stats

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%

Overall benchmark stats

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

Classical warmup occurs for only:

Summary

Classical warmup occurs for only:

67.2%–70.3% of process executions

Summary

Classical warmup occurs for only:

67.2%–70.3% of process executions

37.8%–40.0% of (VM, benchmark) pairs

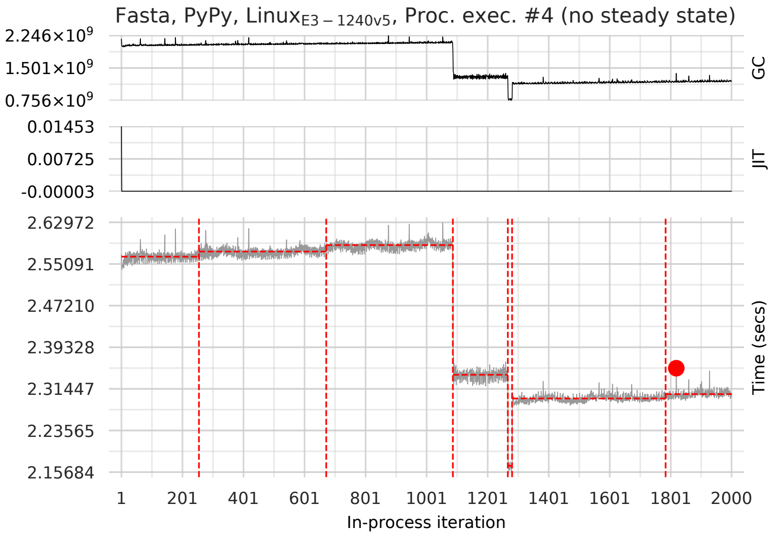
Classical warmup occurs for only:

67.2%–70.3% of process executions

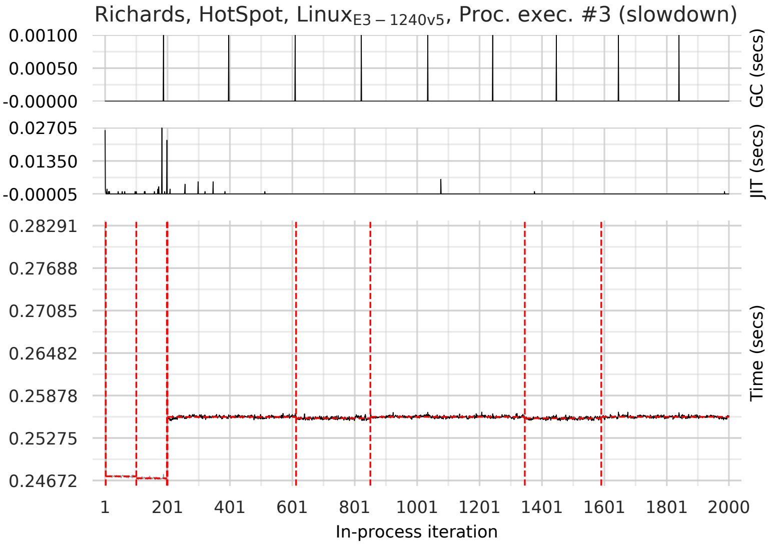
37.8%–40.0% of (VM, benchmark) pairs

12.5% of benchmarks for (VM, benchmark, machine) triples

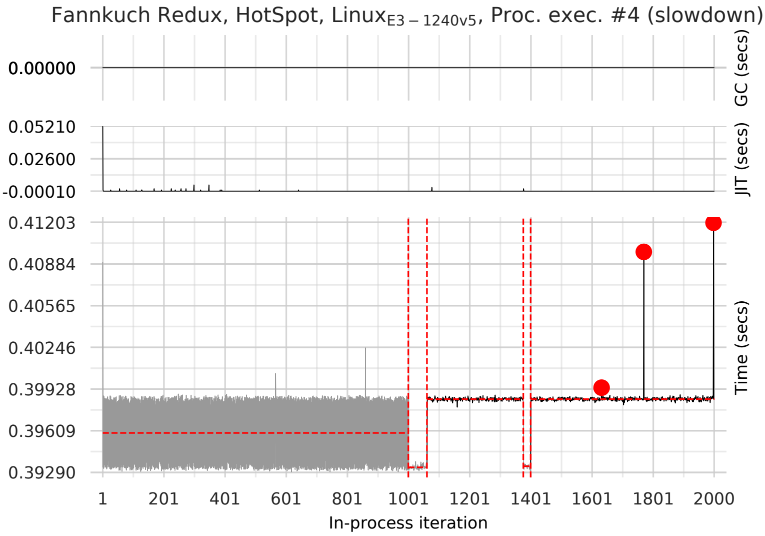
Are odd effects correlated with compilation and GC?



