Java Virtual Machine Profiler Interface

by D. Viswanathan
S. Liang

We present the Java™ Virtual Machine Profiler Interface (JVMPI), which defines a general-purpose and portable mechanism for obtaining comprehensive profiling data from the Java virtual machine. We show that it is extensible, nonintrusive, and powerful enough to suit the needs of different profilers and virtual machine implementations. With the JVMPI, most profiler vendors will not need to build custom instrumentation in the Java virtual machine. In addition, we solve challenges to profiler design and implementation posed by the multithreading and garbage collection support provided by the Java virtual machine. Profilers based on the JVMPI can produce thread-aware CPU time profiles, uncover heavy memory allocation sites, detect unnecessary object retention, pinpoint scalability problems caused by high monitor contention, reveal thread deadlocks, and perform interactive profiling with minimum overhead. We also describe HPROF, a profiler based on JVMPI, developed by us to demonstrate the power of JVMPI.

Profiling[1] is an important step in software development. We use the term *profiling* to mean, in a broad sense, the ability to monitor and trace events that occur during run time, the ability to track the cost of these events, and the ability to attribute the cost of the events to specific parts of the program. For example, a profiler may provide information about what portion of the program consumes the most amount of CPU time, or about what portion of the program allocates the greatest amount of memory.

This paper is mainly concerned with profilers that provide information to programmers, as opposed to profilers that provide feedback to the compiler or run-time system. Although the fundamental principles of profiling are the same, there are different requirements in designing these two kinds of profilers. For example, a profiler that sends feedback to the run-time system must incur as little overhead as possible so that it does not slow down program execution. In contrast, a profiler that constructs the complete call graph may be permitted to slow down the program execution significantly.

In this paper, we discuss techniques for profiling support in the Java** virtual machine.[2] Java applications are written in the Java programming language[3] and compiled into machine-independent binary class files, which can then be executed on any implementation of the Java virtual machine. The Java virtual machine is a multithreaded and garbage-collected execution environment that generates various events of interest for the profiler. For example:

- The profiler may measure the amount of CPU time consumed by a given method in a given class. In order to pinpoint the exact cause of inefficiency, the profiler may need to isolate the total CPU time of a method A.f called from another method B.g, and ignore all other calls to A.f. Similarly, the profiler may only want to measure the cost of executing a method in a particular thread.
- The profiler may inform the programmer why there is excessive creation of object instances that belong to a given class. The programmer may want to know, for example, that many instances of class

©**Copyright** 2000 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8670/00/$5.00 © 2000 IBM

D are allocated in method C.h. More specifically, it is also useful to know that the majority of these allocations occur when B.g calls C.h, *and* only when A.f calls B.g.
- The profiler may show why garbage is not being collected for a certain object. The programmer may want to know, for example, that garbage collection is not performed for an instance of class C because it is referred to by an instance of class D, which is then referred to by a local variable in an active stack frame of method B.g.
- The profiler may identify the monitors that are contended by multiple threads. It is useful to know, for example, that two threads, T_1 and T_2, repeatedly contend to enter the monitor associated with an instance of class C.
- The profiler may inform the programmer what causes a given class to be loaded. Class loading not only takes time, but also consumes memory resources in the Java virtual machine. By knowing the exact reason why a class is loaded, the programmer can optimize the code to reduce memory usage.

We present the Java Virtual Machine Profiler Interface (JVMPI) in this paper. Using this interface, profilers can obtain profiling data from the Java virtual machine. The contribution of this paper is to describe the design of the JVMPI and to justify its design choices. Following are the significant properties of the JVMPI:

- Comprehensive—Profilers based on the JVMPI can produce thread-aware CPU time profiles, uncover heavy memory allocation sites, detect unnecessary object retention, pinpoint scalability problems caused by high monitor contention, and reveal thread deadlocks. The JVMPI enables profilers to derive the above types of information by providing a mechanism to trace a key set of relevant Java virtual machine run-time events, assign costs to them, and attribute the costs to specific execution contexts.
- General-purpose—Profilers need not rely on custom instrumentation in the virtual machine. The JVMPI is efficient and powerful enough to suit the needs of different profilers and Java virtual machine implementations. First, it supports a variety of profiling techniques. For example, both code instrumentation and statistical sampling are supported. Second, it supports interactive profiling with minimum overhead. Profilers may accept interactive input from users and perform selective profiling, or they may simply write the profile data to a disk file.
- Portable—The JVMPI is designed to be completely independent of the underlying Java virtual machine implementation. For example, the heap profiling interface does not rely on the allocation and garbage collection algorithm used by the virtual machine.
- Nonintrusive—When profiling is disabled, the Java virtual machine incurs only a test and branch overhead for each event traced by the JVMPI. Most events occur in code paths that can tolerate the overhead of an added check. As a result, the Java virtual machine can be deployed with profiling support in place. Profiling measurement can be performed with minimal discrepancy between the profiling environment and the actual run-time environment.
- Extensible—The JVMPI can easily be extended to keep up with Java virtual machine evolution. It is easy to add new virtual machine events to the existing framework.

