Virtual Machine Warm-up Blows Hot and Cold

Edd Barrett
Software Development Team, Department of Informatics, King's College London
`http://eddbarrett.co.uk/`

Carl Friedrich Bolz
Software Development Team, Department of Informatics, King's College London
`http://cfbolz.de/`

Rebecca Killick
Department of Mathematics and Statistics, University of Lancaster
`http://www.lancs.ac.uk/~killick/`

Vincent Knight
School of Mathematics, Cardiff University
`http://vknight.org/`

Sarah Mount
Software Development Team, Department of Informatics, King's College London
`http://snim2.org/`

Laurence Tratt
Software Development Team, Department of Informatics, King's College London
`http://tratt.net/laurie/`

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 *warm-up* phase. The traditional view of JIT compiled VMs is that program execution is slow during the warm-up phase, and fast afterwards, when *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 generally aim to measure peak performance. Typically, benchmarks are run repeatedly in a single process, then data prior to peak performance is discarded.

The aim of this paper is to test the following hypothesis:

H1 Small, deterministic programs exhibit traditional warm-up behaviour.

In order to test this hypothesis, we present a carefully designed experiment where a number of simple benchmarks are run on a variety of VMs 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 deliberately treat VMs as black boxes: we simply run benchmarks and record timing data.

While some benchmarks on some VMs run as per traditional expectations, we found a number of surprising cases. At the most extreme, some benchmarks never warm up, staying at their initial performance levels indefinitely and some even slow down. Of the eight VMs we looked at, each had at least one benchmark where the VM did not follow the traditional model.

Our results clearly invalidate H1: the traditional view of warm-up is not valid. We are not aware that anyone has systematically examined this problem before, let alone take it into account when benchmarking. This suggests that many published VM benchmarks (including our own) may have presented results which are misleading in some situations.

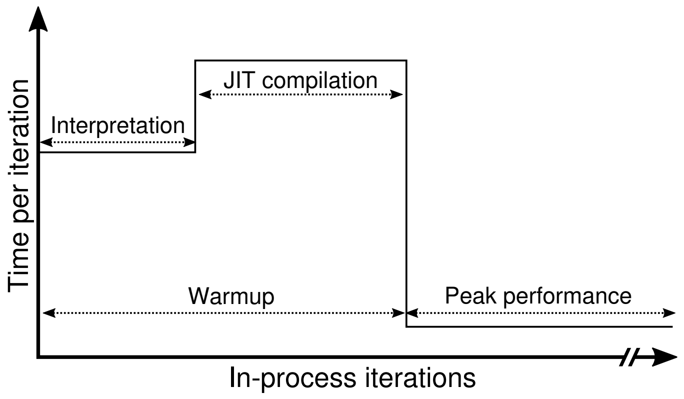


Figure 1. The traditional notion of warm-up: a program starts slowly executing in an interpreter; once hot parts of the program are identified, they are translated by the JIT compiler to machine code; at this point warm-up is said to have completed, and peak performance reached.

We believe that accurate VM benchmarking is paramount, for both VM authors, and for many end users. VM authors need to know if optimisations have an effect distinguishable from noise (and many optimisations have only a small effect). Similarly, end-users with latency sensitive workloads (e.g. games or other soft real-time systems) rely upon accurate benchmarking during their evaluation phase. Our results suggest that current benchmarking methods are potentially leading these parties astray.

This extended abstract is based upon a draft version of a full-length paper. The draft is available at:

`http://arxiv.org/abs/1602.00602/`

2. Background

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 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 traditionally said to have finished warming up, and the program to be executing at peak performance.[1]

Figure 1 illustrates a program subject to the conventional model of warm-up. Exactly how long warm-up takes is highly dependent on the program and the JIT compiler, but this basic assumption about the performance model is shared by almost all benchmarks of JIT compiling VMs (Kalibera and Jones 2013).

