Performance Monitoring of Java Applications

M. Harkema D. Quartel B.M.M. Gijsen R.D. van der Mei

Department of Computer Science
University of Twente
The Netherlands
harkema,quartel@cs.utwente.nl

Expertise Group QoS Control
KPN Research
The Netherlands
b.m.m.gijsen@kpn.com

Mathematics and Computer Science
Free University Amsterdam
The Netherlands
r.d.van.der.mei@kpn.com

ABSTRACT

Over the past few years, Java has evolved into a mature platform for developing enterprise applications. A critical factor for the commercial success of these applications is end-to-end performance, e.g., in terms of response times, throughput and availability. This raises the need for the development, validation and analysis of performance models to predict performance metrics of interest. To develop and validate performance models, insight in the execution behavior of the application is essential, requiring advanced monitoring capabilities.

In this paper we introduce our Java Performance Monitoring Toolkit (JPMT). JPMT represents internal execution behavior of Java applications by event traces. An event represents the occurrence of some activity, such as thread creation, method invocation, and locking contention. Events are 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 scripting. JPMT supports event filtering during and after application execution. 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. Furthermore, the instrumentation itself is carefully optimized.

This paper discusses the architecture and implementation of the toolkit in detail and reports on our experience in applying the toolkit to model a CORBA implementation.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *performance measures.*

General Terms

Algorithms, Measurement, Performance, Design, Experimentation.

WOSP '02, July 24-26, 2002, Rome, Italy.

Keywords

Java, performance measurement, performance monitoring.

1. INTRODUCTION

The drastic growth of the Internet, the ongoing developments in the hardware and software industry, and the recent advances in networking technology have boosted the emergence of Information and Communication Technology (ICT). ICT systems enable applications to be divided in components that can be executed on geographically distributed information systems. These applications are commonly referred to as distributed applications. Distributed computing provides the fundamental technology for the realization of enterprise-wide and even global information systems.

The Quality of Service (QoS) provided by these enterprise applications is often a critical success factor. For example, unavailability of e-business applications directly leads to loss of revenue. Various QoS types can be identified, such as performance, availability, and security. In this paper, we focus on performance.

To ensure applications perform as required one could perform tests using a real system. However, often performance testing is unfeasible for a variety of reasons. For instance, testing might not be allowed because it would disrupt the operation of a running system. Building a test environment similar to the real world environment is often not possible, since it would require too much time and money. Sometimes testing is not possible since it's too difficult or expensive to reproduce certain workloads in the test environment, or simply because an implementation of the application to test is not yet available.

An alternative to performance testing is performance modeling. A performance model of a system is an abstraction of that system describing the parts of the system that are relevant to performance. Performance models provide performance *predictions*, rather than performance observations. These performance predictions are extremely useful to identify performance bottlenecks and to timely anticipate on future performance problems for projected growth volumes. Using, for instance, discrete event simulation, numeric approximations, or analytic techniques, the performance of a given scenario can be predicted. Performance analysis using models is often faster and cheaper than real-world testing. Contrary to testing, an implementation or prototype of the application is not needed for performance modeling. Performance models can be used to validate application design decisions before they are implemented. Since performance models provide only

predictions, and not observations, modeling results can become inaccurate if the model does not capture the dynamic behavior of the application close enough. Building accurate performance models of a system requires insight in the internal behavior of the system and good quality performance measures. As such performance monitoring is an important step towards useful performance models.

Over the past few years Java [5] has evolved into a mature platform. Its portability makes Java a popular language for implementing enterprise applications and off-the-shelf components, such as middleware and business logic. The objective of this paper is twofold: (i) present a performance monitoring toolkit for Java, and (ii) to illustrate how this toolkit can be applied for the development and validation of performance models of software components implemented in Java.

The remainder of this paper is structured as follows. Section 2 presents the requirements for the toolkit. Section 3 discusses the architecture. Section 4 discusses the implementation of our prototype. Section 5 applies the toolkit to monitor the performance of a CORBA object-middleware implementation for Java. Section 6 discusses related work. Section 7 lists future work. Section 8 concludes.

2. REQUIREMENTS

To build performance models of a system, a description of its execution behavior is needed. The description should include performance annotations so that the performance analyst is able to identify the behavior relevant for performance modeling. Accurate performance models require a precise description of the behavior, and good quality performance estimates or measures. [28] Our objective is to design a performance monitoring toolkit for Java that obtains both a description of the behavior of a Java program, and high-resolution performance measurements.

The toolkit should be able to monitor the following elements of execution behavior:

- The invocation of methods. The sequence of method invocations should be represented by a call-tree. To produce call-trees we need to monitor method entry and exit.
- Object allocation and release. In Java, objects are the entities that invoke and perform methods. The monitoring tool should be able to report information on these entities.
- Thread creation and destruction. Java allows multiple threads of control, in which method invocations can be processed concurrently. The monitoring tool should be able to produce call-trees for each thread.
- Mutual exclusion and cooperation between threads. Java uses monitors [6] to implement mutual exclusion and cooperation. The monitoring tool should be able to detect contention due to mutual exclusion (Java's `synchronized` primitive), and measure the duration. Furthermore, the monitoring tool should be able to measure how long an object spends waiting on a monitor, before the object is notified (`wait()`, `notify()`, and `notifyAll()` in Java).

The monitoring results should include attributes that can be used to calculate performance measures. For instance, to calculate the wall-clock completion time of a method invocation the timestamps of the method entry and exit are needed. The timestamps, and other attributes used, should have a high-resolution. For instance, timestamps with a granularity of 10ms are not very useful to calculate the performance of method invocations, since a lot of invocations may use less than 10ms.

Performance modeling is a top-down process. At various performance modeling stages, performance analysts may have different performance questions. During the early modeling stages the analyst is interested in a global view of the system to be modeled. The analyst tries to identify the aspects relevant for performance modeling. In later stages the analyst has more detailed performance questions about certain aspects of the system. The monitoring toolkit should support this way of working.

Instrumentation of a Java program is required to obtain information on its execution behavior. For performance monitoring it is important to keep the overhead introduced by instrumentation minimal. So, we only want to instrument for the behavior we are interested in. During the early modeling stages, when the performance analyst wishes to obtain a global view of the behavior, the overhead introduced by instrumentation is not a major issue. However, when the analyst needs to measure the performance of a certain part of the system it is important to keep the instrumentation overhead to a minimum, since the measurements need to be accurate. This means that we need different levels of instrumentation depending on the performance questions. Manually instrumenting the Java program for each performance question is too cumbersome and time consuming. Therefore we require some sort of automated instrumentation based on a description of the behavioral aspects the performance analyst is interested in.

Tools are required to analyze and visualize the monitoring results. Since performance questions may be domain specific it's important that custom tools can be developed to process the monitoring results. Hence, the monitoring results should be stored in an open data format. An application programming interface (API) to the monitoring data should be provided to make it easy to build custom tools.

3. ARCHITECTURE

Our architecture is based on event-driven monitoring. In general, two types of monitoring can be distinguished: time-driven monitoring and event-driven monitoring [12].

Time-driven monitoring observes the state of the monitored system at certain time intervals. This approach, also known as sampling, is often used to determine performance bottlenecks in software. For instance, by observing the call-stack every millisecond a list of methods using the most processing time can be obtained. Time-driven monitoring doesn't provide complete behavioral information, only snapshots.

Event-driven monitoring is a monitoring technique where events in the system are observed. An event represents a unit of behavior, e.g., the creation of a new thread. Our monitoring toolkit should implement the event-driven monitoring approach, since we require complete behavioral information, not just snapshots.

