Java Performance Evaluation Using External Instrumentation

Georgios Gousios, Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece, 155 62
Email: {gousiosg,dds}@aueb.gr

Abstract

The performance of programs written in the Java programming language is not trivial to analyse. The Java Virtual Machine hides the details of bytecode execution while not providing an accessible profiling mechanism. Most tools used for Java performance evaluations are based on sampling and only present engineers with sampled data aggregations. In this paper, we present the Java DTrace Toolkit, a collection of scripts that is specifically designed to assist engineers in identifying the roots of various performance problems observed with other tools.

1. Introduction

In recent years, there has been a trend toward developing and running Internet and network servers using safe languages and processes-level Virtual Machine (VM)-based runtime environments. This trend is justified; VMs offer a more secure execution environment than native platforms, while they allow programs to be portable. Those facilities come at a cost: process-level VMs introduce several layers of indirection in the chain of management of computing resources. VM runtime environments offer services complementary to those of operating systems, such as processing time sharing and pre-emptive multithreading, memory management, and I/O request handling. Despite the advances in automatic memory management and Just-In-Time (JIT) compilation, which brought the number crunching abilities of process VMs to nearly-native levels, there is a very small number, if any, of VM-based server software applications that can match the performance of widely deployed, natively compiled network servers, such as Apache or Samba.

Apart from the I/O performance handicap that current resource sharing mechanisms inflict on Java programs, there is another factor that contributes to the situation described above: the performance of Java programs is notoriously difficult to analyse, especially when considering layered, network-centric systems. In this paper, we present the Java DTrace Toolkit (JDT) a collection of tools that exploit the DTrace instrumentation facilities, and more specifically the Java probe provider, to gather performance and debugging information from live programs. We also present examples of how each tool can be used to evaluate the performance of Java software.

The remainder of the paper is organised as follows: in Section 2, we present an overview of the tools and techniques used for monitoring the performance of Java programs. In Section 3, we present the collection of tools we have developed and give motivating examples of their use. Finally, in Sections 4 and 5 we briefly describe the tool's implementation and discuss its strenghts and weaknesses.

2. Java Performance Monitoring

Monitoring the performance of Java systems, or managed runtime systems in general, is considered a difficult task. Performance-wise, the JVM is a black box to the engineer. In native systems, it is straightforward to obtain the exact distribution of CPU cycles a particular program will consume even if there are library dependencies, as the developer has access to both the executed code [8] and directly to performance monitoring counters on the CPU [2, 1]. Also, depending on the implementation language, the amount of memory a program allocates can be traced back to requests, as memory allocation is performed mostly manually. The

[1]In In Stefanos Gritzalis, Dimitris Plexousakis, and Dionysios Pnevmatikatos, editors, *PCI 2008: 12th Panhellenic Conference on Informatics*, August 2008. IEEE Computer Society. (doi:10.1109/PCI.2008.14)

JVM hides those performance indicators by abstracting the underlying hardware and employing custom bytecode execution mechanisms. In addition, different JVMs employ different JIT compilers and different garbage collection algorithms, thus further complicating the understanding of program execution flow.

Moreover, the behaviour of the Java execution environment is not predictable. While in the case of native programs the only source of disruption in program flow can be hardware interrupts or signals, in Java a number of events can lead to non-deterministic behaviour and consequently to difficulties in assessing performance, especially for short execution time programs. Firstly, all JVMs use timer-based method execution sampling to drive the JIT compilation process: different executions of the same program may result in different samples being taken and different methods being compiled [12]. Also, the fact that the JVM relies on the operating system for scheduling threads on processors means that different program executions can be subject to different scheduling plans, which can affect benchmarking results. Combined differences in JIT compilation and thread execution can change the benchmark execution profile, which in turn may affect the frequency of garbage collections, especially when compilation-driven techniques such as escape analysis [5] are employed to limit object allocations. In reference [7], Georges et al. show that the difference between the best and mean execution time for a single benchmark can be as high as 40%.

The majority of profiler tools for Java either rely on events generated by the JVM [13] or on bytecode instrumentation [9] to gather performance related events, even though source code based techniques have also been proposed [11]. Current production JVMs emit performance-related events and enable tools that consume those events to be attached to them through the JVM Tool Interface [6]. The Java Management Extensions [10] enable access to performance-related counters in the JVM over a well-defined protocol. A variety of tools for JVM monitoring and application profiling is based on either of those protocols. Among the most well known are the Netbeans profiler, JProfiler and the Websphere console. Those tools can provide a good overview of application behaviour, but are restricted by the underlying protocol's inability to extend analysis beyond the JVM boundaries.