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

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

Benchmarks guide our optimisations

Are they correct guides?

Octane

17 JavaScript benchmarks from V8

Octane

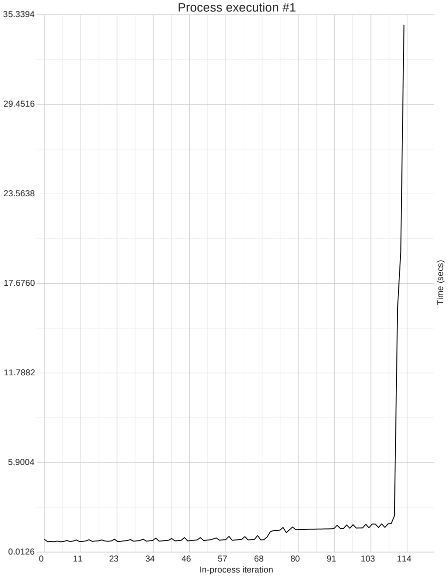
17 JavaScript benchmarks from V8

Let’s make each benchmark run for 2000 iterations

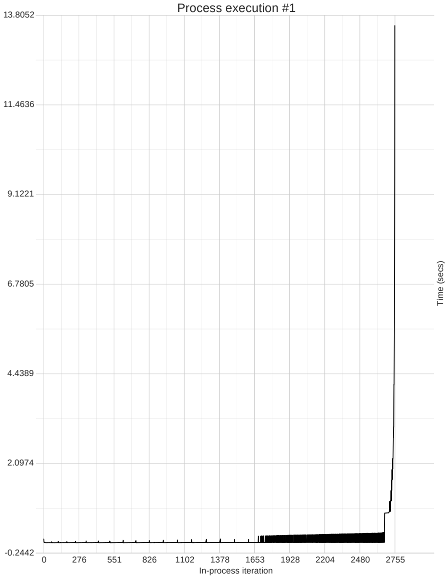
Octane: pdf.js explodes

```
$ 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
```

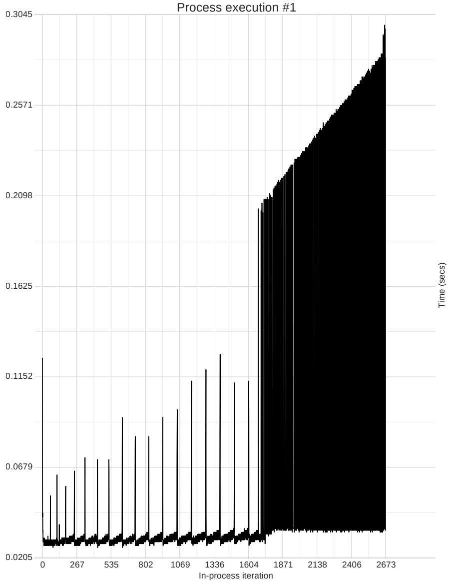
Octane: analysing pdf.js



Octane: analysing pdf.js



Octane: analysing pdf.js



Octane: debugging

pdfjs.js (/tmp/octane) - GVIM2

pdfjs.js | buffers

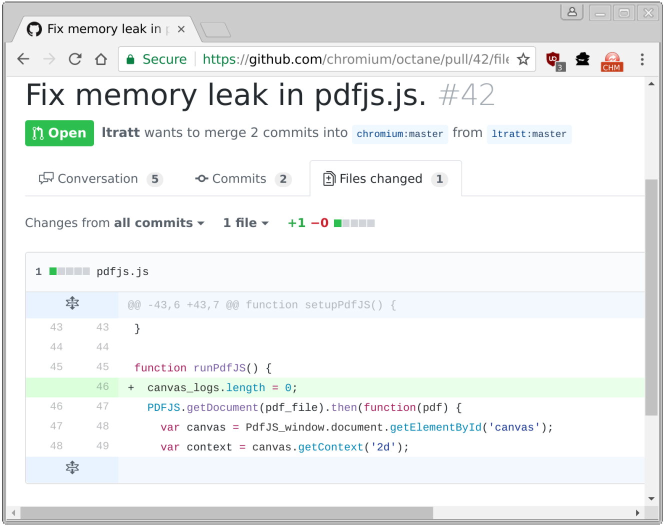
```
///////// Benchmark set-up. (c) by Google. ////////

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?

Summary

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

My thesis: benchmarking *and* benchmarks are performance destiny.

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

My thesis: benchmarking *and* benchmarks are performance destiny.

Ours 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. Collect more benchmarks.

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. Collect more benchmarks.
6. Focus on predictable performance.

The big question

The big question

Can we fix existing VMs?

The big question

Can we fix existing VMs?

At least a bit... but a lot? Unclear.

The big question

Can we fix existing VMs?

At least a bit... but a lot? Unclear.

In case we can't, I have an idea...

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
  elif instr == INSTR_IF:
    result = stack.pop()
    if result == True:
      program_counter += 1
    else:
      program_counter +=
        read_jump_if_instruction()
  elif instr == INSTR_ADD:
    lhs = stack.pop()
    rhs = stack.pop()
    if isinstance(lhs, int)
     and isinstance(rhs, int):
      stack.push(lhs + rhs)
    else: ...
    program_counter += 1
```

