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

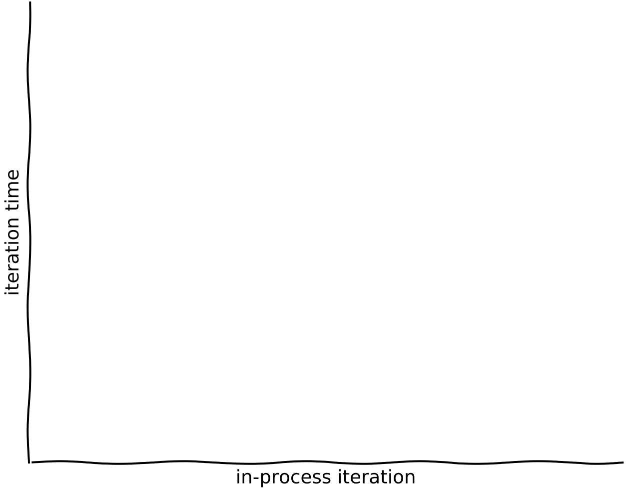
Software Development Team

2018-01-22

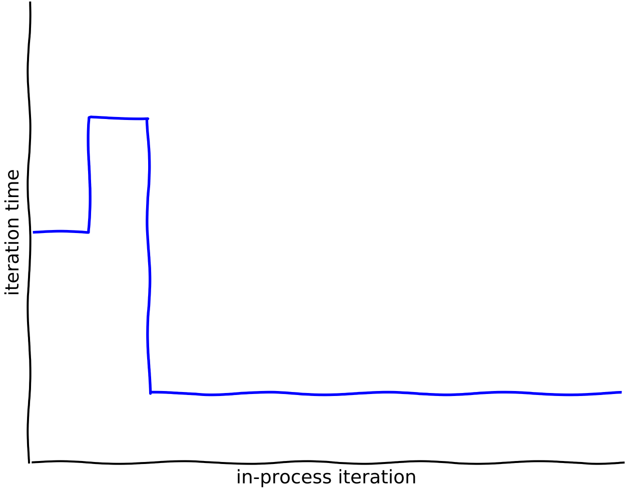
JVMs bring "gcc -O2" to the masses

–*Cliff Click*: A JVM does that?

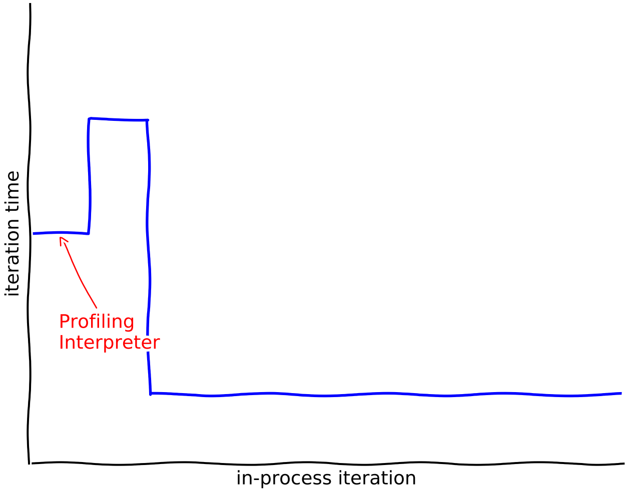
What do VM claims pertain to?



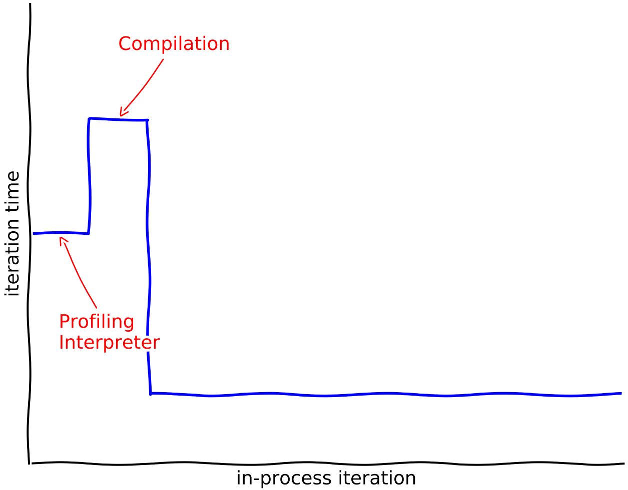
What do VM claims pertain to?



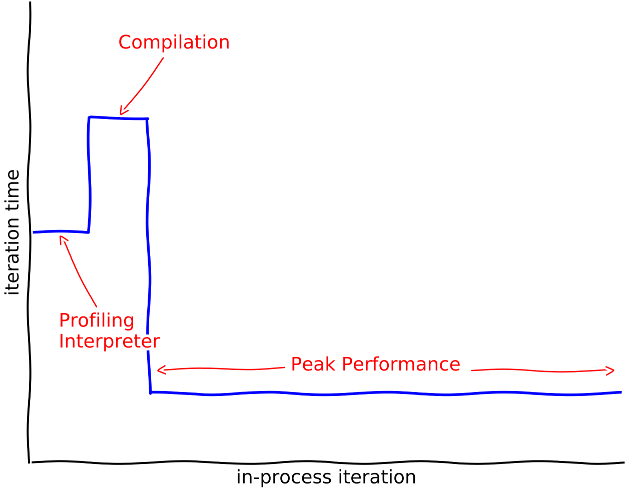
What do VM claims pertain to?



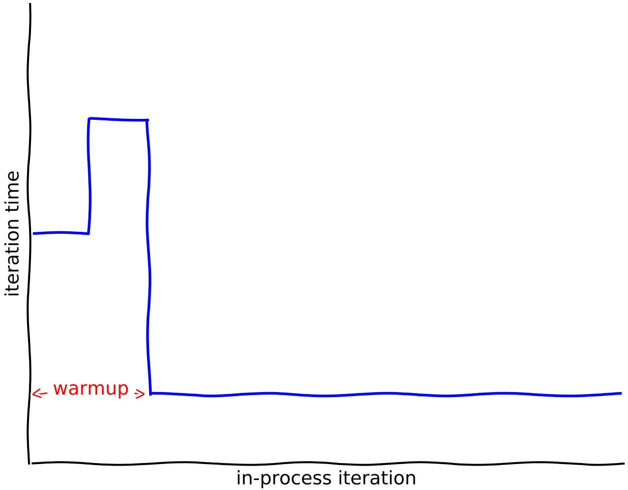
What do VM claims pertain to?