DTrace [4] is a instrumentation tool that is designed to enable application and system performance evaluations with minimal effect. Its strength comes from the fact that it can combine in a single profiling session data from various layers of the execution stack, ranging from methods in a Java program to low-level in-kernel routines. Also, DTrace can place instrumentation hooks while the system is running, a feature that can help users identify performance problems that only appear under special circumstances. Profiling scenarios for DTrace are written in a custom domain specific language, although the majority of DTrace-based tools use a scripting language to wrap user input and format results. A JVM probe provider is available with recent versions of the Sun JVM.

3. The Java DTrace Toolkit

To address the issues described above, namely the lack of tools for performance evaluation across different layers of the Java execution stack and the need to debug performance problems when they occur, we designed and implemented the JDT. As its name implies, the JDT is a set of tools based on DTrace that can help a user identify performance issues in complex application scenarios. The JDT is designed to comply with the following functional requirements:

Least possible intervention: The probes should have minimal effect to program execution. Specifically, the JDT explicitly avoids probes that have more than 5 entry points and defers result aggregation for the end of the tracing session.

Focused output: The JDT offers runtime switches to filter captured data that are not of interest to the user.

Connection to live systems: The fact that that JDT is based on DTrace allows it to connect to live systems. However, for expensive probes, special start switches must be enabled in the JVM configuration.

Parseable output: Following the Unix tradition, our tools can be used in combination with other tools. Special care was put on producing parseable output.

Table 1 presents an overview of the tools that comprise the JDT. In the following sections, we describe the design and the functionality of each individual tool and also provide examples of each tool's output.

3.1. `jprofiler`

The `jprofiler` program implements a classic application profiler: by default, it instruments all program methods, counts their invocations and returns the number of calls and their total execution time. On user request, it can instrument only specific methods or classes and aggregate the results by package name. The following code snippet is the per-package aggregated output of methods required by the Tomcat web server to serve a static web page. The method calls to standard Java libraries have been stripped out.

```
Package                                cnt
=====================================
org/apache/coyote/http11/filters        6
org/apache/catalina/servlets           12
```

Table 1. DTrace toolkit scripts and their use

Tool	Usage
jprofiler	Reports the methods and classes that consume most execution time. Can aggregate results by package name.
jmemstats	Report object allocation statistics. Can aggregate results by package name and filter results by specific package names.
jlockstat	Reports methods initiating locking operations in native code.
jiosnoop	File management statistics: which classes cause I/O traffic?
jcallgraph	Display a Java function call graph from Java to the OS kernel.
jgcsnoop	Reports garbage collection statistics: frequency, duration.

```
org/apache/tomcat/util/http/mapper  14
org/apache/tomcat/util/http         16
org/apache/naming/resources         18
org/apache/catalina/core            23
org/apache/coyote                   25
org/apache/coyote/http11            28
org/apache/catalina/connector       28
org/apache/tomcat/util/buf          51
```

3.2. jmemstats

Java uses automated memory management to manage the executed program's runtime heap. The advantages of garbage collection include very fast, potentially uncontented, memory allocation, minimal memory fragmentation, and cache effectiveness. The main disadvantage is that it consumes computing cycles for memory reclamation and it stops the execution of application threads while memory is reclaimed and compacted [3]. However, the negative effects of garbage collection can be minimized if allocation rates are constrained. To restrict a program's memory allocation rate to the absolute minimum, one has to be careful to select the appropriate data structures for intermediate results, while also understanding the object allocations performed by external libraries. For example, XML parsers, GUI toolkits and container objects and are notorious for poor memory behaviour.

The purpose of the `jmemstats` tool is to help developers to keep track of object allocations and identify the methods whose allocation rates are high. The main difference of this tool from other similar tools is that it reports on the classes that perform excessive allocations, instead of the reporting on the types of objects that are allocated. This helps developers to identify immediately hot allocation sites. The following extract is from the tool's output when applied on the Tomcat web server, displaying the methods that have allocated the most objects.

```
Java Method                                    objects alloc
================================================================
java/lang/StringCoding$StringEncoder.encode      486
java/util/HashMap.newKeyIterator                 221
org/[...]/buf/CharChunk.toStringInternal         142
java/io/UnixFileSystem.resolve                   131
java/lang/String.substring                       122
java/lang/StringCoding$StringDecoder.decode      108
java/net/Socket.getInputStream                   100
java/lang/AbstractStringBuilder.expandCapacity    99
java/lang/Object.clone                            91
java/io/UnixFileSystem.list                       74
```

3.3. jlockstat

Locking is required to avoid concurrent modification of common computing resources. Java implements locking operations at the language level, while current generation JVMs are able to remove the majority of locks by deeply analysing the bytecode, thereby increasing performance. However, the JVM is often required to access operating system resources which, being shared among processes, must protect their critical areas. In several occasions, JVM threads accessing the operating system must be blocked or even stopped outside of the JVM, which in turn forces the JVM to switch to another thread or block. Involuntary thread switching is known for deteriorating the performance of Java, for reasons ranging from invalidation of processor caches to lost opportunities for method optimisation.

