DOs and DON'Ts of Conducting Performance Measurements in Java

Vojtěch Horký Peter Libič Antonín Steinhauser Petr Tůma

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
{horky,libic,steinhauser,tuma}@d3s.mff.cuni.cz

ABSTRACT

The tutorial aims at practitioners – researchers or developers – who need to execute small scale performance experiments in Java. The goal is to provide the attendees with a compact overview of some of the issues that can hinder the experiment or mislead the evaluation, and discuss the methods and tools that can help avoid such issues. The tutorial will examine multiple elements of the software execution stack that impact performance, including common virtual machine mechanisms (just-in-time compilation and garbage collection together with associated runtime adaptation), some operating system features (timers) and hardware (memory) – although the focus will be on Java, some of the take away points should apply even in a more general performance experiment context.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements; D.2.8 [**Metrics**]: Performance measures

General Terms

Performance, Measurement

Keywords

performance measurement; performance evaluation; Java

1. INTRODUCTION

Some quarter century ago, a programmer using a linked list could rely on its performance being fairly transparent. The timing of the individual list operations could be computed merely by considering their algorithmic complexity together with the timing of the individual algorithmic steps, whose source – perhaps in Pascal – was compiled into known assembly instructions with known durations.

Today, programmers do not have that luxury. The same list – this time perhaps written in Java – is repeatedly compiled in multiple stages by adaptive compilers, it is not easy to figure out which assembly instructions will eventually be used to execute the list operations, and even if it were, the instructions do not have predictable durations anymore.

In absence of performance transparency, performance experiments become an attractive alternative. Rather than guessing what performance a linked list may exhibit, a programmer can simply measure it. The same approach is adopted in many similar situations – system administrators may use small scale performance experiments to determine optimal machine settings, computer scientists back their research into software performance with experimental evaluation, and so on.

Intuitively, performance experiment results should represent the ground truth in software performance – after all, what can give a better idea of true software performance than an experiment that measures that very performance ? In truth, however, the very lack of performance transparency that makes performance experiments useful can also make them very much misleading. Even when the measurements themselves report the true performance, which is not always a given, the lack of performance transparency means it is difficult to interpret the measurements and to extrapolate the interpretation beyond the experiment.

There is much evidence that it is easy to unwittingly conduct performance experiments that produce tricky results [6, 7, 4, 8, 21, 3, 14]. On contemporary execution platforms, even small scale performance experiments are turning from an easy and reliable way of evaluating software performance into a difficult exercise of tracking a multitude of technical details that must be dealt with even in otherwise very simple scenarios.

The goal of this tutorial is to help practitioners – programmers, administrators, researchers – who need to evaluate software performance by providing a compact technical overview of the essential problems and the available solutions in performance experiments. By way of a disclaimer, we acknowledge that our work does not tackle the core problem of insufficient performance transparency, but it may help execute performance experiments with less effort and more trust in the results.

The tutorial focuses on contemporary desktop environments running Java. Outside this environment, the content is mostly relevant in general, but the technical particulars can obviously differ. Due to space constraints, this tutorial paper contains only a dense list of topics that we believe deserve attention, together with references to related

ICPE'15, Jan. 31–Feb. 4, 2015, Austin, Texas, USA.

http://dx.doi.org/10.1145/2668930.2688820.

resources. The tutorial will explore the topics in more depth where appropriate.

2. WARMING UP: FAST AND STEADY

Perhaps the most well-known issue related to performance experiments is warm up. The term refers to the fact that software performance can be influenced by one-time artifacts, often visible shortly after software start. When the goal of the experiment is to examine sustainable performance, measurements influenced by these warm up artifacts need to be recognized and discarded.

Understanding the sources of the warm up artifacts is essential. Contrary to expectations, the warm up time can be very long, and observing a long period of steady performance does not necessarily imply all warm up is done. Besides the start of an experiment, the warm up artifacts can also appear between individual experimental stages, especially if these involve changes in workload.

Two major sources of the warm up artifacts discussed in the first tutorial section are class loading and just-in-time compilation. Other sources are discussed later.