What do VM claims pertain to?



What do VM claims pertain to?



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

Measure warmup of modern language implementations

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

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

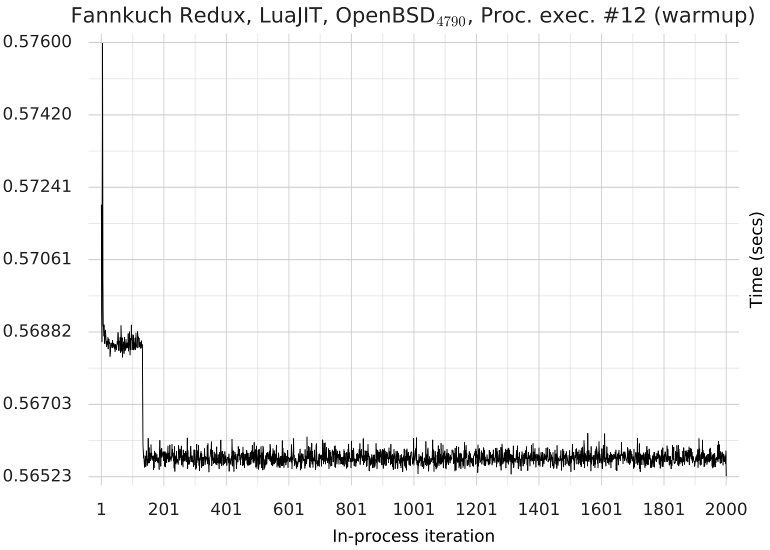
Benchmark runner: tries to control as many confounding variables as possible

Method 5: Krun

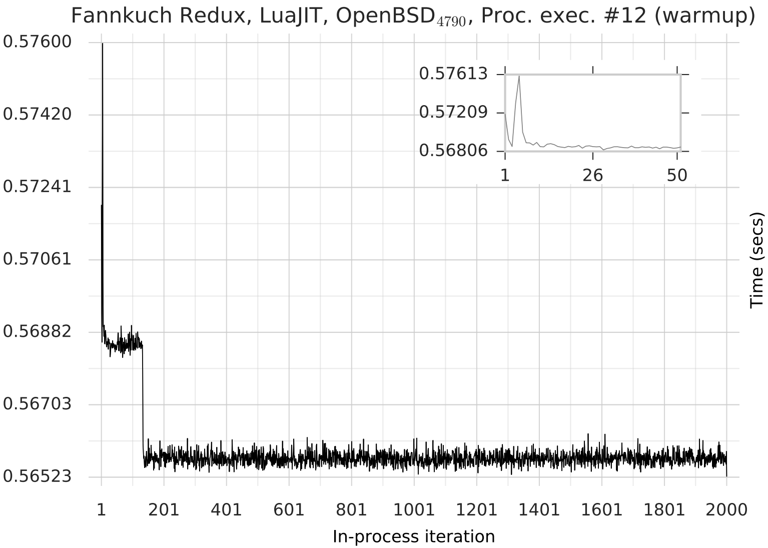
Benchmark runner: tries to control as many confounding variables as possible e.g.:

- 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, ...)

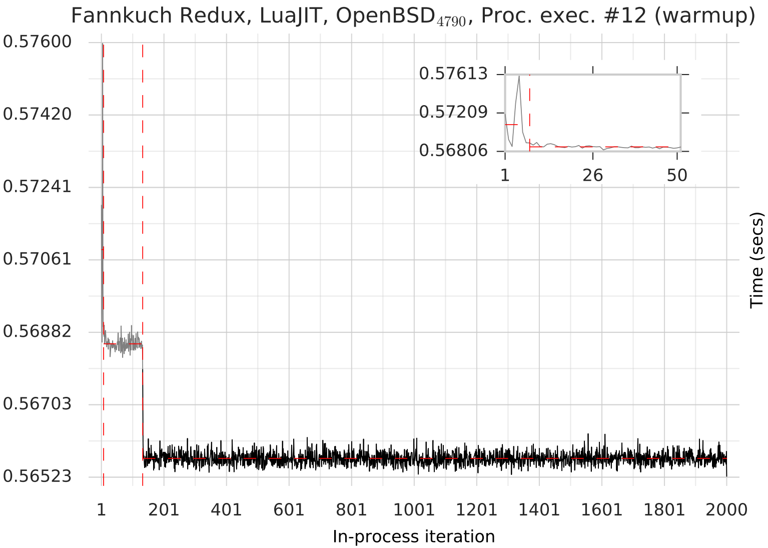
Warmup & flat (1)



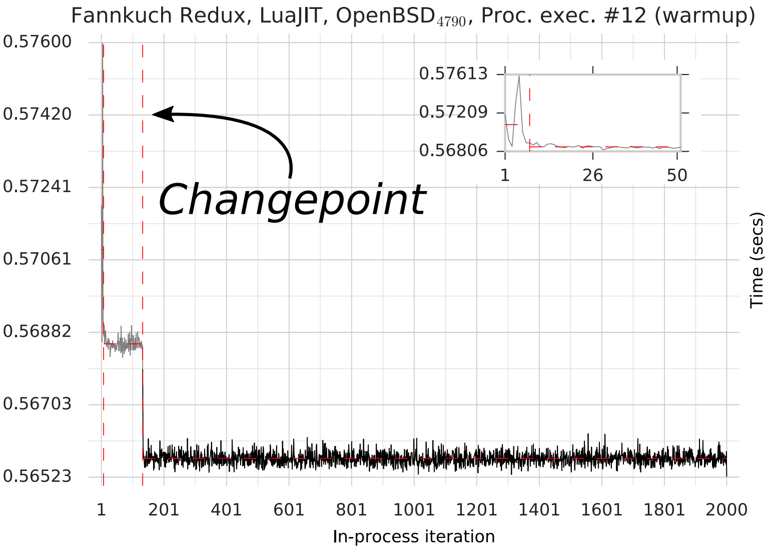
Warmup & flat (1)

