A Java Performance Monitoring Tool

Marcel Harkema, Dick Quartel, Rob van der Mei, Bart Gijsen

Abstract— **This paper describes our Java Performance Monitoring Tool (JPMT), which is developed for detailed analysis of the behavior and performance of Java applications. JPMT represents internal execution behavior of Java applications by event traces, where each event represents the occurrence of some activity, such as thread creation, method invocation, and locking contention. JPMT supports event filtering during and after application execution. Each event is annotated by high-resolution performance attributes, e.g., duration of locking contention and CPU time usage by method invocations. JPMT is an open toolkit, its event trace API can be used to develop custom performance analysis applications. JPMT comes with an event trace visualizer and a command-line event trace query tool for shell scripting purposes. The instrumentation required for monitoring the application is added transparently to the user during run-time. Overhead is minimized by only instrumenting for events the user is interested in and by careful implementation of the instrumentation itself.**

Index Terms—**Measurement, Software performance.**

I. Introduction

OVER the past few years Java [1] has evolved into a mature programming platform. Its portability makes Java a popular choice for implementing enterprise applications and off-the-shelf components, such as middleware and business logic. The performance of this software is of particular importance when it is used by businesses to implement services to their customers: software performance can have an immediate impact on the quality of service that the customers perceive, and thus also on revenue.

There are many approaches to address performance, such as load/stress testing of components and applications, performance monitoring of deployed components and applications, performance prediction using models or rules of thumb, etcetera [8]. We developed the tool to aid us in constructing performance models of Java applications. However, we'd like to emphasize that the tool is also usable for performance testing, and such.

The construction of performance models requires insight in the application execution behavior (how does the application work) and *good-quality* performance measurements (how fast is the application performing its work) [17]. The insight in application execution behavior can be used to develop the structure of the performance model. The performance measurements can be used to identify the parts of the execution behavior that need modeling, the model parameters, the values for the model parameters, and to validate performance models. For Java, the relevant aspects of execution behavior are threading, synchronization and cooperation between threads, method invocation, dynamic object allocation, and garbage collection.

In this paper we present our tool for monitoring these execution behavior aspects in Java applications. We developed this tool because we found that existing tools did not provide us with the functionality we require, which is: (1) monitoring results of the aforementioned aspects of execution behavior, (2) precise timestamps and CPU usage measurements, (3) an open data format for the monitoring results, so that custom post-processing scripts can be developed, and (4) automated instrumentation at run-time based on a user-provided configuration file, which defines the execution behavior the user is interested in.

Profiler tools, such as IBM Jinsight [5], Compuware NuMega DevPartner TrueTime [10], Sitraka Jprobe [14], OptimizeIt [16] and Intel VTune [6], do not provide complete traces of events that have occurred in the virtual machine; they employ call stack sampling (observing the state of the system at certain time intervals) to inform the user of execution hot spots, i.e., parts of the code that use the most execution time. These profiler tools are used to find performance problems and to optimize programs. Our goal is different; we want to measure the performance of user specified methods and provide the user with a complete method invocation trace. Therefore JPMT instruments the software to log events, such as method invocations, as they happen.

Most profiler tools restrict the user to a GUI that provides a fixed view on the performance. Instead of providing fixed views, JPMT logs event traces in an open data format, allowing users to build custom tools to execute the performance queries they require. Rational Quantify [13] and IBM Jinsight [5] allow exporting of data to, e.g., spreadsheets or scripts.

Often, profilers only support filtering after program execution. An exception is Rational Quantify [13], which allows the user to indicate which data is to be collected and reported, and the level of detail of the data. JPMT supports both online and offline filtering of interesting events (i.e., at

Marcel Harkema is a computer science PhD student at the University of Twente, Enschede, The Netherlands (e-mail: harkema@cs.utwente.nl).

Dick Quartel is with the Department of Computer Science, University of Twente, Enschede, The Netherlands (e-mail: quartel@cs.utwente.nl).