Benchmarking of JIT compiled VMs typically focusses on peak performance. In large part because the widespread assumption has been that warm-up is both fast and inconsequential to users. With that assumption in mind, the methods 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 are then discarded, on the basis that warm-up will have completed at some point before $n + 1$. It is common for n to be a hard-coded number, e.g. 5. The more sophisticated Kalibera & Jones benchmarking method (Kalibera and Jones 2012, 2013) (recently used in (Barrett et al. 2015; Grimmer et al. 2015)) improves upon this by having the user manually inspect run-sequence plots (or trace plots) for each process execution. The method also suggests a method for dealing with benchmarks which are cyclic in nature (e.g. that produce a sawtooth wave when plotted).

While the Kalibera & Jones method is certainly an improvement over straightforward benchmarking methods, our experience has been that there remain cases where it is hard to produce satisfying benchmarking statistics. Crucially, the method does not provide a firm way of determining when warm-up 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" (Seaton 2015).

3. Method

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. So as to collect high-quality data, we have carefully designed our experiment to be repeatable and to control as many potentially confounding variables as is practical.

The micro-benchmarks we use are as follows: *binary trees*, *spectralnorm*, *n-body*, *fasta*, and *fannkuch redux* from the Computer Language Benchmarks Game (CLBG); and *Richards*. Readers can be forgiven for initial scepticism about this set of micro-benchmarks. The choice was in fact deliberate. These small and deterministic benchmarks are precisely the kind of program we would expect to warmup in a well-behaved fashion. The fact that the benchmarks are small should mean that the amount of code which needs to be compiled at runtime should be small. Further since the benchmarks are deterministic, there should be no control flow variation between in-process iterations of the benchmarks, meaning that the JIT compiler is unlikely to be invoked during later in-process iterations of benchmarking. Finally, since VM authors use these benchmarks as optimisation targets, they should be some of the most well-behaved benchmarks available.

[1] This traditional notion 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, and tolerates slower compilation time for better machine code generation).

We used C, Java, Javascript, Python, Lua, PHP, and Ruby versions of each benchmark.[2] Since most of these benchmarks have multiple implementations in any given language, we picked the same versions used in (Bolz and Tratt 2015), which represented the fastest performers at the point of that publication. We were forced to skip some benchmark and VM pairings which either ran prohibitively slowly (Fasta/JRubyTruffle and Richards/HHVM), or caused the VM to crash (SpectralNorm/JRubyTruffle). The benchmarks were audited for "CFG determinism", meaning that no two in-process iterations take different paths through the control flow graph of the benchmark. We also checked that the different language versions of the benchmarks all computed the same result. We did not interfere with any VM's Garbage Collection (GC) (e.g. we did not force a collection after each iteration).

Our benchmarking hardware consisted of three machines:

Linux1/i7-4709K Quad-core i7-4790K 4GHz, 24GB of RAM, running Debian 8.

Linux2/i7-4790 Quad-core i7-4790 3.6GHz, 32GB of RAM, running Debian 8.

OpenBSD/i7-4790 Identical hardware to Linux2/i7-4790, but running OpenBSD 5.8.

We disabled any hardware features (where possible) that could possibly induce variation into our results. For example turbo boost and hyper-threading were disabled. The two Linux machines have exactly the same packages and package versions installed.

The benchmarks were then run on the following VMs: GCC 4.9.3; Graal 0.13; HHVM 3.12.0; a recent GIT version of JRuby/Truffle[3]; HotSpot 8u72b15; LuaJIT 2.0.4; PyPy 4.0.1; and V8 4.9.385.21. Although not a VM, GCC serves as a baseline to compare the VMs against. HHVM and JRuby/Truffle do not yet run on OpenBSD, and were thus skipped on the OpenBSD benchmarking machine. We use a script to download, configure, and build the above versions of the VMs, ensuring we can easily repeat builds. All VMs were compiled with a manually built GCC/G++ 4.9.3 (the same used for C benchmarks), thus eliminating the GCC versions themselves as a source of variation.