2.1 Class Loading

Java code is loaded class by class on demand, where demand is first use, not first declaration [16]. Class loading disrupts performance both directly, because execution cannot proceed until the required classes are loaded, and indirectly, because some compiler optimizations are based on assumptions related to presence or absence of classes.

Once loaded, classes are rarely collected, because of many links that connect the classes to other objects, with the exception of classes loaded by the anonymous class loader.

2.2 Just-In-Time Compliation

Java program is first compiled statically from source code into bytecode. Bytecode is the portable representation that the Java Virtual Machine (JVM) loads and executes. Although it is possible to execute the bytecode in an interpreter, it is more common to compile bytecode into native code, which executes directly. This is done by Just-In-Time (JIT) compiler.

Triggering compilation. Inherent to JIT compilation is the trade off between initial performance loss, due to executing the compilation, and later performance gain, due to executing the compiled code. The JVM is therefore selective in submitting code for JIT compilation – only methods that have been observed to execute often enough are compiled with complex optimizations.

JIT compilation can be configured with various levels of optimization and include various levels of profiling code that enable better optimization later. Current JIT compilers use tiered JIT compilation, where methods gradually move towards higher compilation levels. A common criterion for triggering a compilation is the number of times a method was invoked or the number of times a loop iterated. Thresholds for lower compilation levels default to values around thousands, thresholds for higher compilation levels to values around tens of thousands. The thresholds can be adjusted [18].

Inlining methods. Inlining is an optimization performed by the JIT compiler which replaces method calls by method bodies where appropriate. The immediately obvious but relatively small benefit of inlining is removing the call overhead. More importantly, inlining improves other optimizations by providing more code to work on.

Whether inlining happens depends on multiple factors, especially the size of the method to be inlined (hence instrumentation influences inlining) and the ability of the JIT compiler to determine the method call target (hence class loading influences inlining).

A performance experiment may inline both more and less than desired or expected. Sometimes it is possible to control inlining or display inlining decisions [18, 19].

Determinism. The exact result of JIT compilation depends on many timing factors. Multiple executions of the same performance experiment will not necessarily exhibit the same JIT behavior and therefore will not use the same code. Support for making the behavior of JIT compilation deterministic is not yet common [5, 1], in absense of such support multiple JVM executions can be used to examine the possible spectrum of experiment behaviors.

Initial performance. So far, we have considered experiments that examine sustainable performance. When initial rather than sustainable performance is of interest, multiple JVM executions are required. Warm up artifacts exist even in this context, for example the very first execution is likely to fetch data from disk, while the subsequent executions will benefit from disk cache. Even the opposite can be true, when all but the first execution pay the cost of flushing the disk cache left dirty by the previous execution.

3. TOO SMART: MORE COMPILATION

In a performance experiment, the dangers of JIT compilation are not only that it progresses gradually or that it is not deterministic, but also that it may be rather sensitive to minute differences between the performance experiment and the real environment the experiment approximates. As a result, the experiment may see optimizations systematically different from reality.

3.1 Optimizing Experiment Workload

It is common for a performance experiment to repeat the measured code multiple times. Also, a performance experiment often cares only about the timing, throwing away the result that would be used in reality. Without careful coding, this may lead to optimizations such as moving loop-invariant parts of the measured code out of the measurement loop, simplifying the measured code through constant propagation from outside the measurement loop, merging multiple iterations of the measured code by loop unrolling and common subexpression elimination, and more.

The black hole support in JMH is particularly helpful in efficiently preventing similar optimizations [21, 19].

As the flip side of the same coin, a performance experiment should not attempt to measure isolated code that would not be isolated in reality.

3.2 Polymorphic Invocation

Calls whose target address is difficult to predict entail overhead at the processor instruction level. Optimizations used to make polymorphic invocations predictable include class hierarchy analysis and target caching. Both are possibly sensitive to performance experiment conditions, such as loading or exercising a relatively limited subset of classes.