We have implemented the JVMPI in the Java 2 SDK, Standard Edition, version 1.2.[4] To demonstrate the comprehensive profiling support provided by the JVMPI, we have developed a profiler called HPROF based on the JVMPI. Numerous tool vendors have already built profilers that rely on the JVMPI. Commercially available examples include OptimizeIt!** 3.0 from Intuitive Systems,[5] JProbe** from KL Group,[6] and TrueTime** and TrueCoverage** from Compuware NuMega.[7]

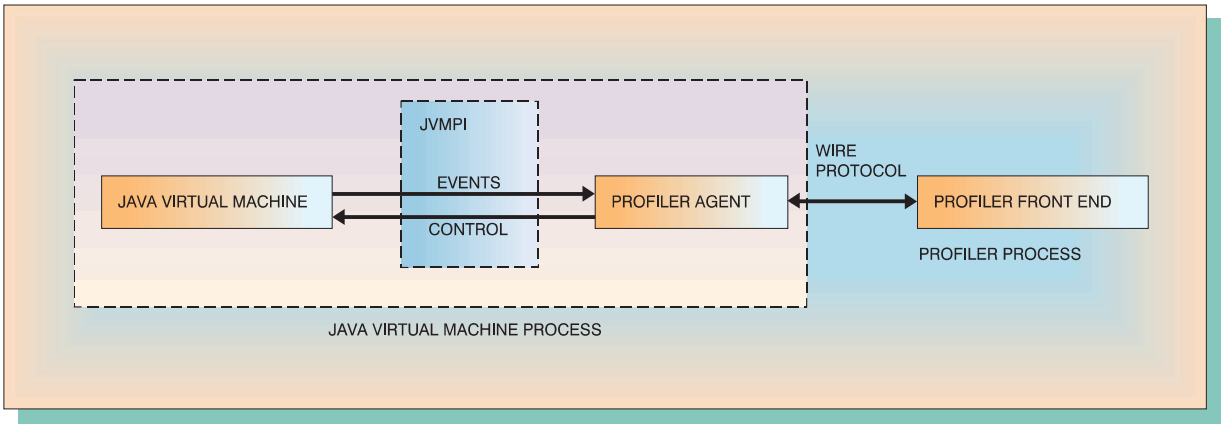
We begin by describing the design of the JVMPI and giving an overview of the HPROF profiler. We then justify our design choices for the JVMPI. We assume the reader is familiar with the basic concepts of the Java programming language[3] and the Java virtual machine.[2]

JVMPI design

Why design a profiling interface? A profiling interface, as opposed to direct profiling support in the virtual machine implementation, offers the following advantages:

Different profilers may be used with the same virtual machine, allowing various presentations of the profiling information. For example, one profiler may simply record profile data in a trace file, whereas another may process the profile data and present the data interactively through a user interface. Similarly,

Figure 1 Profiler architecture



the same profiler can work with different virtual machine implementations, as long as they all support the same profiling interface. This flexibility allows tool vendors and virtual machine vendors to leverage one another's products effectively.

Profilers need not rely on custom instrumentation in the virtual machine. Therefore, users need not have a copy of a virtual machine for each profiling tool in use. In addition, profiling tool vendors need not maintain virtual machine source code on their target platforms.

A profiling interface, although providing flexibility, also has potential shortcomings. On the one hand, profiler front ends may be interested in a diverse set of events that occur in the virtual machine. On the other hand, virtual machine implementations from different vendors may be different enough that it is impossible to expose all the interesting events through a general-purpose interface.

The contribution of our work is to reconcile these differences. We have designed a general-purpose profiling interface that is suitable for a wide variety of virtual machine implementations and profiler front ends.

We describe next the key role played by the JVMPI in the overall profiling architecture.

Role in overall profiling architecture. Figure 1 illustrates the overall profiling architecture. The JVMPI is a binary function-call interface between the Java virtual machine and a *profiler agent* that runs in the same process. The JVMPI provides a mechanism for the profiler agent to interact with the Java virtual machine and present profiling data to the front end. The in-process function-call interface allows maximum control with minimum intrusion on the part of a profiling tool.

Note that although the profiler agent runs in the same process as the virtual machine, the profiler front end typically resides in a different process, or even on a different machine. The profiler front end is separated to prevent it from interfering with the application. Process-level separation ensures that resources consumed by the profiler front end are not attributed to the profiled application. Our experience shows that it is possible to write compact profiler agents that delegate resource-intensive tasks to the profiler front end, so that running the profiler agent in the same process as the virtual machine does not overly distort the profiling information.

The profiler agent is typically implemented as a dynamically loaded library. The Java virtual machine loads the profiler agent at startup and looks for a prespecified entry point. Then, the virtual machine and the profiler agent initialize a JVMPI function pointer table to their implementations of the JVMPI. The function table consists of two sets of functions, one implemented by the profiler agent and the other implemented by the virtual machine. The Java virtual machine makes function calls to inform the profiler agent about various events that occur during the execution of the Java application. The agent in turn

receives profiling events and calls back into the Java virtual machine to issue control requests or to obtain more information in response to particular events.

Using function calls is a good approach to an efficient binary interface between the profiler agent and different virtual machine implementations. Sending profiling events through function calls is somewhat slower than directly instrumenting the virtual machine to gather specific profiling information. However, as we will see in a later subsection on the overhead of disabled profiling events, a majority of the profiling events are sent in situations where the Java virtual machine can tolerate the additional cost of a function call.

Event notification. The profiler agent implements only one function, NotifyEvent. This function is called by the virtual machine to inform the profiler agent of run-time events. Events are represented by data structures consisting of an integer indicating the type of the event and the identifier of the thread whose execution caused the event, followed by information specific to the event. To illustrate, we list the definition of the JVMPI_Event structure and one of its variants gc_info below. The gc_info variant records information about an invocation of the garbage collector. The event-specific information indicates the number of live objects, total space used by live objects, and the total heap size.

```
typedef struct {
    jint event_type;
    JNIEnv *thread_id;
    . . .
    union {
        . . .
        struct {
            jlong used_objects;
            jlong used_object_space;
            jlong total_object_space;
        } gc_info;
        . . .
    } u;
} JVMPI_Event;
```

The following are key types of JVMPI events:

- THREAD_START, THREAD_END. Events of these types are issued when threads start and end in the Java virtual machine.
- CLASS_LOAD, CLASS_UNLOAD. Events of these types are issued when classes are loaded and unloaded in the Java virtual machine.
- CLASS_FOR_INSTRUMENT. Events of this type are issued when classes become ready to be instrumented by the profiler agent. The virtual machine passes a pointer to a class file in the event data. The profiler can instrument the class file and set the pointer to the newly instrumented class.
- METHOD_ENTER, METHOD_EXIT. Events of these types are issued when the virtual machine starts and finishes method executions.
- NEW_ARENA, DELETE_ARENA. We introduce the abstract notion of a heap *arena*, in which objects are allocated. Events of these types are issued when heap arenas are created and deleted from memory. These events, along with the other allocation and garbage-collection related events described in the following two items, are described later in detail.
- NEW_OBJECT, DELETE_OBJECT. Events of these types are issued when objects are allocated and subject to garbage collection from a heap arena.
- MOVE_OBJECT. Events of this type are issued when objects are relocated to different heap arenas during garbage collection.
- COMPILED_METHOD_LOAD, COMPILED_METHOD_UNLOAD. Events of these types are issued when a just-in-time (JIT) compiler compiles methods to native code and loads them in memory and when the compiled code is unloaded from memory.
- MONITOR_CONTENDED_ENTER. Events of this type are issued when a thread blocks as it attempts to enter a monitor already owned by another thread.
- MONITOR_CONTENDED_ENTERED. Events of this type are issued when a thread finishes waiting to enter a monitor and acquire the monitor.
- MONITOR_CONTENDED_EXIT. Events of this type are issued when threads exit a monitor and discover that another thread is waiting to enter the same monitor.
- MONITOR_WAIT. Events of this type are issued when threads wait on a condition variable.
- MONITOR_WAITED. Events of this type are issued when threads finish waiting on a condition variable.
- HEAP_DUMP. Events of this type are issued to dump the state of the heap.
- MONITOR_DUMP. Events of this type are issued to dump the state of all the monitors and the threads in the system.

We will not go into the details of the event-specific data for each event type in this paper. But they are fully described in the JVMPI documentation that is

shipped with the Java 2 SDK, Standard Edition, releases version 1.2 and onward.[4]

To efficiently pass data during event notification, the JVMPI uses unique identifiers to refer to virtual machine entities such as threads, classes, methods, and objects. An identifier is assigned to an entity by the virtual machine during its defining event when all the information associated with that identifier is sent. For example, the defining event for a class identifier, the CLASS_LOAD event, contains, among other entries, the name of the class. An identifier is valid until its undefining event arrives, after which it may be reused. For example, a class identifier becomes invalid after the corresponding CLASS_UNLOAD event is notified.

Assigning unique identifiers during run time should be cheap in most Java virtual machine implementations. A thread is identified by its JNIEnv interface pointer. Other entities may be uniquely identified by their address in memory. Since objects may be relocated during garbage collection, we specify that MOVE_OBJECT events invalidate an object identifier. The MOVE_OBJECT event-specific data contain, among other fields, the new object identifier.

Java virtual machine call-backs. The Java virtual machine implements a set of call-back functions to enable the profiler agent to set control parameters and obtain more information in response to event notification. Following are the key call-back functions:

- EnableEvent, DisableEvent. These functions are used by the profiler agent to selectively enable or disable notification of a specified type of event at run time. The virtual machine returns a code indicating success, failure, or that notification for the event type is not supported. For example, all the events notified for heap profiling may be turned off when CPU time profiling is being performed. Notification of event types may be enabled or disabled any number of times while the application is running.
- RequestEvent. This function is used by the profiler agent to request the notification of certain event types such as HEAP_DUMP and MONITOR_DUMP. Such events happen only at the request of the profiler agent and cannot be enabled or disabled. This call-back can also be used to request the virtual machine to send the event-specific data for the defining event of a particular Java virtual machine entity, given its valid JVMPI identifier. We provide more details of why this event is needed in the subsection on interactive profiling.
- GetCallTrace. This function is called by the profiler agent to obtain the dynamic stack trace up to a specified depth of a given thread at a given time.
- SuspendThread, ResumeThread. These functions are used by the profiler agent to suspend and resume threads.
- ThreadHasRun. This function is used to determine whether a thread has run since the last time it was suspended. Details on how this call-back may be implemented are given later in the discussion on statistical sampling.
- GetThreadStatus. This function is called to obtain the status of a thread, whether it is runnable, blocked, or waiting on a monitor, and whether it has been suspended or interrupted.
- EnableGC, DisableGC, and RunGC. These functions are called to enable, disable, or run the garbage collector.

We will not describe the arguments taken by the Java virtual machine call-backs here. Please refer to the documentation shipped with the Java 2 SDK, Standard Edition, version 1.2 and onward[4] for a detailed description of the Java virtual machine call-backs.

The JVMPI also provides for utility call-backs into the virtual machine to create system threads, store thread local storage, obtain the accumulated CPU time consumed by a thread, and create and use raw monitors.

Our design makes the JVMPI completely extensible. Extending the JVMPI to evolve with the Java virtual machine would typically require defining new events and their JVMPI_Event variants. The existing framework does not need to be changed.

The HPROF profiler. To illustrate the power of the JVMPI and show how it may be utilized, we describe some of the features in the HPROF agent, a simple profiler agent shipped with Java 2 SDK, Standard Edition, version 1.2 and onward. The HPROF agent is a dynamically linked library. It interacts with the JVMPI and presents profiling information either to the user directly or through profiler front ends.

We can invoke the HPROF agent by passing a special option to the Java virtual machine:

```
java -Xrunhprof ProgName
```

ProgName is the name of a Java application. Note that we pass the -Xrunhprof option to java, the op-

Figure 2 HPROF heap allocation profile

SITES BEGIN (ordered by live bytes) Thu May 13 14:13:28 1999

rank	percent		live		allocated		stack trace	class name
	self	accum	bytes	objs	bytes	objs		
1	6.88%	6.88%	160004	1	160004	1	1006	int[ ]
2	5.15%	12.03%	119688	9974	199128	16594	3162	spec/benchmarks/_205_raytrace/ObjNode
3	3.12%	15.15%	72540	1395	72540	1395	2550	spec/benchmarks/_205_raytrace/TriangleObj
4	2.77%	17.93%	64428	1239	64428	1239	3344	spec/benchmarks/_205_raytrace/TriangleObj
...								

timized version of the Java virtual machine. We need not rely on a specially instrumented version of the virtual machine to support profiling.