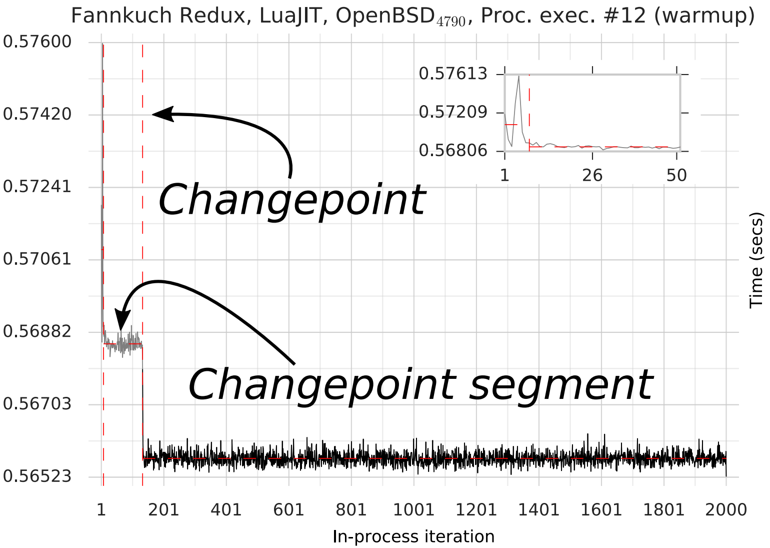


Warmup & flat (1)

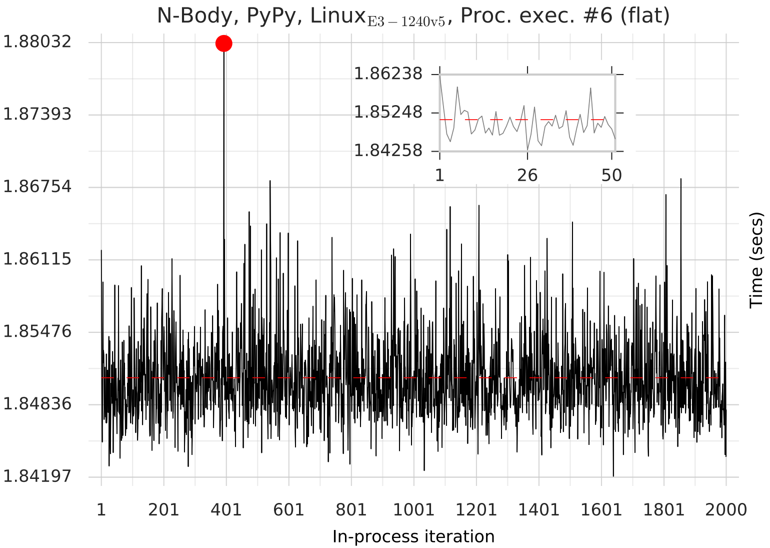


Warmup & flat (1)




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
Changepoint
Changepoint segment
Time (secs)
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration

Warmup & flat (1)



Classification algorithm (steps in order):

All segs are equivalent: *flat*

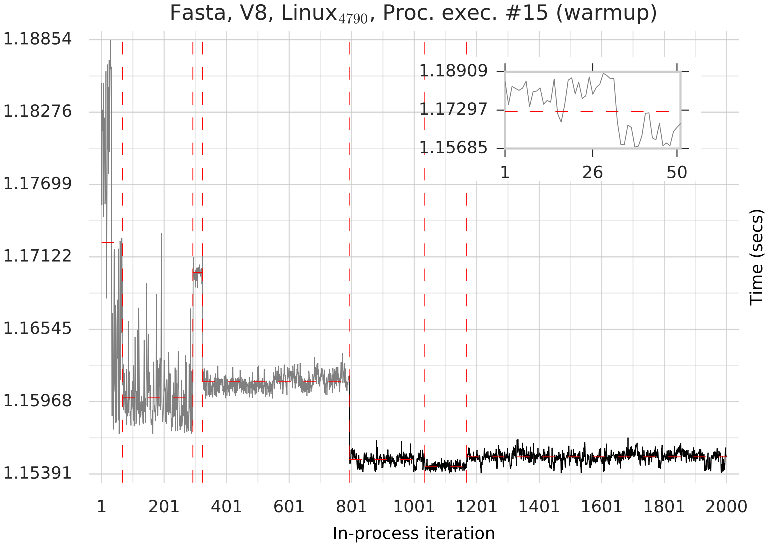
Method 7: Classification

Classification algorithm (steps in order):