The benchmarks were run under a specially developed tool called Krun[4], which measures in-process iteration times (using a monotonic clock) in a fashion which minimises the effect of external sources of variation. For example, Krun: reboots the system prior to the first process execution and after each process execution; ensures that each process execution starts with the same system temperatures ($\pm 3^\circ$C); runs benchmarks as an otherwise unused user account; and (on Linux, where this is possible) runs the kernel with adaptive-tick mode CPU cores (tic 2016).

4. Preliminary Results

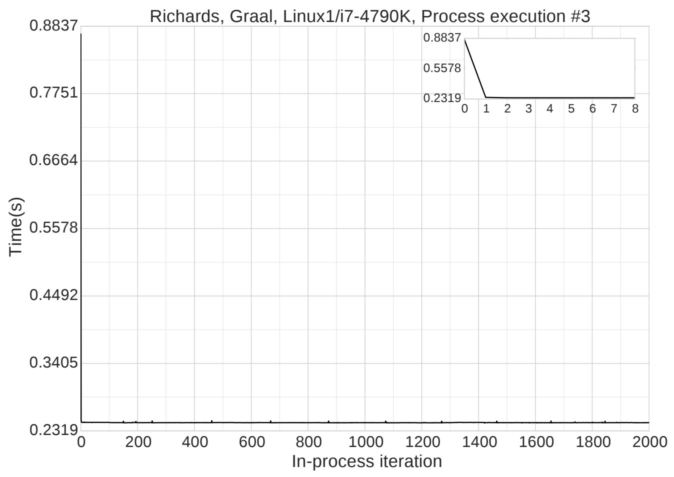
Our experiment runs 450 unique process executions, giving a total of 900 000 in-process iteration readings. For each process execution we generate a run-sequence graph, with the in-process iteration number on the x axis, and the time (in seconds) on the y axis. A full set of graphs, as well as our raw data, can be downloaded from:

`https://archive.org/download/softdev_warmup_experiment_artefacts/v0.2/`

[2] Our need to have implementations in a wide variety of languages restricted the micro-benchmarks we could use.

[3] GIT hash: `f82ac77137da265d2447d723ce5973a04459a609`

[4] `github.com/softdevteam/krun`



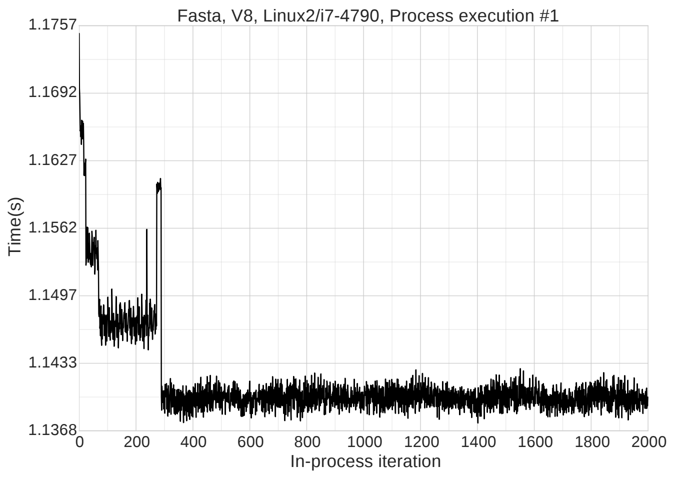
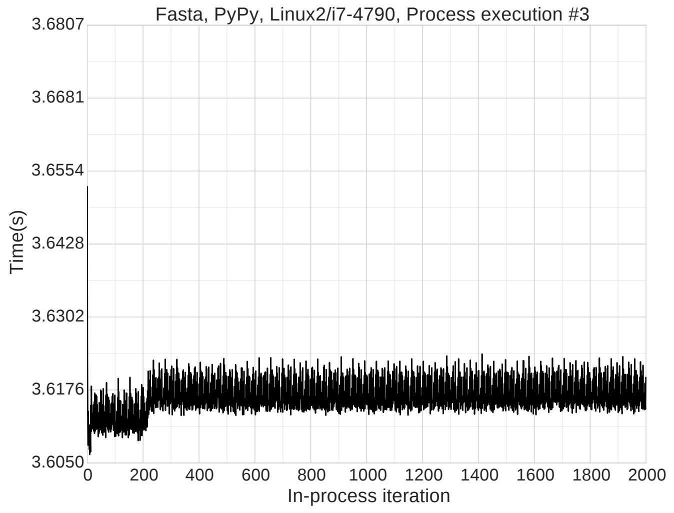


Figure 2. Process executions warming up under the classical model.