Depending on the type of profiling requested, HPROF instructs the virtual machine to send it the relevant profiling events. It gathers the event data into profiling information and outputs the result by default to a file. For example, the following command obtains the heap allocation profile for running a program:

```
java -Xrunhprof:heap=sites ProgName
```

Figure 2 contains a part of the heap allocation profile generated during a run of the _227_mtrt benchmark from the SPECjvm98 suite.[8] We show only parts of the profiler output here. A crucial piece of information in the heap profile is the amount of allocation that occurs in various parts of the program. An integer array occupies 6.88 percent of the live bytes according to the SITES record in Figure 2. We can also see that 5.15 percent of the live bytes is occupied by 9974 instances of the spec/benchmarks/_205_raytrace/ObjNode class and that it is only a fraction of the total allocation of instances (16 594) of the same class at that site; the rest has been collected as garbage.

A good way to relate allocation sites to the source code is to record the dynamic stack traces that led to the heap allocation. Figure 3 shows another part of the profiler output that illustrates the stack traces referred to by the four allocation sites presented in Figure 2.

A stack trace consists of the identifier of the thread to which the trace belongs and a set of frames. Each frame contains a class name, method name, source file name, and the line number where the method was executing. The user can set the maximum number of frames collected by the HPROF agent. The default limit is four. Stack traces reveal not only which methods performed heap allocation, but also which methods were ultimately responsible for making calls that resulted in memory allocation. For example, in the heap profile shown in Figure 3, both traces 3344 and 2550 cause allocation of spec/benchmarks/_205_raytrace/TriangleObj class instances. Each trace originated from different methods executing in different threads.

The HPROF agent has built-in support for profiling CPU usage. For example, Figure 4 is part of the generated output after the HPROF agent performs sampling-based CPU time profiling on a run of the _202_jess benchmark from the SPECjvm98 suite.[8]

The HPROF agent periodically samples the stack of all running threads to record the most frequently active stack traces. The count field in Figure 4 indicates how many times a particular stack trace was found

Figure 3 HPROF stack traces

```
THREAD START  (obj=1d7ff8, id = 1, name="main", group="main")
THREAD START  (obj=3468f8, id = 6, name="Thread-1", group="main")
THREAD START  (obj=3467b8, id = 7, name="Thread-2", group="main")
…
TRACE 1006: (thread=1)
        spec/benchmarks/_205_raytrace/Canvas.<init>(Canvas.java:82)
        spec/benchmarks/_205_raytrace/RayTracer.run(RayTracer.java:76)
        spec/benchmarks/_205_raytrace/RayTracer.inst_main(RayTracer.java:57)
        spec/benchmarks/_227_mtrt/Main.runBenchmark(Main.java:24)

TRACE 3162: (thread=6)
        spec/benchmarks/_205_raytrace/OctNode.CreateChildren(OctNode.java:233)
        spec/benchmarks/_205_raytrace/OctNode.CreateChildren(OctNode.java:250)
        spec/benchmarks/_205_raytrace/OctNode.CreateChildren(OctNode.java:250)
        spec/benchmarks/_205_raytrace/OctNode.CreateChildren(OctNode.java:250)

TRACE 3344: (thread=7)
        spec/benchmarks/_205_raytrace/Scene.ReadPoly(Scene.java:345)
        spec/benchmarks/_205_raytrace/Scene.LoadSceneOrig(Scene.java:119)
        spec/benchmarks/_205_raytrace/Scene.LoadScene(Scene.java:70)
        spec/benchmarks/_205_raytrace/Scene.<init>(Scene.java:591)


TRACE 2550: (thread=6)
        spec/benchmarks/_205_raytrace/Scene.ReadPoly(Scene.java:345)
        spec/benchmarks/_205_raytrace/Scene.LoadSceneOrig(Scene.java:119)
        spec/benchmarks/_205_raytrace/Scene.LoadScene(Scene.java:70)
        spec/benchmarks/_205_raytrace/Scene.<init>(Scene.java:591)
…
```

Figure 4 HPROF profile of CPU usage hot spots

```
CPU SAMPLES BEGIN  (total = 226)  Thu May 13 14:29:55 1999
 rank    self     accum      count  trace                    method

    1  15.93%   15.93%         36     30  java/io/FileInputStream.readBytes
    2   9.73%   25.66%         22    170  spec/benchmarks/_202_jess/jess/ValueVector.equals
    3   7.52%   33.19%         17      8  java/io/UnixFileSystem.getBooleanAttributes0
    4   4.87%   38.05%         11     18  java/lang/ClassLoader.defineClass0
    5   3.10%   41.15%          7    164  spec/benchmarks/_202_jess/jess/Value.equals
    6   2.21%   43.36%          5     14  java/io/FileInputStream.open
    7   2.21%   45.58%          5    163  spec/benchmarks/_202_jess/jess/Node2.appendToken
    8   2.21%   47.79%          5    166  spec/benchmarks/_202_jess/jess/Node2.findInMemory
    9   1.77%   49.56%          4    174  spec/benchmarks/_202_jess/jess/Token.data_equals
   10   1.33%   50.88%          3    172  spec/benchmarks/_202_jess/jess/Node2.findInMemory
   ...
CPU SAMPLES END
```

to be active. These stack traces correspond to the CPU usage hot spots in the application.

The HPROF agent can also report complete heap dumps and monitor contention information. We will not list more examples of how the HPROF agent presents the information obtained through the profiling interface in this paper. Instead, we will discuss the details of how various profiling interface features are supported in the virtual machine.

