Why VM Benchmarking is Probably Misleading You
Curry On 2018, Amsterdam

Edd Barrett
Carl Friedrich Bolz-Tereick
Rebecca Killick (Lancaster)
Sarah Mount
Laurence Tratt

Software Development Team
2018-07-17
Virtual Machine Warmup Blows Hot and Cold*

EDD BARRETT, King’s College London, UK
CARL FRIEDRICH BOLZ-TEREIICK, King’s College London, UK
REBECCA KILICK, Lancaster University, UK
SARAH MOUNT, King’s College London, UK
LAURENCE TRATT, King’s College London, UK

Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: the initial warmup phase determines which parts of a program would most benefit from dynamic compilation, before JIT compiling those parts into machine code; subsequently the program is said to be at a steady state of peak performance. Measurement methodologies almost always discard data collected during the warmup phase such that reported measurements focus entirely on peak performance. We introduce a fully automated statistical approach, based on changepoint analysis, which allows us to determine if a program has reached a steady state and, if so, whether that represents peak performance or not. Using this, we show that even when run in the most controlled of circumstances, small, deterministic, widely studied microbenchmarks often fail to reach a steady state of peak performance on a variety of common VMs. Repeating our experiment on 3 different machines, we found that at most 43.5% of (VM, benchmark) pairs consistently reach a steady state of peak performance.

CCS Concepts: • Software and its engineering → Software performance; Just-in-time compilers; Interpreters;

Additional Key Words and Phrases: Virtual machine, JIT, benchmarking, performance

ACM Reference Format:
The lifetime of a VM invocation
The lifetime of a VM invocation

“process execution”
BENCHMARKING CONCEPTS

- The lifetime of a VM invocation
  - "process execution"

- Run benchmark in a loop
Benchmarking Concepts

- The lifetime of a VM invocation
  - “process execution”

- Run benchmark in a loop
  - “in-process iterations”
The Current State of the Art of Benchmarking
The Current State of the Art of Benchmarking

Just In Time (JIT) Compilation

Iteration Time vs. In-Process Iteration
The Current State of the Art of Benchmarking

**JUST IN TIME (JIT) COMPILATION**

ITERATION TIME

IN-PROCESS ITERATION

PROFILING INTERPRETER
The Current State of the Art of Benchmarking

JUST IN TIME (JIT) COMPILATION

Iteration Time

In-Process Iteration

Compilation

Profiling Interpreter
The Current State of the Art of Benchmarking

Just in Time (JIT) Compilation

Compilation

Profiling Interpreter

Peak Performance

Iteration Time

In-process Iteration
The Current State of the Art of Benchmarking

More realistic VM warmup

Iteration time vs. in-process iteration
The Current State of the Art of Benchmarking

MORE REALISTIC VM WARMUP

ITERATION TIME

IN-PROCESS ITERATION

SOME NOISE
The Current State of the Art of Benchmarking

More Realistic VM Warmup

Iteration Time

In-Process Iteration

Compilation Tiers

Some Noise
The Current State of the Art of Benchmarking

MORE REALISTIC VM WARMUP

ITERATION TIME

IN-PROCESS ITERATION

SOME NOISE

COMPILATION TIERS

GC SPIKES
The Current State of the Art of Benchmarking

THE WARMUP PHASE

ITERATION TIME

IN-PROCESS ITERATION

WARMUP
Warmup Matters

1.5 SessionDislike Poor Warmup.

AUTHORS Dislike Poor Warmup.

Warmup is important!

6/36 http://soft-dev.org/
Warmup Matters

1. Users dislike poor warmup.
1. Users dislike poor warmup.

2. VM authors dislike poor warmup.
Warmup Matters

1. Users dislike poor warmup.

2. VM authors dislike poor warmup.

Warmup is important!
We should measure the warmup of modern language implementations
We should measure the warmup of modern language implementations

*Hypothesis*: Small, deterministic programs reach a steady state of peak performance.
Method 1: Which Benchmarks?

BENCHMARKS ARE PERFECT FOR US (UNUSUALLY)

http://benchmarksgame.alioth.debian.org/

EREMOVED ANY CFG NON-DETERMINISM

ADDED CHECKSUMS TO ALL BENCHMARKS

HTTP://SOFT-DEV.ORG/
Method 1: Which Benchmarks?

The CLBG benchmarks are perfect for us (unusually)

http://benchmarksgame.alioth.debian.org/
Method 1: Which Benchmarks?