Rob van der Mei is with TNO Telecom, Center of Excellence Quality of Service, Leidschendam, The Netherlands, and with the Free University, Faculty of Exact Sciences, Amsterdam, The Netherlands (e-mail: r.d.vandermei@telecom.tno.nl).

Bart Gijsen is with TNO Telecom, Center of Excellence Quality of Service, Leidschendam, The Netherlands (e-mail: b.m.m.gijsen@telecom.tno.nl).

run-time and during event trace post-processing).

The remainder of this paper is structured as follows. Section 2 presents an overview of the main features of the tool. Section 3 illustrates the use of the tool by applying it to a simple example. Section 4 contains some concluding remarks.

II. Features

The Java Performance Monitoring Tool (JPMT) is based on event-driven monitoring. JPMT represents the execution behavior of applications by event traces, in which each event represents the occurrence of some activity, such as a method invocation or the creation of a new thread of execution. These vent traces are similar to call-trees, but in addition to method invocations they also contain other event types. Our monitoring tool implements the event-driven monitoring approach, since we require complete behavioral information (not just snapshots of the system, like tools using the sampling technique provide).

The core of JPMT is an agent that is inserted into the Java virtual machine using the Java Virtual Machine Profiler Interface (JVMPI). JVMPI agents are implemented as dynamically linked libraries written in C++. The agent uses JVMPI to subscribe to events that occur inside the virtual machine, collecting information on what happens inside the virtual machine on behalf of the tool. An important feature of JVMPI is its portability - Java implementations by Sun and IBM support JVMPI out-of-the-box in production virtual machines.

JPMT does not use Java or JVMPI to obtain timing information. The timestamps provided by Java and JVMPI are too coarse, they have a granularity of 10 milliseconds on most systems. Also, Java and JVMPI do not provide information on the CPU usage. Instead of using Java or JVMPI, JPMT uses native operating system APIs. For instance, on the Linux operating system (and other UNIX systems) `gettimeofday(2)` can be used to obtain wall-clock timestamps with microsecond resolution (around 7 usec on an Intel Pentium Pro 180 MHz and 1 usec on an AMD Athlon XP 2000+). The perfctr API [12], a kernel add-on for Linux, can be used to obtain detailed information on CPU usage. On the Windows platform similar APIs are available, such as the `QueryPerformance(Win32)` API.

The following elements of Java's execution behavior can be monitored:

Threading: Java applications can have multiple threads of execution. The creation and destruction of these threads is monitored. Each thread has its own event trace.

Thread synchronization and cooperation: Java uses *monitors* [3] to implement thread synchronization (Java's synchronized primitive) and cooperation (Java's `wait()`, `notify()`, `notifyAll()` methods in the `Object` class). JPMT can report how long a thread has contended for a monitor, how long it held a monitor, how long it spend waiting for a signal from another thread (cooperation), etc.

Method invocation: The sequence of method invocations is represented in a call-tree notation for each thread. A method invocation event includes the timestamp of when it occurred, the used wall-clock time in microseconds, the used CPU time in microseconds, and whether or not the method was compiled from byte-code into native code by the Java virtual machine. The wall-clock and CPU time event parameters of the caller method include the wall-clock and CPU time of callee methods. Mutual exclusion, cooperation, garbage collection, process scheduling events, and such, are shown within the method in which they occurred. While JVMPI does have the ability to monitor method invocations, it does not have the ability to select which methods are to be monitored and which methods not. When subscribed to JVMPI method invocation events, events are generated for every method invocation (including the ones the user may not be interested in), which can add significant monitoring overhead. Because we wanted to keep the monitoring overhead down to a minimum, we chose to develop our own method invocation monitoring mechanism. We have implemented a library written in C++ that can *rewrite Java byte-code* (the compiled form of Java source code). Using the library Java methods can be instrumented to trigger the method invocation event handlers present in our JVMPI agent. The library is linked with the JVMPI agent, and rewriting the byte-code occurs at run-time.