JVMPI design justifications

What were the motivations behind the design choices made for JVMPI? As we have stated before, our high-level goal is to create a general-purpose and portable profiling interface for the Java virtual machine. Our design must be flexible enough to suit the needs of a variety of Java virtual machine implementations and profiler front ends. The JVMPI must provide comprehensive profiling support, i.e., the user must be able to perform various types of profiling such as CPU time profiling, heap profiling, and thread and monitor profiling. The JVMPI must support interactive profiling, and its implementation must be nonintrusive when profiling is turned off. The JVMPI must provide a way to attribute event costs to specific execution contexts of the program for effective performance analysis. In addition, it must solve the challenges posed by garbage collection and multithreading support provided by the Java virtual machine. In the following subsections we demonstrate how our design achieves the above goals.

Cost attribution to specific execution contexts. How do we attribute event costs to specific execution contexts? In particular, how do we do so effectively in a multithreaded environment?

We had a number of design options—present information at the method call level, or at a finer granularity such as basic blocks or different execution paths inside a method. On the basis of our experience with tuning Java applications, we believe that there is little reason to attribute cost to a finer granularity than a method and the source code line number where the method was executing. Programmers typically have a good understanding of cost distribution inside a method and can easily pinpoint the inefficiency with the source code line number; methods in Java applications tend to be smaller than, for example, C or C++ functions.

It is not enough to report a flat profile consisting only of the portion of time in individual methods. If, for example, the profiler reports that a program spends a significant portion of time in the String.getBytes method, how do we know which part of our program indirectly contributed to invoking this method if the program does not call this method directly?

A good way to attribute profiling costs to specific execution contexts is to report the dynamic stack traces executed by the thread that caused the resource consumption. Dynamic stack traces become less informative in some programming languages where it is hard to associate stack frames with source language constructs, such as when anonymous functions are involved. Fortunately, anonymous inner classes in the Java programming language are represented by classes with informative names at run time.

The GetCallTrace call-back provides the dynamic method call stack trace of a thread up to a specified depth at any point. Each stack trace element consists of a method identifier and the line number in the source code where the method was executing. The defining information for a method identifier is notified when the class to which it belongs is loaded into the virtual machine. As we saw earlier, the data sent during any event notification contain the identifier of the thread whose execution caused the event. When the profiler agent receives an event notification, it can use GetCallTrace to determine the current call-stack of the corresponding thread in which the event occurred.

CPU time profiling. JVMPI supports both statistical sampling and code instrumentation. Statistical sampling is less disruptive to program execution, but cannot provide completely accurate information. Code instrumentation, in contrast, may be more disruptive, but allows the profiler to record all the events of interest.

The Java virtual machine is a multithreaded execution environment. One difficulty in building CPU time profilers for such systems is how to properly attribute CPU time to each thread, so that the time spent in a method is accounted for only when the method actually runs on the CPU, not when it is unscheduled and waiting to run.

Statistical sampling. The basic thread-aware CPU time-sampling algorithm that is supported by the JVMPI is as follows:

```
while (true) {
   sleep for a short interval;

   for each thread T {
     call ThreadSuspend on T;
   }

   for each thread T {
     if ThreadHasRun is true for T {
       GetCallTrace for T;
       assign a cost unit to the stack trace;
     }
   }

   for each thread T {
     call ThreadResume on T;
   }
}
```

The profiler needs to suspend the thread while collecting its stack trace, otherwise a running thread may change the stack frames as the stack trace is being collected.

The main difficulty in the above scheme is how to implement ThreadHasRun, i.e., how to determine whether a thread has run in the last sampling interval. We should not attribute cost units to threads that are waiting for an I/O operation or waiting to be scheduled in the last sampling interval. Ideally, this problem would be solved if the scheduler could inform the profiler of the exact time interval in which a thread is running, or if the profiler could find out the amount of CPU time a thread has consumed at each sampling point.

Unfortunately, modern operating systems such as Windows NT** and Solaris** neither expose the kernel scheduler nor provide ways to obtain accurate per-thread CPU time. For example, the GetThread-

Times call on Windows NT returns per-thread CPU time in 10 millisecond increments, too inaccurate for profiling needs.

Our solution is to determine whether a thread has run in a sampling interval by checking whether its register set has changed. If a thread has run in the last sampling interval, it is almost certain that the contents of the register set have changed.

The information gathered for the purpose of profiling need not be 100 percent reliable. It is extremely unlikely, however, that a running thread maintains an unchanged register set, which includes such registers as the stack pointer, the program counter, and all general-purpose registers. One pathological example of a running program with a constant register set is the following C code segment, where the program enters into an infinite loop that consists of one instruction:

```
loop: goto loop;
```

In practice, we find that it suffices to compute and record a checksum of a subset of the registers, thus further reducing the overhead of the profiler.

An important aspect of our sampling algorithm worth pointing out is that it does not depend on the number of processors on the machine. Therefore, it would work equally well on uniprocessor and multiprocessor machines.

The cost of suspending all threads and collecting their stack traces is roughly proportional to the number of threads running in the virtual machine. A minor enhancement to the sampling algorithm discussed earlier is that we need not suspend and collect stack traces for threads that are blocked on monitors managed by the virtual machine. This enhancement significantly reduces the profiling overhead for many multithreaded programs in which most threads are blocked most of the time. From our measurements, profiling a multithreaded program (10 total threads, two-thirds runnable at a time), with our sampling-based CPU time profiler with a sampling interval of one millisecond, incurs less than 20 percent overhead on platforms such as Windows NT and Solaris.

Alternatively, the profiler may perform program counter sampling. The Java virtual machine notifies COMPILED_METHOD_LOAD events when methods get compiled into native code and loaded into memory. When they are unloaded, the Java virtual machine sends COMPILED_METHOD_UNLOAD events. The start and end address of the compiled code, along with the source code line number mapping, is sent as a part of the COMPILED_METHOD_LOAD event-specific data. Thus, the profiler agent has enough information to attribute the sampled program counter value to a particular method and its source code.

