In Search of Accurate Benchmarking

BenchWork 2018, Amsterdam

Edd Barrett

Sarah Mount

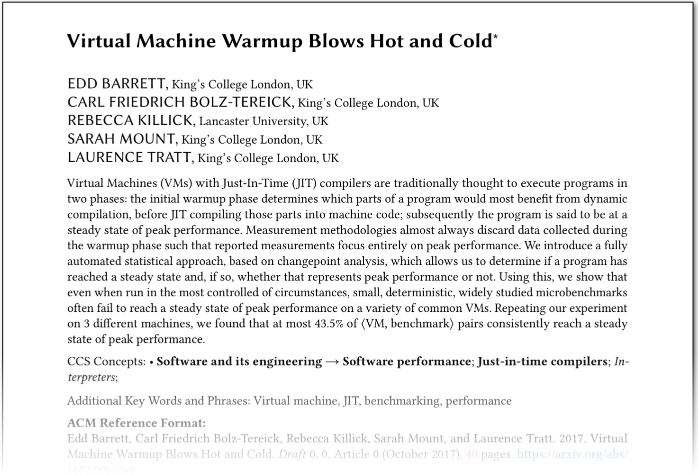
Laurence Tratt

KING'S College LONDON

Software Development Team

2018-07-18

Back-story



Virtual Machine Warmup Blows Hot and Cold*

EDD BARRETT, King's College London, UK
CARL FRIEDRICH BOLZ-TEREICK, King's College London, UK
REBECCA KILLICK, Lancaster University, UK
SARAH MOUNT, King's College London, UK
LAURENCE TRATT, King's College London, UK

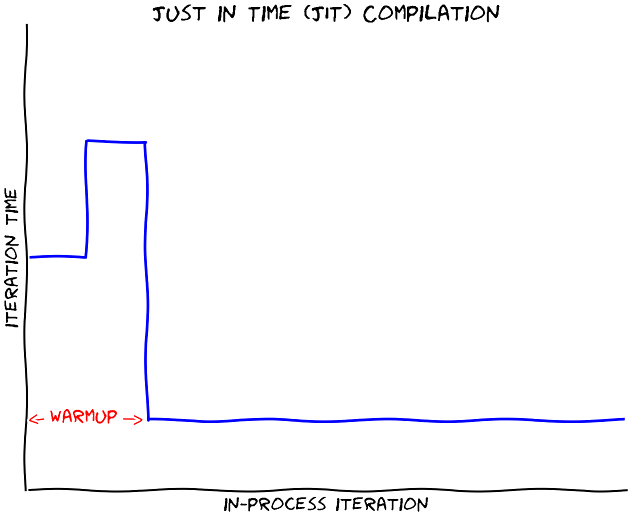
Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: the initial warmup phase determines which parts of a program would most benefit from dynamic compilation, before JIT compiling those parts into machine code; subsequently the program 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*;

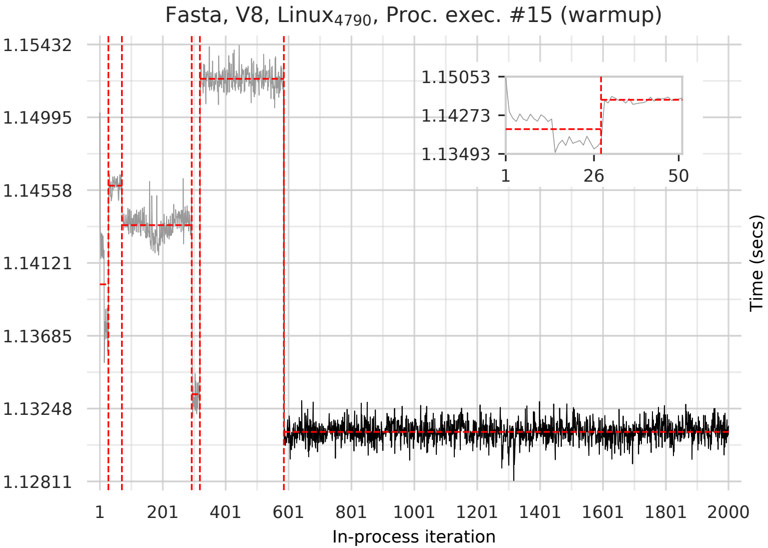
Additional Key Words and Phrases: Virtual machine, JIT, benchmarking, performance

