Virtual Machine Warmup Blows Hot and Cold

Blinded for submission

Abstract

Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: first the *warmup* phase determines which parts of a program would most benefit from dynamic compilation and JIT compiles them into machine code; after compilation has occurred, the program is said to be at *peak performance*. When measuring the performance of JIT compiling VMs, data collected during the warmup phase is generally discarded, placing the focus on peak performance. In this paper we first run a number of small, deterministic benchmarks on a variety of well known VMs, before introducing a rigorous statistical model for determining when warmup has occurred. Across 3 benchmarking machines only 43.3–56.5% of (VM, benchmark) pairs conform to the traditional view of warmup and none of the VMs consistently warms up.

1. Introduction

Many modern languages are implemented as Virtual Machines (VMs) which use a Just-In-Time (JIT) compiler to translate 'hot' parts of a program into efficient machine code at run-time. Since it takes time to determine which parts of the program are hot, and then compile them, programs which are JIT compiled are said to be subject to a *warmup* phase. The traditional view of JIT compiled VMs is that program execution is slow during the warmup phase, and fast afterwards, when a steady state of *peak performance* is said to have been reached (see Figure 1 for a simplified view of this). This traditional view underlies most benchmarking of JIT compiled VMs, which usually require running benchmarks several times within a single VM process, discarding any timing data collected before warmup is complete, and reporting only peak performance figures.

The fundamental aim of this paper is to test the following hypothesis, which captures a constrained notion of the traditional view of warmup:

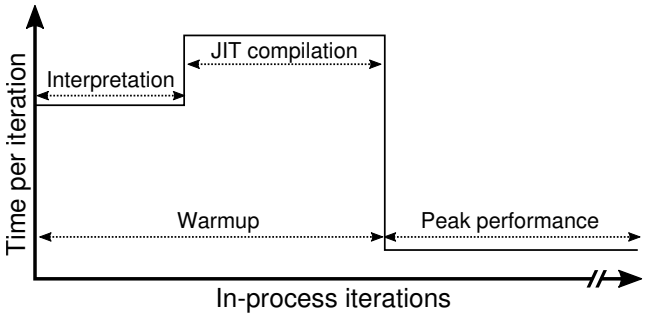


Figure 1: The traditional view of warmup: a program starts slowly executing in an interpreter; once hot parts of the program are identified, they are translated by the JIT compiler to machine code; at this point warmup is said to have completed, and a steady state of peak performance reached.

H1 Small, deterministic programs exhibit traditional warmup behaviour.

We present a carefully designed experiment where a number of simple benchmarks are run on seven VMs and GCC for a large number of *in-process iterations* and repeated using fresh *process executions* (i.e. each process execution runs multiple in-process iterations). We also introduce the first automated approach to determining *when* warmup has completed, based on statistical changepoint analysis.

While our results show that some benchmarks on some VMs run as per the traditional view, many surprising cases exist: some benchmarks slowdown rather than warmup; some never hit a steady state; and some perform very differently over different process executions. Of the seven VMs we looked at, none consistently warms up.

Our results clearly invalidate Hypothesis H1, showing that the traditional view of warmup is no longer valid (and, perhaps, that it may not have held in the past). This is of importance to both VM developers and users: much prior VM benchmarking is likely to be partly misleading: and it is likely to have allowed some ineffective, and perhaps some deleterious, optimisations to be included in production VMs.

In order to test Hypothesis H1, we first present a carefully designed experiment (Section 3). We then introduce a new statistical method for automatically classifying benchmarks' warmup style, and present steady-state in-process iteration

times (when possible) based upon the classifications (Section 4). After presenting the results from our main experiment (Section 5), we run a smaller experiment to understand whether JIT compilation and Garbage Collection (GC) are responsible for unusual warmup patterns (Section 5.3). As a useful side bonus of our main experiment, we present data for VM *startup* time: how long it takes until a VM can start executing any user code (Section 6).

We present supporting material in the Appendix: we show that our statistical method can be applied to the DaCapo and Octane benchmark suites, both run in a conventional manner (Appendix A); and we present a curated series of plots of interesting data (Appendix C). A separate document contains the complete plots of all data from all machines.

The repeatable experiment we designed, as well as the specific results used in this paper, can be downloaded from:

Blinded for submission

2. Background

Figure 1 shows the traditional view of warmup. When a program begins running on a JIT compiled VM, it is typically (slowly) interpreted; once 'hot' (i.e. frequently executed) loops or methods are identified, they are dynamically compiled into machine code; and subsequent executions of those loops or methods use (fast) machine code rather than the (slow) interpreter. Once machine code generation has completed, the VM is said to have finished warming up, and the program to be executing at a steady state of peak performance.[1] While the length of the warmup period is dependent on the program and JIT compiler, all JIT compiling VMs assume this performance model holds true [12].

Benchmarking of JIT compiled VMs typically focusses on peak performance, partly due to an assumption that warmup is both fast and inconsequential to users. The methodologies used are typically straightforward: benchmarks are run for a number of in-process iterations within a single VM process execution. The first n in-process iterations (typically a small value such as 5) are then discarded, on the basis that warmup *should* have completed in that period, whether or not it has actually done so.

A more sophisticated VM benchmarking methodology was developed by Kalibera & Jones [11, 12]. After a specific VM / benchmark combination has been run for a small number of process executions, a human must determine at which in-process iteration warmup has occurred. A larger number of VM process executions are then run, and the previously determined cut-off point applied to each process's iterations. The Kalibera & Jones methodology observes that some benchmarks do not obviously warm up; and that others follow cyclic patterns post-warmup (e.g. in-process iteration m is slow, $m + 1$ is fast, for all even values of $m > n$). In the latter case, the Kalibera & Jones methodology requires a consistent in-process iteration in the cycle (ideally the first post-warmup iteration) be picked for all process executions, and used for statistical analysis.

Despite its many advances, the Kalibera & Jones methodology's reliance on human expertise means that it cannot provide a fully repeatable way of determining when warmup has completed. Because of this "determining when a system has warmed up, or even providing a rigorous definition of the term, is an open research problem" [17].

3. Methodology

To test Hypothesis H1, we designed an experiment which uses a suite of micro-benchmarks: each is run with 2000 in-process iterations and repeated using 10 process executions. We have carefully designed our experiment to be repeatable and to control as many potentially confounding variables as is practical. In this section we detail: the benchmarks we used and the modifications we applied; the VMs we benchmarked; the machines we used for benchmarking; and the Krun system we developed to run benchmarks.

3.1 The Micro-benchmarks

The micro-benchmarks we use are as follows: *binary trees*, *spectralnorm*, *n-body*, *fasta*, and *fannkuch redux* from the Computer Language Benchmarks Game (CLBG) [3]; and *Richards*. Readers can be forgiven for initial scepticism about this set of micro-benchmarks. They are small and widely used by VM authors as optimisation targets. In general they are more effectively optimised by VMs than average programs; when used as a proxy for other types of programs (e.g. large programs), they tend to overstate the effectiveness of VM optimisations (see e.g. [16]). In our context, this weakness is in fact a strength: small, deterministic, and widely examined programs are our most reliable means of testing Hypothesis H1. Put another way, if we were to run arbitrary programs and find unusual warmup behaviour, a VM author might reasonably counter that "you have found the one program that exhibits unusual warmup behaviour".

For each benchmark, we provide versions in C, Java, JavaScript, Python, Lua, PHP, and Ruby. Since most of these benchmarks have multiple implementations in any given language, we picked the versions used in [6], which represented the fastest performers at the point of that publication. We lightly modified the benchmarks to integrate with our benchmark runner (see Section 3.5). For the avoidance of doubt we did not interfere with any VM's GC (e.g. we did not force a collection after each iteration).

3.1.1 Ensuring Determinism

User programs that are deliberately non-deterministic programs are unlikely to warm-up in the traditional fashion. We

[1] The traditional view applies equally to VMs that perform immediate compilation instead of using an interpreter, and to those VMs which have more than one layer of JIT compilation (later JIT compilation is used for 'very hot' portions of a program, trading slower compilation time for better machine code generation).

therefore wish to guarantee that our benchmarks are, to the extent controllable by the user, deterministic: that they take the same path through the Control Flow Graph (CFG) on all process executions and in-process iterations.[2]

To check whether the benchmarks were deterministic at the user-level, we created versions with `print` statements at all possible points of CFG divergence (e.g. `if` statements' true and false branches). These versions are available in our experimental suite. We first ran the modified benchmarks with 2 process executions and 20 in-process iterations, and compared the outputs of the two processes. This was enough to show that the *fasta* benchmark was non-deterministic in all language variants, due to its random number generator not being reseeded. We fixed this by moving the random seed initialisation to the start of the in-process iteration main loop.

In order to understand the effects of compilation non-determinism, we then compiled VMs and ran our modified benchmarks on two different machines. We then observed occasional non-determinism in Java benchmarks. This was due to the extra class we had added to each benchmark to interface between it and the benchmark runner: sometimes, the main benchmark class was lazily loaded after benchmark timing had started in a way that we could observe. We solved this by adding an empty static method to each benchmark, which our extra classes then call via a static initialiser, guaranteeing that the main benchmark class is eagerly loaded. Note that we do not attempt to eagerly load other classes: lazy loading is an inherent part of the JVM specification, and part of what we need to measure.

3.2 Measuring Computation and Not File Performance

By their very nature, micro-benchmarks tend to perform computations which can be easily optimised away. While this speaks well of optimising compilers, benchmarks whose computations are entirely removed are rarely useful [17]. To prevent optimisers removing such code, many benchmarks write intermediate and final results to `stdout`. However, this then means that one starts including the performance of file routines in libraries and the kernel in measurements, which can become a significant part of the eventual measure.

To avoid this, we modified the benchmarks to calculate a checksum during each in-process iteration. At the end of each in-process iteration the checksum is compared to a predetermined value; if the comparison fails then the incorrect checksum is written to `stdout`. Using this idiom means that optimisers can't remove the main benchmark code even though no output is produced. We also use this mechanism to give some assurance that each language variant is performing the same work, as we use a single checksum value for each benchmark, irrespective of language.

[2] Note that non-determinism beyond that controllable by the user (i.e. non-determinism in low-level parts of the VM) is part of what we need to test for Hypothesis H1.

