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

Software Development Team
2017-05-27
What to expect from this talk

The gap
What to expect from this talk

What we tell users to expect

The gap
What to expect from this talk

What we tell users to expect

The gap

What users experience
What to expect from this talk

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?

```rust
fn f(a: Option<i64>) {
    match a {
        Some(i) => ..., 
        None => ...
    }
}
```
Is this just about ‘dynamic typing’?

Is this statically or dynamically typed? Both!

```rust
fn f(a: Option<i64>) {
    match a {
    Some(i) => ...,  
    None => ...,  
    }
}
```
The best VMs are close in performance to, and sometimes faster than,
gcc -O2
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

![Graph showing iteration time vs. in-process iteration with labels for Compilation and Profiling Interpreter.]
What our claims pertain to

![Graph showing compilation and profiling interpreter with peak performance on in-process iteration time.

Compilation
Profiling Interpreter
Peak Performance

iteration time

in-process iteration

http://soft-dev.org/
What our claims pertain to

![Graph](http://soft-dev.org/)

- Compilation
- Profiling Interpreters
- Peak Performance

iteration time vs. in-process iteration
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
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\textsubscript{4790}, Debian 8, 24GiB RAM
- Linux\textsubscript{E3-1240v5}, Debian 8, 32GiB RAM
- OpenBSD\textsubscript{4790}, OpenBSD 6.0, 32GiB RAM
Method 4: Machines

- Linux\textsubscript{4790}, Debian 8, 24GiB RAM
- Linux\textsubscript{E3-1240v5}, Debian 8, 32GiB RAM
- OpenBSD\textsubscript{4790}, OpenBSD 6.0, 32GiB RAM

- Turbo boost and hyper-threading disabled
- Network card turned off.
- Daemons disabled (cron, smtpd)
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

More Realistic VM Warmup

iteration time

in-process iteration
Method 6: Expectations

More Realistic VM Warmup

iteration time

in-process iteration

some noise
Method 6: Expectations

More Realistic VM Warmup

iteration time

some noise

compilation tiers

in-process iteration
Method 6: Expectations

More Realistic VM Warmup

- Some noise
- Compilation tiers
- GC spikes
Fannkuch Redux, LuaJIT, OpenBSD, Proc. exec. #12 (warmup)
Fannkuch Redux, LuaJIT, OpenBSD, Proc. exec. #12 (warmup)
Fannkuch Redux, LuaJIT, OpenBSD, Proc. exec. #12 (warmup)
Fannkuch Redux, LuaJIT, OpenBSD\textsuperscript{4790}, Proc. exec. #12 (warmup)

\textbf{Changepoint}
Changepoint

Changepoint segment
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
Fasta, V8, Linux, Proc. exec. #15 (warmup)

In-process iteration

Time (secs)

1.15391
1.15968
1.16545
1.17122
1.17699
1.18276
1.18854
1.15685
1.17297
1.18909

18/45 HTTP://SOFT-DEV.ORG/
Spectral Norm, PyPy, Linux_{E3 − 1240v5}, Proc. exec. #13 (warmup)

Time (secs)

In-process iteration

0.46483
0.46632
0.46781
0.46931
0.47080
0.47229
0.47378

0.46492
0.46942
0.47393
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*
In-process iteration analysis of Fasta, V8, Linux, Proc. exec. #14 (slowdown)

Time (secs)

In-process iteration

1.14063
1.14575
1.15087
1.15599
1.16111
1.16623
1.17135

1.14086
1.15142
1.16197

Fasta, V8, Linux, Proc. exec. #14 (slowdown)
In-process iteration

0.49034
0.49320
0.49607
0.49893
0.50180
0.50466
0.50753

Time (secs)

Binary Trees, V8, Linux, Proc. exec. #24 (no steady state)

0.49764
0.50266
0.50769
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 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?