FL Interpreter

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

Meta-tracing JITs

FL Interpreter	User program (lang *FL*)

```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
if x < 0:
  x = x + 1
else:
  x = x + 2
x = x + 3
```

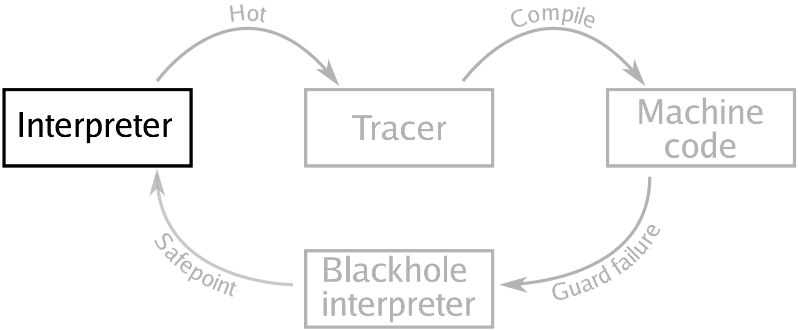
Meta-tracing JITs

FL Interpreter	Initial trace

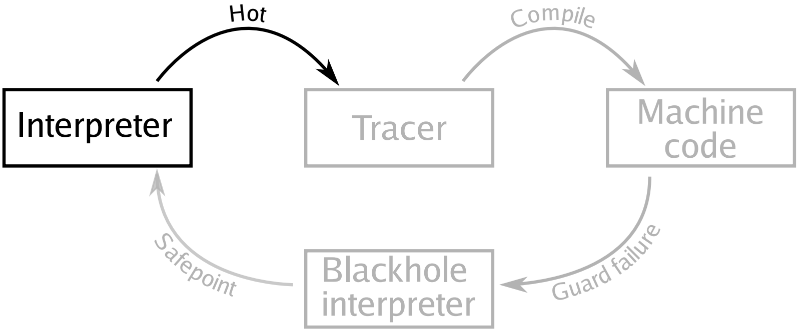
```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = stack.pop()
    program_counter += 1
  elif instr == INSTR_INT:
    stack.push(read_int_from_instruction())
    program_counter += 1
  elif instr == INSTR_LESS_THAN:
    rhs = stack.pop()
    lhs = stack.pop()
    if isinstance(lhs, int) and isinstance(rhs, int):
      if lhs < rhs:
        stack.push(True)
      else:
        stack.push(False)
    else: ...
    program_counter += 1
```

```
v0 = <program_counter>
v1 = <stack>
v2 = <vars>
v3 = load_instruction(v0)
guard_eq(v3, INSTR_VAR_GET)
v4 = dict_get(v2, "x")
list_append(v1, v4)
v5 = add(v0, 1)
v6 = load_instruction(v5)
guard_eq(v6, INSTR_INT)
list_append(v1, 0)
v7 = add(v5, 1)
v8 = load_instruction(v7)
guard_eq(v8, INSTR_LESS_THAN)
v9 = list_pop(v1)
v10 = list_pop(v1)
guard_type(v9, int)
guard_type(v10, int)
guard_not_less_than(v9, v10)
list_append(v1, False)
v11 = add(v7, 1)
v12 = load_instruction(v11)
guard_eq(v12, INSTR_IF)
v13 = list_pop(v1)
guard_false(v13)
...
```