Code instrumentation. Code instrumentation is also supported in two ways. The profiler agent may measure the time spent on the CPU between METHOD_ENTER and METHOD_EXIT events by the thread generating the events for all methods. Naturally, this approach introduces additional C function call overhead to each profiled method.

A less disruptive way that is supported by JVMPI is to allow the profiler agent to dynamically inject profiling code directly into the profiled program. This type of code instrumentation is easier on platforms using the Java language than on traditional CPUs, because there is a standard class file format. The JVMPI allows the profiler agent to instrument every class file before it is loaded by the virtual machine using the CLASS_FOR_INSTRUMENT event. The profiler agent may, for example, insert a custom bytecode sequence that records method invocations, control flow among the basic blocks, or other operations performed inside the method body. When the profiler agent changes the content of a class file, it must ensure that the resulting class file is still valid according to the Java Virtual Machine Specification.

Heap profiling. Heap profiling serves a number of purposes: pinpointing the part of a program that performs excessive heap allocation, revealing the performance characteristics of the underlying garbage collection algorithm, and detecting the causes of unnecessary object retention.

Excessive heap allocation leads to performance degradation for two reasons: the cost of the allocation operations themselves and, because the heap is filled up more quickly, the cost of more frequent garbage collections. With the JVMPI, the profiler agent can track all NEW_OBJECT events. At each event, the agent can obtain the stack trace that serves as a good identification of the heap allocation site. The programmer should concentrate on optimizing busy heap allocation sites. The profiler agent can also monitor DELETE_OBJECT events to keep track of

how many objects allocated from a given site are being kept alive.

Unnecessary object retention occurs when an object is no longer useful but is being kept alive by another object that is in use. For example, a programmer may insert objects into a global hash table. These objects cannot be collected as garbage as long as any entry in the hash table is useful and the hash table is kept alive.

An effective way to find the causes of unnecessary object retention is to analyze the heap dump received in a HEAP_DUMP event. The heap dump contains information about all the garbage collection roots, all live objects, and how objects refer to one another. This information can be processed and analyzed by the profiler front end.

An alternative way to track unnecessary object retention is to provide the direct support in the profiling interface for finding all objects that refer to a given object. The advantage of this incremental approach is that it requires less temporary storage than complete heap dumps. The disadvantage is that unlike heap dumps, the incremental approach cannot present a consistent view of all heap objects that are constantly being modified during program execution.

In practice, we do not find the size of heap dumps to be a problem. Typically, the majority of the heap space is occupied by primitive arrays. Because there are no internal pointers in primitive arrays, elements of primitive arrays need not be part of the heap dump.

Algorithm-independent allocation and garbage collection events. Many memory allocation and garbage collection algorithms are suitable for different Java virtual machine implementations. Mark-and-sweep, copying, generational, and reference counting are some examples. This presents a challenge to designing a portable profiling interface: Is there a set of events that can uniformly handle a wide variety of garbage collection algorithms?

We have designed a set of profiling events that cover all garbage collection algorithms that we are currently concerned with. As we have seen earlier, the virtual machine issues the following set of events:

- NEW_ARENA(arena identifier, or ID)
- DELETE_ARENA(arena ID)
- NEW_OBJECT(arena ID, object ID, class ID)
- DELETE_OBJECT(object ID)
- MOVE_OBJECT(old arena ID, old object ID, new arena ID, new object ID)

Our notation encodes the event-specific information in a pair of parentheses, immediately following the event type. Let us go through some examples to see how these events may be used with different garbage collection algorithms:

- A mark-and-sweep collector issues NEW_OBJECT events when allocating objects and issues DELETE_OBJECT events when adding objects to the free list. Only one arena ID is needed.
- A mark-sweep-compact collector additionally issues MOVE_OBJECT events. Again, only one arena is needed; the old and new arena IDs in the MOVE_OBJECT events are the same.
- A standard two-space copying collector creates two arenas. It issues MOVE_OBJECT events during garbage collection and issues a DELETE_ARENA event followed by a NEW_ARENA event with the same arena ID to free up all remaining objects in the semi-space.
- A generational collector issues a NEW_ARENA event for each generation. When an object is scavenged from one generation to another, a MOVE_OBJECT event is issued. All objects in an arena are implicitly freed when a DELETE_ARENA event arrives.
- A reference-counting collector issues NEW_OBJECT events when new objects are created and issues DELETE_OBJECT events when the reference count of an object reaches zero.

In summary, the simple set of heap allocation events supports a wide variety of garbage collection algorithms.

Monitor profiling. Monitors are the fundamental synchronization mechanism in the Java programming language. Programmers are generally concerned with two issues related to monitors: the performance impact of *monitor contention* and the cause of *deadlocks*.

Monitor contention is the primary cause of the lack of scalability in multiprocessor systems. Monitor contention is typically caused by multiple threads holding global locks too frequently or too long.

Frequently contended monitors can be tracked by tracing MONITOR_CONTENDED_ENTER events. Monitors being held for unnecessarily long periods

of time can be tracked by tracing MONITOR_CONTENDED_ENTERED events that indicate the amount of elapsed time the current thread was blocked before it entered the monitor. Tracing MONITOR_CONTENDED_EXIT events indicates possible performance problems caused by the current thread holding the monitor for too long.

During all these events, the overhead of issuing the event is negligible compared to the performance impact of the blocked monitor operation. The profiler agent can obtain the stack trace of the current thread and thus attribute the monitor contention events to the parts of the program responsible for issuing the monitor operations.

If every thread is waiting to enter monitors that are owned by another thread, the system runs into a deadlock situation. The profiler agent can request a MONITOR_DUMP event to find the cause of this kind of deadlock.[9]

The MONITOR_DUMP event data include information about the owner of each contended monitor, the list of threads waiting to enter the monitor, and the stack trace of all the threads. To obtain a consistent view of all threads and all monitors, all threads must be suspended using the SuspendThread callback.