The following figure illustrates our architecture in terms of the main building blocks of our toolkit, and the way they are related (e.g., via input and output files).

First, the events of interest are specified in a configuration file. By using event filters, events can be included or excluded

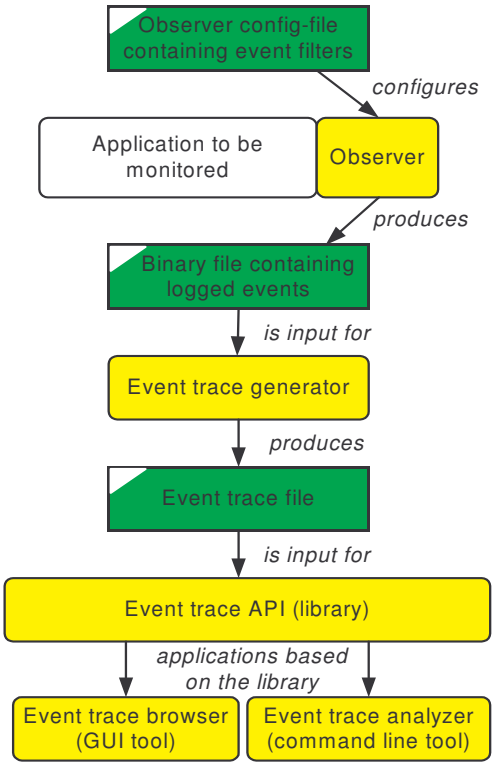

Figure 1: Monitoring architecture

from the set of interesting events. For example, certain methods of a given class may be interesting, while the rest should not be monitored.

During run-time the observer component collects the events of interest. Instrumentation is added to enable generation, observation, and logging of these events. The observed events are stored in a binary file.

After monitoring the binary file containing the collected events can be analyzed. An event trace generator produces event traces from the collection of events. One event trace is produced for each thread of control. An event trace represents events in a tree-like structure. Event traces are very similar to call-trees, but in addition to method invocations they also contain other event types.

The event traces can be accessed using our event trace API. This API is the main building block for tools that analyze or visualize event traces. The event trace API allows the performance analyst to build custom analysis tools.

The toolkit provides two applications based on the event trace API; an event trace browser and an event trace analyzer. The implementation of this architecture and the tools are discussed in the next section.

4. IMPLEMENTATION

4.1 The Java Virtual Machine Profiler Interface

The Java Virtual Machine Profiler Interface (JVMPI) plays an important role in our observer implementation. JVMPI allows a user provided *profiler agent* to observe events in the Java virtual machine. The profiler agent is a dynamically linked library written in C or C++. By subscribing to the events of interest, using JVMPI's event subscription interface, the profiler agent can collect profiling information on behalf of the monitoring tool. We use JVMPI and a profiler agent to implement the observer component in our architecture. Figure 2 depicts the implementation of the observer.

An important feature of JVMPI is its portability; its specification is independent of the virtual machine implementation. The same interface is available on each virtual machine implementation that supports the JVMPI specification. Furthermore, JVMPI does not require the virtual machine to be in debugging mode, it is enabled by default. The Java virtual machine implementations by Sun and IBM support JVMPI.

JVMPI supports both time-driven monitoring and event-driven monitoring. This section only discusses the functionality in JVMPI that is relevant for event-driven monitoring.

The profiler agent is notified of events through a callback interface. The following C++ fragment illustrates a profiler agent's event handler:

```
void NotifyEvent(JVMPI_EVENT *ev) {
  switch (ev->event_type) {
  case JVMPI_CLASS_LOAD:
    // Handle 'class load' event.
    break;
  case JVMPI_CLASS_UNLOAD:
    // Handle 'class unload' event.
    break;
  ..
  }
}
```

The JVMPI_EVENT structure includes the type of the event, the environment pointer (the address of the thread the event occurred in), and event specific data:

```
typedef struct {
  jint event_type;
  JNIEnv *env_id;
  union {
    struct {
    // Event specific data for 'class load'.
    } class_load;
    ..
  } u;
} JVMPI_EVENT;
```

JVMPI uses unique identifiers to refer to threads, classes, objects, and methods. Information on these identifies is obtained by subscribing to the *defining events*. For instance, the 'thread start' event, notifying the profiler agent of thread creation, *defines*

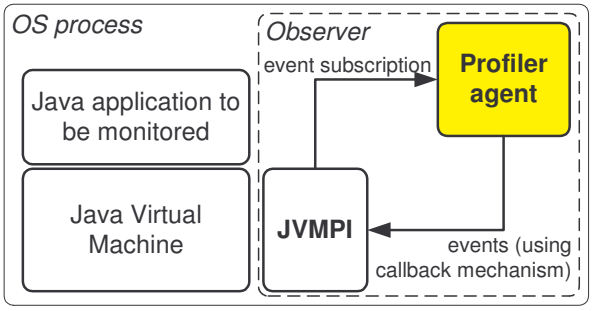

Figure 2: Implementation of the observer

the identifier of that thread and has attributes describing the thread (e.g., the name of the thread). The 'thread end' event *undefines* the identifier. For certain identifiers it is not required to be subscribed to their defining events to obtain information on the identifier. Instead, the defining events may be requested at a later time. For instance, defining events for object identifiers can be requested at any time using the `RequestEvent()` method of the JVMPI API.

JVMPI profiler agents have to be multithread aware, since JVMPI may generate events for multiple threads of control at the same time. Profiler agents can implement mutual exclusion on its internal data structures using JVMPI's raw monitors. These monitors are similar to Java monitors, but are not attached to a Java object.

The remainder of this subsection discusses the events that are supported by JVMPI.

JVM start and shutdown events. These events are triggered when the Java virtual machine starts and exits, respectively. These events can be used to initialize the profiler agent when the virtual machine is started and to release resources (e.g., close log file) when the virtual machine exits.

Class load and unload events. These events are triggered when the Java virtual machine loads a class file or unloads (removes) a class. The attributes of the class load event include the names and signatures of the methods it contains, the class and instance variables the class contains, etc. The class loading and unloading events are useful for building and maintaining state information in the profiler agent. For instance, when JVMPI informs the profiler agent of a method invocation it uses an internal identifier to indicate what method is being invoked. The class load event contains the information that is needed to map this identifier to the class that implements the method and the method signature.

Class ready for instrumentation. This event is triggered after loading a class file. It allows the profiler agent to instrument the class. The event attributes are a byte array that contains the byte-code implementing the class, and the length of the array. Using the Java virtual machine specification, profiler agents may interpret the byte array, and change (instrument) the implementation of the class and its methods. JVMPI doesn't provide interfaces to instrument class objects though. We use this event to instrument methods we want to monitor. This is described in the next subsection.

Thread start and exit. These events are triggered when the Java virtual machine spawns and deletes threads of control. The events attributes include the name of the thread, the name of the thread-group, and the name of the parent thread.

Method entry and exit. Method entry events are triggered when a method implementation is entered. Method exit events are triggered when the method exits. The time period between these events is the wall-clock completion time of the method.

Compiled method load and unload. These events are issued when just-in-time (JIT) compilation of a method occurs. Just-in-time compilation of a method compiles the (virtual machine) byte-code of the method into real (native) machine instructions. Suns hot-spot technology automatically detects often-used methods, and compiles them to native machine instructions automatically.