The CLBG benchmarks are perfect for us (unusually)

http://benchmarksgame.alioth.debian.org/

We removed any CFG non-determinism
Method 1: Which Benchmarks?

The CLBG benchmarks are perfect for us (unusually)

http://benchmarksgame.alioth.debian.org/

We removed any CFG non-determinism

We added checksums to all benchmarks
Method 2: How Long to Run?
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
- Linux\(_{E3-1240v5}\), Debian 8, 32GiB RAM
- OpenBSD\(_{4790}\), 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 off.
- Hyper-threading off.
Method 5: Benchmark Harness

Krun

Control as many confounding variables as possible
Method 5: Benchmark Harness

Krun

Control as many confounding variables as possible

- 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, ...)

12 / 36
Method 6: Changepoint Analysis and Classification
Method 6: Changepoint Analysis and Classification

OUTLIERS IDENTIFIED AND IGNORED

ITERATION TIME

IN-PROCESS ITERATION
Method 6: Changepoint Analysis and Classification

Changepoints Identified

Iteration Time

In-Process Iteration
Method 6: Changepoint Analysis and Classification
One changepoint segment: 

\text{FLAT} (−)
Method 6: Changepoint Analysis and Classification

- One changepoint segment: \texttt{FLAT (—)}

- No changepoint after 1500 in-process iterations:
  - The last seg is the fastest: \texttt{WARMUP (¯)
Method 6: Changepoint Analysis and Classification

- One changepoint segment: \textbf{FLAT} (─)

- No changepoint after 1500 in-process iterations:
  - The last seg is the fastest: \textbf{WARMUP} (⊃)
  - not the fastest seg: \textbf{SLOWDOWN} (∥)
Method 6: Changepoint Analysis and Classification

- One changepoint segment: **FLAT (−)**
- No changepoint after 1500 in-process iterations:
  - The last seg is the fastest: **WARMUP (⌜)**
  - not the fastest seg: **SLOWDOWN (⌞)**
- Otherwise: **NO STEADY STATE (≈)**
Very close segments are considered equivalent
Results
Results

- The results I’m showing are from v1.5 of the data.
  https://archive.org/download/softdev_warmup_experiment_artefacts/v1.5/

- The OOPSLA paper uses v0.8 data.
  https://archive.org/download/softdev_warmup_experiment_artefacts/v0.8/
Results: Flat

N-Body, PyPy, Linux\textsubscript{4790}, Proc. exec. #24 (flat)
Results: Warmup

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

In-process iteration

Time (secs)

1.12811
1.13248
1.13685
1.14121
1.14558
1.14995
1.15432

1.13493
1.14273
1.15053

19/36 HTTP://SOFT-DEV.ORG/
Results: Warmup

Spectral Norm, PyPy, Linux$E_3-1240v5$, Proc. exec. #5 (warmup)

- In-process iteration
- Time (secs)

Graph showing the spectral norm values over time for PyPy on Linux with $E_3-1240v5$, execution #5, during the warmup phase.
Results: Slowdown

Richards, HotSpot, Linux$_{E3-1240v5}$, Proc. exec. #8 (slowdown)
Results: Slowdown

In-process iteration

Time (secs)

Fasta, V8, Linux, Proc. exec. #26 (slowdown)
Results: No Steady State

Binary Trees, V8, Linux, Proc. exec. #6 (no steady state)
Results: Inconsistent Process-executions

(Binary Trees, V8, Linux4790, Proc. exec. #15 (warmup))

(Binary Trees, V8, Linux4790, Proc. exec. #6 (no steady state))

