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

Laurence Tratt

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

KING’S College LONDON

Software Development Team
2017-11-21
What to expect from this talk
What to expect from this talk

The gap
What 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
What to expect from this talk

The gap is bigger than we think
A stroppy user? Or rightfully disappointed?
A stroppy user? Or rightfully disappointed?

“You told me I’d get a 10x speed-up, but I only saw 1.2x”
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

iteration time

in-process iteration
What our claims pertain to
What our claims pertain to

Profiling Interpreter

(iteration time)

in-process iteration
What our claims pertain to

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

- Compilation
- Profiling Interpreter
- Peak Performance

iteration time

in-process iteration
What our claims pertain to

**Diagram:**

- **Compilation**
- **Profiling**
- **Interpreted**
- **Peak Performance**

**Axes:**
- **Iteration time**
- **In-process iteration**
What our claims pertain to

This is Barry
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?
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)
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$_{4790}$, Debian 8, 24GiB RAM
- Linux$_{E3-1240v5}$, 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)
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, ...)

14/48 HTTP://SOFT-DEV.ORG/
Method 6: Expectations

More Realistic VM Warmup
Method 6: Expectations

More Realistic VM Warmup

iteration time

in-process iteration

some noise
Method 6: Expectations

More Realistic VM Warmup

iteration time

in-process iteration

compilation tiers

some noise

15 / 48 http://soft-dev.org/
Method 6: Expectations

More Realistic VM Warmup

iteration time

in-process iteration

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)

Changepoint
Warmup & flat (1)

Fannkuch Redux, LuaJIT, OpenBSD_{4790}, Proc. exec. #12 (warmup)

Changepoint

Changepoint segment
Warmup & flat (1)

In-process iteration

1.84197
1.84836
1.85476
1.86115
1.86754
1.87393
1.88032

Time (secs)

N-Body, PyPy, Linux$_{E3-1240v5}$, Proc. exec. #6 (flat)

16/48 HTTP://SOFT-DEV.ORG/
Method 7: Classification

Classification algorithm (steps in order):
All segs are equivalent: flat
Classification algorithm (steps in order):

All segs are equivalent: *flat*

Final seg is in fastest set: *warmup*
Warmup & flat (2)

Spectral Norm, PyPy, Linux\textsubscript{E3 – 1240v5}, Proc. exec. #13 (warmup)
Richards, Hotspot, Linux\textsuperscript{E3-1240v5}, Proc. exec. #3 (slowdown)
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
In-process iteration

Time (secs)

Fasta, V8, Linux, Proc. exec. #14 (slowdown)
No steady state (1)

Binary Trees, V8, Linux\textsuperscript{4790}, Proc. exec. #24 (no steady state)
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?

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

In-process iteration

Time (secs)

In-process iteration: 1 201 401 601 801 1001 1201 1401 1601 1801 2000

Time (secs):

