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

Laurence
Tratt

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

KING'S College LONDON

Software Development Team

2016-10-31

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    => ...
  };
}
```

VM claims

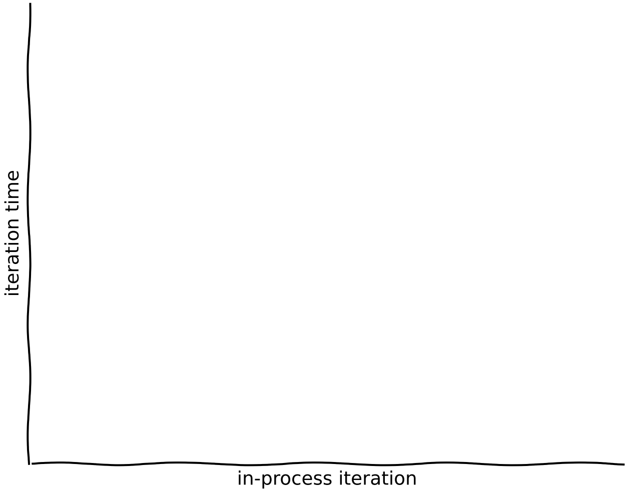
The best VMs are close in performance to,
and sometimes faster than,
gcc -O2

VM claims

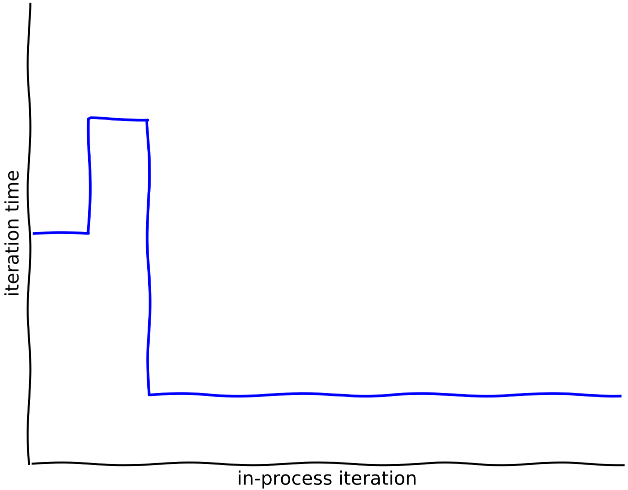
The best VMs are close in performance to,
and sometimes faster than,
gcc -O2

What’s being measured?

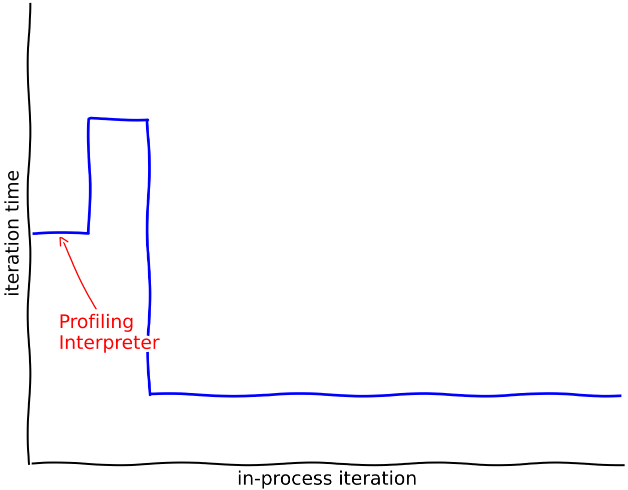
What our claims pertain to



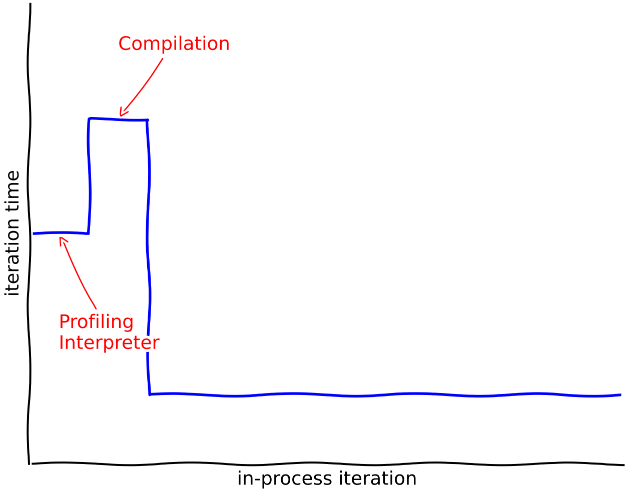
What our claims pertain to



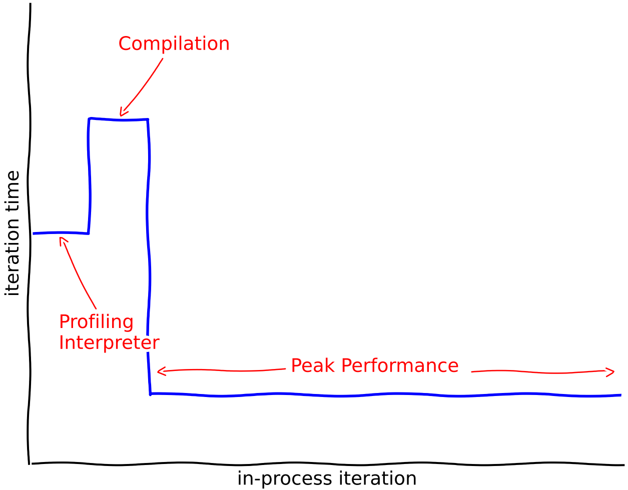
