USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA, May 3–7, 1999

Comprehensive Profiling Support in the Java Virtual Machine

Sheng Liang and Deepa Viswanathan
Sun Microsystems Inc.

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Comprehensive Profiling Support in the Java™ Virtual Machine

Sheng Liang Deepa Viswanathan
Sun Microsystems Inc.
901 San Antonio Road, CUP02-302
Palo Alto, CA 94303
{sheng.liang,deepa.viswanathan}@eng.sun.com

Abstract

Existing profilers for Java applications typically rely on custom instrumentation in the Java virtual machine, and measure only limited types of resource consumption. Garbage collection and multi-threading pose additional challenges to profiler design and implementation.

In this paper we discuss a general-purpose, portable, and extensible approach for obtaining comprehensive profiling information from the Java virtual machine. Profilers based on this framework can uncover CPU usage hot spots, heavy memory allocation sites, unnecessary object retention, contended monitors, and thread deadlocks. In addition, we discuss a novel algorithm for thread-aware statistical CPU time profiling, a heap profiling technique independent of the garbage collection implementation, and support for interactive profiling with minimum overhead.

1 Introduction

Profiling [14] 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, as well as 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 most amount of memory.

This paper is mainly concerned with profilers that provide information to programmers, as opposed to profilers that 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. A profiler that constructs the complete call graph, on the other hand, may be permitted to slow down the program execution significantly.

This paper discusses techniques for profiling support in the Java virtual machine [17]. Java applications are written in the Java programming language [10], and compiled into machine-independent binary class files, which can then be executed on any compatible implementation of the Java virtual machine. The Java virtual machine is a multi-threaded 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 `D` are allocated in method `C.h`. More specifically, it is also useful to know that majority of these allocations occur when `B.g` calls `C.h`, *and* only when `A.f` calls `B.g`.
- The profiler may show why a certain object is not being garbage collected. The programmer may want to know, for example, that an instance of class `C` is not garbage collected 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 con-

tended 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. Knowing the exact reason that a class is loaded, the programmer can optimize the code to reduce memory usage.

The first contribution of this paper is to present a general-purpose, extensible, and portable Java virtual machine profiling architecture. Existing profilers typically rely on custom instrumentation in the Java virtual machine and measure limited types of resource consumption. In contrast, our framework relies on an interface that provides comprehensive support for profilers that can be built independent of the Java virtual machine. A profiler can obtain information about CPU usage hot spots, heavy memory allocation sites, unnecessary object retention, monitor contention, and thread deadlocks. Both code instrumentation and statistical sampling are supported. Adding new features typically requires introducing new event types, and does not require changes to the profiling interface itself. The profiling interface is portable. It is not dependent on the internal implementation of the Java virtual machine. For example, the heap profiling support is independent of the garbage collection implementation, and can present useful information for a wide range of garbage collection algorithms. The benefit of this approach is obvious. Tools vendors can ship profilers that work with any virtual machine that implements the interface. Equivalently, users of a Java virtual machine can easily take advantage of the profilers available from different tools vendors.

The second contribution of this paper is to introduce an algorithm that obtains accurate CPU-time profiles in a multi-threaded execution environment with minimum overhead. It is a standard technique to perform statistical CPU time profiling by periodically sampling the running program. What is less known, however, is how to obtain accurate per-thread CPU time usage on the majority of operating systems that do not provide access to the thread scheduler or a high-resolution per-thread CPU timer clock. In these cases, it is difficult to attribute elapsed time to threads that are actually running, as opposed to threads that are blocked, for example, in an I/O operation. Our solution is to determine whether a thread has run in a sampling interval by comparing the check sum of its register sets. To our knowledge, this is the most portable technique for obtaining thread-aware CPU-time profiles on modern operating systems.

The third contribution is to demonstrate how our approach supports interactive profiling with minimum overhead. Users can selectively enable or disable different types of profiling while the application is running. This is achieved with very low space and time overhead. Neither the virtual machine, nor the profiler need to accumulate large amounts of trace data. The Java virtual machine incurs only a test and branch overhead for a disabled profiling event. 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.