3.3 VMs under investigation

We ran the benchmarks on the following language implementations: GCC 4.9.3; Graal #9405be47 (an alternative JIT compiler for HotSpot); HHVM 3.14.0 (a JIT compiling VM for PHP); JRuby+Truffle #170c9ae6; HotSpot 8u72b15 (the most widely used Java VM); LuaJIT 2.0.4 (a tracing JIT compiling VM for Lua); PyPy 5.3.0 (a meta-tracing JIT compiling VM for Python 2.7); and V8 5.1.281.65 (a JIT compiling VM for JavaScript). A repeatable build script downloads, patches, and builds fixed versions of each VM. All VMs were compiled with GCC/G++ 4.9.3 (and GCC/G++ bootstraps itself, so that the version we use compiled itself) to remove the possibility of variance through the use of different compilers.

On OpenBSD, we skip Graal, HHVM, and JRuby+Truffle, which have not yet been ported to it. We skip *fasta* on JRuby+Truffle as it crashes; and we skip *Richards* on HHVM since it takes as long as every other benchmark on every other VM put together.

3.4 Benchmarking Hardware

With regards to hardware and operating systems, we made the following hypothesis:

H2 Moderately different hardware and operating systems have little effect on warmup.

We deliberately use the word 'moderately', since significant changes of hardware (e.g. x86 vs. ARM) or operating system (e.g. Linux vs. Windows) imply that significantly different parts of the VMs will be used (see Section 7).

In order to test Hypothesis H2, we used three benchmarking machines: $Linux_{4790K}$, a quad-core i7-4790K 4GHz, 24GB of RAM, running Debian 8; $Linux_{4790}$, a quad-core i7-4790 3.6GHz, 32GB of RAM, running Debian 8; and $OpenBSD_{4790}$, a quad-core i7-4790 3.6GHz, 32GB of RAM, running OpenBSD 5.8. Linux_{4790K} and Linux_{4790} have the same OS (with the same updates etc.) but different hardware; Linux_{4790} and OpenBSD_{4790} have the same hardware (to the extent we can determine) but different operating systems.

We disabled turbo boost and hyper-threading in the BIOS. Turbo boost allows CPUs to temporarily run in an higher-performance mode; if the CPU deems it ineffective, or if its safe limits (e.g. temperature) are exceeded, turbo boost is reduced [7]. Turbo boost can thus substantially change one's perception of performance. Hyper-threading gives the illusion that a single physical core is in fact two logical cores, inter-leaving the execution of multiple programs or threads on a single physical core, leading to a less predictable performance pattern than on physical cores alone.

3.5 Krun

Many confounding variables occur shortly before, and during the running of, benchmarks. In order to control as many of these as possible, we wrote Krun, a new benchmark runner. Krun itself is a 'supervisor' which, given a configuration

file specifying VMs, benchmarks (etc.) configures a Linux or OpenBSD system, runs benchmarks, and collects the results. Individual VMs and benchmarks are then wrapped, or altered, to report data back to Krun in an appropriate format.

In the remainder of this subsection, we describe Krun. Since most of Krun's controls work identically on Linux and OpenBSD, we start with those, before detailing the differences imposed by the two operating systems. We then describe how Krun collects data from benchmarks. Note that, although Krun has various 'developer' flags to aid development and debugging benchmarking suites, we describe only Krun's full 'production' mode.

3.5.1 Platform Independent Controls

A typical problem with benchmarking is that earlier process executions can affect later ones (e.g. a benchmark which forces memory to swap will make later benchmarks seem to run slower). Therefore, before each process execution (including before the first), Krun reboots the system, ensuring that the benchmark runs with the machine in a (largely) known state. After each reboot, Krun is executed by the init subsystem; Krun then pauses for 3 minutes to allow the system to fully initialise; calls `sync` (to flush any remaining files to disk) followed by a 30 second wait; before finally running the next process execution.

The obvious way for Krun to determine which benchmark to run next is to examine its results file. However, this is a large file which grows over time, and reading it in could affect benchmarks (e.g. due to significant memory fragmentation). On its initial invocation, and before the first reboot, Krun therefore creates a simple schedule file. After each reboot this is scanned line-by-line for the next benchmark to run; the benchmark is run; and the schedule updated (without changing its size). Once the process execution is complete, Krun can safely load the results file in and append the results data, knowing that the reboot that will occur shortly after will put the machine into a (largely) known state.

Modern systems have various temperature-based limiters built in: CPUs, for example, lower their frequency if they get too hot. After its initial invocation, Krun waits for 1 minute before collecting the values of all available temperature sensors. After each reboot's `sync`-wait, Krun waits for the machine to return to these base temperatures (± 3°C) before starting the benchmark, fatally aborting if this temperature range is not met within 1 hour. In so doing, we aim to lessen the impact of ambient temperature changes.

Krun fixes the heap and stack `ulimit` for all VM process executions (in our case, 2GiB heap and a 8MiB stack).[3] Benchmarks are run as the '`krun`' user, whose account and home directory are created afresh before each process execution to prevent cached files affecting benchmarking.

[3] Note that Linux allows users to inspect these values, but to allocate memory beyond them.

User-configurable commands can be run before and after benchmark execution. In our experiment, we switch off as many Unix daemons as possible (e.g. smtpd, crond) to lessen the effects of context switching to other processes. We also turn off network interfaces entirely, to prevent outside sources causing unbounded (potentially performance interfering) interrupts to be sent to the processor and kernel.

In order to identify problems with the machine itself, Krun monitors the system's `dmesg` buffer for unexpected entries (known 'safe' entries are ignored), informing the user if any arise. We implemented this after noticing that one machine initially ear-marked for benchmarking occasionally overheated, with the only clue to this being a line in `dmesg`. We did not use this machine for our final benchmarking.

A process's environment size can cause measurement bias [15]. The diversity of VMs and platforms in our setup makes it impossible to set a unified environment size across all VMs and benchmarks. However, the `krun` user does not vary its environment (we recorded the environment seen by each process execution and verified their size), and we designed the experiment such that, for a particular (machine, VM, benchmark) triple the additional environment used to configure the VM is of constant size.

3.5.2 Linux-specific Controls

On Linux, Krun controls several additional factors, sometimes by checking that the user has correctly set controls which can only be set manually.

Krun uses `cpufreq-set` to set the CPU governor to `performance` mode (i.e. the highest non-overclocked frequency possible). To prevent the kernel overriding this setting, Krun verifies that the user has disabled Intel P-state support in the kernel by passing `intel_pstate=disable` as a kernel argument.

As standard, Linux interrupts ('ticks') each core `CONFIG_HZ` times per second (usually 250) to decide whether to perform a context switch. To avoid these repeated interruptions, Krun checks that it is running on a 'tickless' kernel [2], which requires recompiling the kernel with the `CONFIG_NO_HZ_FULL_ALL` option set. Whilst the boot core still 'ticks', other cores only 'tick' if more than one runnable process is scheduled.

Similarly, Linux's `perf` profiler may interrupt cores up to 100,000 times a second. We became aware of `perf` when Krun's `dmesg` checks notified us that the kernel had decreased the sample-rate by 50% due to excessive sampling overhead. This is troubling from a benchmarking perspective, as the change of sample rate could change benchmark performance mid-process execution. Although `perf` cannot be completely disabled, Krun sets it to sample at most once per second, minimising interruptions.

Finally, Krun disables Address Space Layout Randomisation (ASLR). While ASLR is a sensible security precaution

for everyday use, it introduces obvious non-determinism between process executions.[4]

3.5.3 OpenBSD-specific Controls

Relative to Linux, OpenBSD exposes many fewer knobs to users. Nevertheless, there are two OpenBSD specific features in Krun. First, Krun sets CPU performance to maximum by invoking `apm -H` prior to running benchmarks (equivalent to Linux's `performance` mode). Second, Krun minimises the non-determinism in OpenBSD's malloc implementation, for example not requiring `realloc` to always reallocate memory to an entirely new location. The `malloc.conf` flags we use are `sfghjpru`.

3.5.4 The Iterations Runners

To report timing data to Krun, we created an *iterations runner* for each language under investigation. These take the name of a specific benchmark and the desired number of in-process iterations, run the benchmark appropriately, and once it has completed, print the times to `stdout` for Krun to capture. For each in-process iteration we measure (on Linux and OpenBSD) the wall-clock time taken, and (Linux only) core cycle, APERF, and MPERF counters.

We use a monotonic wall-clock timer with sub-millisecond accuracy (`CLOCK_MONOTONIC_RAW` on Linux, and `CLOCK_MONOTONIC` on OpenBSD). Although wall-clock time is the only measure which really matters to users, it gives no insight into multi-threaded computations: we therefore also record core cycle counts using the `CPU_CLK_UNHALTED.CORE` counter to see what work each core is actually doing. In contrast, we use the ratio of APERF/MPERF deltas solely as a safety check that our wall-clock times are valid. The `IA32_APERF` counter increments at a fixed frequency for each instruction executed; the `IA32_MPERF` counter increments at a rate proportional to the processor's current frequency. With an APERF/MPERF ratio of precisely 1, the processor is running at full speed; below 1 it is in a power-saving mode; and above 1, turbo boost is being used.

A deliberate design goal of the in-process iterations runners is to minimise timing noise and distortion from measurements. Since system calls can have a significant overhead (on Linux, calling functions such as `write` can evict as much as two thirds of an x86's L1 cache [18]), we avoid making any system calls other than those required to take measurements. We avoid in-benchmark I/O and memory allocation by storing measurements in a pre-allocated buffer and only writing measurements to `stdout` after all in-process iterations have completed (see Listing 2 for an example). However, the situation on Linux is complicated by our need to read core cycle and APERF/MPERF counts from

```
void krun_measure(int mdata_idx) {
  struct krun_data *data = &(krun_mdata[mdata_idx]);
  if (mdata_idx == 0) { // start benchmark readings
    for (int core = 0; core < num_cores; core++) {
      data->aperf[core] = read_aperf(core);
      data->mperf[core] = read_mperf(core);
      data->core_cycles[core] = read_core_cycles(core);
    }
    data->wallclock = krun_clock_gettime_monotonic();
  } else {              // end benchmark readings
    data->wallclock = krun_clock_gettime_monotonic();
    for (int core = 0; core < num_cores; core++) {
      data->core_cycles[core] = read_core_cycles(core);
      data->aperf[core] = read_aperf(core);
      data->mperf[core] = read_mperf(core);
    }
  }
}
```

Listing 1: `krun_measure`: Measuring before (the `if`'s true branch) and after (its false branch) a benchmark. Since wall-clock time is the most important measure, it is innermost; since the APERF/MPERF counters are a sanity check, they are outermost. Note that the APERF/MPERF counters must be read in the same order before and after a benchmark.

```
wallclock_times = [0] * iters
for i in xrange(iters):
    krun_measure(0)   # Start timed section
    bench_func(param) # Call the benchmark
    krun_measure(1)   # End timed section
    wallclock_times[i] = \
        krun_get_wallclock(1) - krun_get_wallclock(0)
js = { "wallclock_times": wallclock_times }
sys.stdout.write("%s\n" % json.dumps(js))
```

Listing 2: An elided version of the Python in-process iterations runner (with core cycles etc. removed).