What our claims pertain to



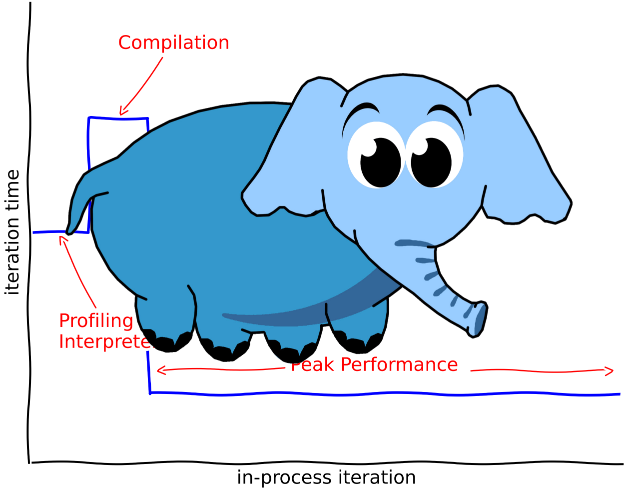
What our claims pertain to



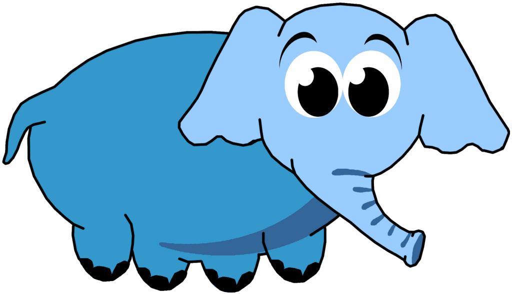
What our claims pertain to



What our claims pertain to

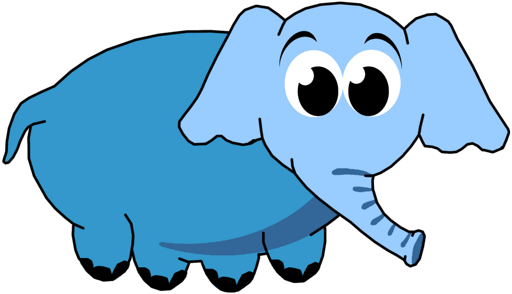


What our claims pertain to



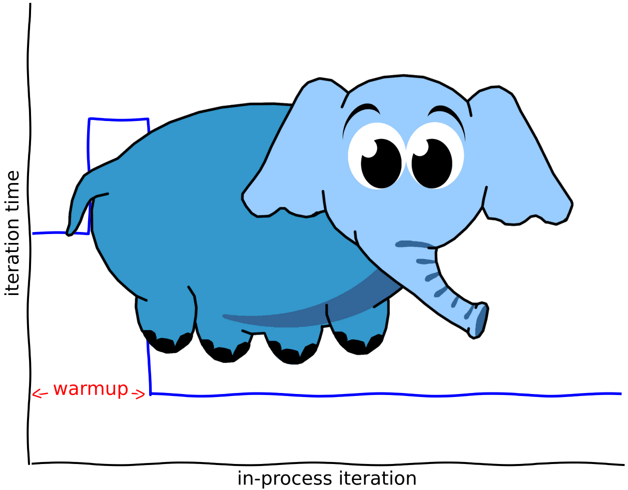
This is Barry

What our claims pertain to



This is Barry: the benchmarking elephant in the room

What our claims pertain to



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 exhibit classical warmup behaviour

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*

10 *process executions*

Method 3: VMs

- Graal-0.13
- HHVM-3.12.0
- JRuby/Truffle (git #f82ac771)
- Hotspot-8u72b15
- LuaJit-2.0.4
- PyPy-4.0.1
- V8-4.9.385.21
- GCC-4.9.3