(Same machine)
## Quantitative Results

<table>
<thead>
<tr>
<th>Class.</th>
<th>Steady iter (#)</th>
<th>Steady iter (s)</th>
<th>Steady perf (s)</th>
<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>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Graal  | $\times (27\times 3)\uparrow$ | $32.0$ | $6.60$ | $0.18959$ | Graal  | $\times (27\times 3)\uparrow$ | $8.0$ | $1.22$ | $0.40555$ | $0.005310$
| HHVM   | $\times (24\times 4)\uparrow,2\downarrow$ | $7.0$ | $1.19$ | $0.168279$ | HHVM   | $\times (16\times 11)\uparrow,3\downarrow$ | $2.0$ | $0.14$ | $0.13869$ | $0.0000042$
| HotSpot| $\times (25\times 5)\uparrow$ | $1082.0$ | $2219.59$ | $2.05150$ | HotSpot| $\times (25\times 5)\uparrow$ | $69.0$ | $17.95$ | $0.20644$ | $0.0000158$
| JRuby+Truffle | $\rightarrow$ | $5000.00$ | $5000.00$ | $0.000016$ | JRuby+Truffle | $\rightarrow$ | $5000.00$ | $5000.00$ | $0.000016$ | $0.0000042$
| LuaJIT | $\times (25\times 5)\uparrow,2\downarrow,1\uparrow$ | $0.25$ | $0.49237$ | $1.0$ | $0.00$ | $0.24138$ | $1.0$ | $0.00$ | $0.24138$
| PyPy   | $\times (27\times 3)\uparrow$ | $1.5$ | $0.09318$ | $1.0$ | $0.00$ | $0.24138$ | $1.85835$ | $0.012848$
| V8     | $\times (15\times 9)\uparrow,6\downarrow$ | $5000.00$ | $5000.00$ | $0.000016$ | V8     | $\times (15\times 9)\uparrow,6\downarrow$ | $5000.00$ | $5000.00$ | $0.000016$ | $0.0000042$

<table>
<thead>
<tr>
<th>Class.</th>
<th>Steady iter (#)</th>
<th>Steady iter (s)</th>
<th>Steady perf (s)</th>
<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>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Graal  | $\times (21\times 6)\uparrow,2\downarrow,1\uparrow$ | $10.0$ | $52.66$ | $1.35779$ | Graal  | $\times (19\times 5)\uparrow,4\downarrow,2\downarrow$ | $1021.0$ | $917.30$ | $0.89590$ | $0.0002759$
| HHVM   | $\times (28\times 1)\uparrow,1\downarrow$ | $390.0$ | $153.70$ | $0.36202$ | HHVM   | $\times (28\times 1)\uparrow,1\downarrow$ | $1021.0$ | $917.30$ | $0.89590$ | $0.0002759$
| HotSpot| $\rightarrow$ | $1016.5$ | $1039.04$ | $1.08833$ | HotSpot| $\rightarrow$ | $1016.5$ | $1039.04$ | $1.08833$ | $0.0000997$
| JRuby+Truffle | $\rightarrow$ | $999.00$ | $1025.1$ | $0.56285$ | JRuby+Truffle | $\rightarrow$ | $999.00$ | $1025.1$ | $0.56285$ | $0.0000997$
| LuaJIT | $\rightarrow$ | $2.0$ | $1.57$ | $0.020149$ | LuaJIT | $\rightarrow$ | $2.0$ | $1.57$ | $0.020149$ | $0.0000997$
| PyPy   | $\times (15\times 13)\uparrow,2\downarrow$ | $2.0$ | $0.31$ | $0.30401$ | PyPy   | $\times (15\times 13)\uparrow,2\downarrow$ | $2.0$ | $0.31$ | $0.30401$ | $0.0000154$
| V8     | $\times (19\times 11)\uparrow,2\downarrow$ | $2.0$ | $0.31$ | $0.30401$ | V8     | $\times (19\times 11)\uparrow,2\downarrow$ | $2.0$ | $0.31$ | $0.30401$ | $0.0000154$

<table>
<thead>
<tr>
<th>Class.</th>
<th>Steady iter (#)</th>
<th>Steady iter (s)</th>
<th>Steady perf (s)</th>
<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>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Graal  | $\times (29\times 1)\uparrow$ | $150.00$ | $12.40$ | $0.89293$ | Graal  | $\times (29\times 1)\uparrow$ | $150.00$ | $12.40$ | $0.89293$ | $0.0000047$
| HHVM   | $\times (27\times 2)\uparrow,1\downarrow$ | $35.00$ | $139.18$ | $1.04609$ | HHVM   | $\times (27\times 2)\uparrow,1\downarrow$ | $35.00$ | $139.18$ | $1.04609$ | $0.0000047$
| HotSpot| $\times (18\times 12)\uparrow$ | $7.0$ | $1.90$ | $0.31470$ | HotSpot| $\times (18\times 12)\uparrow$ | $7.0$ | $1.90$ | $0.31470$ | $0.0000009$
| JRuby+Truffle | $\rightarrow$ | $307.00$ | $191.24$ | $0.83633$ | JRuby+Truffle | $\rightarrow$ | $307.00$ | $191.24$ | $0.83633$ | $0.0000042$
| LuaJIT | $\rightarrow$ | $75.00$ | $34.43$ | $0.46489$ | LuaJIT | $\rightarrow$ | $75.00$ | $34.43$ | $0.46489$ | $0.0000046$
| PyPy   | $\rightarrow$ | $3.00$ | $0.55$ | $0.24963$ | PyPy   | $\rightarrow$ | $3.00$ | $0.55$ | $0.24963$ | $0.0000039$
| V8     | $\times (19\times 10)\uparrow,1\downarrow$ | $75.00$ | $34.43$ | $0.46489$ | V8     | $\times (19\times 10)\uparrow,1\downarrow$ | $75.00$ | $34.43$ | $0.46489$ | $0.0000039$