Model Specific Register (MSR) file device nodes,[5] which are relatively slow (see Section 7). Since wall-clock time is the most important measure, we ensure that it is the innermost measure taken (i.e. to the extent we control, it does not include the time taken to read core cycle or APERF/MPERF counts) as shown in Listing 1.

The need to carefully sequence the measurements means that we cannot rely on user-level libraries: not all of the VMs in our experiment give us access to the appropriate monotonic timer. We therefore implemented a small C library (`libkruntime.so`) which exposes these measures (see Listing 1 for an example). When possible (all VMs apart from JRuby+Truffle, HHVM and V8), we used a language's FFI to dynamically load this library in; in the remaining cases, we linked the library directly against the VM, which then required us to add user-language visible functions to access them. Core-cycle, APERF, and MPERF counts are 64-bit unsigned integers; since JavaScript and current versions of LuaJIT do not support integers, and since PHP's maximum integer size varies across OS and PHP versions, we convert the 64-bit unsigned measurements to double-

[4] The Stabilizer system [8] is an intriguing approach for obtaining reliable statistics in the face of features such as ASLR. Unfortunately we were not able to build it on a modern Linux system.

[5] We forked, and slightly modified, Linux's `msr` device driver to allow us to easily access the MSRs as a non-root user.

precision floating point values in those VMs, throwing an error if this leads to a loss of precision.

4. Classifying Warmup

The main data created by our experiment is the time taken by each in-process iteration to run. Formally, this is time series data of length 2000. In this Section we explain how we use statistical changepoint analysis to enable us to understand this time series data and classify the results we see, giving the first automated method for identifying warmup.

4.1 Outliers

As is common with analyses of time series data, we first identify outliers (i.e. in-process iterations with much larger/smaller times than their near neighbours) so that our later analysis can ignore them. In our case, the intuition is that such outliers are likely to be the result of JIT compilation, GC, or of other processes interrupting benchmarks. We use Tukey's method [19], conservatively defining an outlier as one that, within a sliding window of 200 in-process iterations, lies outside the median $\pm 3 * (90\%\text{ile} - 10\%\text{ile})$. In order that we avoid classifying slow warm-up iterations at the start of an execution as outliers (when they are in fact likely to be important warmup data), we ignore the first 200 in-process iterations. Of the 2440000 in-process iterations, 0.4% are classified as outliers, with the most for any single process execution being 15.0% of in-process iterations.

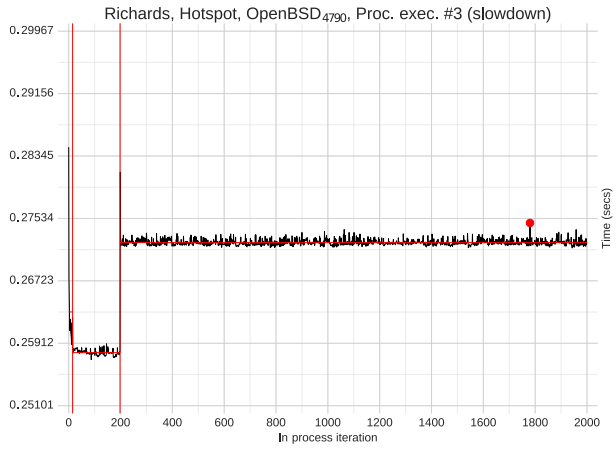


Figure 2: An example of changepoint analysis on a run-sequence plot (a process execution of Richards on HotSpot from OpenBSD_{4790}). This shows two changepoints (the vertical red lines): one soon after startup; and another around iteration 200. The horizontal red lines are the means of the changepoint segments. The red blob denotes a solitary outlier, which was discounted by the changepoint analysis.

4.2 Changepoint Analysis

Intuitively, in order to uncover if/when warmup has completed, we need to determine when the times taken by in-process iterations have become faster within a process execution. Put another way, we expect to see a number of in-process executions taking time t to be followed by a number at time t'. In order to automatically determine when this occurs, we use statistical changepoint analysis (see [9] for an introduction). Formally, a *changepoint* is a point in time where the statistical properties of prior data are different to the statistical properties of subsequent data; the data between two changepoints is a *changepoint segment*. There are various ways that one can determine when a changepoint has occurred, but the best fit for our data is to consider changes in both the mean and variance of in-process iterations.

In order to automate this, we use the `cpt.meanvar` function in the R `changepoint` package [13], passing $15 \log n$ (where n is the time series length minus the number of outliers) to the `penalty` argument. This function uses the PELT algorithm [14], which can detect arbitrary numbers of changepoints; it returns changepoint locations and the mean and variance of each segment. For the particular, somewhat noisy, data we obtain, the algorithm is slightly oversensitive to changes in the variance, particularly for very fast in-process iterations. Although we use a microsecond timer, our experience is that time deltas under 0.001s are at the mercy of the non-determinism inherent in a real machine and OS. We therefore merge consecutive sequences of segments (from left-to-right) whose confidence intervals (calculated using the inter-quartile range) overlap, and/or whose means are within 0.001s of each other. Figure 2 shows an example of a run-sequence plot with outliers, changepoints, and changepoint segments superimposed upon it.

4.3 Classifications

Building atop changepoint analysis, we can then define useful classifications of time-series data from VM benchmarks.

First, we define classifications for individual process executions. A process execution with no changepoints is classified as *flat* (−). Since all our benchmarks run for 2000 in-process iterations we (somewhat arbitrarily) define that a process execution reaches a steady-state if the last 500 in-process iterations are part of a single changepoint segment. A process execution with a changepoint in its last 500 in-process iterations is classified as *no steady state* (⇜⇝). A steady-state process execution whose final segment: is the fastest execution time of all segments is classified as *warmup* (⌞); is not the fastest execution time of all segments is classified as *slowdown* (⌜).

Second, we define classifications for a (VM, benchmark) pair as follows: if its process executions all share the same classification (e.g. warmup) then we classify the pair the

Class.	$\text{Linux}_{4790\text{K}}$	Linux_{4790}	OpenBSD_{4790}
	(VM, benchmark) pairs		
–	21.7%	26.1%	23.3%
⌊	28.3%	30.4%	20.0%
⌋	8.7%	8.7%	6.7%
⇜	4.3%	2.2%	0.0%
⋍	37.0%	32.6%	50.0%
	Process executions		
–	29.6%	29.6%	45.0%
⌊	45.9%	42.6%	37.7%
⌋	16.7%	19.1%	15.7%
⇜	7.8%	8.7%	1.7%

Table 1: Relative proportions of classifiers across benchmarking machines. Classifiers key: –: flat, ⌊: warmup, ⌋: slowdown, ⇜: no steady state, ⋍: inconsistent.

same way (in this example, warmup); otherwise we classify the pair as *inconsistent* (⋍).

Informally, we suggest that benchmarks whose behaviour is either flat or warmup are 'good' (flat benchmarks may be unobservably fast warmup), while benchmarks which are either slowdown or no steady state are 'bad'. Inconsistent benchmarks are sometimes easily judged through the classification of their process executions, which we report in brackets. A benchmark which is ⋍ (6⌋, 4⇜) (i.e. inconsistent, with 6 process executions classified as slowdown, 4 as no steady state) is clearly 'bad'. However, there are some more subjective cases. For example, consider a benchmark which is ⋍ (8⌊, 2–). Manual investigation of the individual process execution's data may show that there is considerable difference between the warmup and flat process executions and that it is clearly 'bad'. However, if the warmups are small in magnitude (and thus are not far from being classified as flat) some may consider it 'good' whilst some will consider it 'neutral', on the basis that they should all be consistent. To avoid such subjectivity, we consider inconsistent benchmarks to be either (obviously) 'bad' or 'neutral'.

4.4 Timings

For benchmarks which reach a steady state (–, ⌊, or ⌋) we report the number of iterations and the time in seconds to reach the steady state, as well as the performance of the steady state itself in seconds (defined as the final changepoint segment's mean). For the latter two measures, we report the median time over all 10 process executions, and report 99% confidence intervals (bootstrapped with 10000 iterations, sampling with replacement).

5. Results

Our experiment consists of 1220 process executions and 2440000 in-process iterations. Table 1 summarises the (VM, benchmark) pairs and process executions for each benchmarking machine. Taking $\text{Linux}_{4790\text{K}}$ as a representative example, 50.0% of (VM, benchmark) pairs and 75.5% of process executions have 'good' warmup (–, ⌊) (for comparison, Linux_{4790} is 56.5% and 72.2% respectively and OpenBSD_{4790} is 43.3% and 82.7% respectively, though the latter runs fewer benchmarks). Fewer (VM, benchmark) pairs than process executions have 'good' warmup because some inconsistent (VM, benchmark) pairs have some 'good' warmup process executions. On $\text{Linux}_{4790\text{K}}$ 8.7% of (VM, benchmark) pairs slowdown and 37.0% are inconsistent (with over two thirds of those being obviously 'bad'); Linux_{4790} and OpenBSD_{4790} are broadly similar. Taking $\text{Linux}_{4790\text{K}}$ as an example, one third of its (VM, benchmark) pairs are 'bad', clearly invalidating Hypothesis H1.

Table 1 clearly shows that the results from $\text{Linux}_{4790\text{K}}$ and Linux_{4790} (both running Linux but on moderately different hardware) are comparable, with the relative proportions of all measures being only a few percent different. OpenBSD_{4790} (running OpenBSD, but on the same hardware as $\text{Linux}_{4790\text{K}}$) is somewhat different. This is in large part because it is unable to run benchmarks on Graal, HHVM, and JRuby+Truffle (see Section 3.3): of the benchmarks it can run, most behave similarly, with the notable exception of LuaJIT, whose benchmarks are more often classified as non-flat (see Table 7 in the Appendix). With that in mind, we believe that our results are a fairly clear validation of Hypothesis H2.