Monitor contented enter, entered, and exit. These events can be used to monitor the contention of Java monitors (due to mutual exclusion). The *monitor contented enter* event is issued when a thread attempts to enter a Java monitor that is owned by another thread. The *monitor contented entered* event is issued when the thread that waited for the monitor enters the monitor. The *monitor contented exit* event is issued when a thread leaves a monitor for which another thread is waiting.

Monitor wait and waited. The *monitor wait* event is triggered when a thread is about to wait on an object. The *monitor waited* event is triggered when the thread finishes waiting on the object. These events are triggered due to waiting on condition variables for the purpose of cooperation between different threads.

Garbage collection start and finish. These events are triggered before and after garbage collection by the virtual machine. These events can be used to measure the time spent on collecting garbage.

New arena and delete arena. These events are sent when arenas (areas of memory) for objects are created and deleted. (Currently not implemented in JVMPI)

Object allocation, free, and move. These are triggered when an object is created, released, or moved in the heap due to garbage collection.

4.2 Profiler agent

Our first profiler agent prototype logged events in a human readable (text) ASCII data format. Threads writing to the log file needed to obtain a global lock for the file. This lock is held while writing the event to the file.

Our second prototype replaced the text format with a binary format, which is significantly faster. The binary log file is memory-mapped into the profiler agent's address space. Different threads can write simultaneously to the memory map. Of course, threads should not log events in the same location of the memory map. A position in the memory map is assigned using a position counter. Incrementing the position counter in the memory map requires mutual exclusion. The position counter is increased with the size of the data that is to be written.

The combined use of a binary data format, memory-mapped I/O, and mutual exclusion to a position counter instead of the whole file, makes this solution much faster than using text files. There is a downside though; portability is sacrificed. Each operating system may implement different interfaces for memory-mapped files. For example, on POSIX `mmap(2)` [25] is used, while Microsoft Windows has `CreateFileMapping(Win32)`.

Initialization, configuration, and maintenance of internal tables. During its initialization the profiler agent reads a user specified configuration file, containing event filters. For example, the user may indicate which classes and methods are to be monitored, or whether locking contention is to be monitored. An example of a configuration file is depicted below:

```
output log/test5.bin
observe_monitor_contention
observe_object_allocation
bytecode_rewriting
observe_method_invocations
excludemethod * <init>
excludemethod * main
includemethod net.qahwah.jpmt.test.* *
excludemethod * *
```

After specifying the log file, the above configuration turns on observation of monitor contention, object allocation, and method invocations. Furthermore, it indicates that byte-code rewriting is to be used to instrument for method invocation monitoring. Finally, it specifies the methods to be observed, using include and exclude filters. These filters take two arguments: the class name and the method name. Wildcards are allowed in these filters.

When the virtual machine loads a class file the JVMPI informs the profiler agent about it using a 'class load' event. This event has the following attributes:

```
struct {
  char *class_name;
  char *source_name;
  jint num_interfaces;
  jint num_methods;
  JVMPI_Method *methods;
  jint num_static_fields;
  JVMPI_Field *statics;
  jint num_instance_fields;
  JVMPI_Field *instances;
  JobjectID class_id;
} class_load;
```

The profiler agent uses the attributes from the 'class load' event to build hash tables that map class and method identifiers to class and method information, respectively. The hash tables only keep information for classes and methods that are to be monitored.

A 'class unload' event causes the information related to the class to be removed from the hash tables.

The following code fragment depicts how the information on classes and methods are stored in hash tables.

```
class ClassInfo
{
public:
  jobjectID classID;
  char *name;
  bool dumped;
};

class MethodInfo
{
public:
  jmethodID methodID;
  ClassInfo *classInfo;
  char *name;
  char *signature;
  bool dumped;
};

typedef hash_map<
  jobjectID,
  ClassInfo *,
  hash<jobjectID> > ClassMap;

typedef hash_map<
  jmethodID,
  MethodInfo *,
  hash<jmethodID> > MethodMap;
```

Class and method information is written to the log file on demand. For instance, when a monitored method is invoked for the first time, information on that method is logged. The 'dumped' fields in the ClassInfo and MethodInfo classes indicate whether the information has been logged or not. The following class information is logged:

- The 'classID' field, which uniquely identifies a class.
- The 'name' field, which contains the name of the class.

The following information is logged for methods:

- The 'methodID' field, which uniquely identifies the method.
- The class identifier of the class the method is part of, obtained via the 'classInfo' reference to the object that holds the class information.
- The name of the method, which is stored in the 'name' field.
- The signature (return type and types of the parameters) of the method, which is stored in the 'signature' field.

Monitoring thread spawning and deletion. The profiler agent records the spawning and deletion of every thread of control, using the 'thread start' and 'thread exit' JVMPI events.

A thread is identified by its environment pointer (`thread_env_id`). The profiler agent maintains a hash mapping of this environment pointer to a structure that contains information on the thread.

When the thread exits the profiler agent removes the thread information from the hash map.

The following code fragment shows the attributes of the 'thread start' event, and how thread information is stored in a hash table.

```
struct {
  char *thread_name;
  char *group_name;
  char *parent_name;
  jobjectID thread_id;
  JNIEnv *thread_env_id;
} thread_start;

class ThreadInfo
{
public:
  char *name;
  char *groupName;
  char *parentName;
};

typedef hash_map<
  JNIEnv *,
  ThreadInfo *,
  hash<JNIEnv *> > ThreadMap;
```

Spawning and deletion of threads is recorded in the event log file. The following information is logged when a thread is spawned:

- The environment pointer of the thread.
- The object identifier of the thread, when the profiler agent is configured to monitor object allocations. The object identifier can be used to obtain type information of the thread object, e.g., the name of the class of the thread object.
- The name of the thread, the name of the parent thread,

and the name of the thread group this thread is part of.
- A timestamp, representing the time the event occurred.

When a thread is deleted the following information is logged:
- The environment pointer of the thread.
- A timestamp, representing the time the event occurred.

Monitoring method invocations using JVMPI method entry and exit events. JVMPI notifies the profiler agent of method invocations using the method entry and method exit events. Both JVMPI events have one event specific attribute, the method identifier.

```
struct {
  jmethodID method_id;
} method;
```

When the profiler agent is notified of a method entry it executes the following algorithm:

1. Obtain the thread local storage to store per-thread data. The profiler agent creates this thread local storage when it processes the first method entry in the thread.
2. If the method hash map doesn't contain information for the method identifier (meaning that the method is not to be monitored) then a counter in the thread local storage is increased and the event handler exits. This counter represents the number of un-logged method invocations in the thread of control. It can be used to correct the event trace for perturbation. For example, if a logged method calls 5 other methods that are filtered out (not logged), the CPU usage and completion time of that method includes the time for observing and filtering the un-logged method invocations. By keeping track of the number of un-logged invocations, we can subtract the costs of observing and filtering out these method invocations from the CPU usage and completion time of the logged method.
3. If the method hash map does contain information on the method identifier then information on the method and its class is logged if it hasn't been logged before. The un-logged method-invocation counter is reset and its old value is pushed on a stack in the thread local storage. Subsequently, the information on the method entry is logged, including timing related attributes, and the event handler exits.

A logged method entry contains the following attributes:
- The environment pointer of the thread that executes the method invocation.
- The identifier of the method.
- The value of the un-logged method invocations counter, before it has been reset.
- Information needed to determine the CPU usage of the method.
- A timestamp, representing the time the event occurred.

Unfortunately, each operating system implements interfaces to obtain CPU timing information differently. For example, POSIX (a UNIX standard) offers the `times(2)` call, Windows offers `GetThreadTimes(Win32)` and `GetCurrentThreadCPUTime(Win32)`, BSD UNICES and derivatives have `getrusage(2)`, and Sun Solaris has `gethrtime(2)` and `gethrvtime(2)`.