Interactive profiling. An approach to support interactive profiling is to specify that all events must be recorded by the profiler agent and selectively passed onto the profiler front end. But this puts too much overhead on the agent and does not meet the requirements of programmers and tools vendors.

In JVMPI, event notification can be selectively enabled or disabled at run time using EnableEvent and DisableEvent. The need for dynamically enabling and disabling profiling events requires added checks in the virtual machine code paths that lead to the generation of these events. But we see in the next subsection that this requirement poses minimum overhead for the majority of events.

A problem that arises when profiler events can be enabled and disabled is that the profiler agent receives incomplete, or partial, profiling information. This problem has been characterized as the *partial profiling problem*.[10] For example, if the profiler agent enables CLASS_LOAD events after a number of classes have been loaded and a number of instances of these classes have been created, the agent may encounter NEW_OBJECT events that contain an unknown class identifier. This situation can occur for any identifier if its defining event was disabled.

A straightforward solution is to require the virtual machine to record all profiling events in a trace file, whether or not these events are enabled by the profiler agent. The virtual machine is then able to send the appropriate information for any entities unknown to the profiler agent. This approach is undesirable because of the potentially unlimited size of the trace file and the overhead when profiling events are disabled.

We solve the partial profiling problem based on one observation: The Java virtual machine keeps track of information internally about the valid entities whose identifiers can be sent with profiling events. The virtual machine does not need to keep track of outdated entities (such as a class that has been loaded and unloaded), because they will not appear in profiling events. When the profiler agent receives an unknown entity (such as an unknown class identifier), the entity is still valid, and thus the agent can immediately obtain all the relevant information from the virtual machine. At this point RequestEvent comes in. Using this call-back, the profiler can request information about unknown entities received as part of a profiling event. For example, when the profiler agent encounters an unknown class identifier, it may request the virtual machine to send the same information that is contained in a CLASS_LOAD event for this class.

Certain entities need to be treated specially by the profiling agent in order to deal with partial profiling information. For example, if the profiling agent disables the MOVE_OBJECT event, it must immediately invalidate all object IDs it knows about because they may be changed by future garbage collections. With the MOVE_OBJECT event disabled, the agent can request the virtual machine to send the class information about unknown object IDs. However, such requests must be made only when garbage collection is disabled by using the DisableGC call-back. Otherwise, garbage collection may generate a MOVE_OBJECT event asynchronously and invalidate the object ID before the virtual machine obtains the class information for this object ID.

Overhead of disabled profiling events. If the overhead of disabled profiling events is minimal, JVMPI can be shipped with production Java virtual machines. It offers two advantages: applications can be

profiled directly on the same virtual machines on which they run normally, and the user is saved from the space and administrative overheads of maintaining a special version for profiling.

The need for dynamically enabling and disabling profiling events requires added checks in the code paths that lead to the generation of these events.

The majority of profiling events are issued relatively infrequently. Examples of these types of events are class loading and unloading, thread start and end, garbage collection, and Java Native Interface (JNI) global reference creation and deletion. The JVMPI can easily support interactive low-overhead profiling by placing checks in the corresponding code paths without having a performance impact on normal program execution.

Heap profiling events, in particular NEW_OBJECT, DELETE_OBJECT, and MOVE_OBJECT, could be quite frequent. An added check in every object allocation may have a noticeable performance impact on program execution, especially if the check is inserted in the allocation fast path that typically is inlined into the code generated by the JIT compilers. Fortunately, garbage-collected memory systems by definition need to check for possible heap exhaustion conditions in every object allocation, even in the fast path. We can thus enable heap allocation events by forcing every object allocation into the slow path with a false heap exhaustion condition, and check whether heap profiling events have been enabled and whether the heap is really exhausted in the slow path. Because no change to the allocation fast path is needed, object allocation runs in full speed when heap profiling is disabled.

The METHOD_ENTER and METHOD_EXIT events are also generated frequently. They can be easily supported by the JIT compilers that can dynamically patch the generated code and the virtual method dispatch tables.

Related work

Some of the design rationales of a comprehensive profiling interface such as the JVMPI have been presented by the authors in an earlier paper.[11] The current paper, however, is the first complete account of the JVMPI design and implementation.

Extensive work has been done in CPU time profiling. The gprof tool,[12] for example, is a sample-based profiler that records call graphs, instead of flat profiles. Recent research[13–15] has improved the performance and accuracy of time profilers based on code instrumentation. Analysis techniques have been developed such that instrumentation code may be inserted with as little run-time overhead as possible.[16,17] Our sampling-based CPU time profiling uses stack traces to report CPU usage hot spots and is the most similar to the technique of call graph profiling.[18] Sansom et al.[19] investigated how to properly attribute costs in profiling higher-order lazy functional programs. Appel et al.[20] studied how to efficiently instrument code in the presence of code inlining and garbage collection. None of the above work addresses the issues in profiling multithreaded programs, however.

Issues similar to profiling multithreaded programs arise in parallel programs,[21,22] where the profiler typically executes concurrently with the program and can selectively profile parts of the program.

Heap profiling similar to that reported in this paper has been developed for C, Lisp,[23] and Modula-3.[24] To our knowledge, our work is the first that constructs a heap profiling interface that is independent of the underlying garbage collection algorithm.

We have a general-purpose profiling architecture, but sometimes it is also useful to build custom profilers[25] that target specific compiler optimizations.

There have been numerous earlier experiments (for example, see Reference 26) on building interactive profiling tools for Java applications. These approaches are typically based on placing custom instrumentation in the Java virtual machine implementation.

Conclusions

We have presented the Java virtual machine profiler interface (JVMPI) and justified its design choices. We have demonstrated that its scope includes multithreaded CPU usage profilers, heap allocation and garbage collection profilers, monitor contention profilers, and thread deadlock detectors. In addition, we have shown that the JVMPI supports interactive profiling and carries extremely low run-time overhead.