Looking at one machine's data brings out further detail. For example, Table 2 (with data from $\text{Linux}_{4790\text{K}}$) enables to make us several observations. Of the 6 benchmarks, only *n-body* comes close to 'good' warmup behaviour on all VMs on all machines (the one C slowdown excepted): the other 5 benchmarks all have at least one VM which consistently slows down. Second, there appears to be a bimodal pattern for the number of in-process iterations required to reach a steady state (when it exists): VMs tend either to reach it very quickly (often in 10 or fewer in-process iterations) or take hundreds of in-process iterations. Those that take a long time to stabilise are troubling, because previous benchmarking methodologies will not have run them long enough to see this steady state emerge. Finally, there seems to be very little correlation between the time it takes a VM to reach a steady state and the performance of that steady state: i.e. it does not seem to be the case that VMs which take a long time to warmup are creating more efficient steady-state code.

5.1 Warmup Plots

To understand why we see so many unexpected cases, we created a series of plots for process executions. All plots have in-process iteration number on the x-axis. For y-axes: *run-sequence plots* show wall-clock time; and (on Linux) *core-cycle plots* show the processor cycle count per-core. The run-sequence and core-cycle plots in Figures 3, 6, and 4 show examples of traditional warmup, slowdown, and no

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*binary trees*	⇜				*n-body*	⇹ (9−, 1⌋)			
Graal		⌊	12 ±5	2.78 ±1.047	0.17329 ±0.001097		⌊	4 ±0	0.75 ±0.036	0.15095 ±0.000003
HHVM		⌊	73 ±11	151.71 ±24.233	2.06641 ±0.002518		⌊	132 ±0	905.10 ±2.404	2.71942 ±0.000306
HotSpot		⌊	6 ±7	1.12 ±1.296	0.17220 ±0.000601		⌊	1 ±0	0.16 ±0.000	0.15616 ±0.000007
JRuby+Truffle		⇹ (7⌊, 3⌋)					⌊	68 ±0	19.93 ±0.161	0.23255 ±0.000603
LuaJIT		⇹ (5−, 4⌋, 1⌊)					−			0.26651 ±0.000046
PyPy		⇹ (6⇜, 4⌋)					−			1.63684 ±0.000594
V8		⇹ (9⌊, 1⌋)					⌊	366 ±12	92.20 ±3.156	0.25066 ±0.000049
C	*fannkuch redux*	−			0.37885 ±0.000073	*Richards*	−			0.68768 ±0.000340
Graal		⇹ (9⌊, 1⌋)					⇹ (7⌊, 2⌋, 1⇜)			
HHVM		⌊	9 ±0	39.64 ±0.003	1.32396 ±0.000415					
HotSpot		⇹ (9⇜, 1⌊)					⌋	201 ±1	46.25 ±1.040	0.24374 ±0.003678
JRuby+Truffle		⌋	998 ±0	928.46 ±6.095	1.05923 ±0.009511		⇹ (7⌋, 3⌊)			
LuaJIT		−			0.50777 ±0.000048		−			2.60442 ±0.000459
PyPy		⇹ (8−, 2⌊)					⌊	3 ±4	2.88 ±3.712	0.88289 ±0.000215
V8		−			0.27360 ±0.000030		⇹ (5⌊, 5⌋)			
C	*fasta*	−			0.06649 ±0.000040	*spectralnorm*	⇹ (7⌊, 3−)			
Graal		⌊	3 ±1	0.62 ±0.221	0.13348 ±0.000214		⌋	13 ±0	6.08 ±0.247	0.93004 ±0.000024
HHVM		⇹ (9⌊, 1⌋)					⌊	355 ±402	596.22 ±579.975	1.48467 ±0.000291
HotSpot		⇹ (9⌊, 1⌋)					⇹ (8⌊, 2−)			
JRuby+Truffle							⇹ (7⌋, 3⌊)			
LuaJIT		−			0.29931 ±0.000750		−			0.42033 ±0.000026
PyPy		⇜					⇹ (9−, 1⌊)			
V8		⌋	523 ±0	533.02 ±0.106	1.03610 ±0.000318		⌊	2 ±0	0.84 ±0.000	0.42035 ±0.000013

Table 2: Benchmark results for $\text{Linux}_{4790\text{K}}$. For processes which reach a steady state: *steady iter (#)* is the median in-process iteration that the steady state was reached at; *steady iter (s)* is the wall-clock time since the beginning of the process execution that the steady state was reached; *steady perf (s)* is the median performance of the steady state segment across all process executions. We give the constituent classifications of inconsistent benchmarks e.g. ⇹ (2⌋, 8⌊) means 2 of the process executions were slowdown, and 8 warmup.

steady state respectively. Figure 5 shows an example of inconsistent process executions.

Core-cycle plots help us understand how VMs use, and how the OS schedules, threads. Benchmarks running on single-threaded VMs are characterised by a high cycle-count on one core, and very low (though never quite zero) values on all other cores. Such VMs may still be migrated between cores during a process execution, as can be clearly seen in Figure 4. Although multi-threaded VMs can run JIT compilation and / or GC in parallel, it is typically hard to visually detect such parallelism as it tends to be accompanied by frequent migration between cores. However, it can often easily be seen that several cores are active during the first few in-process iterations whilst JIT compilation occurs.

5.2 Cyclic data

We suspected that enough benchmarks would exhibit cyclic behaviour (as seen in Figure 7) to require special support in our automated analysis. To quantify this, we took one process execution from each benchmark run on $\text{Linux}_{4790\text{K}}$, removed the first 1500 in-process iterations, and generated an autocorrelation plot for the remainder (in similar fashion to Kalibera & Jones). 4 (VM, benchmark) pairs showed what we have come to call 'fully cyclic' behaviour, where all in-

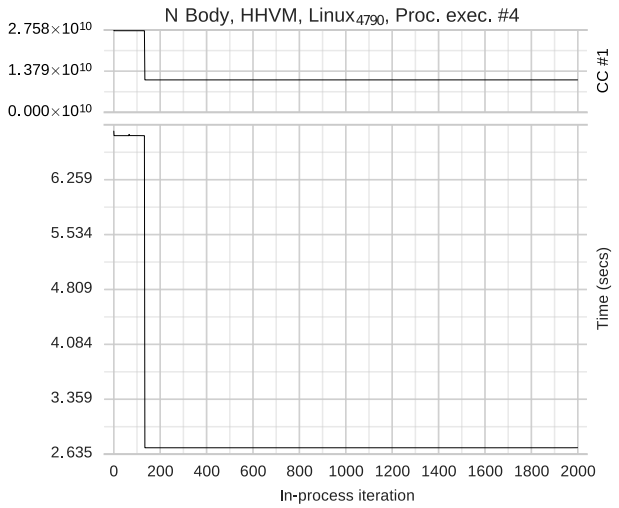


Figure 3: An example of traditional warmup behaviour with the n-body benchmark on HHVM (process execution 4 of 10 run on $\text{Linux}_{4790\text{K}}$). The run-sequence plot (bottom) shows that warmup completed by in-process iteration #134. The core-cycle plot (top) shows that the main benchmark computation occurred almost entirely on core 1: cores 0, 2 and 3 Cycle Counts (CC) were low enough for us to classify them as 'inactive', and we have thus elided them.

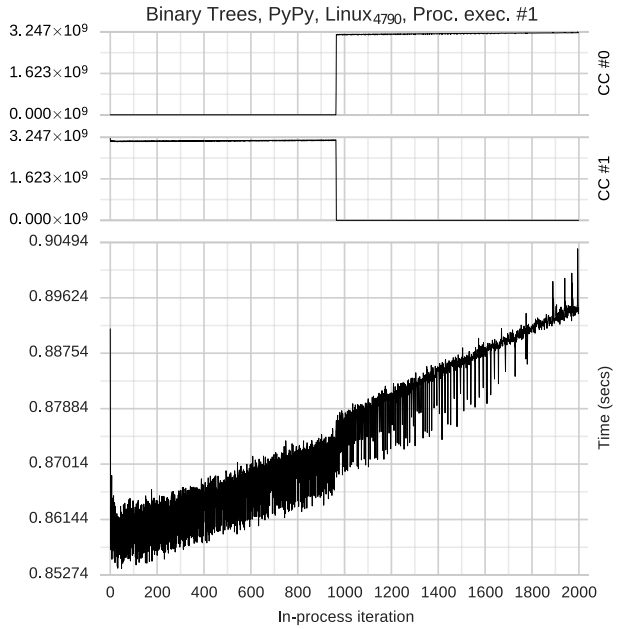


Figure 4: No steady state. There is a small behavioural shift around iteration 950, where the VM migrates from (tickless) core 1 to (ticked) core 0.

process iterations repeat in a predictable pattern; 8 showed 'partially cyclic' behaviour, where most iterations are noise around a mean, but every n iterations there is a predictable and significant peak (with n varying from 5 to 19).

For partially cyclic data, or fully cyclic data with a large period (as in Figure 7), our automated analysis handles the situation appropriately (classifying Figure 7 as no steady state). However, changepoint analysis cannot currently identify small cycle periods in fully cyclic data. Since this only happens for 2 of 47 (VM, benchmark) pairs, both of which have a small absolute difference between the cycle's elements, we consider this a minor worry.

5.3 The Effects of Compilation and GC

The large number of non-warmup cases in our data led us to make the following hypothesis:

H3 Non-warmup process executions are largely due to JIT compilation or GC events.

To test this hypothesis, we made use of the fact that both HotSpot and PyPy allow information about the duration of JIT compilation and GC to be recorded. Since recording this additional data could potentially change the results we collect, it is only collected when Krun is explicitly set to 'instrumentation mode'. An example instrumentation-plot can be seen in the slowdown of Figure 6 where there is a clear correlation (though we cannot be entirely sure that it is a causation) between a JIT compilation and the slowdown. However, some other notably odd examples have no such correlation, as can be seen in Figure 8.

The relatively few results we have with GC and JIT compilation events, and the lack of a clear message from them, means that we feel unable to validate or invalidate Hypothesis H3. Whilst some non-warmups are plausibly explained by GC or JIT compilation events, many are not, at least on HotSpot and PyPy. When there is no clear explanation, we have very little idea what might be the cause of the unexpected behaviour. It may be that obtaining similar data from other VMs may help clarify this issue, but not all VMs support this feature and, in our experience, those that do support it do not always document it accurately.