Dynamic object allocation: Object allocation and release monitoring can be used to track down excessive object allocation and release, which can cause performance problems, not only because of the allocation and release cost, but also because it may trigger garbage collection cycles.

Garbage collection: Garbage collection cycles can have a significant impact on the performance of applications, since execution is suspended during garbage collection. JPMT can report when and where garbage collection cycles occur, and how long they take.

Process scheduling information (Linux specific): This provides information on the operating system process scheduling changes involving the monitored thread. There are two events that represent process scheduling changes in JPMT: *thread-in* and *thread-out*. Together they describe when a thread was actually running and when other processes where running. Currently, these events are not supported on other platforms than Linux. The Linux Trace Toolkit [18], a kernel tracing toolkit, is used to obtain this monitoring information.

JPMT is configured using a configuration file for each application it monitors. This configuration file is read by the JPMT when the Java virtual machine that will run the Java application is started. The configuration file allows the user to choose an output file to log events to, whether or not object allocation, garbage collection, method invocation, use of Java monitors are to be logged (that implement synchronization and cooperation mechanisms), whether or not to use byte-code instrumentation to monitor method invocations (instead of using JVMPI's method invocation monitoring mechanism),

and whether or not to monitor operating system process scheduling. Using the method and thread filters the user can specify which methods and threads should be monitored, and which should not be monitored. JPMT applies these filters in the same order as they are specified in the configuration file (from top to bottom). By default all threads are monitored and no method invocations are monitored, i.e. `include_thread * ` and `exclude_method * *` are the default thread and method filter settings. All 'yes/no' configuration directives (*Object_Allocation*, *Garbage_Collection*, *Monitor_Waiting*, *Monitor_Contention, Method_Invocation*, *Bytecode_Rewriting*, and *LTT_Process_Scheduling*) default to 'no'.

The following configuration file example logs events to log/mylogfile.bin, tells JPMT to monitor all method invocations in the `net.qahwah.jpmt.test` package using byte-code rewriting, and excludes all other method invocations from monitoring.

```
Output log/mylogfile.bin
Method_Invocation yes
Bytecode_Rewriting yes
Include_Method net.qahwah.jpmt.test.* *
Exclude_Method * *
```

During run-time our JVMPI agent collects the events of interest. The JVMPI agent instruments the software to be monitored when the software is loaded into the Java virtual for execution. The collected events are stored into a binary memory-mapped [15] file. After monitoring this binary file with the collected events can be analyzed. First, event traces are generated from the event collection. These event traces can be inspected using the event trace API, available for C++ and Ruby (a scripting language). Using the C++ event trace API we have implemented an event trace visualizer GUI for browsing event traces. The event trace API for Ruby is useful for implementing custom event trace post-processing scripts.

We have tried to keep the overhead as low as possible by allowing the user to specify what is to be monitored, implementing a fast event logging mechanism, and by instrumenting methods to be monitored using Java byte-code rewriting. The overhead depends on how many events the user wishes to monitor. The cost of monitoring and logging one method invocation (using byte-code rewriting) on an AMD Athlon XP 2000+ system (running at 1.67 GHz) is 5 microseconds CPU time. To put this into perspective, on the same machine we measured the CPU time of the method invocation `System.out.println("monitoring information")` to be 19 microseconds (after being compiled into native code by the Java HotSpot VM).

More information on the implementation (of an older version) of our tool is available from [2].

III. An Example

In this section we briefly illustrate how JPMT has been used to study performance behavior of Java software. In this example we look at the CPU processing overhead of marshaling and un-marshaling a sequence of floating point numbers in a CORBA object middleware implementation.

The Common Object Request Broker Architecture (CORBA) [11] [4] is the de-facto object middleware standard. CORBA mediates between application objects that may reside on different machines, by implementing an object-oriented RPC mechanism. CORBA provides programming language, operating system, and machine architecture transparency. For instance, it allows C++ objects to talk to Java objects that run on a remote host. To support this transparency a common data representation (CDR) is needed. Marshaling is a process that translates programming language, operating system, and machine architecture specific data types to a common data representation (CDR). Un-marshaling does the reverse. Marshaling and un-marshaling are used in every CORBA method invocation (although some CORBA implementations do not use marshaling/un-marshaling for local method invocations, to optimize these).