The JVMPI solves the problem of profilers relying on custom instrumentation. It is general-purpose and powerful enough to suit the needs of different profiling tool vendors and virtual machine vendors.

We believe that our work lays a foundation for building advanced profiling tools for the Java language.

Acknowledgments

We wish to thank a number of colleagues at IBM, in particular Robert Berry, Alan Stevens, and members of the Jinsight team, for numerous comments, proposals, and discussions that led to many improvements in the JVMPI.

**Trademark or registered trademark of Sun Microsystems, Inc., Intuitive Systems, Inc., KL Group, Inc., Compuware Corporation, or Microsoft Corporation.

Cited references and note

1. D. Ingalls, "The Execution Profile as a Programming Tool," *Design and Optimization of Compilers*, R. Rustin, Editor, Prentice-Hall, Englewood Cliffs, NJ (1972).
2. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
3. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).
4. *Java 2 SDK, Standard Edition, v 1.2*, Java Software, Sun Microsystems, Inc., Palo Alto, CA (1998). Available at http://java.sun.com/products/jdk/1.2.
5. OptimizeIt 3.0 Professional, Intuitive Systems, Inc., Cupertino, CA (1999), available at http://www.optimizeit.com/index.html.
6. JProbe, KL Group, Inc., Toronto (1999), available at http://www.klgroup.com/.
7. TrueTime and TrueCoverage, NuMega, Compuware Corp., Farmington Hills, MI (1999), available at http://www.numega.com/.
8. SPECjvm98 Benchmarks, The Standard Performance Evaluation Corporation (SPGC), Manassas, VA (1998), available at http://www.spec.org/osg/jvm98/.
9. Deadlocks may also be caused by implicit locking and ordering in libraries and system calls, such as I/O operations.
10. The Jinsight team, IBM Corporation, private communication (July 1998).
11. S. Liang and D. Viswanathan, "Comprehensive Profiling Support in the Java Virtual Machine," *USENIX Conference on Object-Oriented Technologies (COOTS)* (May 1999), pp. 229–240.
12. S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (June 1982), pp. 120–126.
13. M. Bishop, "Profiling Under UNIX by Patching," *Software—Practice and Experience* **17**, No. 10, 729–739 (October 1987).
14. C. Ponder and R. J. Fateman, "Inaccuracies in Program Profilers," *Software—Practice and Experience* **18**, No. 5, 459–467 (May 1988).
15. J. F. Reiser and J. P. Skudiarek, "Program Profiling Problems, and a Solution via Machine Language Rewriting," *ACM SIGPLAN Notices* **29**, No. 1, 37–45 (January 1994).
16. T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems* **16**, No. 4, 1319–1360 (July 1994).
17. G. Ammons, T. Ball, and J. R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1997).
18. R. J. Hall and A. J. Goldberg, "Call Path Profiling of Monotonic Program Resources in UNIX," *Proceedings of Summer 1993 USENIX Technical Conference* (June 1993), pp. 1–13.
19. P. M. Sansom and S. L. Peyton Jones, "Time and Space Profiling for Non-Strict Higher-Order Functional Languages," *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, ACM Press, New York (January 1995), pp. 355–366.
20. A. W. Appel, B. F. Duba, D. B. MacQueen, and A. P. Tolmach, *Profiling in the Presence of Optimization and Garbage Collection*, Technical Report CS-TR-197-88, Princeton University, Princeton, NJ (1988).
21. Z. Aral and I. Gernter, "Nonintrusive and Interactive Profiling in Parasight," *Proceedings of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems* (July 1988), pp. 21–30.
22. J. M. D. Hill, S. A. Jarvis, C. Siniolakis, and V. P. Vasilev, "Portable and Architecture Independent Parallel Performance Tuning Using a Call-Graph Profiling Tool: A Case Study in Optimizing SQL," *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing—PDP'98* (January 1998), pp. 286–294.
23. B. Zorn and P. Hilfinger, "A Memory Allocation Profiler for C and Lisp Programs," *Proceedings of Summer USENIX'88 Conference Proceedings* (June 1988), pp. 223–237.
24. D. L. Detlefs and B. Kalsow, "Debugging Storage Management Problems in Garbage-Collected Environments," *USENIX Conference on Object-Oriented Technologies (COOTS)* (June 1995), pp. 69–82.
25. M. Cierniak and S. Srinivas, "Java and Scientific Programming: Portable Browsers for Performance Programming," *Java for Computational Science and Engineering—Simulation and Modeling II* (June 1997).
26. J. J. Barton and J. Whaley, "A Real-Time Performance Visualizer for Java," *Dr. Dobb's Journal* **24,** 44–48 (March 1998).

Accepted for publication September 30, 1999.

Deepa Viswanathan *Extricity Software, Inc., 555 Twin Dolphin Drive, Suite 600, Redwood Shores, California 94065 (electronic mail: dviswana@yahoo.com).* Ms. Viswanathan is at present working as a software engineer with Extricity Software, a provider of business-to-business e-commerce software. Prior to that she worked at Sun Microsystems in the Java run-time group. She received an M.S. in computer science from Indiana University in 1997 and a B.S. in computer science from Birla Institute of Technology and Science, India, in 1995.

Sheng Liang *Sun Microsystems, Inc., 901 San Antonio Road, CUP02-302, Palo Alto, California 94303 (electronic mail: sheng.liang@eng.sun.com).* Dr. Liang is a member of the Java team at Sun Microsystems and is a leading expert in Java virtual machine technologies. He led the Java virtual machine development for the first release of the Java 2 Platform and is the author of the Addison-Wesley Java Series book on the Java Native Interface. He holds a Ph.D. in computer science from Yale University.

Reprint Order No. G321-5717.