The `jlockstat` tool implements a native level lock analyser; it uses the DTrace `plockstat` provider to extract user space locking operations on mutexes and reader-writer locks. More interestingly, it correlates the lock acquisition at the native level to the Java method or stack frame that produced the native call. It can aggregate results in {`Java method, native method`} tuples and filter out locking operations based on user specified package-based filters. Using this information, the developer can understand what locking operations are taking place in the native level and how those affect the execution of Java code. The following output extract presents the locking operations at the native library level performed while servicing a series of web page requests on a Tomcat web server.

```
Java Method                                    Native       cnt
================================================================
java/io/UnixFileSystem.getBooleanAttributes0 malloc       367
java/io/UnixFileSystem.getBooleanAttributes0 free         363
java/lang/ClassLoader.defineClass1           free         288
java/lang/ClassLoader.defineClass1           malloc       290
java/io/UnixFileSystem.getLastModifiedTime   free         136
java/io/UnixFileSystem.getLastModifiedTime   malloc       136
java/io/UnixFileSystem.list                  readdir64_r  134
```

3.4. jiosnoop

Current generation JVMs use the operating system services to perform I/O operations. This means that for each read or write operation to an I/O device, such as a hard disk or a network card, the JVM thread must be stopped while

the operating system serves the request. If the I/O operations are small and frequent, then the effect in performance can be disastrous. While the I/O interfaces in Java are well defined and therefore can be instrumented, it is common for certain types of programs that include external components or do not have centralised I/O facilities to experience poor performance.

The `jiosnoop` tool is designed to correlate Java I/O operations with operating system I/O primitives, namely the `read(2)`, `write(2)`, `send(2)` and `sendfile(2)` system calls. The tool's output is a table listing of the I/O initiating class, the corresponding system call, the number of times this I/O path has been followed, and the total time spent for I/O per path. To the best of our knowledge, the JDT is the first tool that allows this kind of analysis for Java I/O.

```
Java Method                                                  syscall  cnt
============================================================================
org/apache/coyote/Response.action                           _so_send  7
org[...]Http11Processor.action                              _read     7
org/[...]OutputStreamOutputBuffer.doWrite                   _so_send  2
org/[...]Http11ConnectionHandler.processConnect             _read     1
org[...]/mapper/Mapper.internalMapWrapper                   _read     1
```

3.5. `jcallgraph`

Java is a dynamic language, in the sense that executable code can be substituted by code that is loaded at runtime. In programs featuring complex inheritance or composition relationships and using reflection to invoke functionality on runtime-loaded code, it is often difficult to determine the code path that is executed in a particular context. In such cases, it is useful to know at runtime the call hierarchy for a specific method. The `jcallgraph` tool outputs the call hierarchy for a user specified Java method both in Java code and, if the method call results in a native function call, the call hierarchy in the native context up to the operating system boundary. The user can also specify a thread identifier to monitor invocations of the method in a specific thread. The Java portion of the call graph for the `accept` method of the `java.net.ServerSocket` class is presented in the following code extract.

```
java/net/ServerSocket.accept Call graph
=======================================
java/net/ServerSocket.accept
 java/net/ServerSocket.isClosed
 java/net/ServerSocket.isBound
 java/net/Socket.<init>
  java/lang/Object.<init>
  java/lang/Object.<init>
 java/net/ServerSocket.implAccept
  java/net/Socket.setImpl
   java/net/SocksSocketImpl.<init>
    java/net/PlainSocketImpl.<init>
     java/net/SocketImpl.<init>
      java/lang/Object.<init>
      [6 entries removed for brevity]
     java/lang/Object.<init>
     java/lang/Object.<init>
   java/net/SocketImpl.setSocket
    java/net/InetAddress.<init>
     java/lang/Object.<init>
    java/io/FileDescriptor.<init>
     java/lang/Object.<init>
    java/net/ServerSocket.getImpl
    java/net/PlainSocketImpl.accept
     java/net/PlainSocketImpl.acquireFD
     java/net/PlainSocketImpl.socketAccept
```

3.6. `jgcsnoop`

Java uses garbage collection to manage heap memory. The default garbage collector must stop all application threads in order to operate. To reduce the time spent in garbage collection, the heap is divided into various memory zones, while different collection algorithms are used per zone. As its name implies, the `jgcsnoop` tool is a simple script that reports garbage collection invocations, the affected memory zone and the time required to finish. The tool's output monitoring the Tomcat web server can be seen below.

```
Memzone                 ZoneMgr Time
=====================================
GC: Code Cache           (Copy):15 msec
GC: Eden Space           (Copy):19 msec
GC: Survivor Space       (Copy):19 msec
GC: Tenured Gen          (Copy):19 msec
GC: Perm Gen             (Copy):19 msec
GC: Perm Gen [shared-ro](Copy):19 msec
GC: Perm Gen [shared-rw](Copy):19 msec
```

4. Implementation

Due to result and input processing limitations present in the language used by DTrace for specifying instrumentation scenaria (usually referred to as the D programming language), the majority of the tools are written as Perl scripts that encapsulate D programs. The host language parses command line arguments, starts the profiling session and finally parses and aggregates the DTrace tool output. The Perl scripts themselves are minimal; most functionality is provided through a shared component, the `DTrace::Parse` Perl library, which we intend to submit to the CPAN repository.