3.3 Optimization Fallback

Compared to static compilers, performance experiments with dynamic compilers require a subtly different mindset. Where a static compiler needs to prove a particular optimization correct before using it, a dynamic compiler may optimize tentatively and surround the optimized code with a test that guards the correctness prerequisites. When the guard fails, the assumptions are corrected and compilation redone.

To guarantee realistic optimizations, a performance experiment must emit workload that violates unrealistic compiler assumptions. This may entail taking unusual branches, throwing unexpected exceptions, or loading and exercising classes that are not otherwise essential to the workload.

3.4 On Stack Replacement

Normally, a method compiled at lower optimization level is replaced by a method at higher optimization level between calls. This poses a problem for methods with long loops, where the more optimized version may wait long for the less optimized version to exit.

Replacement of an executing method is possible but may require converting stack layout and limiting available optimizations. Both may impact the performance experiment.

4. MANAGED MEMORY

Java heap uses garbage collection (GC) with many performance implications. Importantly, a performance experiment must decide whether the GC cost should be included or avoided. Using large enough heap and forcing collections between iterations may avoid most GC costs, but from the practical perspective, including GC cost may provide more relevant results.

Heap size. Heap size has significant impact on GC cost [2]. Adaptive heuristics that tune heap size are therefore complicating performance experiments [15, 12].

Heap content. A performance experiment may generate heap content that is less diverse than reality. When using a large heap, most objects are in the young generation, with better locality and fewer references between generations. Experiments that discard results between iterations lower GC costs by reducing live data. Collections between iterations may also encounter few live objects, making GC cheaper.

Sometimes, comparing workloads with different allocation behavior is needed. Additional allocations exhaust allocation buffers (TLAB) sooner, causing more slow-path allocations and more young collections. In turn, this pushes more objects into the old generation. This often leads to higher GC costs.

Many changes to the heap content also impact the behavior of weak reference types.

Optimized allocation. Using escape analysis or escape detection, JVM can allocate objects on stack rather than on heap [17]. As a side effect, measurement may turn stack allocation into heap allocation [11]. This increases the allocation cost and disables some potential optimizations.

Storing measurements. The measurements are often stored on the same heap that the workload uses. This again impacts heap content, in particular when storing the measurements in dynamic structures such as lists. The very presence of a long list may decrease the throughput of parallel collectors, because the list must be traversed sequentially, also part of the list will likely be in the young generation, with impact as above.

Dynamic structures suffer from bloat that may not be immediately apparent [13]. Using an array instead of a dynamic structure is less flexible, but may be more efficient and more predictable, especially when the array items are primitive.

When necessary, it is possible to store measurements outside the heap, using native memory through JNI or through the sun.misc.unsafe API.

5. PARALLELISM

When collecting data from more threads, proper workload synchronization is essential. Where blocking is reasonable, the java.util.concurrent API may suffice. In some cases, it is necessary to keep workload threads running and only synchronize their data collection phases.

Biased locking. Parallel workload often involves synchronization. Synchronization may be heavily optimized in favor of certain common cases, such as the same thread repeatedly acquiring a lock without contention [20]. To assess realistic synchronization performance, a performance experiment must therefore use realistic contention.

Synchronization optimizations may exhibit anomalies due to internal implementation details. For example, biased locking is disabled for certain time after JVM start, and does not work for objects whose identity hashcode was queried [20].

Memory sharing. Contemporary architectures utilize multiple levels of memory caches, some local to cores, some shared among cores. Although JVM may use thread local allocation buffers (TLAB) to prevent threads on different cores from allocating data near each other, various workload patterns may lead to excessive cache traffic.

Particularly disruptive issue is that of false sharing, where unrelated variables occupy the same cache line. Modifying such variables from multiple cores generates needless cache coherency traffic whose reason is difficult to discern at source code level. In artificial (especially extremely regular) workloads, similar effects may occur with internal JVM structures, such as card tables.

Randomizing workload parameters tends to reduce the chance of encountering performance anomalies due to artificial memory sharing patterns.

NUMA. On NUMA architectures, memory access performance depends on node locality. Automated partitioning of data to improve locality is a technically difficult problem [22]. Solutions that avoid some worst-case scenarios include thread-local allocations and interleaving data in shared heap spaces [17].