JVMPI offers `GetCurrentThreadCpuTime()`, which is supposed to return the CPU time in nano-seconds. However, on Linux it returns the same value as `gettimeofday(2)` does: the current wall-clock time in micro-seconds.

Often, CPU time information has a 10ms resolution, e.g. POSIX' `times(2)`, BSD's `getrusage(2)`, and Windows' `GetThreadTimes(Win32)` have a 10ms granularity. This is too coarse to be used as a performance measure for Java method invocations. This is caused by the frequency of the clock interrupt timer. Most operating systems are configured to generate 100 clock interrupts per second.

There is a solution for this problem: architecture specific hardware performance counters. All modern micro processors, including Intel's Pentium family, IBM/Motorola's PowerPC, and Compaq/DEC's Alpha, implement such performance counters.

These hardware counters don't have the granularity problem, but they still have an interfacing problem: there is no common interface to access these counters. Libraries such as PAPI [2] provide a common interface to these counters. The current implementation of the profiler agent doesn't use such a library, but implements similar functionality.

The profiler agent accesses the hardware performance counters using an operating system specific device driver. On Linux Mikael Pettersson's `perfctr` [20] package is used.

Besides propagating hardware performance counter values to user-land, these device drivers could also implement virtual performance counters for each process (thread). Virtual counters are only incremented when a process is executing, not when it is waiting in the operating system's process scheduler queue. So, in contrast to global counters, these counters provide precise timing information for the process.

On the Intel platform information that can be obtained using hardware performance counters includes the number of processor cycles since the processor was booted [10]. We use this counter to calculate the CPU usage of a method invocation. The information that can be obtained using hardware performance counters can be much more detailed, e.g. efficiency of the caches, and efficiency of branch prediction. For our purposes this is much too detailed.

The CPU usage of a method can be calculated by subtracting the number of processor cycles at method entry from the number of processor cycles at method exit.

Using byte-code rewriting to monitor method invocations. Processing method entry and exit notifications sent by the JVMPI for every method invocation introduces a lot of overhead. At minimum, two hash table lookups are required (to see whether or not to log the method entry and exit). To reduce the overhead introduced due to monitoring method invocations we have also implemented another monitoring approach: instrumentation of the byte-code of method implementations. This instrumentation mechanism only inserts instrumentation in methods that we want to monitor. This is different from JVMPI's

method entry and exit events, which are triggered for every method invocation, similar to the interceptor design pattern [4].

JVMPI provides the 'class ready for instrumentation') event, which can be used to insert instrumentation in Java classes. However, no user-friendly interface is provided to change the implementation of the classes. For Java, libraries are available that provide the user with APIs for rewriting the implementation of classes, such as the Byte Code Engineering Library (BCEL) [3]. Unfortunately, we cannot use these libraries since the profiler agent has to be implemented in C or C++. Rewriting classes before run time is not an option, since that solution is in conflict with the requirement that the insertion of instrumentation should be transparent to the user, and be done at run-time.

We choose to implement the byte-code instrumentation ourselves, in the profiler agent. The 'class ready for instrumentation' event provides us with a byte array that contains the compiled class object. The format of the class object is described in the Java Virtual Machine Specification [16]. The following pseudo-code fragment describes the structure of class objects. Data types represented are: `u2` and `u4`, which are 2-byte and 4-byte unsigned numbers; `cp_info`, `field_info`, `method_info`, and `attribute_info`, which are structures that describe constant pool entries (e.g., names of methods), variables, methods, and attributes, respectively.

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

The byte-code rewriter takes the original byte-code array, rewrites it, and returns the rewritten byte-code array to the virtual machine. A full description of the rewriting process is beyond the scope of this paper. Instead we describe shortly the rewriting algorithm (the rewriting algorithm is only executed if the class contains methods that we should monitor):

- All bytes of the class are copied to a newly allocated byte array that keeps the instrumented class.
- The constant pool of the class is parsed and several administration tables are built, containing information about constants. The constant pool contains, for instance, references and names of classes the class refers to, and references and names of methods it calls.
- New constant pool entries for the class and methods that implement the instrumentation are added to the instrumented class.
- Subsequently, the list of methods is iterated. If a method is to be monitored, instrumentation is inserted at the method entry point and before every method exit. The instrumentation contains byte-code that notifies the profiler agent of method entry and exit. The insertion of byte-code instructions renders branching offsets, and position counters of exception handlers invalid. The rewriting algorithm fixes these relative and absolute addresses. In addition, the maximum stack size may need to be updated, since the instrumentation contains method invocations of the profiler agents method entry and exit handlers. Method invocations use the stack to store parameters.

Java source:

```
public static int test2b(int i) {
  if (i == 2) return test2a(i+1);
  else return test2a(i);
}
```

Byte-code before instrumentation:

```
PC   OPCODE            OPERANDS
0x0  0x1a iload_0
0x1  0x05 iconst_2
0x2  0xa0 if_icmpne    0x0 0xa
0x5  0x1a iload_0
0x6  0x04 iconst_1
0x7  0x60 iadd
0x8  0xb8 invokestatic 0x0 0x2
0xb  0xac ireturn
0xc  0x1a iload_0
0xd  0xb8 invokestatic 0x0 0x2
0x10 0xac ireturn
```

Byte-code after instrumentation:

```
PC   OPCODE            OPERANDS
0x0  0x10 bipush       0x01
0x2  0xb8 invokestatic 0x0 0x1e
0x5  0x1a iload_0
0x6  0x05 iconst_2
0x7  0xa0 if_icmpne    0x0 0xf
0xa  0x1a iload_0
0xb  0x04 iconst_1
0xc  0x60 iadd
0xd  0xb8 invokestatic 0x0 0x2
0x10 0x10 bipush       0x01
0x12 0xb8 invokestatic 0x0 0x1f
0x15 0xac ireturn
0x16 0x1a iload_0
0x17 0xb8 invokestatic 0x0 0x2
0x1a 0x10 bipush       0x01
0x1c 0xb8 invokestatic 0x0 0x1f
0x1f 0xac ireturn
```

Figure 3: Java byte-code rewriting illustration

Figure 3 illustrates the rewriting algorithm. It shows how the method body of a simple method is rewritten. The instrumentation is printed in bold face.

Monitoring contention during mutual exclusion. The JVMPI 'monitor contented enter', 'entered', and 'exit' events are logged with timestamps, the environment pointer of the thread, and the object-id of the Java object that is associated with the monitor.