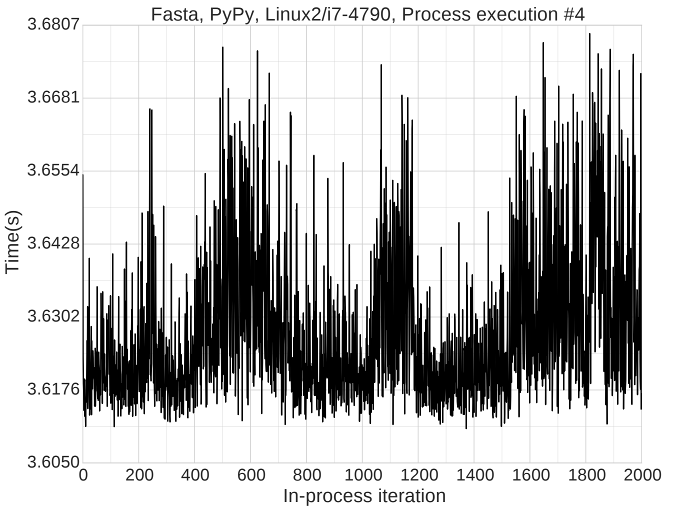


Figure 3. Inconsistent process executions on the same machine.

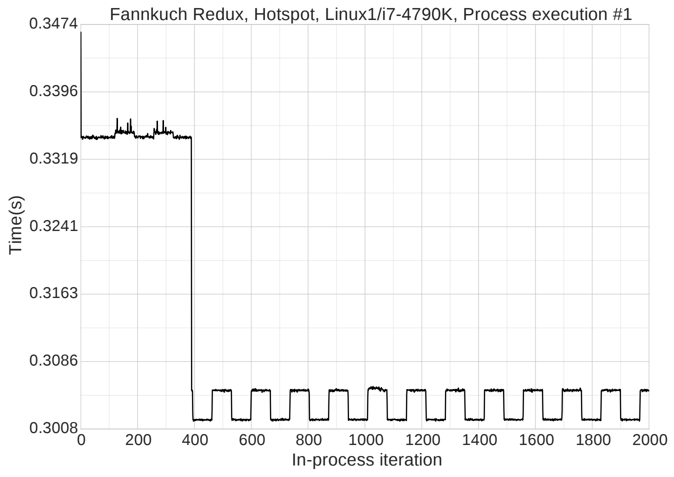


Figure 4. A process execution with cycles.

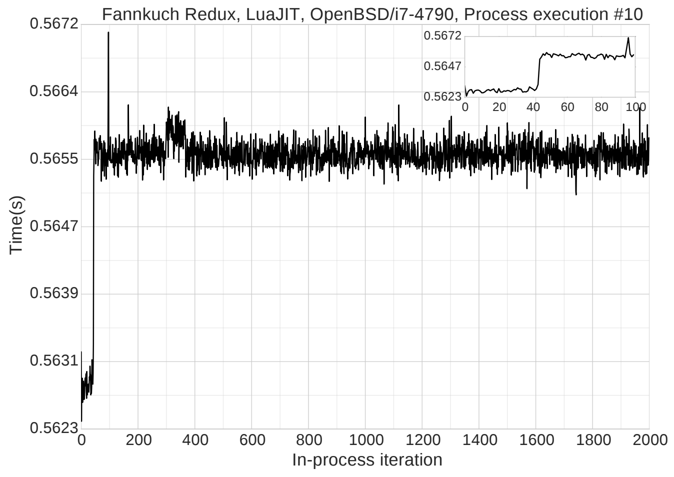


Figure 5. A process execution with slowdown.