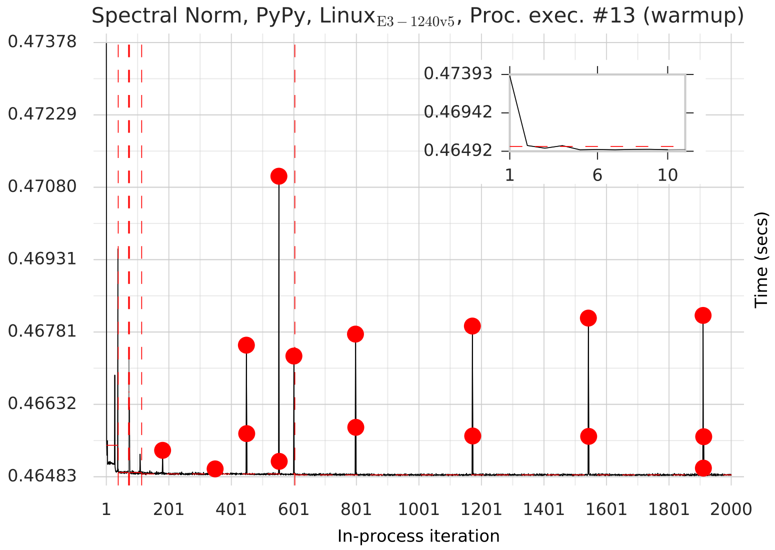
All segs are equivalent: *flat*

Final seg is in fastest set: *warmup*

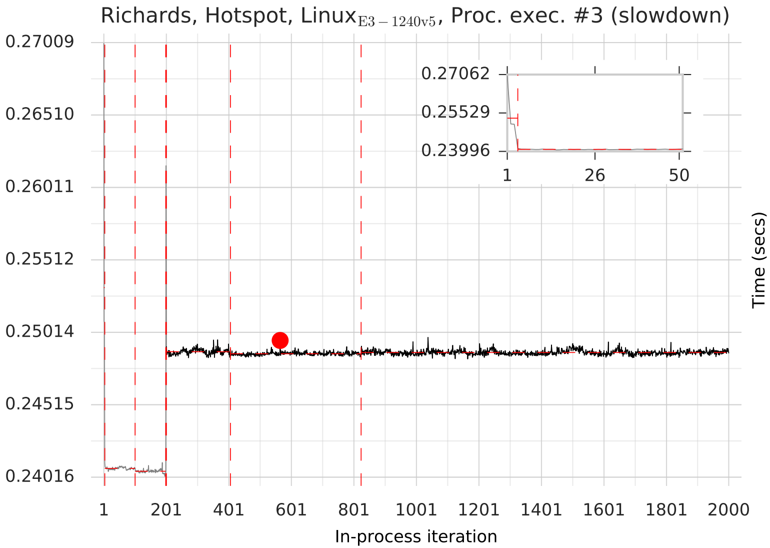
Warmup & flat (2)



Warmup & flat (2)



Slowdown (1)



Richards, Hotspot, Linux$_{\mathrm{E3-1240v5}}$, 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
Time (secs)
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration

Method 7: Classification

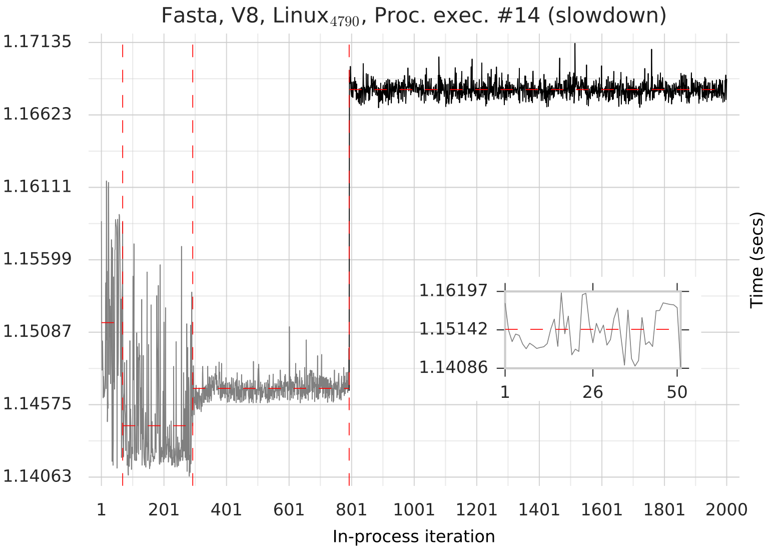
Classification algorithm (steps in order):

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

Classification algorithm (steps in order):

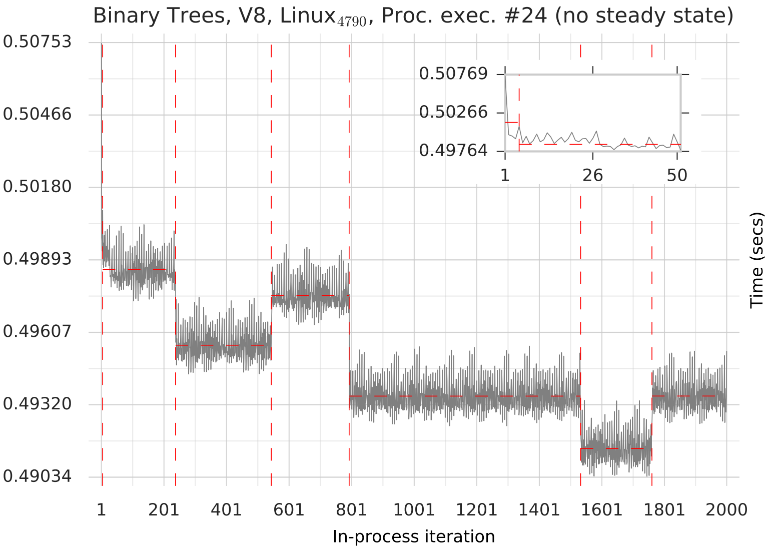
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*

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 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*

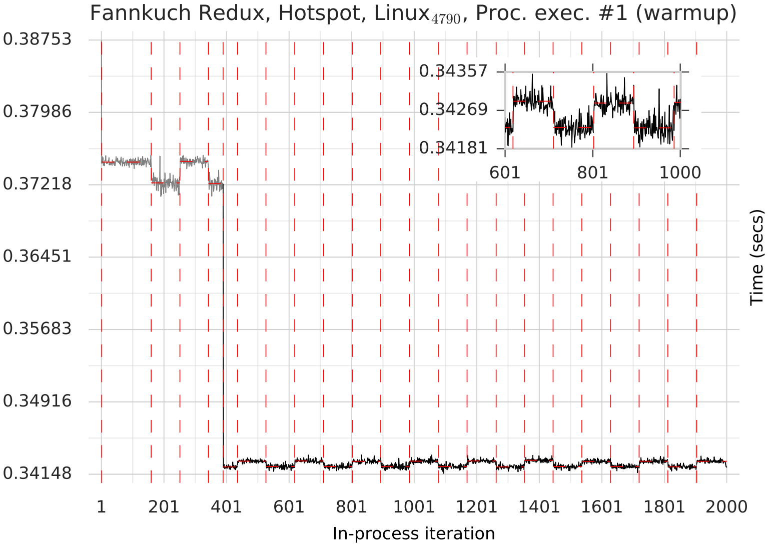
Good

Classification algorithm, in order:

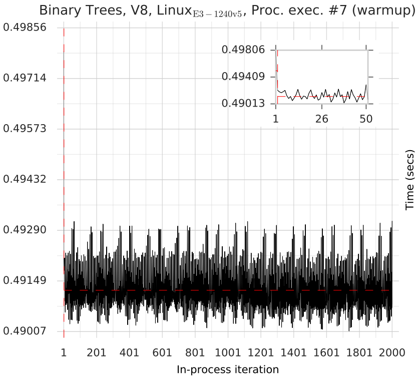
- All segs are equivalent: *flat*
- Final seg is in fastest set: *warmup*
- Final seg is not in fastest set: *slowdown*
- Else: *no steady state*

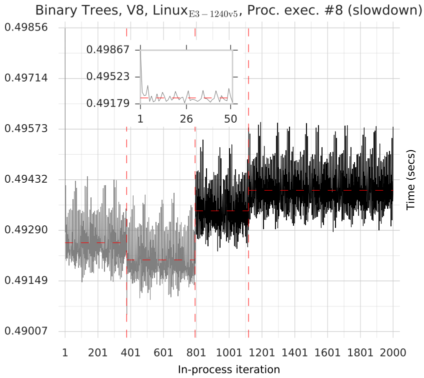
Bad

Warmup or no steady state?



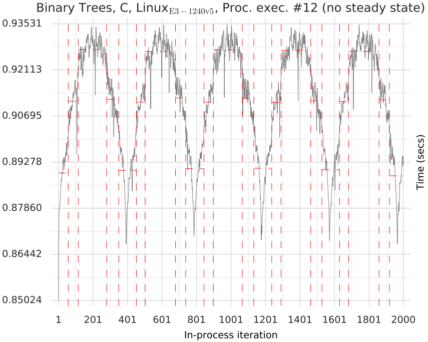
Inconsistent Process-executions

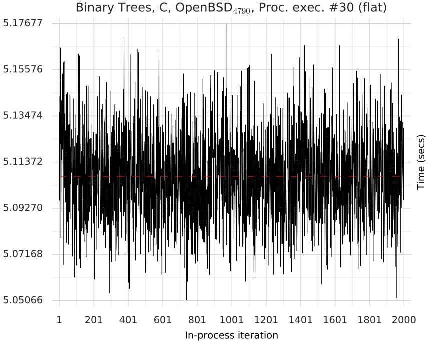




(Same machine)

Inconsistent Process-executions





(Different machines. Bouncing ball Linux-specific)

Individual benchmark stats

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

Classical warmup occurs for only:

72.4%–74.7% of process executions

Summary

Classical warmup occurs for only:

72.4%–74.7% of process executions

43.4%–43.5% of (VM, benchmark) pairs

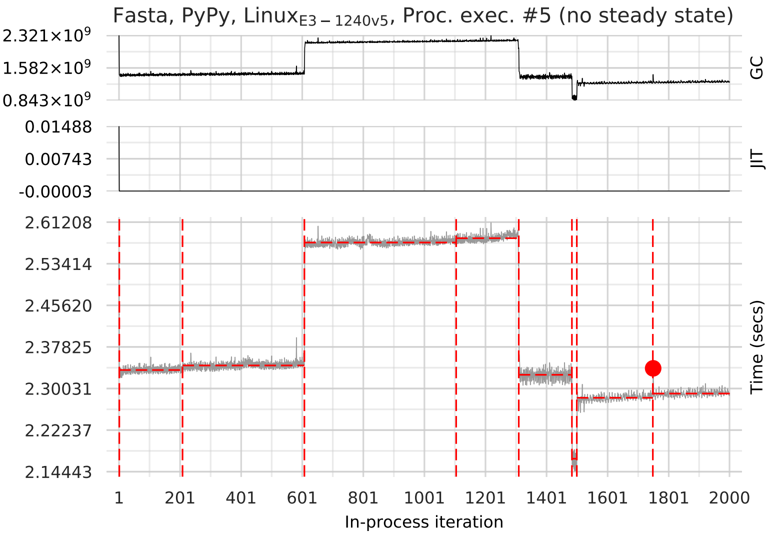
Classical warmup occurs for only:

72.4%–74.7% of process executions

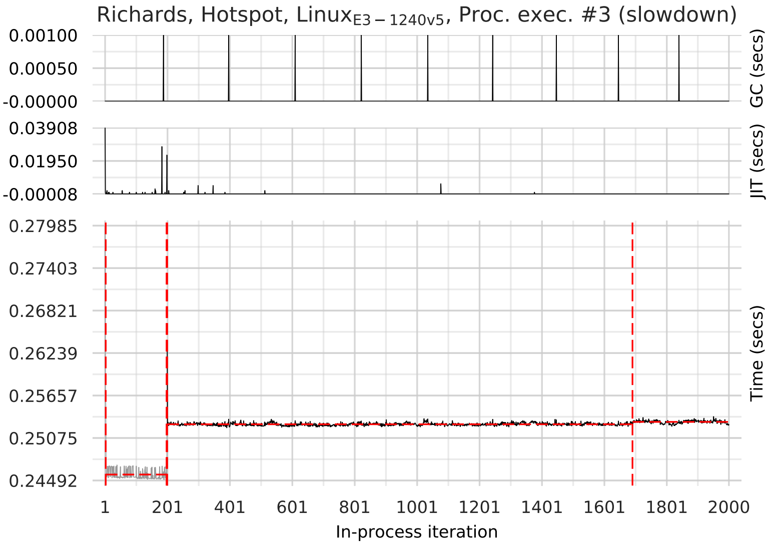
43.4%–43.5% of (VM, benchmark) pairs

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

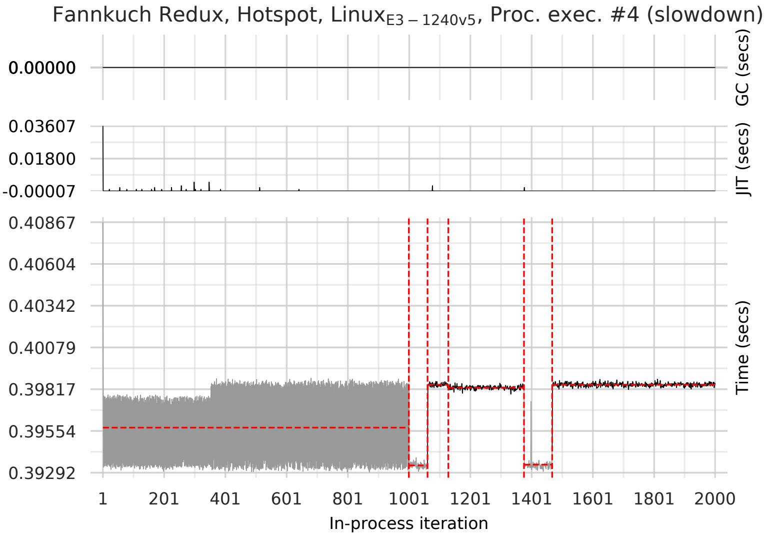
Are odd effects correlated with compilation and GC?