Note: same GCC (4.9.3) used for all compilation

Method 4: Machines

- Linux-Debian8/i4790K, 24GiB RAM
- Linux-Debian8/i4790, 32GiB RAM
- OpenBSD-5.8/i4790, 32GiB RAM

Method 4: Machines

- Linux-Debian8/i4790K, 24GiB RAM
- Linux-Debian8/i4790, 32GiB RAM
- OpenBSD-5.8/i4790, 32GiB RAM

- Turbo boost and hyper-threading disabled
- SSH blocked from non-local machines
- Daemons disabled (cron, smtpd)

Method 5: Krun

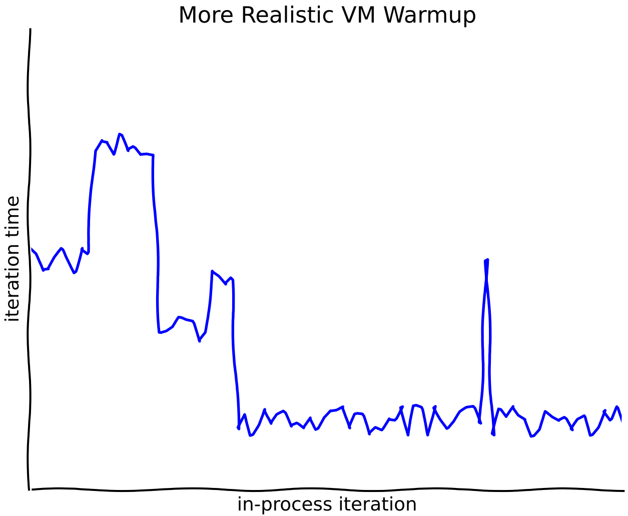
Benchmark runner: tries to control as many confounding variables as possible

Method 5: Krun

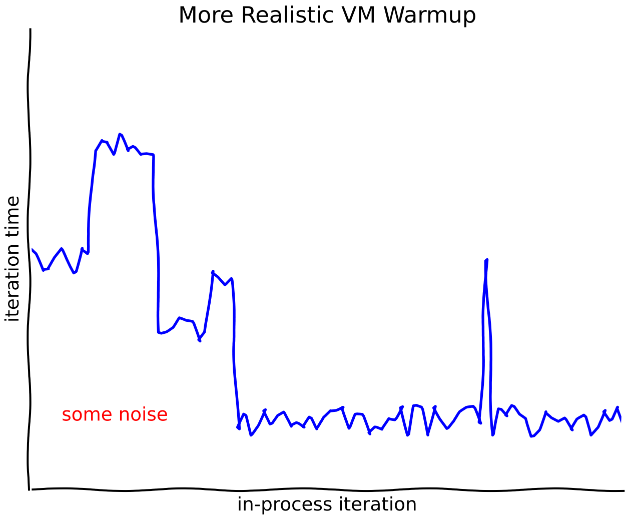
Benchmark runner: tries to control as many confounding variables as possible e.g.:

- 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, ...)