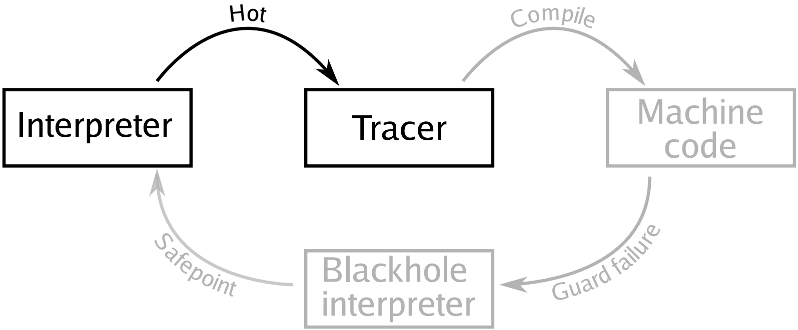
Meta-tracer states



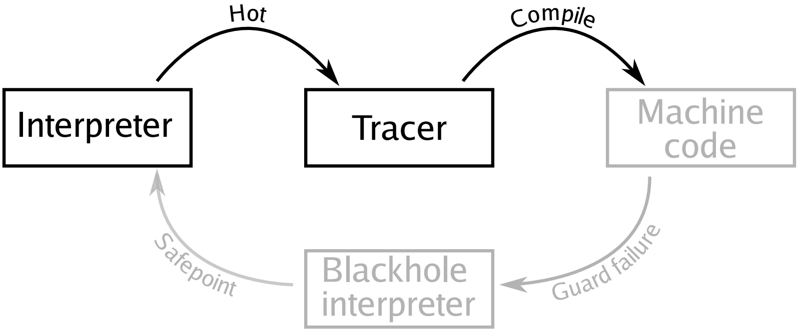
Meta-tracer states



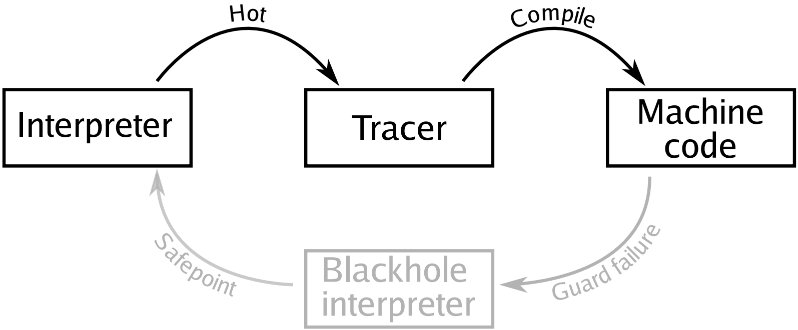
Meta-tracer states



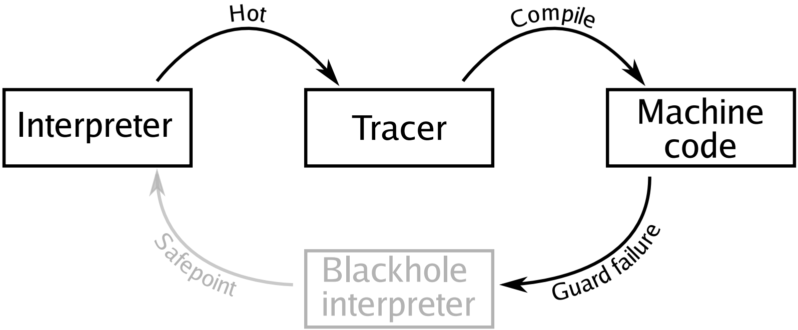
Meta-tracer states



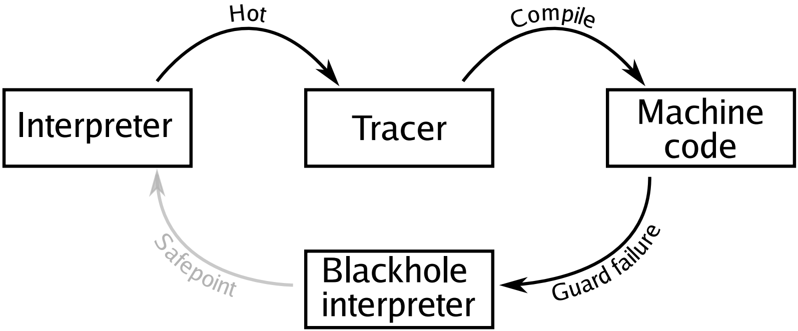
Meta-tracer states