Are odd effects correlated with compilation and GC?



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

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

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)
```

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

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)

Benchmarks guide our optimisations

Benchmarks guide our optimisations

Are they correct guides?

Octane

17 JavaScript benchmarks from V8

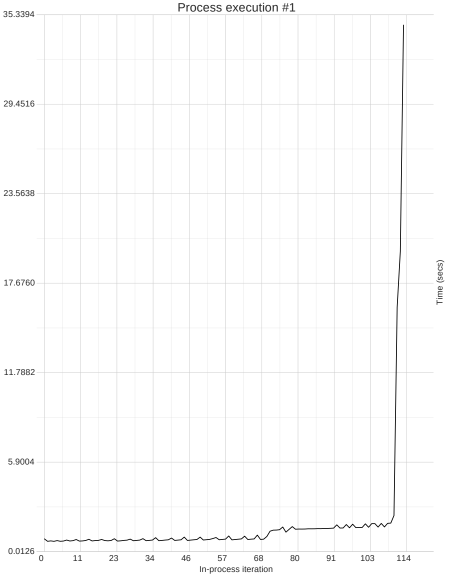
17 JavaScript benchmarks from V8

Let’s make each benchmark run for 2000 iterations

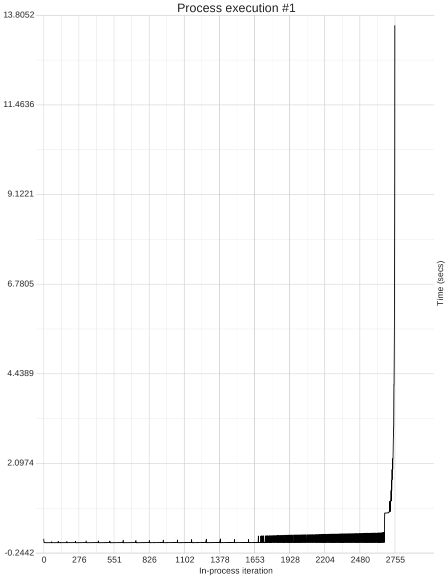
Octane: pdf.js explodes

```
$ 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
```

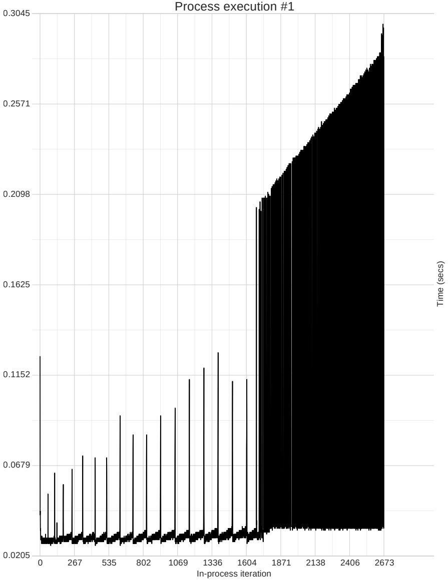
Octane: analysing pdf.js



Octane: analysing pdf.js



Octane: analysing pdf.js

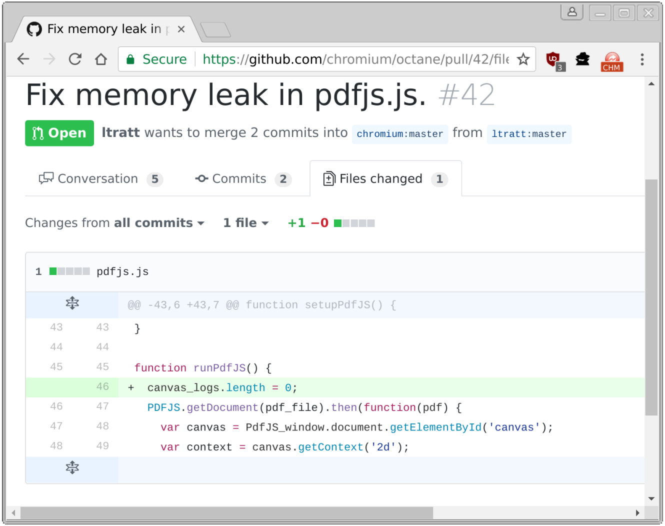


Octane: debugging

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

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.

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?

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
```

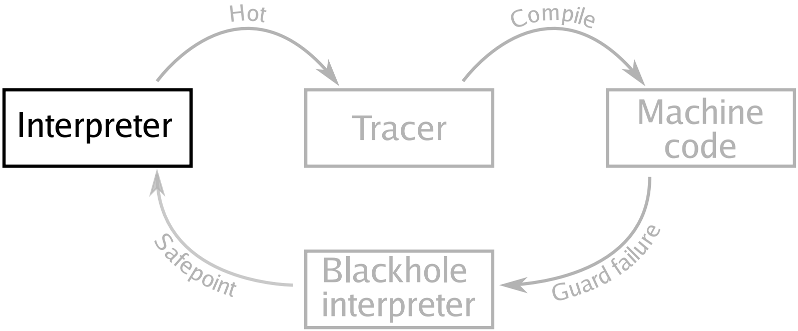
Meta-tracing JITs

FL Interpreter	Initial trace

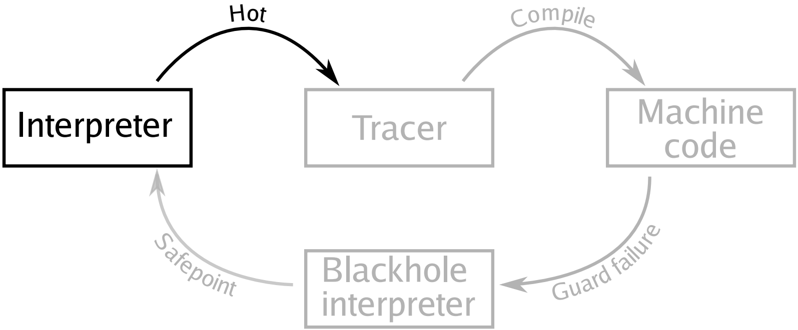
```
program_counter = 0; stack = []
vars = {...}
while True:
  jit_merge_point(program_counter)
  instr = load_instruction(program_counter)
  if instr == INSTR_VAR_GET:
    stack.push(
     vars[read_var_name_from_instruction()])
    program_counter += 1
  elif instr == INSTR_VAR_SET:
    vars[read_var_name_from_instruction()]
     = 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)
...
```