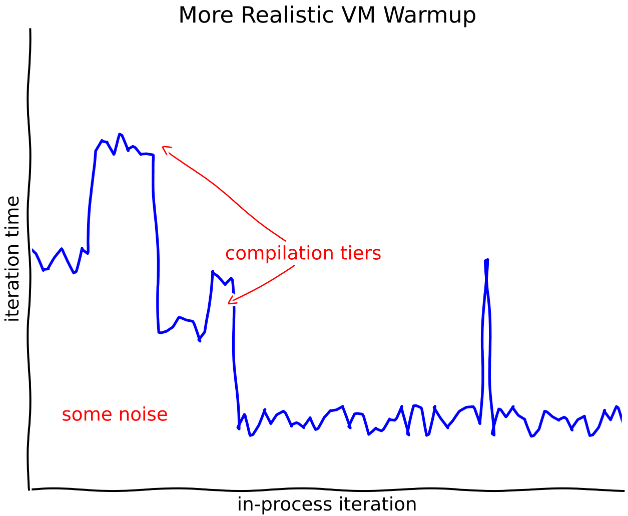
Method 6: Expectations



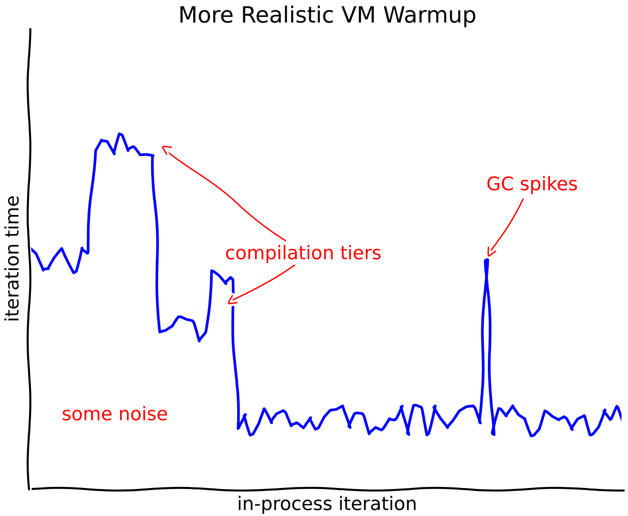
Method 6: Expectations



Method 6: Expectations

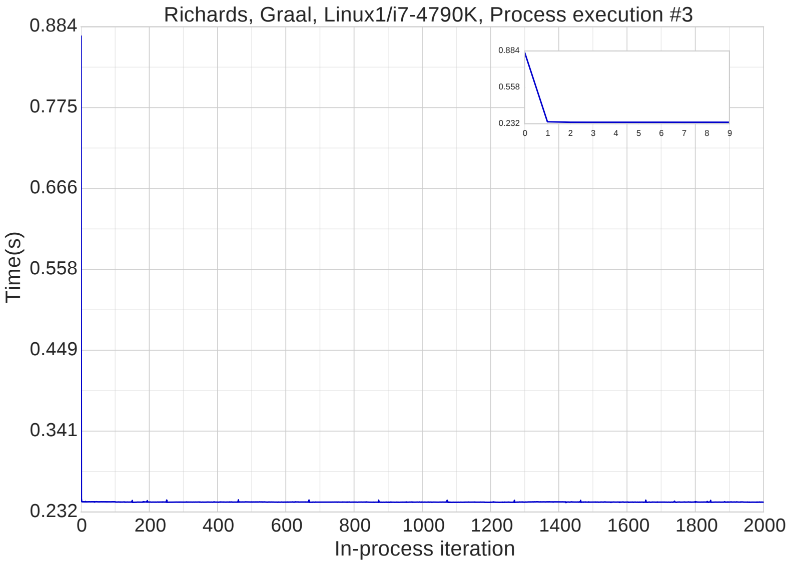


Method 6: Expectations

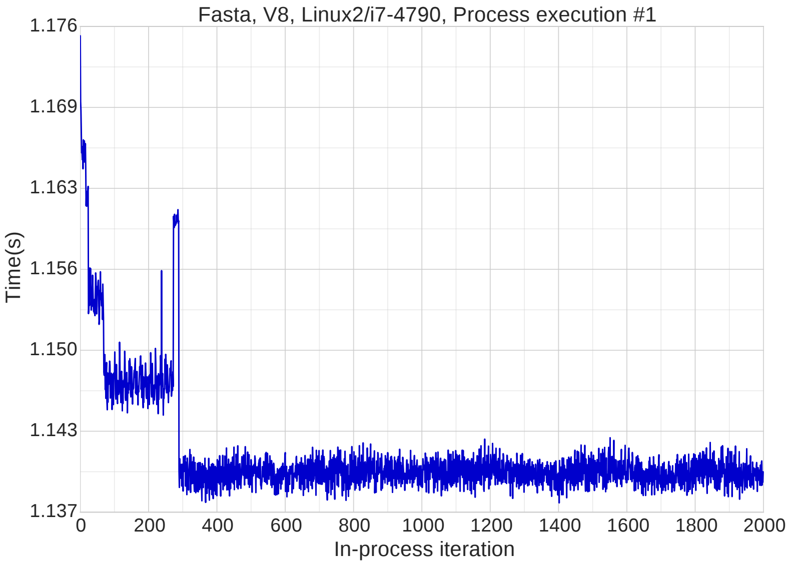


Preliminary results

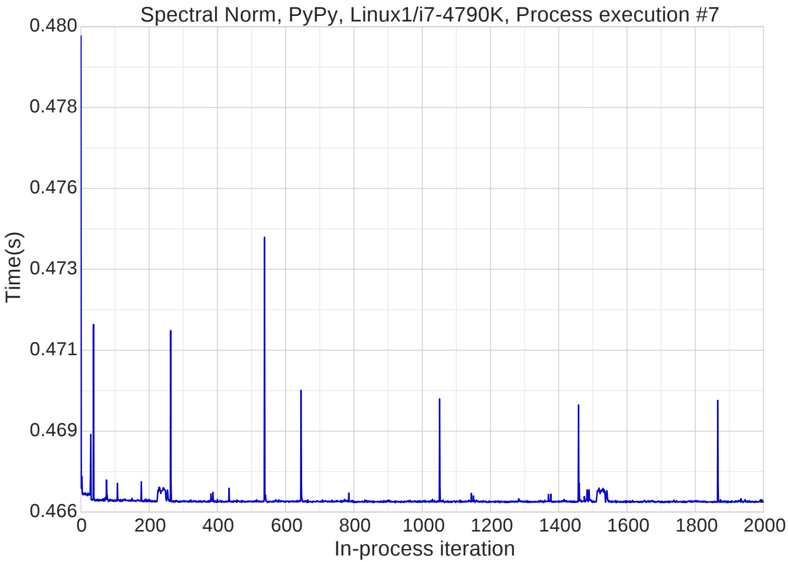
Classical Warmup



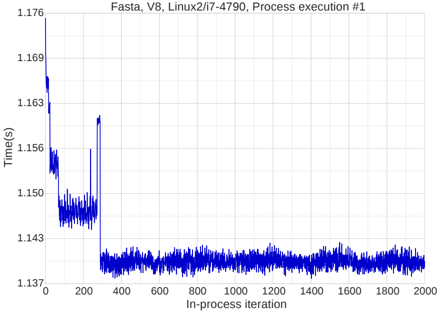
Classical Warmup



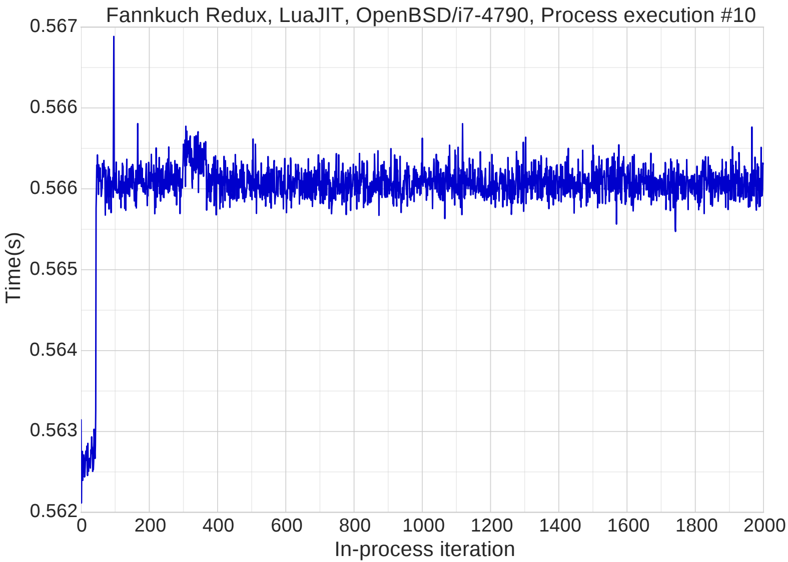
Classical Warmup



Classical Warmup across machines



Slowdown

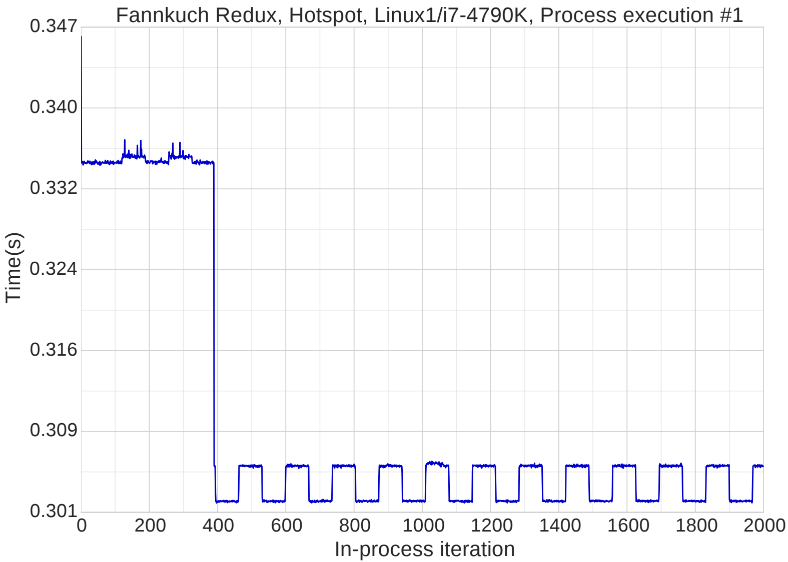


Fannkuch Redux, LuaJIT, OpenBSD/i7-4790, Process execution #10
Time(s)
0.567
0.566
0.566
0.565
0.564
0.563
0.562
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration

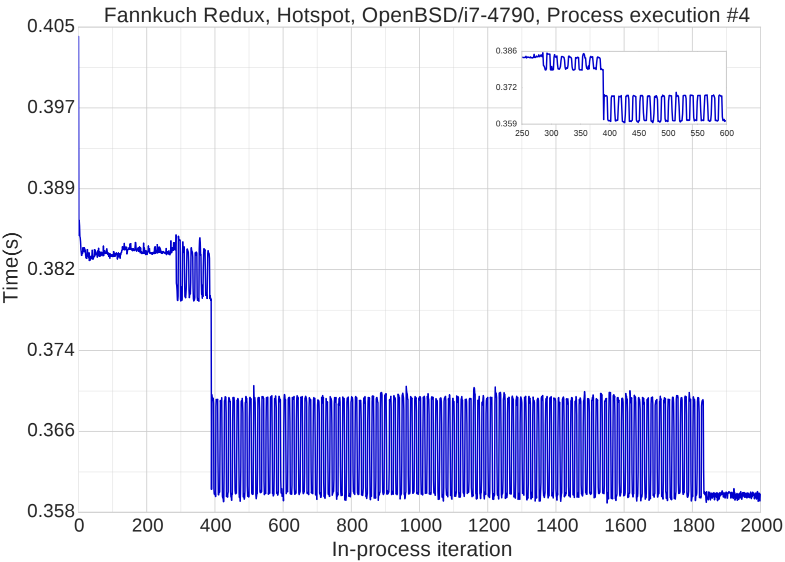
Slowdown

Richards, Hotspot, Linux2/i7-4790, Process execution #2
0.298
0.293
0.287
0.282
0.276
0.271
0.266
Time(s)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration

Cycles

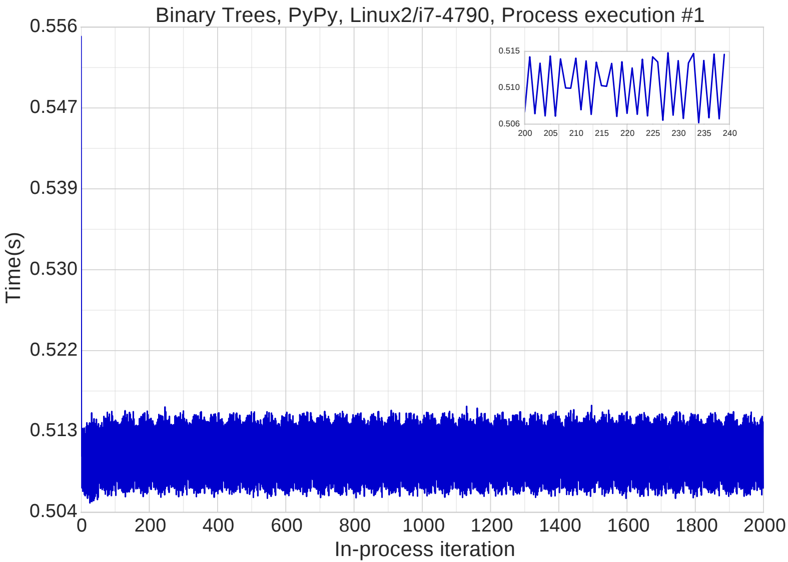

Fannkuch Redux, Hotspot, Linux1/i7-4790K, Process execution #1
Time(s)
0.347
0.340
0.332
0.324
0.316
0.309
0.301
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration

Cycles



Fannkuch Redux, Hotspot, OpenBSD/i7-4790, Process execution #4
0.405
0.397
0.389
0.382
0.374
0.366
0.358
Time(s)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration
0.386
0.372
0.359
250
300
350
400
450
500
550
600

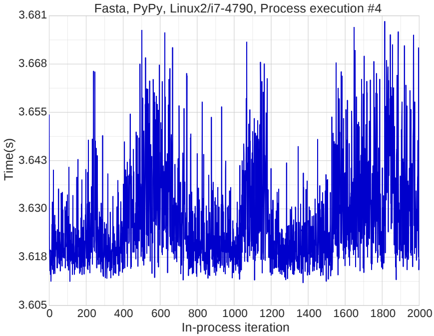