6. SENSORS

Besides paying attention to how the workload of a performance experiment behaves, we also need to pay attention to the sensors used to measure the workload. Perhaps the most obvious sensor is the time source.

Timing accuracy. The most easily accessible time source in Java is calling System.nanoTime. The exact behavior of this method is platform-dependent, for example Open-

JDK 8 on Linux queries the CLOCK_MONOTONIC system clock, OpenJDK 8 on Windows uses the QueryPerformanceCounter call. These are both high precision time sources, however, on older systems the same method can use microsecond or millisecond granularity sources.

Internally, the high precision time sources may use hardware counters with unknown or varying frequency. Such sources are callibrated against counters with known frequency but possibly lower accuracy, this callibration can yield slightly different frequency estimate on each initialization. Also, the CLOCK_MONOTONIC system clock is subject to adjustments on systems with NTP or PTP support, even during measurement.

Performance counters. Hardware performance event counters are another important sensor in performance experiments. Querying the counters from Java requires making native calls, possibly through JNI. The associated overheads may limit usefulness. Libraries for accessing these and other sensors from Java are also available [9].

7. MISCELLANEA

The tutorial will also touch on other topics that we do not describe in detail now. Some diverse examples include anomalous optimization behavior on loop constructs that do not qualify as counted due to choice of the loop control type, caching of some boxed primitive values and impact on memory allocation behavior, overhead associated with invoking native code through JNI, and effects of thermal budget on turbo boosting and frequency scaling [10].

Acknowledgements

This work was partially supported by the Charles University institutional funding and the EU project ASCENS 257414.

8. REFERENCES

[1] Azul. Zing JVM, 2014. http://www.azulsystems.com/products/zing/virtual-machine.

[2] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. SIGMETRICS '04/Performance '04. ACM, 2004.

[3] S. M. Blackburn et al. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8), Aug. 2008.

[4] A. Buble, L. Bulej, and P. Tůma. CORBA benchmarking: a course with hidden obstacles. In *PDPS*, April 2003.

[5] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *OOPSLA*. ACM, 2008.

[6] J. Y. Gil, K. Lenz, and Y. Shimron. A microbenchmark case study and lessons learned. In *SPLASH Workshops*. ACM, 2011.

[7] B. Goetz. Java theory and practice: Anatomy of a flawed microbenchmark, 2005. http://www.ibm.com/developerworks/java/library/j-jtp02225/.

[8] D. Gu, C. Verbrugge, and E. M. Gagnon. Relative factors in performance analysis of Java virtual machines. ACM, 2006.

[9] V. Horký. Java microbenchmark agent, 2014. http://github.com/d-iii-s/java-ubench-agent.

[10] Intel. Intel Turbo Boost technology, 2014. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[11] P. Libič, L. Bulej, V. Horký, and P. Tůma. On the limits of modeling generational garbage collector performance. In *ICPE*. ACM, 2014.

[12] P. Libič, P. Tůma, and L. Bulej. Issues in performance modeling of applications with garbage collection. In *QUASOSS*. ACM, 2009.

[13] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*. ACM, 2007.

[14] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*. ACM, 2009.

[15] Oracle. Memory management in the Java HotSpot virtual machine, 2006. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

[16] Oracle. The Java® virtual machine specification, 2013. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html.

[17] Oracle. Java HotSpot virtual machine performance enhancements, 2014. http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

[18] Oracle. Java invocation documentation, 2014. http://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html.

[19] Oracle. Java microbenchmarking harness (OpenJDK: jmh), 2014. http://openjdk.java.net/projects/code-tools/jmh/.

[20] Oracle. Synchronization (HotSpot internals for OpenJDK), 2014. http://wikis.oracle.com/display/HotSpotInternals/Synchronization.

[21] A. Shipilev. Java microbenchmark harness (the lesser of two evils). Presented at Devoxx, 2013.

[22] M. Tikir and J. Hollingsworth. NUMA-aware Java heaps for server applications. In *IPDPS*, 2005.