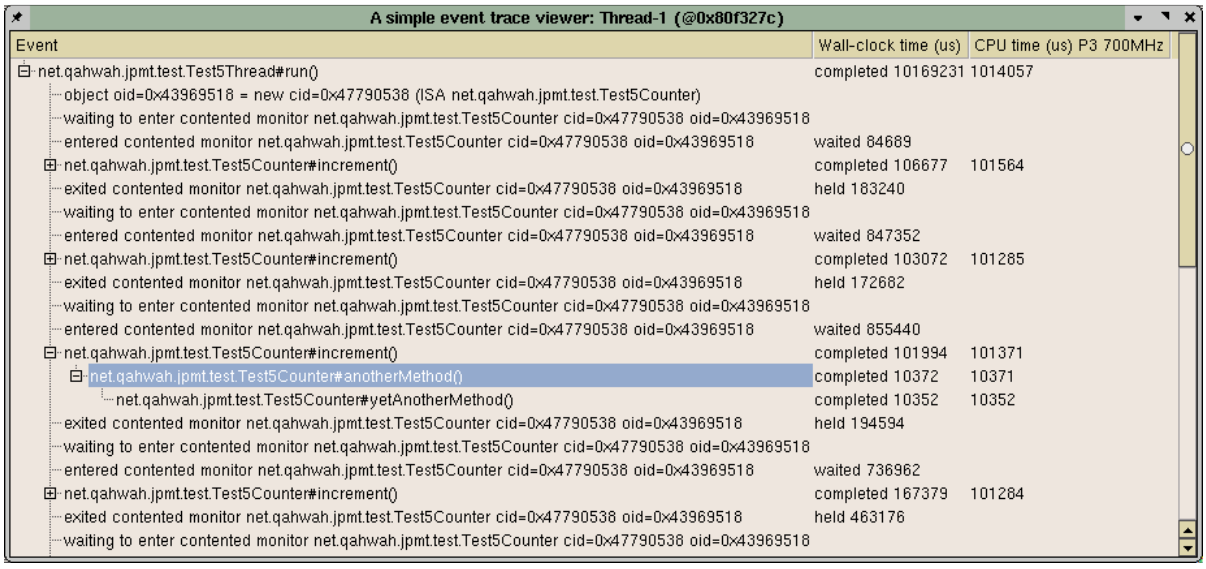

Figure 4: Screenshot of the event trace browser, displaying part of a trace of a particular thread

Monitoring cooperation. The JVMPI 'monitor wait' and 'waited' events are logged with timestamps, the environment pointer of the thread, and the object-id of the Java object that is associated with the monitor.

Object allocation. In case information on Java objects is needed, for instance when there is contention for a monitor, the observer queries JVMPI for object information. This information is stored in the observer's internal data structures. To keep this information valid the observer also needs to monitor 'object free' events, and 'object move' events (an object may be moved to another address during garbage collection). The object information is also logged, so that the analyzer and visualizer tools can display object information of objects associated to monitor contention and cooperation events.

4.3 Tools

The observer produces a binary log file containing the collected events. From this event collection, event traces can be generated. The toolkit provides an event trace generator that produces and event trace for each program thread. Tools are needed to visualize and analyze these event traces.

JPMT provides an event trace API that can be used to implement event trace processing tools. The API provides an object-oriented view on the event traces. C++ classes represent the various event types; events are instances of those classes (objects). Event attributes can be queried by invoking methods on the event objects.

We have implemented two tools on top of this API: (i) an event trace browser (or visualizer), and (ii) a command-line tool for analyzing event traces.

The event trace browser provides a simple graphical user interface for browsing the event trace. Events, including performance attributes, are displayed on the screen in a tree-like structure. The user is able to expand and collapse sub traces, depending on the level of detail the user requires. Figure 4 shows a snapshot of the event browser. The figure contains part of an event trace of one of the threads in an application we used to test the toolkit. In this application various threads increment a shared counter repeatedly. After incrementing the counter they perform some CPU intensive operations. Since the counter is a shared object, we use mutual exclusion. The screenshot shows the monitored events for mutual exclusion (contended monitors) and the invocation of the increment method on the counter object. The events are annotated with performance measures, such as the duration of monitor contention and CPU time usage of methods.

The command-line tool can be used in scripts that post-process the event traces for specific experiments, for example, to obtain input data for GNUplot (a tool to generate plots). The command-line tool can also be used to export the event traces to a human readable text format.

We plan to add language bindings for the Java and Ruby programming languages to the API, such that performance analysis applications can be written in those languages too.

4.4 Overhead

The overhead of monitoring depends on the amount of instrumentation. In section 2 we identified two distinct levels of monitoring: (i) exploring the behavior of the monitored software and (ii) measuring performance of parts of the software to answer specific performance questions. For exploring behavior it doesn't really matter how much overhead is introduced. For performance measurement however, we like to minimize the overhead, since it disrupts the measurements. We can do this by filtering out events we're not interested in. Monitoring of method invocations causes the most overhead. On a Pentium III 700 MHz system the overhead for monitoring and logging a method invocation is 6 microseconds when the byte-code rewriting algorithm is used to instrument methods. This can be split evenly in 3us for monitoring the method entry event, and 3us for monitoring the method exit event. Logging itself costs 1us for each event. The other 2us per event are caused by the invocation of our method invocation event handler and obtaining timestamps and CPU performance counters. To compare, a Java method that does a print-line on the screen (`System.out.println()`), costs

16us. We provide a small utility to measure overhead caused by the instrumentation on other platforms.

5. EXAMPLE: CORBA PERFORMANCE MEASURES

This section illustrates the use of our Java Performance Monitoring Toolkit to obtain performance measures of ORBacus/Java [11], a CORBA implementation [19].

5.1 OMG CORBA Distributed Object Middleware

The Common Object Request Broker Architecture (CORBA) [19] specified by the Object Management Group (OMG) is the de-facto object-middleware standard. CORBA mediates between application objects that may live on different machines, by implementing an object-oriented RPC mechanism. This allows application objects to talk to remote objects in the same way as they talk to local objects. Besides this object location transparency, CORBA also implements programming language transparency. Object interfaces are specified in an interface description language (IDL). These IDL interfaces are compiled to so-called stubs and skeletons, which act as proxies for the client and server objects, respectively. The client and server objects may be implemented in different programming languages, for instance Java and C++. Figure 5 depicts the CORBA method invocation path. The figure shows a two-way (request and reply) remote method invocation.

Activities that make up the functional path of a CORBA remote method invocation. First, the client invokes the method on the stub, which is the local proxy of the target object. A reference to the stub is obtained by narrowing the object reference of the target object.

The stub constructs a CORBA request object and translates the method invocation parameters, which are expressed using programming language, operating system, and architecture specific data types, to a common data representation (CDR). This translation process is called marshaling. The marshaled data is added to the request object. Subsequently, the request object is forwarded to the client-side ORB library.

The client-side ORB library uses a TCP/IP connection to communicate with the server-side ORB library. The address of the server-side ORB is obtained from the target object's *object reference*. This object reference is created by a portable object adapter (POA) on the server-side ORB. Each object is managed by exactly one POA.

A POA implements the adapter design pattern [4] to adapt the programming language specific object interfaces to CORBA interfaces, making the target object implementation accessible from the ORB.

The POA has a map of active objects. This map associates object identifiers with object implementations. Object implementations are called *servants*. The object reference contains server information, such as hostname and port number, the name of the POA, and the object identifier of the target object.

The client-side ORB sends the request object to the server-side ORB.

The server-ORB obtains the target object's POA and object identifier from the request object and forwards the request to the POA.

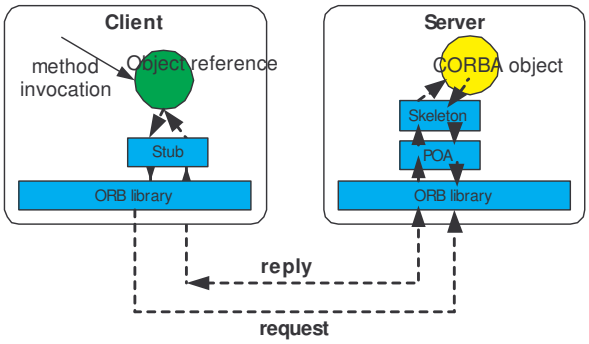

Figure 5: CORBA method invocation path

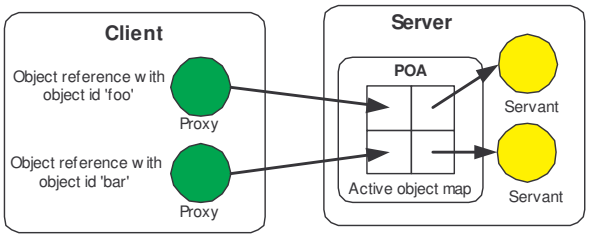

Figure 6: From object reference to servant

The POA looks up the servant in it's active object map using the object identifier and forwards the request to the skeleton of the target object. Figure 6 illustrates the mapping from object-id to servant.