---

Software Development Team

http://soft-dev.org/
## Results

<table>
<thead>
<tr>
<th>Class.</th>
<th>SpectralNorm</th>
<th>Steady iter (#)</th>
<th>Steady iter (s)</th>
<th>Steady perf (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>△ (27, 2−, 1Δ)</td>
<td>775.0 (1.5,780.0)</td>
<td>425.16 (0.246,426.809)</td>
<td>0.54581 ± 0.033116</td>
</tr>
<tr>
<td>Graal</td>
<td>△</td>
<td>14.0 (2.0,94.6)</td>
<td>13.60 (0.830,98.737)</td>
<td>1.05685 ± 0.000126</td>
</tr>
<tr>
<td>HHVM</td>
<td>△ (29, 1ω)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotSpot</td>
<td>□</td>
<td>7.0 (7.0,7.5)</td>
<td>1.91 (1.902,3.645)</td>
<td>0.31472 ± 0.169143</td>
</tr>
<tr>
<td>LuaJIT</td>
<td>□</td>
<td></td>
<td></td>
<td>0.22181 ± 0.000039</td>
</tr>
<tr>
<td>PyPy</td>
<td>△ (27, 3Δ)</td>
<td>1.0 (1.0,45.2)</td>
<td>0.00 (0.000,20.597)</td>
<td>0.46480 ± 0.000085</td>
</tr>
<tr>
<td>TruffleRuby</td>
<td>△ (25Δ, 5ω)</td>
<td>3.0 (3.0,3.0)</td>
<td>0.52 (0.523,0.526)</td>
<td>0.25362 ± 0.000034</td>
</tr>
<tr>
<td>V8</td>
<td>△</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Results: Summary

<table>
<thead>
<tr>
<th>Class.</th>
<th>( \text{Linux}_{4790} )</th>
<th>( \text{Linux}_{1240v5} )</th>
<th>( \text{OpenBSD}_{4790} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \langle \text{VM, benchmark} \rangle ) pairs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sim )</td>
<td>8.9%</td>
<td>11.1%</td>
<td>13.3%</td>
</tr>
<tr>
<td>( \preceq )</td>
<td>20.0%</td>
<td>17.8%</td>
<td>20.0%</td>
</tr>
<tr>
<td>( \preceq )</td>
<td>4.4%</td>
<td>4.4%</td>
<td>3.3%</td>
</tr>
<tr>
<td>( \approx )</td>
<td>4.4%</td>
<td>4.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>( \equiv )</td>
<td>11.1%</td>
<td>8.9%</td>
<td>13.3%</td>
</tr>
<tr>
<td>( \neq )</td>
<td>51.1%</td>
<td>53.3%</td>
<td>50.0%</td>
</tr>
<tr>
<td></td>
<td>Process executions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sim )</td>
<td>22.0%</td>
<td>23.3%</td>
<td>37.7%</td>
</tr>
<tr>
<td>( \preceq )</td>
<td>48.3%</td>
<td>43.9%</td>
<td>35.2%</td>
</tr>
<tr>
<td>( \preceq )</td>
<td>20.1%</td>
<td>22.1%</td>
<td>12.1%</td>
</tr>
<tr>
<td>( \approx )</td>
<td>9.6%</td>
<td>10.8%</td>
<td>15.0%</td>
</tr>
</tbody>
</table>
## Results: Summary