Cycles



Binary Trees, PyPy, Linux2/i7-4790, Process execution #1
0.556
0.547
0.539
0.530
0.522
0.513
0.504
Time(s)
0
200
400
600
800
1000
1200
1400
1600
1800
2000
In-process iteration
0.515
0.510
0.506
200
205
210
215
220
225
230
235
240

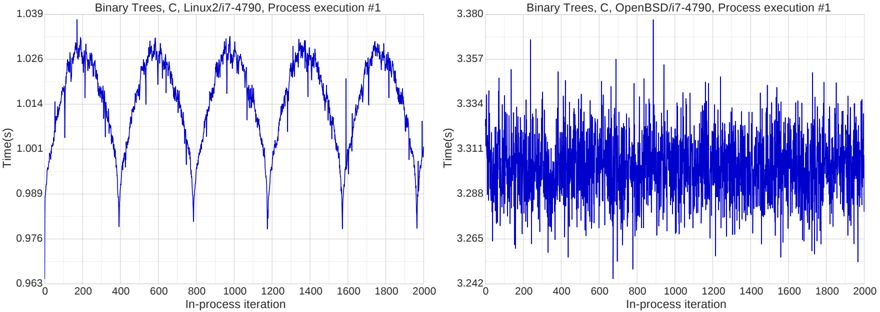
Never-ending Phase Changes

Inconsistent Process-executions



(Note: same machine)

Inconsistent Process-executions



(Note: different machines. Bouncing ball pattern Linux-specific)

Classical warmup occurs for only:

Classical warmup occurs for only:

50% of process executions

Summary

Classical warmup occurs for only:

50% of process executions

25% of (VM, benchmark) pairs

Summary

Classical warmup occurs for only:

50% of process executions

25% of (VM, benchmark) pairs

0% of benchmarks for all VMs, machines & proc execs.

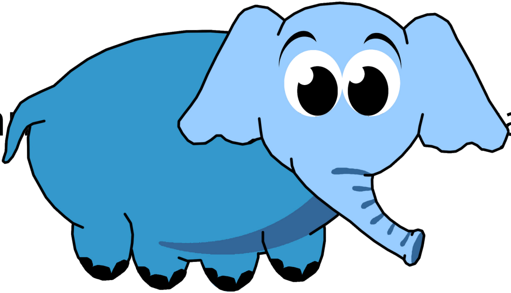
Benchmark suites

Benchmark suites

Benchmarks guide our optimisations

Benchmark suites

Bench



ations

Benchmark suites

Benchmarks guide our optimisations

Are they complete guides?

A war story

Symptom: poor performance of a Pyston benchmark on PyPy

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

A war story: conclusion

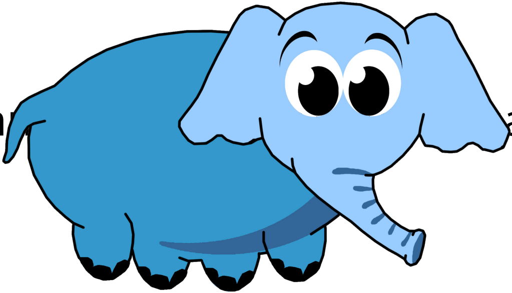
The benchmark suite said 7 levels, so that’s what I suggested

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

Benchations

Benchmarks guide our optimisations

Are they correct guides?