The skeleton unmarshals the method invocation parameters and invokes the request on the target object.

When the method invocation returns, the skeleton creates a CORBA reply object, marshals the return parameters, and inserts the return parameters in the reply object. The reply object is forwarded to the server-side ORB.

The server-side ORB forwards the reply object to the client-side ORB.

The client-side ORB forwards the reply object to the stub.

The stub unmarshals the return parameters and forwards those to the client.

This description is a bit simplified. However, a full description of what happens during a method invocation is outside the scope of this paper. More information is available from [19] [7].

ORBacus/Java using thread pools. ORBacus/Java supports various server-side threading models, such as thread-per-request and thread-pool. In this use case we look at the thread-pool model.

In the thread-pool model the server-side activities described above are distributed over receiver and dispatcher threads. Each client connection has its own receiver thread. The requests are dispatched by receiver threads onto a pool of pre-allocated threads. Requests are queued in the thread pool in a FIFO queue. Idle dispatcher threads take a request from the queue and process the request. The length of the queue is only limited by the available memory. The number of threads in the thread pool is fixed. The maximum number of requests that can be processed simultaneously equals the size of the thread pool. A discussion

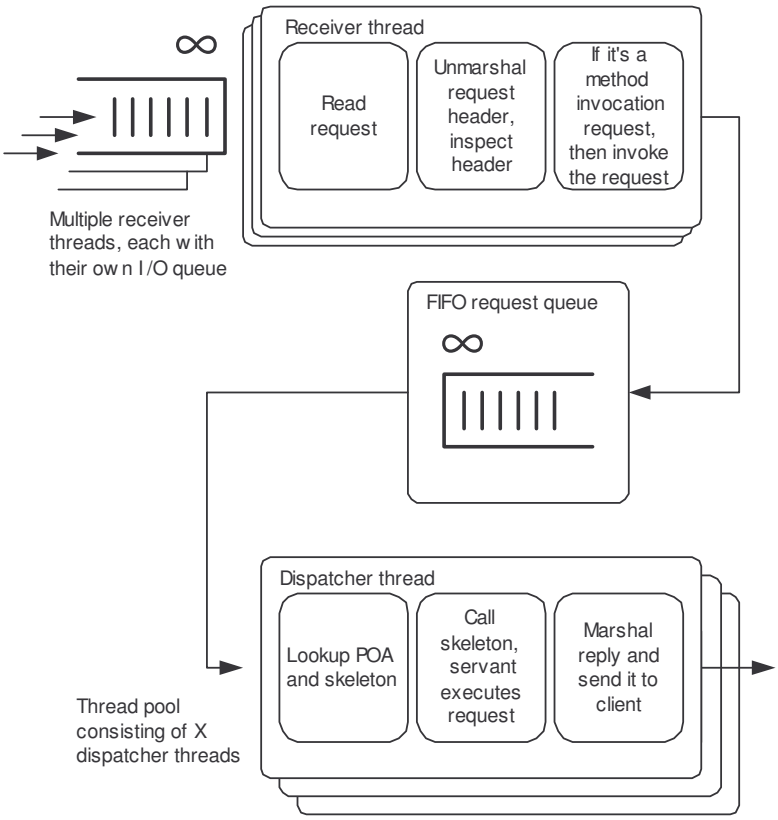

Figure 7: Server side request flow in ORBacus with thread-pool model

on the advantages and disadvantages of various threading models can be found in [22].

The receiver thread receives the request object from the network, gets the POA and object identifier from the request object, constructs a dispatch-request object, and forwards that object to the thread pool. The receiver thread can then process the next request (if any).

The dispatch-request is processed by a dispatcher thread, which forwards the request to the POA, skeleton, and servant. After the request is processed by the servant a reply object is constructed and sent to the client-side ORB. The dispatcher thread is then ready to process the next dispatch-request object.

Figure 7 illustrates the execution path at the server-side of a method invocation in ORBacus, using the thread pool threading model.

5.2 Blackbox performance measurements

To measure end-to-end performance of CORBA under different configurations we have developed a set of benchmarking applications. The set consists of a source and sink application. The source application implements a synthetic workload according to a given scenario. The sink application processes the workload generated by the source application. The interaction between the source and sink applications is illustrated in figure 8.

A scenario consists of three parts:

- The configuration of the client and server ORBacus. For instance, the threading model of the server is described here.
- A description of the workload the source should generate. It describes a number of arrival processes, their distribution, and the routing probabilities to the different target objects (see figure 8).
- A deployment description of the sink application, which describes a number of POAs and objects that are managed by the POAs. It also describes POA policies, such as the POA threading policy. This threading policy is different from the ORB threading policy. The CORBA specification specifies two POA threading policies: ORB controlled and single threaded. If the ORB controlled threading policy is used, the ORB may implement any threading model. If the single thread policy is used, then the POA can only process one request at a time. This POA threading policy can be used when the servants are not multi-thread aware.

The source and sink applications have been developed to make it easier to perform different experiments. Without the source and sink applications and their scenario configuration files, we would have to implement the scenario inside client and server applications. Having to change these client and server applications for different experiments is too tedious.

The source and sink applications report on end-to-end performance measures, such as the response time of each request. These measures are *blackbox* measures, meaning that we don't know how the end-to-end measures are distributed over the activities that make up the functional path of a method invocation

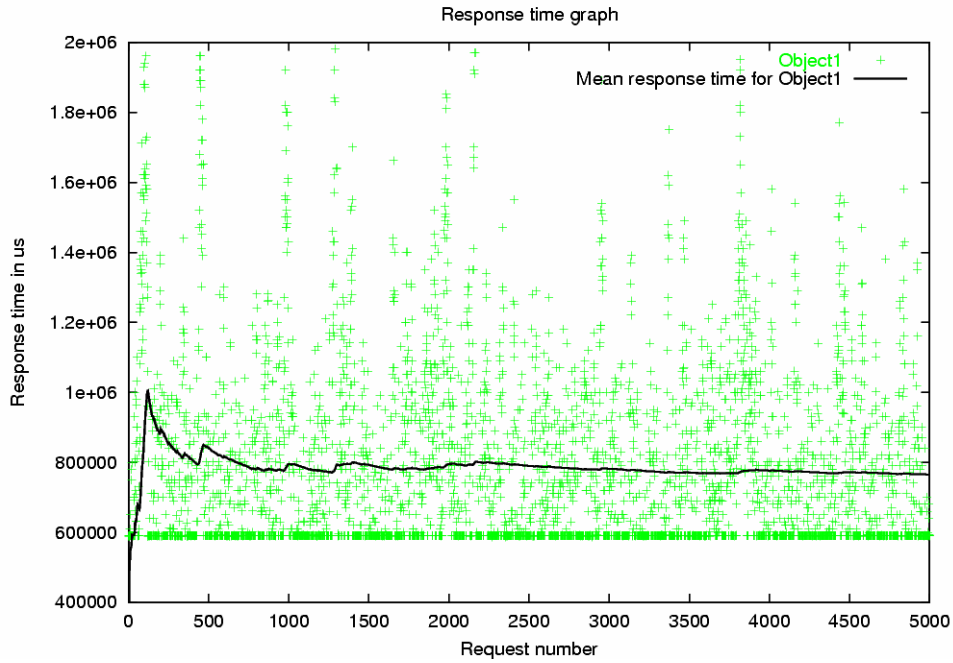

Figure 9: Response time graph for our example scenario

at the server-side ORB. The next paragraph discusses how *whitebox* measures are obtained. The remainder of this paragraph discusses our example scenario.