6. Startup Time

The data presented thus far in the paper has all been collected after the VM has started executing the user program. The period between a VM being initially invoked and it executing the first line of the user program is the VM's *startup* time, and is an important component in understanding a VM's real-world performance.

A small modification to Krun enables us to measure startup. We prepend each VM execution command with a small C wrapper, which prints out wall-clock time before immediately executing the VM itself; and, for each language under investigation, we provide a 'dummy' iterations runner which simply prints out wall-clock time. In other words, we measure the time just before the VM is loaded and at the first point that a user-level program can execute code on the VM;

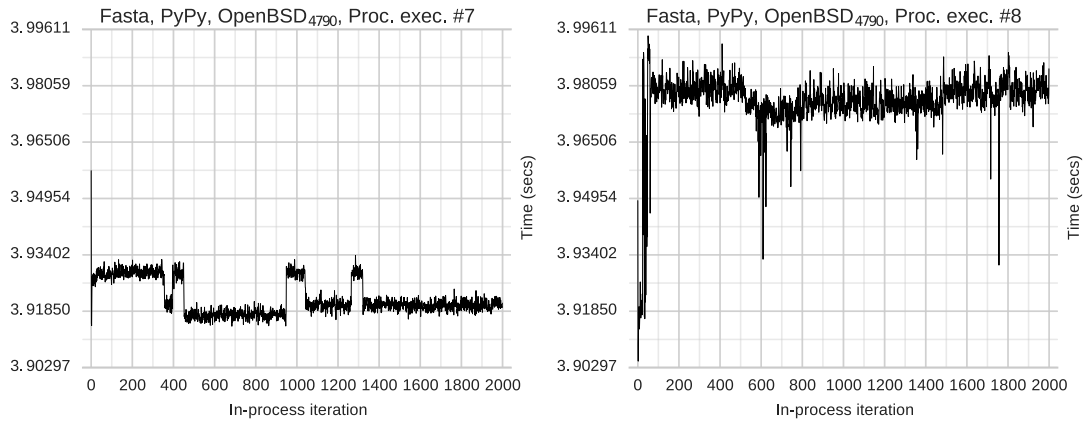


Figure 5: An example of inconsistent process executions for the same (machine, VM, benchmark) triple.

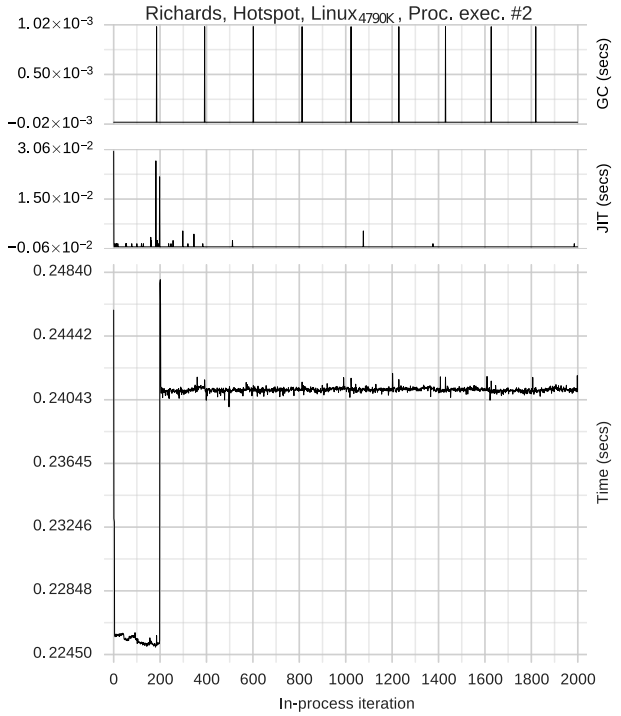


Figure 6: Slowdown at in-process iteration #199, with GC and JIT measurements turned on. Although these additional measures add overhead, in this case they make no observable difference to the main run-sequence graph. They allow us to observe that the slowdown is correlated with two immediately preceding JIT compilation events that may explain the drop in performance. Note that the regular GC spikes have only a small effect on the time of in-process iterations.

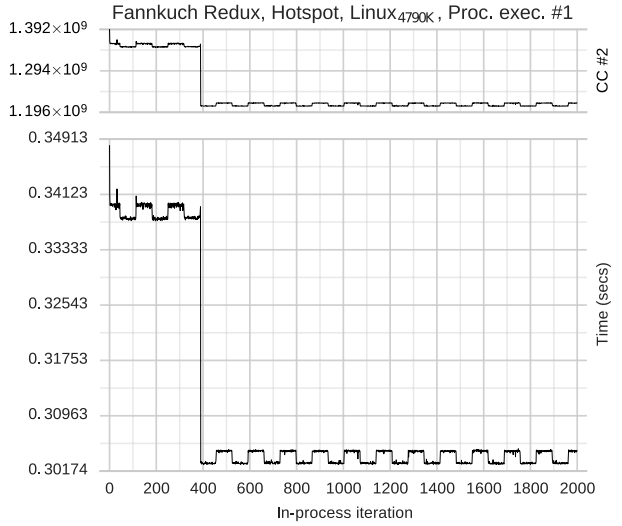


Figure 7: Cycles in wall-clock times and core-cycle counts.

the delta between the two is the startup time. For each VM we run 200 process executions (for startup, in-process executions are irrelevant, as the user-level program completes as soon as it has printed out wall-clock time).

Table 3 shows startup times from both $\text{Linux}_{4790\text{K}}$ and OpenBSD_{4790}. Since Krun reboots before each process execution, these are measures of a 'cold' start and thus partly reflect disk speed etc. (though both machines load VMs from an SSD). As this data clearly shows, startup time varies significantly amongst VMs: taking C as a baseline, the fastest VM (LuaJIT) is around 2x slower, whilst the slowest VM (JRuby+Truffle) is around 1000x slower, to startup.

7. Threats to Validity

While we have designed our experiment as carefully as possible, we do not pretend to have controlled every possibly confounding variable. It is inevitable that there are further confounding variables that we are not aware of, some of which may be controllable, although many may not. It is possible that confounding variables we are not aware of have coloured our results.

We have tried to gain a partial understanding of the effects of different hardware on benchmarks by using machines with the same OS but different hardware. However, while the hardware between the two is different, more dis-

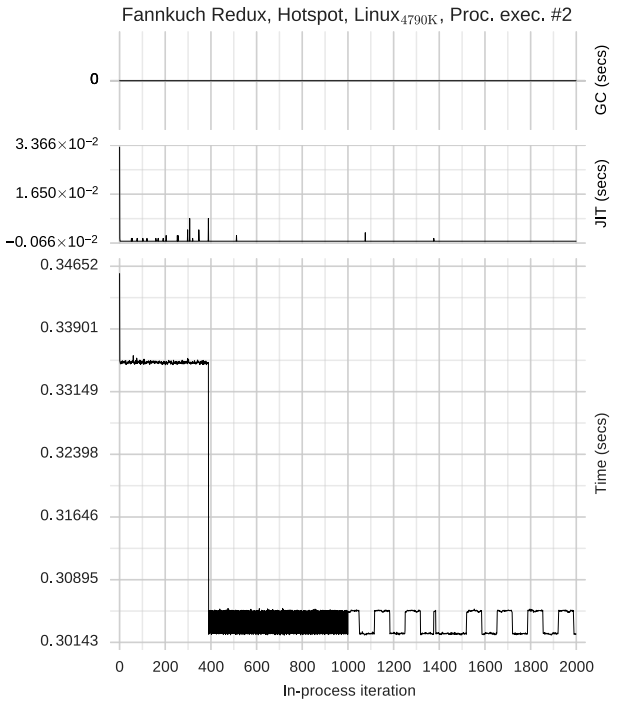


Figure 8: The cyclic example from Figure 7 with GC and JIT compilation instrumentation enabled. The additional overhead changes some of the details in the first half of the process execution, but the second half is unchanged. As this example shows, the cycles are not correlated with GC (of which there is none) or JIT compilation.

	Linux$_{4790K}$	OpenBSD$_{4790}$
C	0.00213 ±0.000005	0.00110 ±0.000004
Graal	0.24744 ±0.000339	
HHVM	0.78364 ±0.001834	
Hotspot	0.10655 ±0.000411	0.19041 ±0.000148
JRubyTruffle	2.21071 ±0.007821	
LuaJIT	0.00432 ±0.000013	0.00673 ±0.000004
PyPy	0.21710 ±0.000563	0.82526 ±0.000235
V8	0.03702 ±0.000097	0.09607 ±0.000048

Table 3: VM startup time (in seconds with 99% confidence intervals).

tinct hardware (e.g. a non-x86 architecture) is more likely to uncover hardware-related differences. However, hardware cannot be varied in isolation from software: the greater the differences in hardware, the more likely that JIT compilers compilers are to use different components (e.g. different code generators). Put another way, an apples-to-apples comparison across very different hardware is often impossible, because the 'same' software is itself different.

We have not systematically tested whether rebuilding VMs effects warmup, an effect noted by Kalibera & Jones, though which seems to have little effect on the performance of JIT compiled code [4]. However, since measuring warmup largely involves measuring code that was not created by a JIT compiler, it is possible that these effects may impact upon our experiment. To a limited extent, the rebuilding of VMs that occurred on each of our benchmarking machines gives some small evidence as to this effect, or lack thereof.

The checksums we added to benchmarks ensure that, at a user-visible level, each performs equivalent work in each language variant. However, it is impossible to say whether each performs equivalent work at the lowest level or not. For example, choosing to use a different data type in a language's core library may substantially impact performance. There is also the perennial problem as to the degree to which an implementation of a benchmark should respect other language's implementations or be idiomatic (the latter being likely to run faster). From our perspective, this possibility is somewhat less important, since we are more interested in the warmup patterns of reasonable programs, whether they be the fastest possible or not. It is however possible that by inserting checksums we have created unrepresentative benchmarks, though this complaint could arguably be directed at the unmodified benchmarks too.