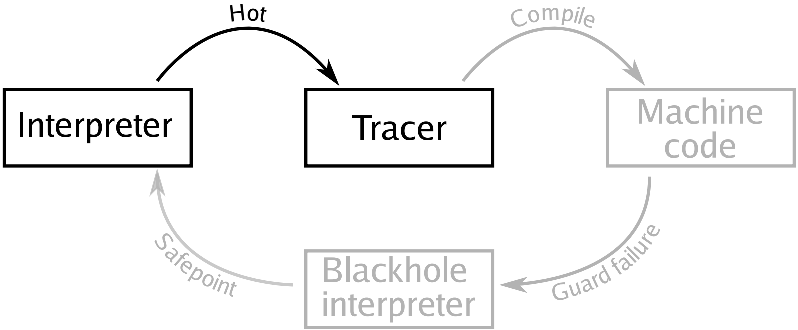
Meta-tracer states



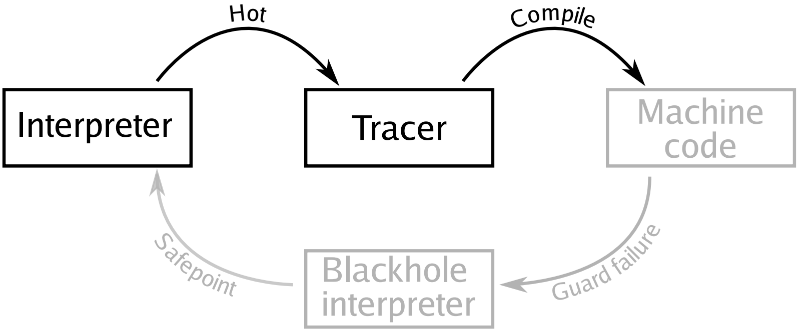
Meta-tracer states



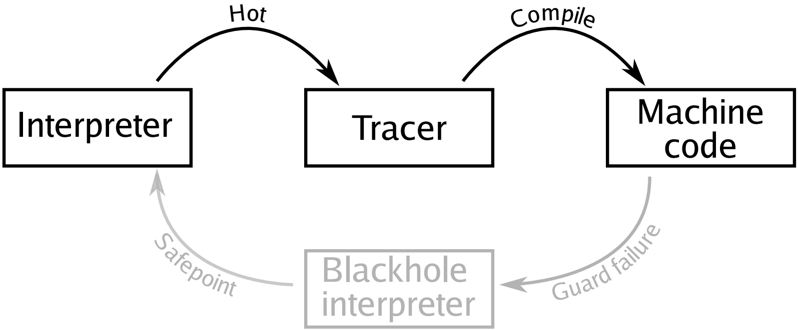
Meta-tracer states



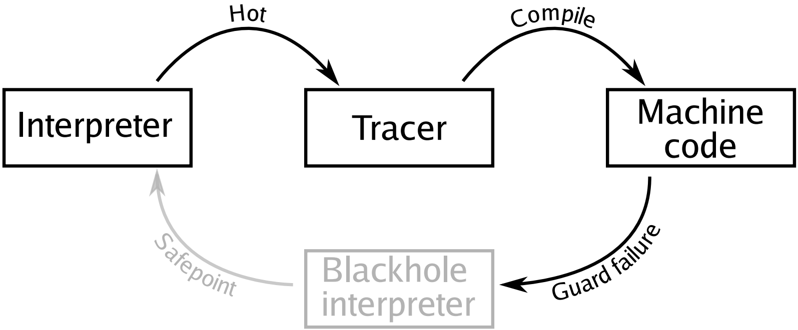
Meta-tracer states

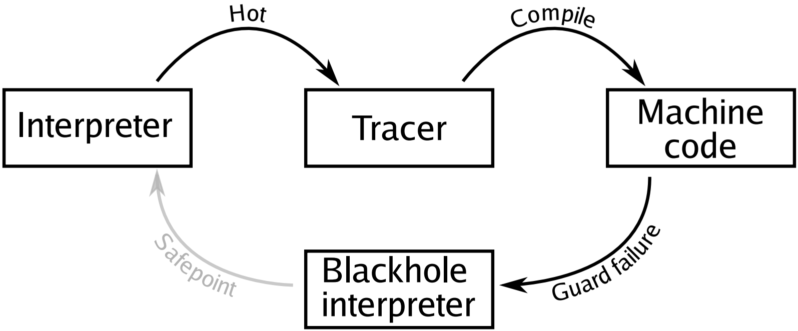


Meta-tracer states

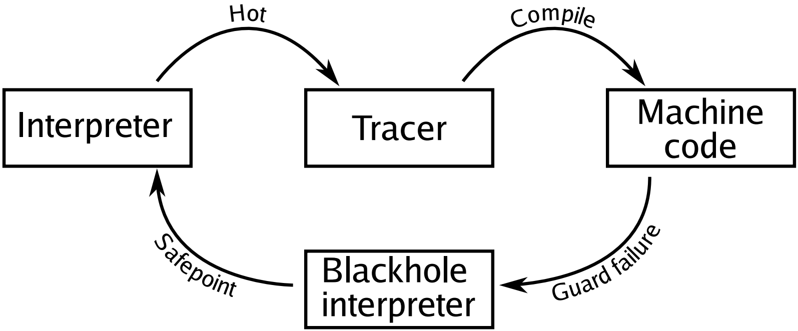


Meta-tracer states

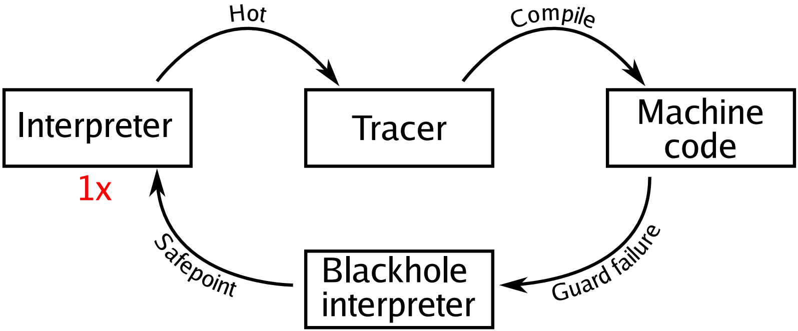



Hot
Compile
Interpreter
Tracer
Machine code
Blackhole interpreter
Guard failure
Safepoint

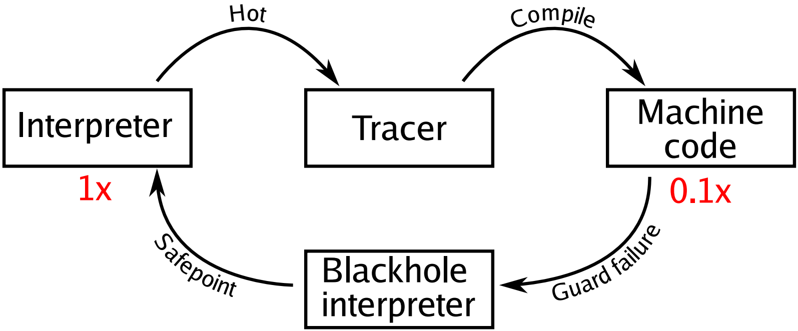
Meta-tracer states



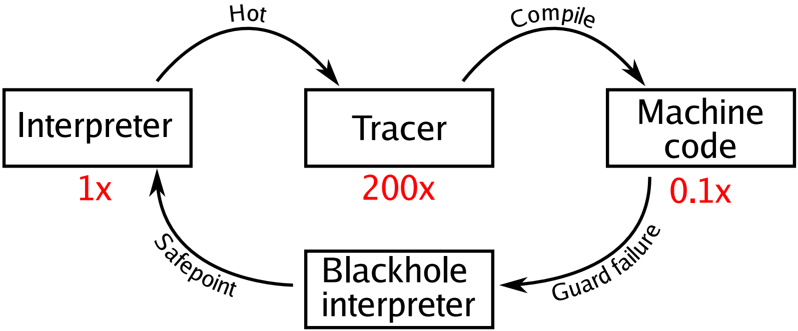