We use the following scenario:

- The workload consists of 5000 requests.
- All requests are sent to the same target object. The target method's name is `ping_with_wait()`. The name of the Java servant class implementing the object is `net.qahwah.jpmt.examples.ORBacus.PerformanceTest1_impl`.
- There is one arrival process, generating 4 requests per second for the target object. This arrival process is exponentially distributed.
- On the server side there is a multithreaded POA, with a thread-pool of 3 threads.
- The `ping_with_wait()` operation in the servant has a constant waiting time (no CPU usage) of 1/1.75 seconds per request.

We have performed the experiment on a uniprocessor Pentium III 700MHz machine with 320MB RAM, running the Linux 2.4 operating system and ORBacus 4.1.0b1. Both the source and sink application run on the same machine. There was no background load.

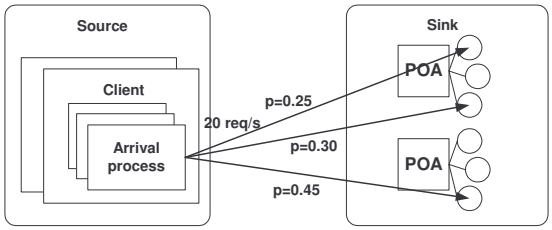

Figure 8: Source generates synthetic workload for

The mean end-to-end response time of a request as measured is 0.7649234997 seconds (with a variance of 0.07222361847 seconds).

Figure 9 shows a plot of the response times, and the mean response time. The variance in the response times is caused by the exponential distribution of the request arrival process.

5.3 Whitebox performance measures

We use JPMT to obtain whitebox measurements. Figure 10 and 11 show monitoring results for request 87 (out of 5000). First, the request is read from the network by the receiver thread (using `receive_detect()`) in two steps: one for the request header, and one for the request body. The `executeHeader()` method determines the request type from the header. The `execute()` method locates the target POA and creates a request object (a Java object representing the incoming request). The request is put in the thread-pool by the `invoke()` method. Both `executeHeader()` and `execute()` unmarshal fields from the request header and request body. Not the whole request is unmarshaled; method invocation parameters are unmarshaled by the skeleton in the dispatcher thread. After adding the request object to the thread-pool (using `add()`), the receiver thread is ready to process the next incoming request (highlighted `receive_detect()` in figure 10). The thread-pool has a FIFO queue in which requests are stored. When a dispatcher thread is ready to process a request it gets the request from the thread-pool using the `get()` method. The request is then invoked. First, the object implementation (servant) is located in the POAs active object map (using `locate()`). Subsequently, the request is dispatched to the servant, via the target object's skeleton (`dispatch()`). The skeleton unmarshals the method invocation parameters (not shown in the trace) and calls the target method in the servant (`ping_with_wait()`). When the

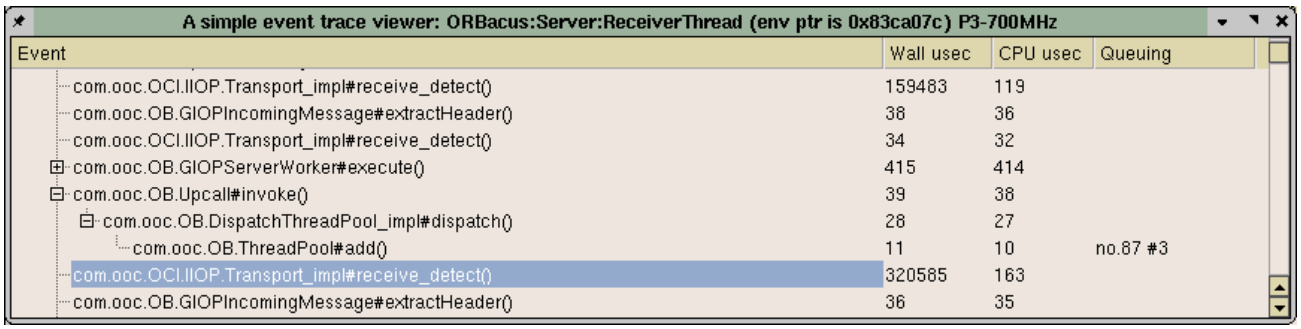

Figure 10: Activities in receiver thread for request 87

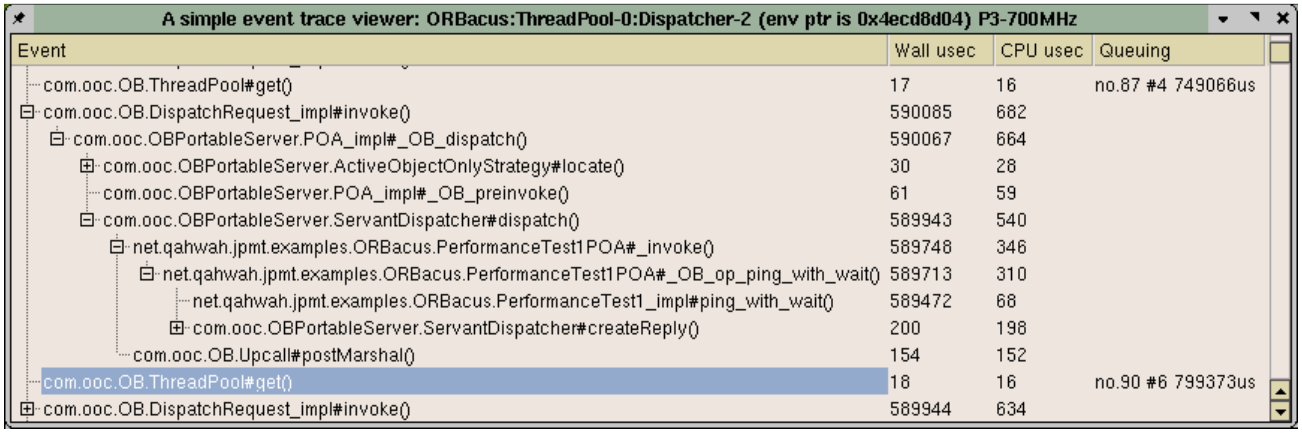

Figure 11: Activities in dispatcher thread for request 87

servant is done, a reply object is created (`createReply()`), and return parameters are marshaled (not shown). The request is sent to the client in the `postMarshal()` method. The dispatcher thread is then ready to process the following request (highlighted `get()` in figure 11).

For each monitored method invocation in the receiver thread and dispatcher thread, we have measured the wall-clock completion time and the CPU time, both in microseconds. Note that the completion time and CPU time of a method invocation includes the completion and CPU time of its children (i.e., the methods it calls). The servant's `ping_with_wait()` method doesn't use a lot of CPU time (68 microseconds) compared to the wall-clock completion time.