0.34148
0.34916
0.35683
0.36451
0.37218
0.37986
0.38753

0.34181
0.34269
0.34357

24/48 HTTP://SOFT-DEV.ORG/
Inconsistent Process-executions

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

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

(Same machine)
Inconsistent Process-executions

(Different machines. Bouncing ball Linux-specific)
## Individual benchmark stats

<table>
<thead>
<tr>
<th>Class</th>
<th>Iter (s)</th>
<th>Perf (s)</th>
<th>Class</th>
<th>Iter (s)</th>
<th>Perf (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>32.0</td>
<td>0.18594</td>
<td></td>
<td>8.0</td>
<td>0.40555</td>
</tr>
<tr>
<td>Graal</td>
<td>(17,0.139)</td>
<td>0.58009</td>
<td>HHVM</td>
<td>(7.5,0.85)</td>
<td>0.13354</td>
</tr>
<tr>
<td>HotSpot</td>
<td>(7.0,0.53)</td>
<td>0.18279</td>
<td></td>
<td>(2.0,0.1)</td>
<td>0.13699</td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td>(999,0.1228)</td>
<td>0.000116</td>
<td></td>
<td>(69,0.179)</td>
<td>0.13644</td>
</tr>
<tr>
<td>LuaJIT</td>
<td>(999,0.1228)</td>
<td>0.000116</td>
<td></td>
<td>(17,7.16,0.127)</td>
<td>0.25399</td>
</tr>
<tr>
<td>PyPy</td>
<td>(31.5,0.6)</td>
<td>0.49237</td>
<td></td>
<td>(0.000,0.87)</td>
<td>0.24138</td>
</tr>
<tr>
<td>V8</td>
<td>(31.5,0.6)</td>
<td>0.49237</td>
<td></td>
<td>(0.000,0.87)</td>
<td>0.24138</td>
</tr>
<tr>
<td>C</td>
<td>10.0</td>
<td>1.0</td>
<td></td>
<td>(199,1)</td>
<td>0.47421</td>
</tr>
<tr>
<td>Graal</td>
<td>(10,0.10)</td>
<td>1.0</td>
<td>HHVM</td>
<td>(999,0.1025)</td>
<td>0.89509</td>
</tr>
<tr>
<td>HotSpot</td>
<td>(253,0.0)</td>
<td>1.0</td>
<td></td>
<td>(901.7,0.0)</td>
<td>0.091118</td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td>(25,0.0)</td>
<td>1.0</td>
<td></td>
<td>(346,7.135)</td>
<td>0.00942</td>
</tr>
<tr>
<td>LuaJIT</td>
<td>(25,0.0)</td>
<td>1.0</td>
<td></td>
<td>(346,7.135)</td>
<td>0.00942</td>
</tr>
<tr>
<td>PyPy</td>
<td>(15,0.3)</td>
<td>1.0</td>
<td></td>
<td>(346,7.135)</td>
<td>0.00942</td>
</tr>
<tr>
<td>V8</td>
<td>(15,0.3)</td>
<td>1.0</td>
<td></td>
<td>(346,7.135)</td>
<td>0.00942</td>
</tr>
</tbody>
</table>
# Individual benchmark stats

<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>0.18594</td>
</tr>
<tr>
<td>Graal</td>
<td>× (27\ell, 3\Gamma)</td>
<td>32.0 (17.0, 193.8)</td>
<td>6.60 (3.729, 36.608)</td>
</tr>
<tr>
<td>HHVM</td>
<td>× (24\ell, 4\Gamma, 2\omega)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotSpot</td>
<td>× (25\ell, 5\Gamma)</td>
<td>7.0 (7.0, 53.5)</td>
<td>1.19 (1.182, 9.703)</td>
</tr>
<tr>
<td>JRuby+Truffle</td>
<td>⊤</td>
<td>1082.0 (999.0, 1232.5)</td>
<td>2219.59 (2039.304, 2516.021)</td>
</tr>
<tr>
<td>LuaJIT</td>
<td>× (23\ell, 4\Gamma, 2--, 1\omega)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PyPy</td>
<td>× (27\ell, 3\omega)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td>× (15--, 9\ell, 6\Gamma)</td>
<td>1.5 (1.0, 794.0)</td>
<td>0.25 (0.000, 391.026)</td>
</tr>
</tbody>
</table>
## Overall benchmark stats

<table>
<thead>
<tr>
<th>Class.</th>
<th>Linux\textsuperscript{4790}</th>
<th>Linux\textsuperscript{1240v5}</th>
<th>OpenBSD\textsuperscript{4790} †</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(VM, benchmark) pairs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>8.7%</td>
<td>13.0%</td>
<td>6.7%</td>
</tr>
<tr>
<td>↓</td>
<td>28.3%</td>
<td>23.9%</td>
<td>10.0%</td>
</tr>
<tr>
<td>∪</td>
<td>6.5%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>≈</td>
<td>4.3%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>=</td>
<td>6.5%</td>
<td>6.5%</td>
<td>13.3%</td>
</tr>
<tr>
<td>≠</td>
<td>45.7%</td>
<td>43.5%</td>
<td>70.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
</tr>
<tr>
<td>↓</td>
</tr>
<tr>
<td>∪</td>
</tr>
<tr>
<td>≈</td>
</tr>
</tbody>
</table>
### Overall benchmark stats