ACM Reference Format:
Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Draft* 0, 0, Article 0 (October 2017), 40 pages. https://arxiv.org/abs/1602.00602v6

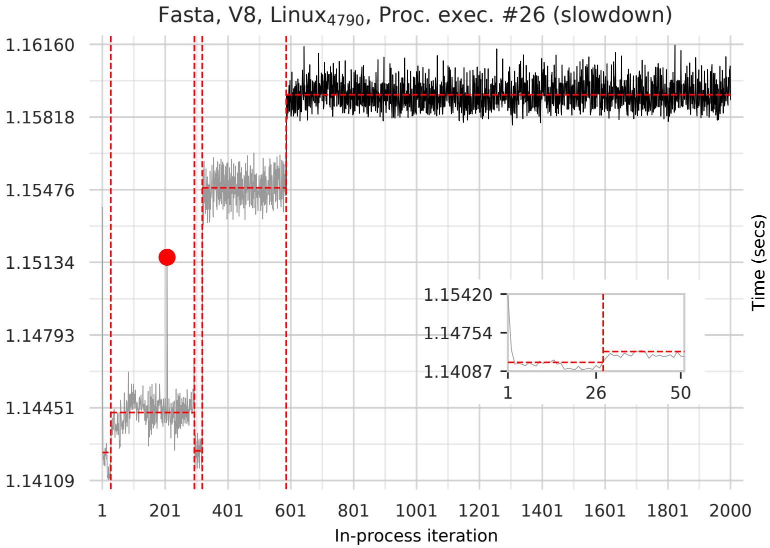
Back-story


JUST IN TIME (JIT) COMPILATION
ITERATION TIME
← WARMUP →
IN-PROCESS ITERATION

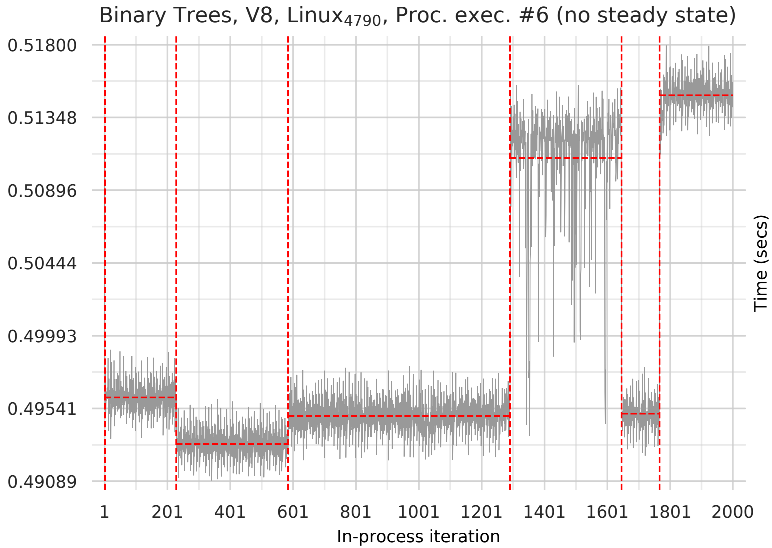
Back-story


Fasta, V8, Linux$_{4790}$, Proc. exec. #15 (warmup)
1.15432
1.14995
1.14558
1.14121
1.13685
1.13248
1.12811
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)
1.15053
1.14273
1.13493
1
26
50