The `DTrace::Parse` can work with two basic types of DTrace output: aggregation results and stack traces. Aggregations are special D language constructs that collect thread-specific data during tracing in a hash map structure per thread. After the end of the profiling session the data are aggregated using the hash map key to identify similar data. Stack traces are produced by calling the `jstack()` D function in the appropriate probe processing body. The `jstack()` function produces stack traces that combine JVM and Java execution frames. Currently, the `DTrace::Parse` library provides methods that operate on text data and can parse, aggregate, filter, dissect and correlate related entries.

5. Discussion

The JDT is a generic framework for the evaluation of the performance of Java applications. As such, it can be used on a variety of scenaria ranging from identification of performance bottlenecks to examination of application flow. However, as is the case for most performance evaluation tools, the tool itself cannot identify performance hotspots, but only offer hints and insight. An interesting possible use of the toolkit is the combination of more than one tools in the same profiling session; for example, the `jlockstat` and `jiosnoop` utilities can be used in parallel to monitor locking while performing I/O. Also, the `jiosnoop` tool can produce output that can then be fed to `jcallgraph` using standard Unix tools to process the intermediate results, i.e. `jiosnoop.pl|cut -f1 -d' '|xargs jcallgraph`

One of the limitations of the JDT is the fact that the underlying instrumentation tool is only available to the Solaris operating system and the corresponding probes in version 1.6 of the JVM. The situation however is currently improving with the recent release of the JVM as open source software and the porting of the DTrace framework on other operating systems such as MacOSX and FreeBSD.

The JDT essentially provides an additional layer of data aggregation and analysis on top of the data collection facilities offered by the DTrace tool. The JDT does not affect the runtime behaviour of DTrace in any way, even though the accuracy of the aggregated data depends solely on the provided output. Currently, both the DTrace Java probe and the DTrace tracing output facilities are not able to handle large volumes of trace data, such as those generated by instrumenting recursive method invocations, object allocations or locking operations in concurrent environments. The JDT can identify and isolate erroneous DTrace output and exclude them for result reporting. Based on our experiments on a moderately loaded system, 10% of the total tracing output returned by DTrace is incomplete or erroneous; however as the tracing load increases and especially in multithreaded environments, the failure rate increases significantly: on a moderately loaded Tomcat web server instance running in a virtual machine inside an otherwise idle computer, about 40% of the result traces are unusable. This fact means that while the JDT is valuable for identifying trends and bottlenecks, it should not be used for accurate profiling.

6. Conclusion and Future Work

Evaluating the performance of Java code is arguably a difficult task. The JDT is a collection of tools that enables developers to identify performance problems, with a particular emphasis on those problems that concern the cooperation between the JVM and the operating system. Currently, we use the JDT along with more specialised DTrace scripts to show that the JVM co-operation with the operating system is not optimal. Specifically, we find unnecessary duplication in the computing resource management duties between the JVM and the OS and we measure its effects on the performance of Java programs. Our long term goal is the creation a Java execution environment that can manage the computing resources internally.

The JDT will be freely available as open source software from `http://istlab.dmst.aueb.gr/~george/sw/jdt` under the Apache license in late 2008.

Acknowledgment This work is partially funded by the Greek Secretariat of Research and Technology thought, the Operational Programme COMPETITIVENESS, measure 8.3.1 (PENED), and partially by the European Community's Sixth Framework Programme under the contract IST-2005-033331 "Software Quality Observatory for Open Source Software" (SQO-OSS)

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 1–14, New York, NY, USA, 1997. ACM.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 25–36. ACM Press, 2004.

[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28. USENIX, 2004.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999.

[6] R. Filed. JSR 163: Java platform profiling architecture. Java Community Process, Dec 2004.

[7] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 57–76, New York, NY, USA, 2007. ACM.

[8] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[9] H. B. Lee and B. G. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec 1997. USENIX.

[10] E. McManus. JSR 255: Java management extensions (JMX) specification, version 2.0. Java Community Process, Dec 2007.

[11] D. Pearce, M. Webster, R. Berry, and P. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, Jun 2006.

[12] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java Just-In-Time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, 2005.

[13] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, 2000.