<table>
<thead>
<tr>
<th>Class</th>
<th>Linux\textsubscript{4790}</th>
<th>Linux\textsubscript{1240v5}</th>
<th>OpenBSD\textsubscript{4790}†</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>8.7%</td>
<td>13.0%</td>
<td>6.7%</td>
</tr>
<tr>
<td>≦</td>
<td>28.3%</td>
<td>23.9%</td>
<td>10.0%</td>
</tr>
<tr>
<td>⪯</td>
<td>6.5%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>≿</td>
<td>4.3%</td>
<td>6.5%</td>
<td>0.0%</td>
</tr>
<tr>
<td>=</td>
<td>6.5%</td>
<td>6.5%</td>
<td>13.3%</td>
</tr>
<tr>
<td>⋈</td>
<td>45.7%</td>
<td>43.5%</td>
<td>70.0%</td>
</tr>
</tbody>
</table>

### ⟨VM, benchmark⟩ pairs

<table>
<thead>
<tr>
<th>Class</th>
<th>Linux\textsubscript{4790}</th>
<th>Linux\textsubscript{1240v5}</th>
<th>OpenBSD\textsubscript{4790}†</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>26.4%</td>
<td>20.9%</td>
<td>34.0%</td>
</tr>
<tr>
<td>≦</td>
<td>48.3%</td>
<td>51.5%</td>
<td>52.1%</td>
</tr>
<tr>
<td>⪯</td>
<td>16.7%</td>
<td>17.9%</td>
<td>11.1%</td>
</tr>
<tr>
<td>≿</td>
<td>8.7%</td>
<td>9.6%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

### Process executions
Classical warmup occurs for only:
Classical warmup occurs for only:

72.4%–74.7% of process executions
Classical warmup occurs for only:

72.4%–74.7% of process executions

43.4%–43.5% of (VM, benchmark) pairs
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?

Fasta, PyPy, Linux\textsubscript{E3 \text{-} 1240v5}, Proc. exec. #5 (no steady state)

\begin{itemize}
\item Time (secs)
\item GC
\item JIT
\end{itemize}

\begin{itemize}
\item 2.321 \times 10^9
\item 1.582 \times 10^9
\item 0.843 \times 10^9
\item 0.01488
\item -0.00003
\item 0.00743
\item 0.01488
\item -0.00003
\item 0.00743
\item 0.01488
\end{itemize}
Are odd effects correlated with compilation and GC?

Richards, Hotspot, Linux\textsubscript{E3 -1240v5}, Proc. exec. #3 (slowdown)
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/metainterpire/pyjitpl.py b/rpython/jit/metainterpire/pyjitpl.py
--- a/rpython/jit/metainterpire/pyjitpl.py
+++ b/rpython/jit/metainterpire/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.metainterpire.perform_call(portal_code, allboxes,
-                    greenkey=greenboxes)
+# we’re about to recurse.
+    inline = True
+ if self.metainterpire.is_main_jitcode(portal_code):
+    for gk, _ in self.metainterpire.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.metainterpire.perform_call(portal_code, allboxes,
+                    greenkey=greenboxes)
Success: slow benchmark now 13.5x faster
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

<table>
<thead>
<tr>
<th></th>
<th>#unrollings</th>
<th>_1</th>
<th>_2</th>
<th>_3</th>
<th>_5</th>
<th>_7</th>
<th>_10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hexiom2</td>
<td>1.3</td>
<td>1.4</td>
<td>1.1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>raytrace-simple</td>
<td>3.3</td>
<td>3.1</td>
<td>2.8</td>
<td>1.4</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>spectral-norm</td>
<td>3.3</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>sympy_str</td>
<td>1.5</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>telco</td>
<td>4</td>
<td>2.5</td>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>polymorphism</td>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
<td>0.08</td>
<td>0.09</td>
</tr>
</tbody>
</table>

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*
Benchmarks guide our optimisations
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>,name=0x4603fbd9c09 <String[12]: JTYMKN+CMR17>,file=0x4603fbf7a18 <String[1]: /usetype1.js>...)

# 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: debugging

```javascript
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)
]);

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

Contribution wants to merge 2 commits from chromium:master from itratt:master

Changes from all commits 1 file  +1 -0

```javascript
function setupPdfJS() {
  // ... -43,6 +43,7 ...
}

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
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)
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.
How to benchmark a bit better

1. Don’t benchmark 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.
1. Run benchmarks for longer to uncover issues.
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 warm-up 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. The more benchmarks, the better.
1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Always report warmup time.
4. Stop over-training on small benchmark suites.
5. The more benchmarks, the better.
6. Focus on predictable performance.
What we’re doing next
References

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

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

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

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