We've added a custom extension to the event trace browser and analyzer tools to show queuing information. In the 'queuing' column the request number is shown (87), the size of the queue after performing the `add()` and `get()` method is shown (3 and 4, respectively), and in the dispatcher thread it is shown how long a request has been queued (request 87 spent 749066 microseconds in the queue). The queuing information is obtained by replaying actions in the event trace related to the thread-pool, i.e., we track the `add()` and `get()` methods of the thread-pool.

The mean queuing time we measured is 0.1739308757 seconds (variance is 0.07204043565 seconds). In our scenario we have configured the `ping_with_wait()` method to use 1/1.75 seconds (0.5714285714 seconds). However, the event trace (figure 11) shows that 0.589472 seconds are used. The difference is caused by the coarse granularity of the Java `Thread.sleep()` method, which is 10ms.

The waiting time in the servant and the queuing time in the thread-pool's FIFO queue are the main contributors to the end-to-end response time. They account for 0.74535945 seconds of the 0.7649234997 seconds. The remaining 0.0195640526 seconds can be attributed to the client ORB processing, loop-back networking between the client and server ORBs, and server ORB processing. The performance monitoring toolkit can be used to precisely quantify these attributes too.

6. RELATED WORK

Performance monitoring requires some kind of instrumentation to measure and trace execution behavior. Various instrumentation techniques are available. In this section we consider manual and automated instrumentation of the source code, automated instrumentation of Java byte-code, and the use of profiler and debugging APIs offered by the Java virtual machine.

The most obvious technique is manual instrumentation of Java source code, by inserting logging and measurement statements in the source code. Tracing and logging libraries, such as Visible Workings' Trace for Java [27] and Apache Log4J [1], make source code instrumentation easier by providing some common tracing and logging functionality. To trace method invocations, the source code of each method to be traced has to be modified, which is cumbersome. Dynamic proxies, supported by Java's reflection package, can be used to selectively instrument certain objects by wrapping the object in another object using the

proxy design pattern. [4] This eliminates the need of having to instrument each method.

Because instrumenting source code manually is tedious, some sort of automation would be preferable. For instance, AspectJ [13], which is an aspect-oriented programming extension to Java, considers tracing as a crosscutting concern, or aspect [14]. The AspectJ compiler can automatically insert instrumentation to facilitate tracing of method invocations.

Another instrumentation technique is byte-code rewriting, which is discussed in section 4. The JMonde Trace tool [17] supports method tracing by rewriting the byte-code of a class when the class is loaded. A custom Java class loader reads the class and uses the BCEL library [3] to rewrite the byte-code of the class, e.g., by inserting logging statements in method implementations. The JMonde Trace tool supports filtering of traces by allowing the user to select classes that are to be included or excluded in the trace. The tool doesn't support performance measurement, and thread monitoring.

Recent versions of Java include the Java Virtual Machine Debugger Interface (JVMDI) [23] and the Java Virtual Machine Profiler Interface (JVMPI) [15]. Both interfaces can be used to implement tracing tools. Like JVMPI (discussed in the previous section), the debugger interface notifies clients through events. Compared to the profiler interface JVMDI provides more context information, such as information on the contents of local variables and method parameters. The disadvantage of JVMDI compared to JVMPI is that it requires the virtual machine to run in debug mode, which causes a performance penalty.

Profiler tools, such as IBM Jinsight [8], Compuware NuMega DevPartner TrueTime [18], Sitraka Jprobe [24], OptimizeIt [26] and Intel VTune [9], obtain profiling data from the virtual machine using JVMPI. Most profilers do not provide complete traces of events that have occurred in the virtual machine; they employ call stack sampling 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.

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 [21] and IBM Jinsight allow exporting of data to, e.g., spreadsheets or scripts.

JPMT also supports both online and offline filtering of interesting events (i.e., at run-time and during event trace processing by analyzer and visualization tools). Most profilers only support filtering after program execution. An exception is Rational Quantify, which allows the user to indicate which data is to be collected and reported, and the level of detail of the data.

7. FUTURE WORK

Our toolkit is under active development and is to be released under an open source license. We're currently adding techniques to quantify the performance of object creation and destruction, and garbage collection. Various improvements are planned for the event trace visualizer, including replaying (animation) of event traces and recognition of execution patterns. Besides improving the usability of the visualizer, we also plan to add features that aid the user with event filter set specification. The specification of these filter sets is currently a manual process. Usually, it takes a few iterations to find a good filter set for measuring the performance aspects the user is interested in. Another useful extension we're planning is the ability to detect and display differences between two event traces. For example, this can be useful when the two event traces are generated for the same experiment, but using different deployment configurations. In this case the differences list the impact of the deployment configuration changes.

8. CONCLUSIONS

In this paper, we have introduced our Java Performance Monitoring Toolkit (JPMT). This toolkit provides insight in the execution behavior of Java programs. JPMT implements event-driven monitoring, i.e., execution behavior is expressed as a series of events. An event represents an individual unit of behavior, such as the creation of a new thread, entering a method, and exiting a method. Events are annotated by performance attributes. For example, the 'method entry' and 'exit' events are annotated with a timestamp (wall-clock time) and the contents of certain CPU registers (called hardware performance counters). These attributes can be used to calculate the wall-clock completion time of a method invocation and its CPU usage.

JPMT allows the user to indicate the events of interest. To monitor these events of interest, instrumentation is added to the Java program. JPMT adds this instrumentation transparently to the user, and during run-time. Instrumentation doesn't require availability of the source code. The instrumentation logs events to a binary formatted file.

From this file, event traces can be produced. An event trace represents the execution behavior as a tree of events; each event may have child events stored in sub-trees. For example, method invocations are child events of the calling method.

Event traces can be analyzed using an event trace API. Tools can be built on top of this API to process the event traces. Two tools are provided: a GUI to browse event traces and a command-line tool to perform event trace analysis.

We have developed the toolkit to gain insight in the execution behavior of Java applications for which we're developing performance models. We found that existing tools didn't offer the functionality we required. Profiler tools focus on performance tuning (finding execution hot-spots) and provide an incomplete view of the execution behavior (no complete event traces). Compared to related tools, JPMT produces complete event traces, offers event filtering during and after execution, and allows custom event trace analysis and visualization tools to be developed. Instrumentation overhead is minimized by only adding instrumentation for events that are to be monitored and by careful optimization of the instrumentation itself.

9. ACKNOWLEDGMENTS

10. REFERENCES

[1] Apache Software Foundation, *Log4J*, http://jakarta.apache.org/log4j/, 2001.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, *A Portable Programming Interface for Performance Evaluation on Modern Processors*, The International Journal of High Performance Computing Applications 14, 3:189-204 (Fall), 2000.

[3] M. Dahm, *The Byte Code Engineering Library*, 2001. http://bcel.sf.net/

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

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

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

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

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

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

[10] Intel Corporation, *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 1997.

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

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

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, *An Overview of AspectJ*, 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, *Aspect-Oriented Programming*, 11th European Conference on Object-Oriented Programming (ECOOP), 1997.

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

[16] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, 2nd ed., Sun Microsystems, 1999.

[17] M. McPhail, *JMonde Trace tool*, 2001. (available from http://jmonde.org/Trace)

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

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

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

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

[22] D. Schmidt, Evaluating Architectures for Multi-threaded CORBA Object Request Brokers, Comm. ACM 41, 10, Special Issue on CORBA, 1998.

[23] Sun Microsystems, *Java Platform Debugger Architecture*, http://java.sun.com/j2se/1.3/docs/guide/jpda/index.html, 1999.

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

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

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

[27] Visible Workings, *Trace for Java*, http://www.visibleworkings.com/trace/, 2001.

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