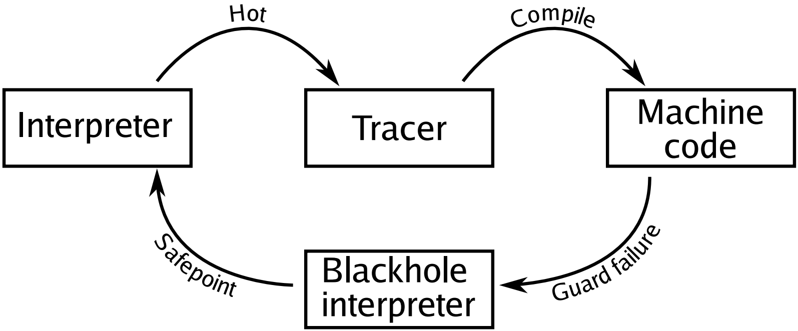
Meta-tracer states



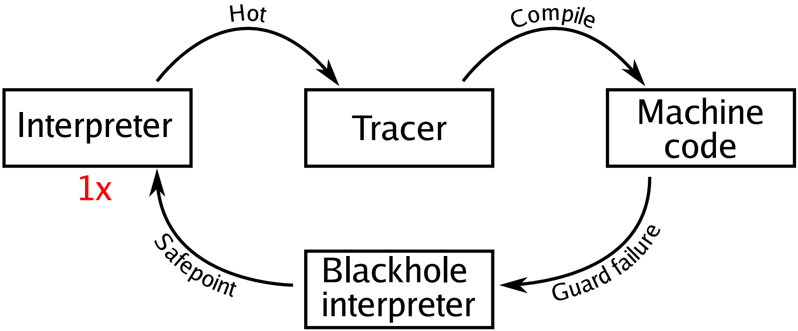
Meta-tracer states



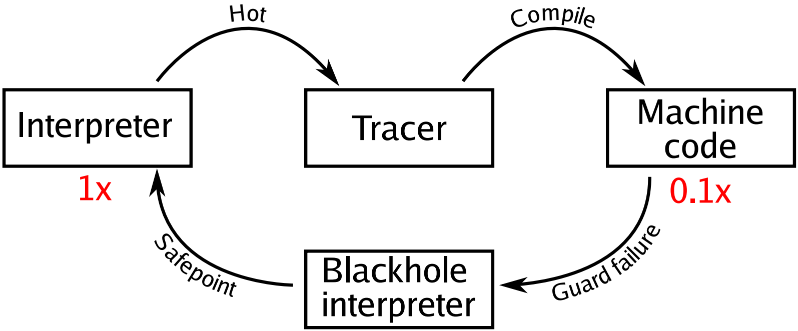
Meta-tracer states



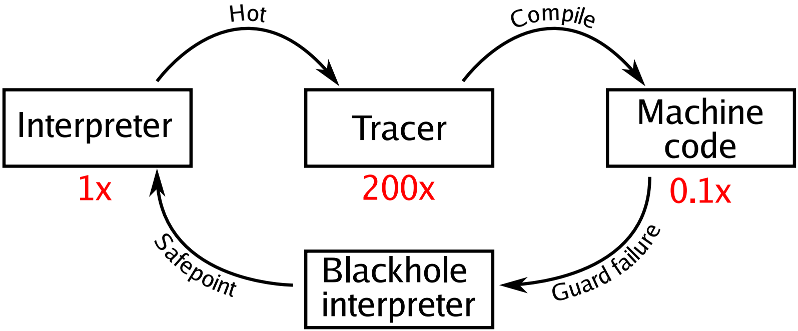
Meta-tracer performance (now)



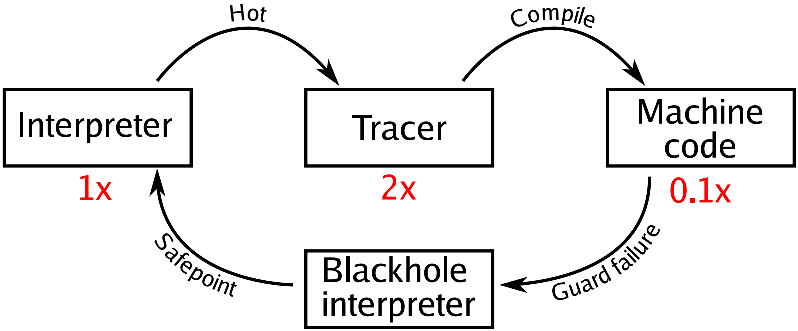
Meta-tracer performance (now)