Meta-tracer performance (now)



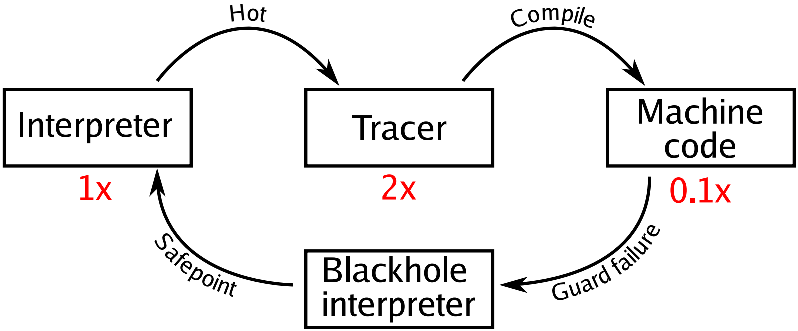
Meta-tracer performance (now)



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

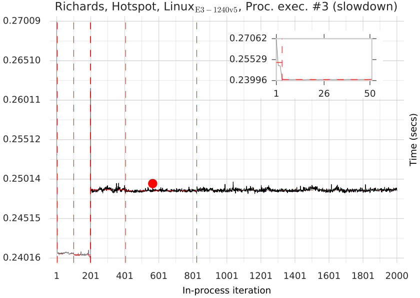
Specialising Dynamic Techniques for Implementing the Ruby Programming Language
C. Seaton (Chapter 4)

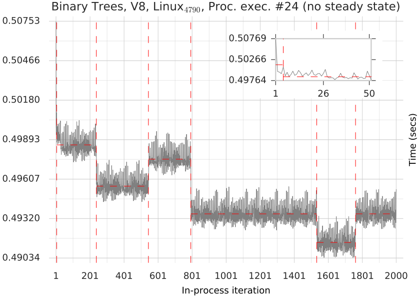
Quantifying performance changes with effect size confidence intervals
T. Kalibera and R. Jones

Thanks

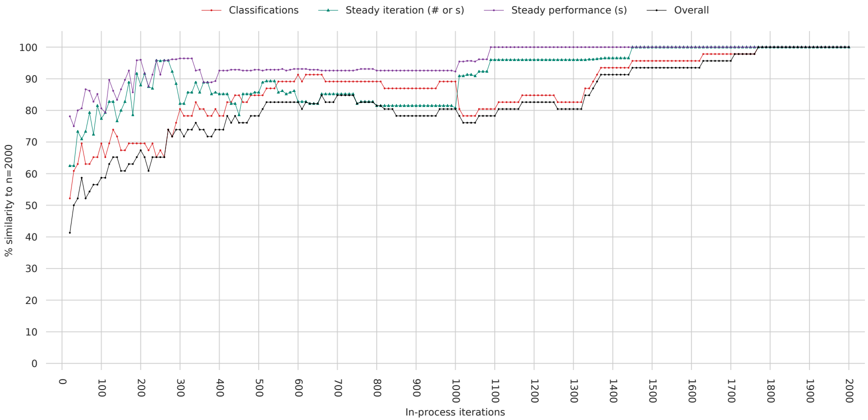
- EPSRC: *COOLER* and *Lecture*.
- Oracle.
- Cloudflare.

Thanks for listening





How long to run things for



How long to run things for