Our test-bed consists of 2 machines interconnected using a local network. The server machine is a Pentium IV 1.7 GHz with 512MB RAM, the client machine is a Pentium III 550 MHz with 256 MB RAM. Both machines run the Linux operating system and the Java 2 standard edition v1.4. The CORBA implementation we use in this example is ORBacus/Java 4.1 [7].

We have configured our CORBA workload generator to send requests consecutively (i.e. there is at most one active CORBA method invocation). We have performed 21 runs of 250 requests. With each run we increase the size of the sequence of floating point numbers with 5120 elements. In our experiment the float sequence is an *inout* parameter, i.e. it is sent to the server as a request parameter, and the server sends the (possibly modified) float sequence back to the client in its reply. The method that is invoked does nothing. The following is the CORBA IDL representation of our method:

```
interface PerformanceTest
{
  typedef sequence<float> FloatSeqType
  void ping_inout_floatSeq(inout
                    FloatSeqType floatSeq);
};
```

Figure 1 presents our findings: the CPU cost of marshaling and un-marshaling is a significant part of the total CPU time used by the CORBA server while processing the request. In figure 1 the top line represents the mean CPU time used for server-side processing of the CORBA request. The line below represents the sum of the mean CPU times used for reading the CORBA request from the network, un-marshaling the float sequence (of the request), and marshaling the float sequence again (for the reply). The bottom three lines represent the individual mean CPU times of marshaling the float sequence for the reply, reading the float sequence from the network, and un-marshaling the float sequence, respectively. The input data for figure 1 is obtained from the generated event traces of the CORBA server application. We can conclude that marshaling

and un-marshaling is an important performance issue in CORBA, and thus needs modeling. From figure 1 we can conclude that the CPU time of marshaling and un-marshaling float sequences can be effectively modeled as a linear function of the size of the float sequence.

In figure 2 we show an example plot of one of the methods involved in the server-side request processing. The top line represents the wall-clock completion time, the line below represents the time spent garbage collecting, the third line from the top represents the CPU time used, and the 'cloud' of dots in the bottom of the plot represents the parts of the completion time where another thread was running (involuntary context switches). We found that particular method interesting since it shows a great gap between CPU time and wall-clock completion time. According to figure 2 the difference is caused by garbage collection occurring sometime during that method invocation. The garbage collection cycles are caused by the large amount of dynamic object allocation and release to store the float sequences. From figure 2 we can conclude that garbage collection can be a significant influence on the wall-clock completion time of a CORBA method invocation. JPMT can be used to determine when garbage collection needs to be considered for performance modeling.

IV. Concluding Remarks

In this paper we presented our Java Performance Monitoring Tool (JPMT) for monitoring the performance of Java applications. JPMT represents Java's execution behavior, such as threading, method invocation, and mutual exclusion, by event traces with high-resolution timestamps and CPU usage information.

Our tool is under active development. We are currently implementing the second version, adding new capabilities to the event trace API and the event trace visualizer. The tool is to be released as open source software.

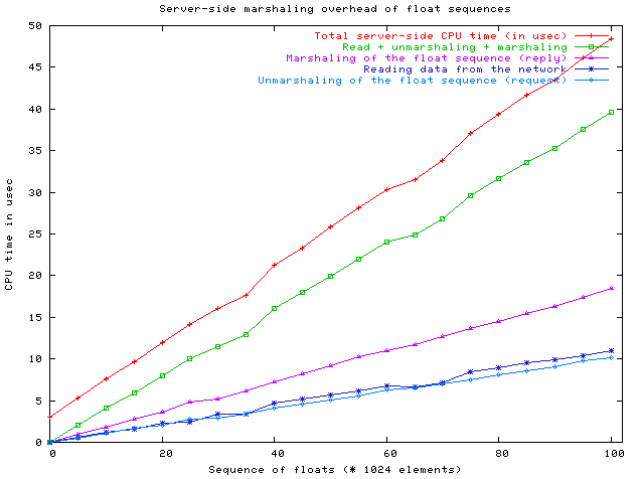

Fig. 1. The overhead of marshaling a sequence of floating point numbers in ORBacus/Java.