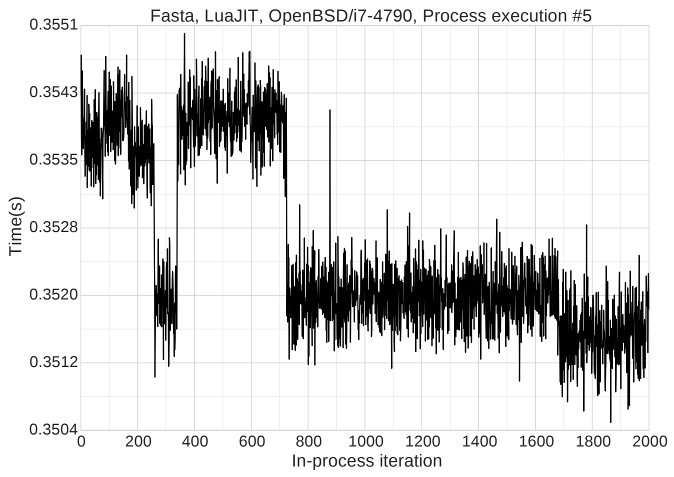


Figure 6. A process execution with changing phases.

Although some of the graphs do show classical warm-up behaviour (e.g. Figure 2), others highlight a number of interesting (and unexpected) behaviours:

Inconsistent process executions Process executions for the same benchmark / VM pair behave differently. Sometimes this occurs on in-process executions on the same machine; sometimes only across machines. E.g. Figure 3.

Cycles In-process iteration times repeat in a predictable pattern. E.g. Figure 4.

Slowdown Performance of in-process iterations drops over time. E.g. Figure 5.

Changing phases The mean of in-process iteration times abruptly changes over time. E.g. Figure 6.

The preliminary results are troubling. Process executions which do not warm up under the classical model cannot have traditional benchmarking methods applied. In other words, if the end of the warm-up phase cannot be identified, then in turn the peak performance phase cannot be identified. This suggests that benchmarking methods need to be rethought.

5. Related work

There are two pieces of prior art we are aware of which explicitly note unusual warm-up patterns. Gil et al.'s main focus is on non-determinism of process executions on HotSpot, and the difficulties this raises in terms of providing reliable benchmarking numbers (Gil et al. 2011). The authors report process executions which we would classify as a slowdowns. 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 (Kalibera and Jones 2013)).

6. Conclusions and Future Work

Warm-up has always been an informally defined term (Seaton 2015). Our preliminary work has shown cases where generally accepted definitions fail to hold. To the best of our knowledge, we are the first to (informally) classify different 'warm-up' styles and note the relatively high frequency of non-traditional classifications such as slowdown and phase changes. However, we have not yet found an acceptable alternative definition of warm-up. Based on our experiences thus far, we think it unlikely that the different styles of warm-up we have seen can be captured in a single metric. We suspect it is more likely that a number of different metrics will be needed to describe and compare warm-up styles.

There are several items of potential future work which would likely lead to a better understanding of the warm-up behaviours: automated classification of warm-up behaviours would help guide us through the vast amount of data we have collected; VM instrumentation may help us to associate VM events, such as compilation and garbage collection, to artefacts in our graphs. Similarly we may find that hardware performance counters could offer insight: perhaps hardware events such as context switches and CPU migrations are a source of some of our findings. Finally, we collect more data by adding further hardware platforms, operating systems, benchmarks and VMs into the experiment. [5]

Acknowledgements: This research was funded by the EPSRC Cooler (EP/K01790X/1) grant and Lecture (EP/L02344X/1) fellowship.

References

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

Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to interpreter composition. *Computer Languages, Systems and Structures*, abs/1409.0757, March 2015.

Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98, Part 3:408–421, Feb 2015.

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

Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *MODULARITY*, Mar 2015.

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

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

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

[5] We have already started instrumenting the VMs and we are also in the process of designing automated analyses for classifying warmup behaviours.