Meta-tracer performance (now)



Meta-tracer performance (our aim)



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

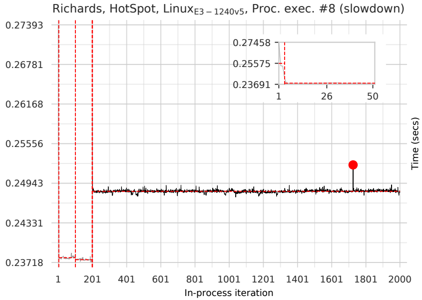
Specialising Dynamic Techniques for Implementing the Ruby Programming Language
C. Seaton (Chapter 4)

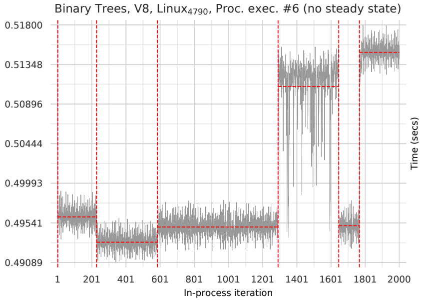
Quantifying performance changes with effect size confidence intervals
T. Kalibera and R. Jones

Thanks

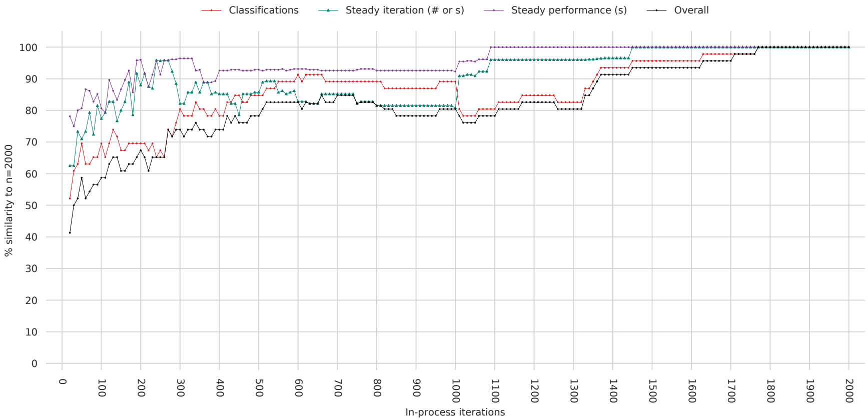
- EPSRC: *COOLER* and *Lecture*.
- Oracle.
- Cloudflare.

Thanks for listening

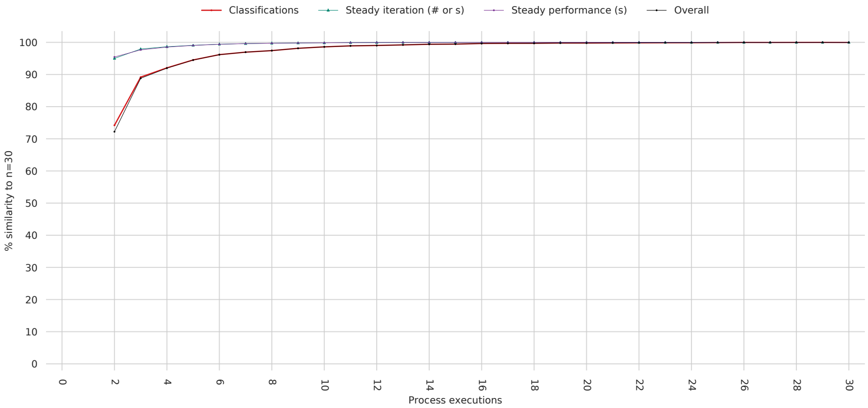




How long to run things for (0.8)



How long to run things for (0.8)



Diffing results (0.8 → 1.5)

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
binarytrees	Graal	⤨ (27⅂, 2⅃, 1⇝)			
fannkuch_redux		⤨ (26⅂, 4⇝)			
fasta		⅂	4.0 (3.0,34.4)	0.75 (0.535,5.873)	0.16188 ±0.000738
nbody		⅂	6.0 δ=−2.0 (5.0,7.0)	0.86 δ=−0.352 (0.704,1.090)	0.13677 δ=+0.00343 ±0.000035
richards		⅂	2.0 (2.0,35.3)	0.95 (0.879,9.745)	0.26465 ±0.007761
spectralnorm		⅃	14.0 (2.0,94.6)	13.60 (0.830,98.737)	1.05685 δ=+0.16392 ±0.000126