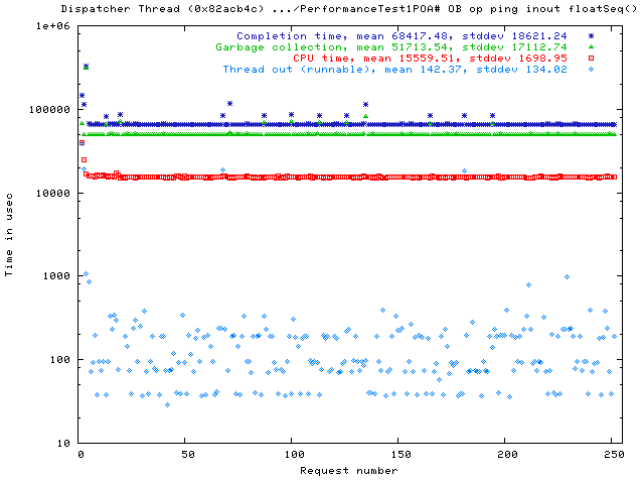

Fig. 2. Plot of the CPU time used by the *skeleton* of our CORBA object (includes un-marshaling and marshaling overhead). The size of the float sequence is 50*1024 elements in this plot.

References

[1] J. Gosling, B. Joy, G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[2] M. Harkema, D. Quartel, B.M.M. Gijsen, R.D. van der Mei, *Performance monitoring of Java applications*, Proc. of the 3rd Workshop on Software and Performance (WOSP), 2002.

[3] C.A.R. Hoare, *Monitors: An Operating System Structuring Concept*, Comm. ACM 17, 10:549-557 (October), 1974.

[4] M. Henning, S. Vinoski, *Advanced CORBA Programming with C++*, ISBN 0201379279, Addison-Wesley, 1999.

[5] IBM Research, *Jinsight project*, http://www.research.ibm.com/jinsight/, 2001.

[6] Intel, VTune Performance Analyzer, http://developer.intel.com/software/products/vtune/, 2001.

[7] IONA Technologies, Object Oriented Concepts Inc., *ORBacus 4 for Java*, 2000.

[8] R, Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, 1991.

[9] S. Liang, D. Viswanathan, *Comprehensive Profiling Support in the Java Virtual Machine*, 5th USENIX Conference on Object-Oriented Technologies and Systems, May 1999.

[10] Compuware NuMega, *DevPartner TrueTime Java Edition*, http://www.compuware.com/products/numega/dps/java/tt_java.htm, 2001.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, revision 2.5, OMG document formal/2001-09-01, 2001.

[12] M. Pettersson, *Linux perfctr OS extensions (sofware)*, 2001. http://www.csd.uu.se/~mikpe/linux/perfctr/

[13] Rational Software Corporation, Rational Quantify, http://www.rational.com/products/quantify_nt/index.jsp, 2001.

[14] Sitraka Inc., *Jprobe Suite*, http://www.klg.com/software/jprobe/, 2001.

[15] Vahalia, *UNIX Internals – The New Frontiers*, Prentice Hall, 1996.

[16] VMGear, *OptimizeIt*, http://www.optimizeit.com/, 2001.

[17] M. Woodside, *Software Performance Evaluation by Models*, In Performance Evaluation, LNCS 1769, 2000.

[18] K. Yaghmour, M. R. Dagenais, *Measuring and Characterizing System Behavior Using Kernel-Level Event Logging*, 2000 USENIX Annual Technical Conference, June 18-23 2000.