We have implemented all the techniques discussed in this paper in the Java Development Kit (JDK) 1.2 [15]. Numerous tool vendors have already built profiling front-ends that rely on the comprehensive profiling support built into the JDK 1.2 virtual machine.

We will begin by introducing the general-purpose profiling architecture, before we discuss the underlying techniques in detail. We assume the reader is familiar with the basic concepts in the Java programming language [10] and the Java virtual machine [17].

2 Profiling Architecture

The key component of our profiling architecture is a general-purpose profiling interface between the Java virtual machine and the front-end responsible for presenting the profiling information. A profiling interface, as opposed to direct profiling support in the virtual machine implementation, offers two main advantages:

First, profilers can present profiling information in different forms. For example, one profiler may simply record events that occur in the virtual machine in a trace file. Alternatively, another profiler may receive input from the user and display the requested information interactively.

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

A profiling interface, while providing flexibility, also has potential shortcomings. On 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 Java Virtual Machine Profiler Interface (JVMPI) that is efficient and powerful enough to suit the needs of a wide variety of virtual machine implementations and profiler front-ends.

Figure 1 illustrates the overall 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 profiler agent is responsible for the communication between the Java virtual machine and the profiler front-end. 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 reason for the separation of the profiler front-end is to prevent the profiler front-end from interfering with the application. Process-level separation ensures that resources consumed by the profiler front-end does not get attributed to the profiled application. Our experience shows that it is possible to write 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.

We will introduce some of the features of the Java virtual machine profiling interface in the remainder of this section, and discuss how such features are supported by the Java virtual machine in later sections.

2.1 Java Virtual Machine Profiler Interface

Figure 1 illustrates the role of the JVMPI in the overall profiler architecture. The JVMPI is a two-way function call interface between the Java virtual machine and the profiler agent.

The profiler agent is typically implemented as a dynamically-loaded library. The 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 accomplish one the the following tasks:

- The agent may disable and enable certain type of events sent through the JVMPI, based on the needs of the profiler front-end.
- The agent may request more information in response to particular events. For example, after the agent receives a JVMPI event, it can make a JVMPI function call to find out the stack trace for the current thread, so that the profiler front-end can inform the user about the program execution context that led to this JVMPI event.

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. As we will see, however, majority of the profiling events are sent in situations where we can tolerate the additional cost of a function call.

JVMPI events are data structures consisting of an integer indicating the type of the event, the identifier of the thread in which the event occurred, 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;
```

Additional details of the JVMPI can be found in the documentation that is shipped with the JDK 1.2 release [15].

2.2 The HPROF Agent

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 JDK 1.2. The HPROF agent is a dynamically-linked library

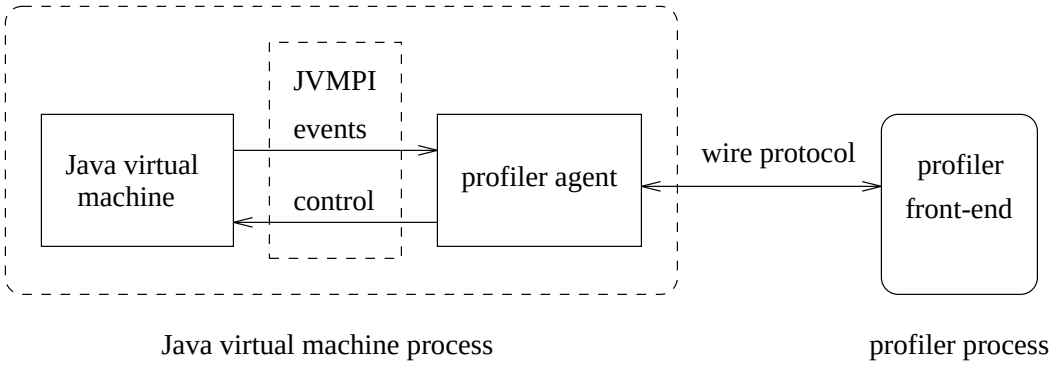

Figure 1: Profiler Architecture

shipped with JDK 1.2. 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 optimized 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 the heap allocation profile generated by running the Java compiler (`javac`) on a set of input files. We only show parts of the profiler output here due to the lack of space. A crucial piece of information in heap profile is the amount of allocation that occurs in various parts of the program. The `SITES` record above tells us that 9.18% of live objects are character arrays. Note that the amount of live data is only a fraction of the total allocation that has occurred at a given site; the rest has been garbage collected.

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.

Each frame in the stack trace contains class name, method name, source file name, and the line number. The user can set the maximum number of frames collected by the HPROF agent. The default limit is 4. 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 above, instances of the same `java/util/Hashtable$Entry` class are allocated in traces 1091 and 1264, each originated from different methods.

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 the `javac` compiler.

The HPROF agent periodically samples the stack of all running threads to record the most frequently active stack traces. The `count` field above indicates how many times a particular stack trace was found 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. Due to the lack of space, we will not list more examples of how the HPROF agent presents the information obtained through the profiling interface. However, we are ready to explain the details of how various profiling interface features are supported in the virtual machine.