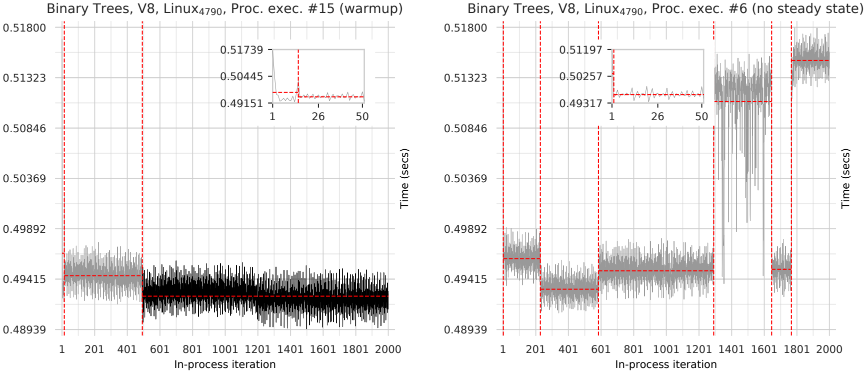
Back-story


Fasta, V8, Linux$_{4790}$, Proc. exec. #26 (slowdown)
1.16160
1.15818
1.15476
1.15134
1.14793
1.14451
1.14109
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)
1.15420
1.14754
1.14087
1
26
50

Back-story


Binary Trees, V8, Linux₄₇₉₀, Proc. exec. #6 (no steady state)
0.51800
0.51348
0.50896
0.50444
0.49993
0.49541
0.49089
1
201
401
601
801
1001
1201
1401
1601
1801
2000
In-process iteration
Time (secs)

Back-story



Same benchmark, same VM, same machine, just repeated.

This Talk

I'm not going to talk about those results. . .

But rather our experiences in designing KRUN

Krun

KRUN: Our benchmark runner

`https://github.com/softdevteam/krun`

- Eliminate variation induced by hardware
- Eliminate variation induced by *unrelated* software
- But without interfering with the full range of VM behaviours

Step 1: Review Conventional Wisdom

- Turn off turbo mode
- Turn off hyper-threading
- Set CPU governor to “performance mode”
- Set process priority on the benchmark
- Pin benchmarks
- Turn off ASLR

Pinning

Limiting the set of cores the benchmark can run on.

Thought to improve benchmark stability.

Further reading:

- `sched_setaffinity(2)`
- `taskset(1)`

Pinning

- We saw slowdowns when pinning multi-threaded VMs

- We opted *not* to pin.
  - We are not experts in all of the VMs we measured

Address Space Layout Randomisation

Randomising relocation of ELF binaries at load time

Again, thought to stabilise benchmarks

Further reading:

`/proc/sys/kernel/randomize_va_space`

ASLR

- ASLR changes VM visible pointer addresses
- Could “pigeon hole” VM behaviour
    - E.g. A hash-table keyed by code-address
- We opted to keep it ON

Beyond Conventional Wisdom

- Turn off turbo mode ✓
- Turn off hyper-threading ✓
- Crank CPU to “performance mode” ✓
- Set process priority on the benchmark ✓
- Pin benchmarks ✗
- Turn off ASLR ✗

What else?

What Else?

- Memory limits (stack and heap)
- Minimise allocation
- Use static-sized allocations where possible
- Use same CPU temperature for all benchmarks
- Create a fresh user account for each benchmark
- Reboot after each benchmark
- Disable the network card
- Use a tickless kernel
- Monitor `dmesg(8)` buffer
- `sync(8)` disks before each benchmark
- Control Intel P-states
- . . .

What Else?

- Memory limits (stack and heap)
- Minimise allocation
- Use static-sized allocations where possible
- Use same CPU temperature for all benchmarks
- Create a fresh user account for each benchmark
- Reboot after each benchmark
- Disable the network card
- Use a tickless kernel
- Monitor `dmesg(8)` buffer
- `sync(8)` disks before each benchmark
- Control Intel P-states
- ...

What Else?

- Memory limits (stack and heap)
- Minimise allocation
- Use static-sized allocations where possible
- Use same CPU temperature for all benchmarks
- Create a fresh user account for each benchmark
- Reboot after each benchmark
- Disable the network card
- Use a tickless kernel
- Monitor `dmesg(8)` buffer
- `sync(8)` disks before each benchmark
- Control Intel P-states
- ...

What Else?

- Memory limits (stack and heap)
- Minimise allocation
- Use static-sized allocations where possible
- Use same CPU temperature for all benchmarks
- Create a fresh user account for each benchmark
- Reboot after each benchmark
- Disable the network card
- Use a tickless kernel
- Monitor `dmesg(8)` buffer
- `sync(8)` disks before each benchmark
- Control Intel P-states
- ...

Intel P-states

A feature in all modern Intel chips relating to “speed step”.

If a core is not fully utilised, you can lower its frequency and save power with no performance cost.