<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>⟨VM, benchmark⟩ pairs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−</td>
<td>8.9%</td>
<td>11.1%</td>
<td>13.3%</td>
</tr>
<tr>
<td>⊥</td>
<td>20.0%</td>
<td>17.8%</td>
<td>20.0%</td>
</tr>
<tr>
<td>⊤</td>
<td>4.4%</td>
<td>4.4%</td>
<td>3.3%</td>
</tr>
<tr>
<td>≈</td>
<td>4.4%</td>
<td>4.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>≛</td>
<td>11.1%</td>
<td>8.9%</td>
<td>13.3%</td>
</tr>
<tr>
<td>※</td>
<td>51.1%</td>
<td>53.3%</td>
<td>50.0%</td>
</tr>
<tr>
<td></td>
<td>Process executions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−</td>
<td>22.0%</td>
<td>23.3%</td>
<td>37.7%</td>
</tr>
<tr>
<td>⊥</td>
<td>48.3%</td>
<td>43.9%</td>
<td>35.2%</td>
</tr>
<tr>
<td>⊤</td>
<td>20.1%</td>
<td>22.1%</td>
<td>12.1%</td>
</tr>
<tr>
<td>≈</td>
<td>9.6%</td>
<td>10.8%</td>
<td>15.0%</td>
</tr>
</tbody>
</table>
“Good” warmup occurs for only:
“Good” warmup occurs for only:

67.2%–70.3% of process executions
“Good” warmup occurs for only:

67.2%–70.3% of process executions

37.8%–40.0% of \( \langle \text{VM, benchmark} \rangle \) pairs
Hypothesis: Small, deterministic programs reach a steady state of peak performance.
Are the Effects due to JIT and GC?

Fasta, PyPy, Linux_{E3-1240v5}, Proc. exec. #4 (no steady state)

In-process iteration

Time (secs)

-0.00003

0.00725

0.01453

JIT

GC

2.62972

2.55091

2.47210

2.39328

2.31447

2.23565

2.15684

1 201 401 601 801 1001 1201 1401 1601 1801 2000

Software Development Team

http://soft-dev.org/
Are the Effects due to JIT and GC?

Richards, HotSpot, Linux_{E3-1240v5}, Proc. exec. #3 (slowdown)

In-process iteration

Time (secs)

JIT (secs)

GC (secs)
However

In many cases, the JIT/GC can’t explain oddness
What Have We Learned?

- Benchmarks often don’t warmup as we expect.

- Repeating a benchmark often gives a different warmup characteristic.

- Have we been misled?
  - Ineffectual or bad optimisations?
What Can We Do?

1. Unbenchmark for longer to uncover issues.
2. Accept that peak performance may not occur.
3. Don't always blame GC or JIT compilation.
4. Always report warmup time.

30/36 HTTP://SOFT-DEV.ORG/
1. Run benchmarks for longer to uncover issues.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. We can’t always blame GC or JIT compilation.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. We can’t always blame GC or JIT compilation.
4. Always report warmup time.
What Can We Do?

1. Run benchmarks for longer to uncover issues.
2. Accept that peak performance may not occur.
3. We can’t always blame GC or JIT compilation.
4. Always report warmup time.
5. Engineer VMs for predictable performance?
OOPSLA paper:

Main experiment repo:
https://github.com/softdevteam/warmup_experiment/

Krun Benchmark Runner:
https://github.com/softdevteam/krun/

Archived data sets:
https://archive.org/details/softdev_warmup_experiment_artefacts/
Backup Slides
The Bouncing Ball Pattern

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

Binary Trees, C, OpenBSD$_{4790}$, Proc. exec. #30 (flat)
For each machine, we summarise each ⟨VM, Benchmark⟩ pairing.
Method 7: Summary Statistics

For each machine, we summarise each \(\langle VM, Benchmark\rangle\) pairing.

<table>
<thead>
<tr>
<th>Consistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>All 30 process executions were classified the same.</td>
</tr>
</tbody>
</table>
Method 7: Summary Statistics

For each machine, we summarise each \( \langle \text{VM, Benchmark} \rangle \) pairing.

### Consistent

All 30 process executions were classified the same.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Inconsistent

A mix of classifications arose within the 30 process executions. E.g.:

- Good inconsistent: \( = (25\downarrow, 5\downarrow) \)
- Bad inconsistent: \( \neq (20\downarrow, 3\uparrow, 7\uparrow) \)
For each machine, we summarise each \( \langle VM, \text{Benchmark} \rangle \) pairing.

### Consistent

All 30 process executions were classified the same.

\[
- \quad \downarrow \quad \bigcirc \quad \nearrow
\]

### Inconsistent

A mix of classifications arose within the 30 process executions. E.g.:

- Good inconsistent: \( = (25\downdownarrow, 5\downarrow) \)
- Bad inconsistent: \( \neq (20\downdownarrow, 3\bigcirc, 7\nearrow) \)

If possible report: steady state performance, time until steady state, etc.