Although we have minimised the number of system calls that our in-process iterations runners make, we cannot escape them entirely. For example, on both Linux and OpenBSD, `clock_gettime()` (which we use to obtain monotonic wall-clock time) contains what is effectively a spin-lock, meaning that there is no guarantee that it returns within a fixed bound. In practise, `clock_gettime` returns far quicker than the granularity of any of our benchmarks, so this is a minor worry at most. The situation on Linux is complicated by our reading of core cycle, APERF, and MPERF counters via MSR device nodes: we call `lseek` and `read` between each in-process iteration (we keep the files open across all in-process iterations to reduce the open/close overhead). These calls are more involved than `clock_gettime`: as well as the file system overhead, reading from a MSR device node triggers inter-processor interrupts to schedule an `RDMSR` instruction on the desired core (causing the running core to save its registers etc.). We are not currently aware of a practical way to lower these costs.

Although Krun does as much to control CPU clock speed as possible, modern CPUs do not always respect operating system requests. On Linux, we use the APERF/MPERF ratio to check for frequency changes. Although the hoped-for ratio is precisely 1, there is often noticeable variation around this value due to the cost of reading these counters. On cores performing active computation, the error is small (around 1% in our experience), but on inactive cores it can be rela-

tively high (10% or more is not uncommon; in artificially extreme cases, we have observed up to 200%). In order to verify that there were no frequency changes during benchmarking, we wrote a simple checking tool. For each in-process iteration, it first checks the corresponding cycle-count data: since active cores generally have cycle-counts in the billions, we conservatively classify inactive cores as those with cycle-counts in the low millions. Active cores are then checked for an APERF/MPERF ratio between 1 ± 0.015. All active cores in our experiment were within these limits, strongly suggesting that our experiment experienced no unexpected CPU frequency changes.

Our experiments allow the kernel to run on the same core as benchmarking code. We experimented extensively with CPU pinning, but eventually abandoned it. After being confused by the behaviour of Linux's `isolcpus` mechanism (whose semantics changes between a normal and a real-time kernel), we used CPU shielding (`cset shield`) to pin the benchmarks to the 3 non-boot cores on our machines. However, we then observed notably worse performance for VMs such as HotSpot. We suspect this is because such VMs query the OS for the number of available cores and create a matching number of compilation threads; by reducing the number of available cores, we accidentally forced two of these threads to compete for a single core's resources.

In controlling confounding variables, our benchmarking environment necessarily deviates from standard configurations. It is possible that in so doing, we have created a system that shows warmup effects that few people will ever see in practise. However, our judgement is that this is preferable to running on a noisy system that is likely to introduce substantial noise into our readings.

8. Related work

There are two works we are aware of which explicitly note unusual warmup patterns. Whilst running benchmarks on HotSpot, Gil et al. [10] observed inconsistent process executions (e.g. recursiveErgodic), and benchmarks that we could classify as no steady state (listBubbleSort) and slowdown (arrayBubbleSort). By running a larger number of (somewhat larger) benchmarks on a number of VMs, and executing them in a more tightly controlled execution environment, our results can be seen as significantly strengthening Gil et al.'s observations. Our work also adds an automated approach to identifying when warmup has occurred. Kalibera & Jones note the existence of what we have called cyclic behaviour (in the context of benchmarking, they then require the user to manually pick one part of the cycle for measurement [12]): the data from our experiment seems to be less often cyclic, though we have no explanation for why.

9. Conclusions

Warmup has previously been an informally defined term [17] and in this paper we have shown cases where the definition fails to hold. Through a carefully designed experiment, and an application of a new statistical method, we hope to have helped give the study of warmup a firmer base.

Although we are fairly experienced in designing and implementing experiments, the experiment in this paper took far more time than we expected — about 2 full person years. In part this is because there is limited precedent for such detailed experiments. Investigating possible confounding variables, understanding how to control them, and implementing the necessary checks, all took time. In many cases, we had to implement small programs or systems to understand a variable's effect (e.g. that Linux allows a process to allocate memory beyond that specified in the soft and hard `ulimit`). However, we are realistic that few people will have the time or energy to institute all the controls that we implemented. An open question is which of the controls are the most significant in terms of producing a reliable experiment. The large number of partly inter-locking combinations means that we estimate that untangling this will require 3–6 months of experimental running time.

Our results have suggested to us some potentially useful advice for VM developers and users. First, simply running benchmarks for a larger number of in-process iterations is a simple way of understanding a VM's long-term performance stability. Second, as a community, we need to accept that a steady state of peak performance is not guaranteed to exist. Third, the significant differences in warmup time between VMs strongly suggest that VM benchmarking should always include warmup time. Fourth, we suspect that some of the odd results we have seen result from over-training VM heuristics on small sets of benchmarks. The approach taken by the machine-learning community may apply equally well to VMs: using a training set to devise heuristics, and then benchmarking the resulting system(s) on a separate validation system. Fifth, we suspect that the general reliance on small suites of benchmarks means that only small parts of VMs are being benchmarked effectively: we are increasingly of the opinion that benchmarking quality and quantity are tightly related, and that VMs need more benchmarks.

Transparency: *Blinded for submission.*

Acknowledgements: *Blinded for submission.*

References

[1] Octane benchmark suite. https://developers.google.com/octane/. Accessed: 2016-11-11.

[2] NO_HZ: Reducing scheduling-clock ticks, Linux kernel documentation. https://www.kernel.org/doc/Documentation-/timers/NO_HZ.txt, 2016. Accessed: 2016-01-21.

[3] Doug Bagley, Brent Fulgham, and Isaac Gouy. The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/. Accessed: 2016-09-02.

[4] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to interpreter composition. *COMLAN*,

abs/1409.0757, March 2015.

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct 2006.

[6] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *SCICO*, 98(3):408–421, Feb 2015.

[7] James Charles, Preet Jassi, Ananth Narayan S, Abbas Sadat, and Alexandra Fedorova. Evaluation of the Intel Core i7 turbo boost feature. In *IISWC*, Oct 2009.

[8] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. In *ASPLOS*, Mar 2013.

[9] Idris Eckley, Paul Fearnhead, and Rebecca Killick. Analysis of changepoint models. In D. Barber, T. Cemgil, and S. Chiappa, editors, *Bayesian Time Series Models*. 2011.

[10] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. A microbenchmark case study and lessons learned. In *VMIL*, Oct 2011.

[11] Tomas Kalibera and Richard Jones. Quantifying performance changes with effect size confidence intervals. Technical Report 4-12, University of Kent, Jun 2012.

[12] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *ISMM*, pages 63–74, Jun 2013.

[13] Rebecca Killick and Idris Eckley. changepoint: An R package for changepoint analysis. *J. Stat. Soft.*, 58(1):1–19, 2014.

[14] Rebecca Killick, Paul Fearnhead, and Idris Eckley. Optimal detection of changepoints with a linear computational cost. *J. Am. Stat. Assoc.*, 107(500):1590–1598, 2012.

[15] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLKS*, pages 265–276, Mar 2009.

[16] Paruj Ratanaworabhan, Benjamin Livshits, David Simmons, and Benjamin Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2009-173, Microsoft Research, Dec 2009.

[17] Chris Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. PhD thesis, University of Manchester, Jun 2015.

[18] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI*, pages 1–8, 2010.

[19] John Tukey. *Exploratory Data Analysis*. 1977.

In this appendix, we first show that our statistical method can be applied to well known benchmarking suites (Appendix A) before showing the complete warmup data from our experiment (Appendix B). We then present a curated selection of interesting run-sequence / core-cycle / APERF/MPERF plots. The complete series of plots is available in a separate document.

A. Applying the Statistical Method to Existing Benchmark Suites

The statistical method presented in Section 4 is not limited to data produced from Krun. To demonstrate this, we have applied it to two standard benchmark suites: DaCapo [5] (Java) and Octane [1] (JavaScript). We ran both for 10 process executions and 2000 in-process iterations without reboots, temperature control etc.: DaCapo on $\text{Linux}_{\text{4790K}}$; and Octane on $\text{Linux}_{\text{E3-1240v5}}$ (a Xeon machine, with a software setup similar to $\text{Linux}_{\text{4790K}}$).[6]

We ran DaCapo (with its default benchmark size) on Graal and HotSpot. As it already has support for altering the number of in-process executions, we used it without modification. However, we were unable to run 4 of its 14 benchmarks: `batik` crashes with a `InvocationTargetException`; `eclipse`, `tomcat`, and (intermittently) `tradesoap` fail their own internal validation checks.

We ran Octane on SpiderMonkey (#465d150b, a JIT compiling VM for JavaScript) and V8. We replaced its complex runner (which reported timings with a non-monotonic microsecond timer) with a simpler alternative (using a monotonic millisecond timer). We also had to decide on an acceptable notion of 'iteration'. Many of Octane's benchmarks consist of a relatively quick 'inner benchmark'; an 'outer benchmark' specifies how many times the inner benchmark should be run in order to make an adequately long running benchmark. We recorded 2000 iterations of the outer benchmark; our runner fully resets the benchmark and the random number generator between each iteration. The `box2d`, `gameboy`, `mandreel` benchmarks do not properly reset their state between runs, leading to run-time errors we have not been able to fix; `typescript`'s reset function, in contrast, frees constant data needed by all iterations, which we were able to easily fix. When run for 2000 iterations, `CodeLoadClosure`, `pdfjs`, and `zlib` all fail due to memory leaks. We were able to easily fix `pdfjs` by emptying a global list after each iteration, but not the others. We therefore include 12 of Octane's benchmarks (including lightly modified versions of `pdfjs` and `typescript`). Because we run fewer benchmarks, our modified runner is unable to fully replicate the running order of Octane's original runner. Since Octane runs all benchmarks in a single process execution, this could affect the performance of later benchmarks in the suite.

Table 4 shows the full DaCapo results. Graal and HotSpot both perform similarly. The luindex benchmark is consistently a slowdown on Graal, and mostly a slowdown on HotSpot. Jython is notably inconsistent on both VMs. In summary, even on this most carefully designed of benchmark suites, only 70% of (VM, benchmark) pairs have 'good' warmup.

Table 5 shows the full Octane results. These are notably less consistent than the DaCapo results, with SpiderMonkey being somewhat less consistent than V8. In summary, only 45.8% of (VM, benchmark) pairs have 'good' warmup.

As these results show, our automated statistical method produces satisfying results even on existing benchmark suites that have not been subject to the Krun treatment. Both DaCapo and (mostly) Octane use much larger benchmarks than our main experiment. We have no realistic way of understanding to what extent this makes 'good' warmup more or less likely. For example, it is likely that there is CFG nondeterminism in many of these benchmarks; however, their larger code-bases may give VMs the ability to 'spread out' VM costs, making smaller blips less likely.