In-process iteration

0.34148
0.34916
0.35683
0.36451
0.37218
0.37986
0.38753

Time (secs)

Fannkuch Redux, Hotspot, Linux, Proc. exec. #1 (warmup)

24/45 HTTP://SOFT-DEV.ORG/
Inconsistent Process-executions

Binary Trees, V8, Linux\textsubscript{E3−1240v5}, Proc. exec. #7 (warmup)

Binary Trees, V8, Linux\textsubscript{E3−1240v5}, Proc. exec. #8 (slowdown)

(Same machine)
Inconsistent Process-executions

(Binary Trees, C, Linux$_{E3-1240v5}$, Proc. exec. #12 (no steady state))

(Binary Trees, C, OpenBSD$_{4790}$, Proc. exec. #30 (flat))

(Different machines. Bouncing ball Linux-specific)
<table>
<thead>
<tr>
<th>Class</th>
<th>Steady iter (#)</th>
<th>Steady iter (s)</th>
<th>Steady perf (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graal</td>
<td>× (27,\varGamma, 3\varGamma)</td>
<td>32.0 (17.0, 193.8)</td>
<td>6.60 (3.729, 36.608)</td>
</tr>
<tr>
<td>HHVM</td>
<td>× (24,\varGamma, 4\varGamma, 2\varGamma)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotSpot</td>
<td>× (25\varGamma, 5\varGamma)</td>
<td>7.0 (7.0, 53.5)</td>
<td>1.19 (1.182, 9.703)</td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td>∼ (binary trees)</td>
<td>1082.0 (999.0, 1232.5)</td>
<td>2219.59 (2039.304, 2516.021)</td>
</tr>
<tr>
<td>LuaJIT</td>
<td>× (23\varGamma, 4\varGamma, 2\varGamma, 1\varGamma)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyPy</td>
<td>× (27\varGamma, 3\varGamma)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td>× (15\varGamma, 9\varGamma, 6\varGamma)</td>
<td>1.5 (1.0, 794.0)</td>
<td>0.25 (0.000, 391.026)</td>
</tr>
</tbody>
</table>
Classical warmup occurs for only:
Classical warmup occurs for only:

77% of process executions
Classical warmup occurs for only:

- 77% of process executions
- 39% of (VM, benchmark) pairs
Summary

Classical warmup occurs for only:

77% of process executions

39% of (VM, benchmark) pairs

0% of benchmarks for (VM, benchmark, machine) triples
Benchmarks guide our optimisations
Benchmark suites
Benchmark suites

Benchmarks guide our optimisations

Are they complete guides?
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
diff --git a/rpython/jit/metainterpreter/pyjitpl.py b/rpython/jit/metainterpreter/pyjitpl.py
--- a/rpython/jit/metainterpreter/pyjitpl.py
+++ b/rpython/jit/metainterpreter/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.metainterpreter.perform_call(portal_code, allboxes,
-                greenkey=greenboxes)
+            inline = True
+            if self.metainterpreter.is_main jitcode(portal_code):
+                for gk, _ in self.metainterpreter.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.metainterpreter.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
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*
Benchmarks guide our optimisations
Benchmark suites (2)
Benchmarks guide our optimisations

Are they correct guides?
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=0x3de358283581 <a type1Parser with map 0x1f822131a411>,stream=0x4603fbdc4e1 <an Uint8Array with map 0x393de2707fe1>)
 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
In-process iteration
-0.2442
2.0974
4.4389
6.7805
9.1221
11.4636
13.8052

Time (secs)
Process execution #1
Octane: debugging

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

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

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();
```
Fix memory leak in pdfjs.js. #42

Itratt wants to merge 2 commits into chromium:master from ltratt:master

Changes from all commits  1 file  +1 -0

```javascript
function runPdfJS() {
  canvas_logs.length = 0;
  PDFJS.getDocument(pdf_file).then(function(pdf) {
    var canvas = PdfJS_window.document.getElementById('canvas');
    var context = canvas.getContext('2d');
  });
}
```

pdfjs isn’t the only problem
Octane: other issues

pdfjs isn’t the only problem

CodeLoadClosure also has a memory leak
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)
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
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
What we can do

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

HTTP://SOFT-DEV.ORG/
1. Run benchmarks for longer to uncover issues.
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.
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
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