Octane

17 JavaScript benchmarks from V8

Octane

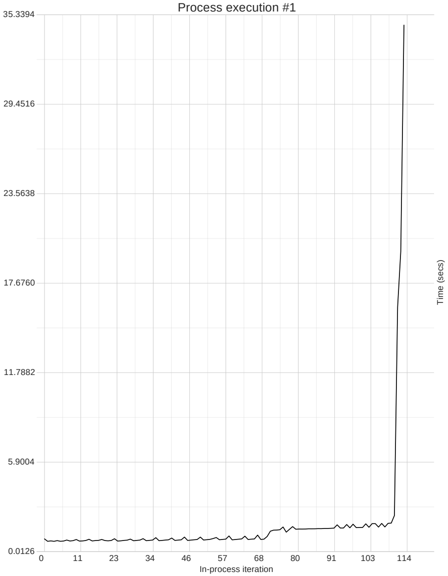
17 JavaScript benchmarks from V8

Let's make each benchmark run for 2000 iterations

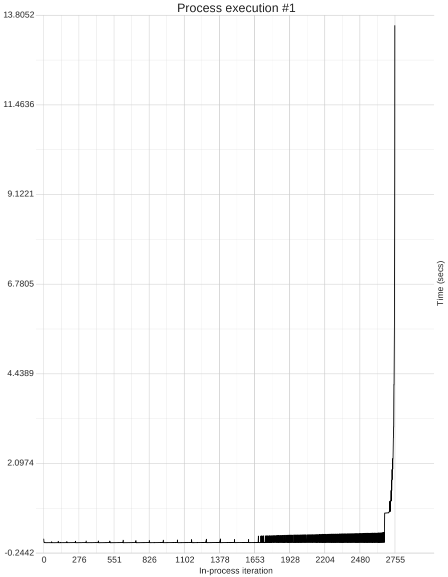
Octane: pdf.js explodes

```
$ 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
```

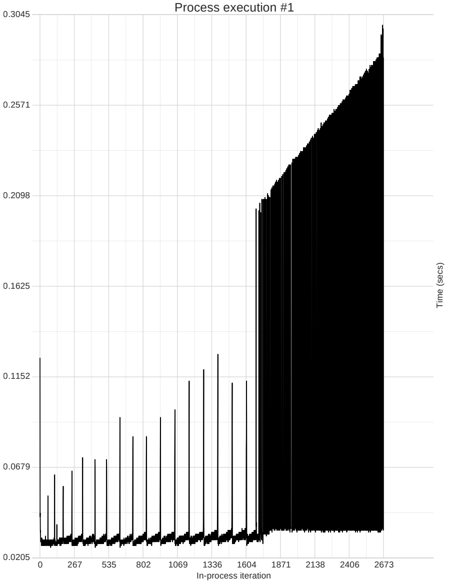
Octane: analysing pdf.js



Octane: analysing pdf.js



Octane: analysing pdf.js



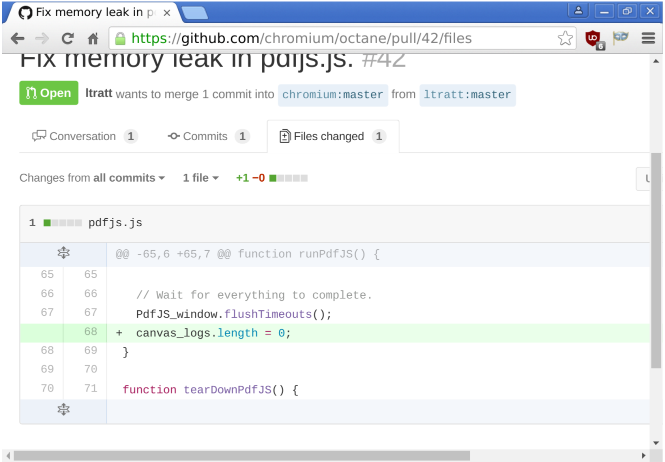
Octane: debugging

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

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

Summary

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

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

What we can do

What we can do

1. Run benchmarks for longer to uncover issues.

What we can do

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

What we can do

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

What we can do

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.

What we can do

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.
5. The more benchmarks, the better.

What we can do

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.
5. The more benchmarks, the better.
6. Focus on predictable performance.

References

VM Warmup Blows Hot and Cold
E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount and L. Tratt.

Rigorous Benchmarking in Reasonable Time
T. Kalibera and R. Jones

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

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

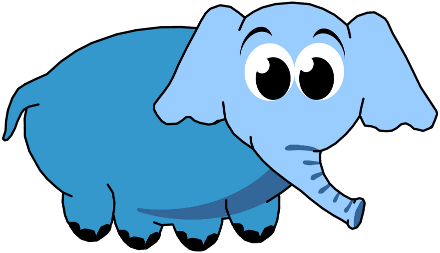
Thanks

- 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