B. Further Results

The main experiment's results for $\text{Linux}_{\text{4790}}$ and $\text{OpenBSD}_{\text{4790}}$ can be seen in Tables 6 and 7.

[6] The large amount of CPU time our experiments require meant that we ran out of time to run Octane on $\text{Linux}_{\text{4790K}}$ before paper submission. For the final paper, we will run DaCapo and Octane on the same machine. Although we do not expect this to result in significant changes to the data, it will reduce a source of variation.

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
avrora	Graal	⌊	2 ±0	5.60 ±1.226	1.68542 ±0.005515	Hotspot	⌊	1 ±0	3.22 ±1.265	1.66918 ±0.006029
fop		≍ (7≁, 2⌋, 1⌊)					≍ (6≁, 4⌋)			
h2		–			2.94680 ±0.019285		–			2.75639 ±0.023803
jython		≍ (6⌊, 3≁, 1⌋)					≍ (7⌊, 3≁)			
luindex		⌋	682 ±76	240.45 ±28.352	0.38489 ±0.001527		≍ (9⌋, 1⌊)			
lusearch		⌊	2 ±1	4.33 ±1.015	0.75815 ±0.001929		⌊	1 ±1	1.82 ±0.696	0.52262 ±0.006121
pmd		⌊	25 ±5	20.06 ±3.709	0.64875 ±0.004969		⌊	32 ±9	22.81 ±6.202	0.60165 ±0.007261
sunflow		⌊	1 ±0	3.34 ±0.067	1.39696 ±0.009948		–			1.52857 ±0.007367
tradebeans		⌊	1 ±0	3.78 ±1.805	2.19572 ±0.021630		⌊	1 ±0	3.89 ±0.997	1.93720 ±0.024628
xalan		⌊	7 ±1	6.53 ±0.665	0.43149 ±0.003218		⌊	8 ±1	5.94 ±0.371	0.37728 ±0.003869

Table 4: DaCapo results.

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
Boyer	Spidermonkey	–			1.72407 ±0.001516	V8	≍ (9⌊, 1–)			
Decrypt		≍ (5–, 3⌊, 2⌋)					⌊	6 ±1	4.98 ±1.167	0.78870 ±0.005202
DeltaBlue		≍ (9–, 1⌋)					≍ (4–, 4⌋, 2⌊)			
Earley		–			1.05912 ±0.002149		⌊	1 ±0	1.17 ±0.002	1.15330 ±0.001862
Encrypt		≍ (8⌊, 2–)					⌊	2 ±1	1.59 ±1.137	0.78309 ±0.000779
NavierStokes		⌊	7 ±7	5.65 ±5.588	0.78486 ±0.000070		≍ (7⌊, 3–)			
PdfJS		⌊	1 ±0	2.10 ±0.906	1.11829 ±0.003161		≍ (8⌊, 1–, 1⌋)			
RayTrace		≍ (8≁, 2⌊)					⌊	1 ±0	0.60 ±0.106	0.52512 ±0.000958
RegExp		≍ (9–, 1⌊)					≍ (9–, 1⌋)			
Richards		≍ (9⌊, 1–)					⌊	2 ±1	1.64 ±1.107	0.81120 ±0.000458
Splay		–			0.51989 ±0.001250		–			0.46197 ±0.002564
Typescript		≍ (7⌊, 3–)					≍ (5–, 5⌊)			

Table 5: Octane results.

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*binary trees*	⩙ (9⇜, 1⌡)				*n-body*	⩙ (9−, 1⌊)			
Graal		⩙ (9⌊, 1⌡)					⌊	4 ±0	0.85 ±0.051	0.16801 ±0.000013
HHVM		⌊	78 ±28	181.45 ±66.989	2.31524 ±0.002691		⌊	132 ±0	1006.83 ±0.078	3.03675 ±0.000224
HotSpot		⌊	2 ±2	0.60 ±0.495	0.19030 ±0.000326		⌊	1 ±0	0.18 ±0.000	0.17379 ±0.000025
JRuby+Truffle		⩙ (5⌊, 4⌡, 1⇜)					⌊	68 ±0	21.98 ±0.198	0.25855 ±0.000431
LuaJIT		⩙ (8⌡, 2−)					–			0.29664 ±0.000062
PyPy		⩙ (7⇜, 3⌡)					–			1.82929 ±0.000782
V8		⌊	1 ±0	0.69 ±0.390	0.48385 ±0.000486		⌊	367 ±4	102.94 ±1.291	0.27909 ±0.000083
C	*fannkuch redux*	–			0.42153 ±0.000124	*Richards*	–			0.76543 ±0.000320
Graal		⌊	1 ±0	0.77 ±0.101	0.37329 ±0.002215		⩙ (7⌊, 2⌡, 1⇜)			
HHVM		⩙ (9⌊, 1⇜)								
HotSpot		⩙ (9⇜, 1⌊)					⩙ (8⌡, 2⇜)			
JRuby+Truffle		⌡	998 ±0	1039.83 ±10.664	1.17529 ±0.020561		⌡	573 ±155	630.39 ±169.475	1.08722 ±0.014276
LuaJIT		–			0.56521 ±0.000032		–			2.89911 ±0.000433
PyPy		–			1.62912 ±0.003134		⌊	1 ±0	1.14 ±0.002	0.98095 ±0.000129
V8		–			0.30454 ±0.000046		⩙ (9⌡, 1⌊)			
C	*fasta*	–			0.07398 ±0.000012	*spectralnorm*	⩙ (6⌊, 2−, 2⌡)			
Graal		⌊	2 ±0	0.67 ±0.102	0.14819 ±0.000253		⌡	13 ±0	6.71 ±0.233	1.03518 ±0.000031
HHVM		⌊	17 ±3	17.21 ±2.557	0.81976 ±0.000458		⌊	32 ±0	128.39 ±1.549	1.65400 ±0.000270
HotSpot		⩙ (7⌡, 3⌊)					⩙ (7⌊, 3−)			
JRuby+Truffle							⩙ (7⌊, 3⌡)			
LuaJIT		–			0.33296 ±0.000671		–			0.46787 ±0.000041
PyPy		⩛					–			0.51937 ±0.000048
V8		⌡	523 ±0	593.15 ±0.197	1.15337 ±0.000826		⌊	2 ±0	0.94 ±0.000	0.46784 ±0.000043

Table 6: Benchmark results for Linux_{4790}.

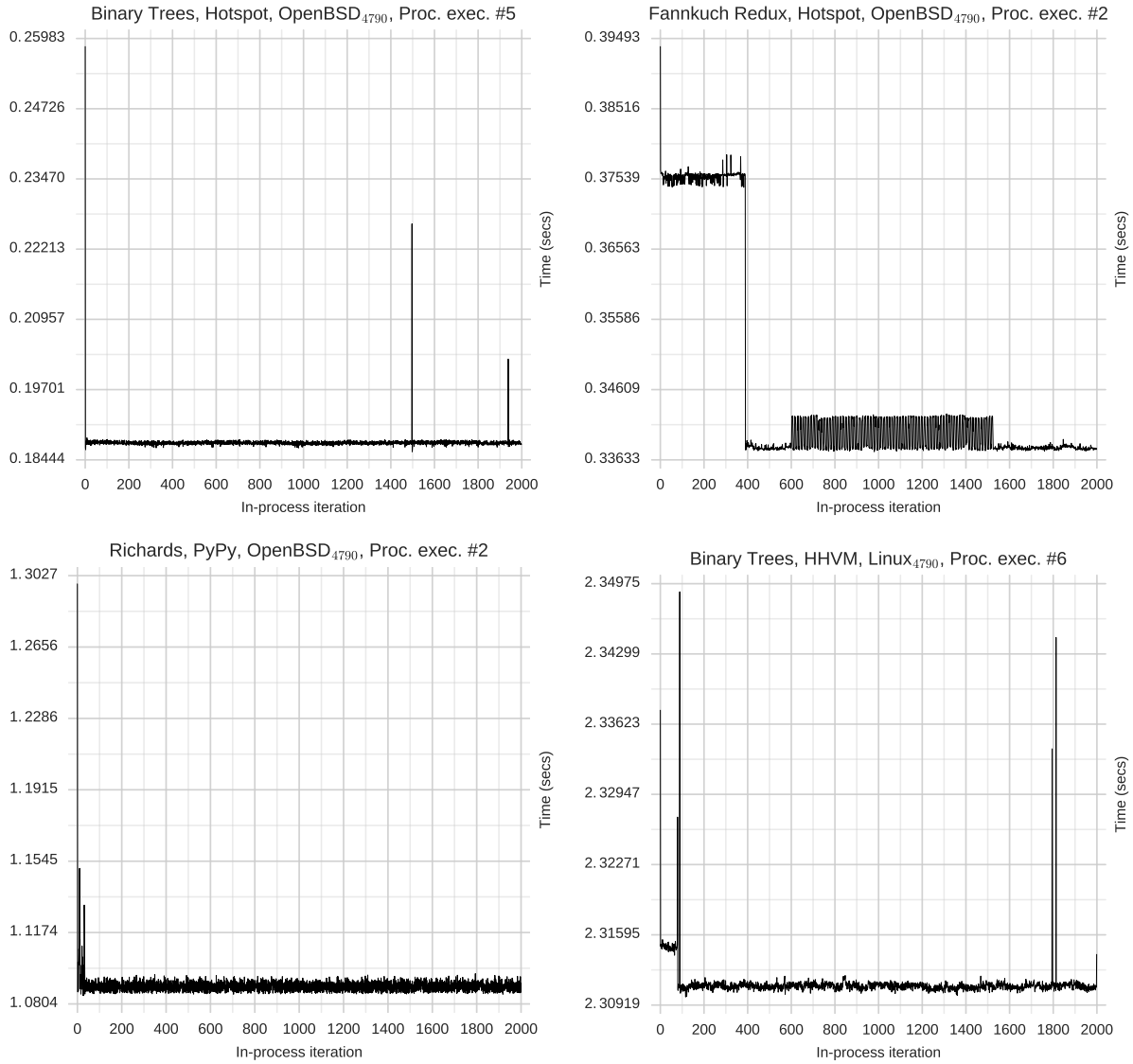
		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*binary trees*	–			3.29854 ±0.014325	*n-body*	⩙ (5⌊, 4–, 1⩍)			
HotSpot		⌊	1 ±0	0.28 ±0.038	0.18688 ±0.000625		⌊	1 ±0	0.18 ±0.000	0.17177 ±0.000215
LuaJIT		⩙ (5⌋, 3–, 2⌊)					–			0.29358 ±0.000771
PyPy		⌊	32 ±62	46.66 ±90.291	1.45425 ±0.006822		–			3.84375 ±0.051145
V8		⩙ (6⌊, 3⌋, 1⩍)					⩙ (7–, 3⌊)			
C	*fannkuch redux*	–			0.41142 ±0.000035	*Richards*	⩙ (8–, 2⌊)			
HotSpot		⌊	195 ±155	73.50 ±58.486	0.35896 ±0.016563		⌋	198 ±0	50.96 ±0.354	0.27327 ±0.002359
LuaJIT		⩙ (8–, 2⌊)					⩙ (8–, 1⌊, 1⩍)			
PyPy		⩙ (9⌊, 1⌋)					⌊	29 ±8	32.23 ±9.555	1.08374 ±0.013451
V8		⩙ (6⌊, 4–)					⩙ (8⌊, 2⌋)			
C	*fasta*	–			0.07415 ±0.000034	*spectralnorm*	⩙ (5–, 3⌋, 2⌊)			
HotSpot		⩙ (8⌋, 2⌊)					⩙ (9–, 1⌊)			
LuaJIT		⩙ (9–, 1⌊)					–			0.46778 ±0.000000
PyPy		⩙ (5⌋, 3⌊, 2⩍)					⌊	110 ±44	57.44 ±24.104	0.51932 ±0.000007
V8		⌋	361 ±86	417.36 ±99.219	1.15646 ±0.004120		–			0.46779 ±0.000001