Intel P-states

Can we disable P-states to get a fixed frequency?

Intel P-states

Can we disable P-states to get a fixed frequency?

> *The idea that frequency can be set to a single frequency is fictional for Intel Core processors.*
>
> *Even if the scaling driver selects a single P-State, the actual frequency the processor will run at is selected by the processor itself*

– Linux Kernel Docs

Intel P-states

- Pass `intel_pstate=disable` on the kernel command line
  - Stops the kernel changing the frequency (hardware still can)
- Monitor the CPU clock speed over the course of benchmarking.
  - Reject runs where the CPU appears to have clocked down
- Further reading:
  - Intel manual: “MPERF and APERF Counters Under HDC”

Data Collection: Know your Clocks

```
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    delta = t2 - t1;
    ...
}
```

Data Collection: Know your Clocks

```
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    delta = t2 - t1;
    ...
}
```

What exactly does `getTime()` do?

Data Collection: Know your Clocks

```
int clock_gettime(clockid_t,  struct timespec *);
```

- `CLOCK_REALTIME`
- `CLOCK_REALTIME_COARSE`
- `CLOCK_MONOTONIC`
- `CLOCK_MONOTONIC_COARSE`
- `CLOCK_MONOTONIC_RAW`
- `CLOCK_BOOTTIME`
- `CLOCK_PROCESS_CPUTIME_ID`
- `CLOCK_THREAD_CPUTIME_ID`

Data Collection: Know your Clocks

We used `CLOCK_MONOTONIC_RAW`

What we did:

- Check if each VM uses `CLOCK_MONOTONIC_RAW`
- If not:
  - Use an FFI to call `clock_gettime` with the right clock
  - Patch the VM to expose `clock_gettime` with the right clock

Data Collection: Sample Storage

No allocation:

```
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    delta = t2 - t1;
    write_to_file(delta);
}
```

Data Collection: Sample Storage

No allocation:

```
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    delta = t2 - t1;
    write_to_file(delta);
}
```

No guarantee as to when this is committed to disk

Data Collection: Sample Storage

Buffer up results, write at the end:

```
deltas = [];
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    deltas.append(t2 - t1);
}
write_all_to_file(deltas);
```

Data Collection: Sample Storage

Buffer up results, write at the end:

```
deltas = [];
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    deltas.append(t2 - t1);
}
write_all_to_file(deltas);
```

Will cause reallocations of the list and memory fragmentation

Data Collection: Sample Storage

Pre-allocation:

```
deltas = num_iters * [-1];
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    deltas[i] = t2 - t1;
}
write_all_to_file(deltas);
```

Data Collection: Sample Storage

Pre-allocation:

```
deltas = num_iters * [-1];
for i in 0..num_iters {
    t1 = getTime();
    run_benchmark();
    t2 = getTime();
    deltas[i] = t2 - t1;
}
write_all_to_file(deltas);
```

Avoids array reallocations *and* excessive IO

And lots more. . .

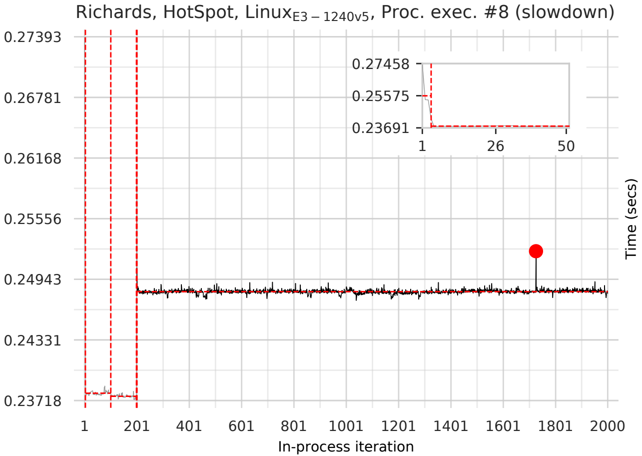
Conclusions

- We engineered a benchmark runner from scratch
    - Measures variance in the VMs/benchmarks and nothing else.
- Current benchmarking practice didn't suit our goals
- This stuff is hard

Future Work

- Which checks/controls matter most?
- What have we missed?

Thanks for Listening



Time for questions!