Table 7: Benchmark results for OpenBSD_{4790}.

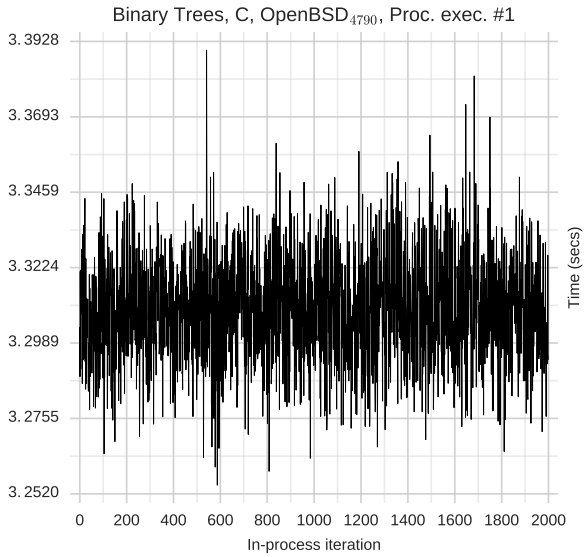
C. Curated Plots

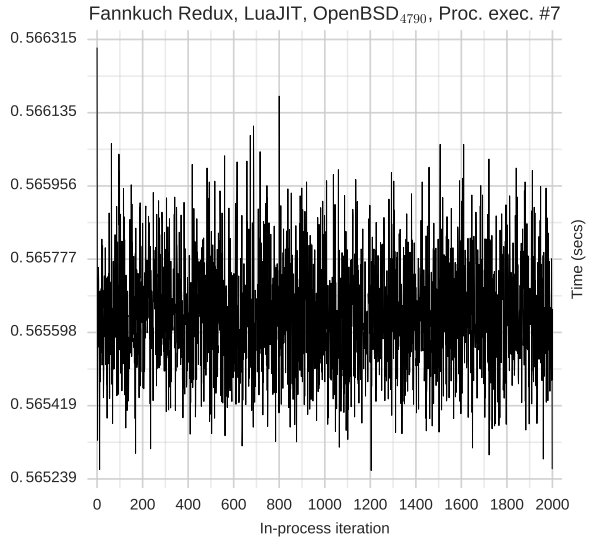
The remainder of this appendix shows curated plots: we have selected 4 interesting plots from each classification, to give readers a sense of the range of data obtained from our experiment. A separate document contains the complete series of plots.

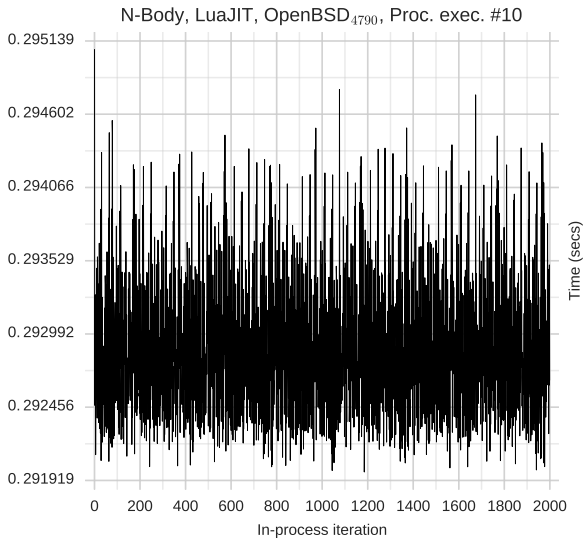
C.1 Examples of Warmup Behaviour

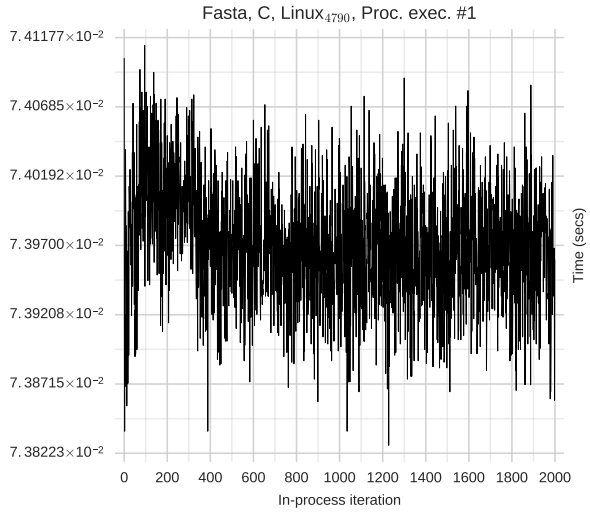


C.2 Examples of Flat Behaviour

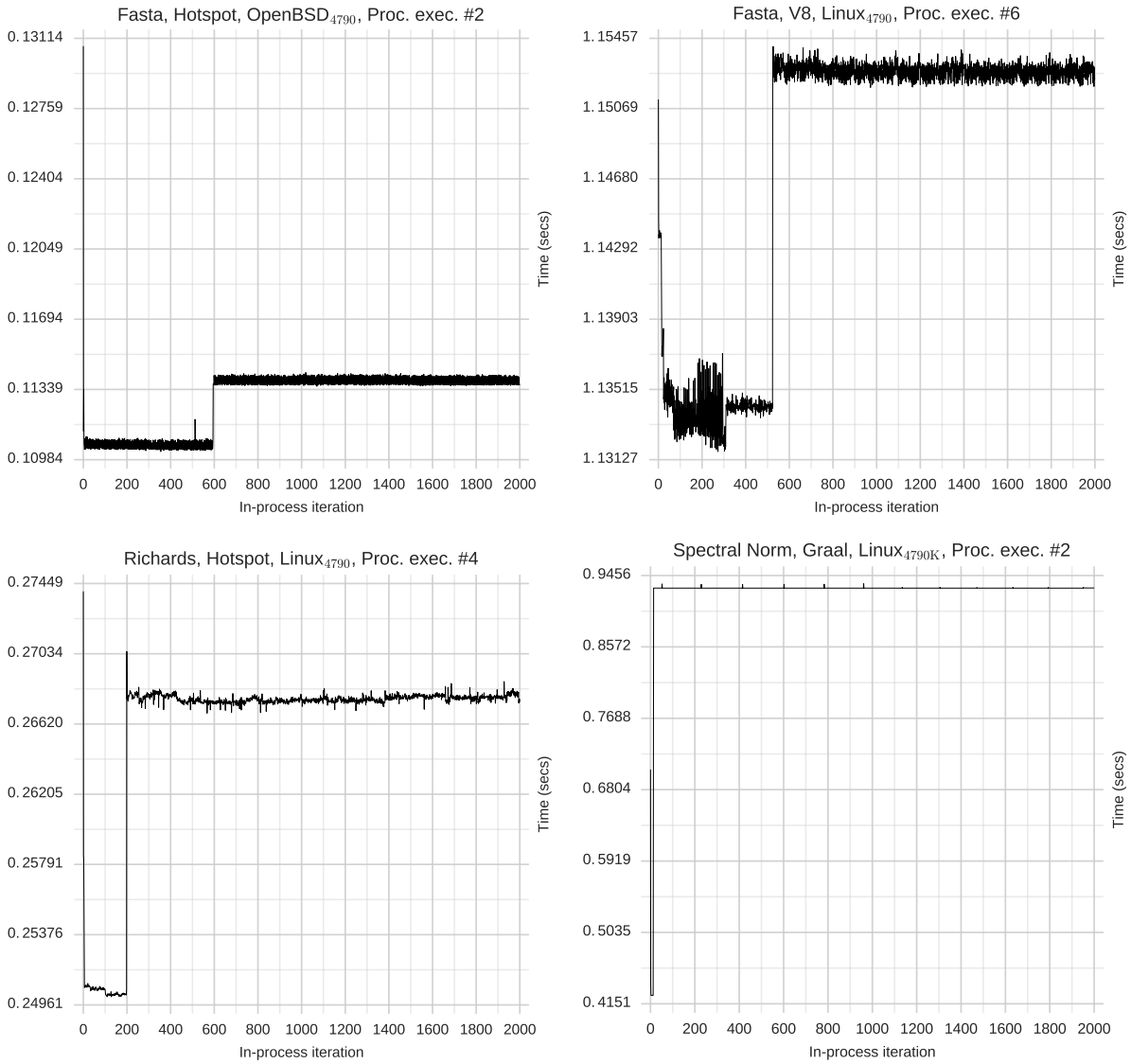








C.3 Examples of Slowdown Behaviour



C.4 Examples of No Steady State Behaviour