```
SITES BEGIN (ordered by live bytes) Wed Oct  7 11:38:10 1998
        percent         live       alloc'ed    stack class
rank  self  accum   bytes objs   bytes   objs trace name
   1 9.18%  9.18%  149224 5916 1984600 129884  1073 char []
   2 7.28% 16.45%  118320 5916  118320   5916  1090 sun/tools/java/Identifier
   3 7.28% 23.73%  118320 5916  118320   5916  1091 java/util/Hashtable$Entry
   ...
   7 3.39% 41.42%   55180 2759   55180   2759  1264 java/util/Hashtable$Entry
   ...
SITES END
```

Figure 2: HPROF Heap Allocation Profile

```
THREAD START (obj=1d6b20, id = 1, name="main", group="main")
...
TRACE 1073: (thread=1)
        java/lang/String.<init>(String.java:244)
        sun/tools/java/Scanner.bufferString(Scanner.java:143)
        sun/tools/java/Scanner.scanIdentifier(Scanner.java:942)
        sun/tools/java/Scanner.xscan(Scanner.java:1281)

TRACE 1090: (thread=1)
        sun/tools/java/Identifier.lookup(Identifier.java:106)
        sun/tools/java/Scanner.scanIdentifier(Scanner.java:942)
        sun/tools/java/Scanner.xscan(Scanner.java:1281)
        sun/tools/java/Scanner.scan(Scanner.java:971)

TRACE 1091: (thread=1)
        java/util/Hashtable.put(Hashtable.java:405)
        sun/tools/java/Identifier.lookup(Identifier.java:106)
        sun/tools/java/Scanner.scanIdentifier(Scanner.java:942)
        sun/tools/java/Scanner.xscan(Scanner.java:1281)

TRACE 1264: (thread=1)
        java/util/Hashtable.put(Hashtable.java:405)
        sun/tools/java/Type.<init>(Type.java:90)
        sun/tools/java/MethodType.<init>(MethodType.java:42)
        sun/tools/java/Type.tMethod(Type.java:274)
```

Figure 3: HPROF Stack Traces

```
CPU SAMPLES BEGIN (total = 252378) Wed Oct 07 13:30:10 1998
rank   self  accum   count trace method
   1  4.96%  4.96%   12514   303 sun/io/ByteToCharSingleByte.convert
   2  3.18%  8.14%    8022   306 java/lang/String.charAt
   3  1.91% 10.05%    4828   301 sun/tools/java/ScannerInputReader.<init>
   4  1.80% 11.85%    4545   305 sun/io/ByteToCharSingleByte.getUnicode
   5  1.50% 13.35%    3783   304 sun/io/ByteToCharSingleByte.getUnicode
   6  1.30% 14.65%    3280   336 sun/tools/java/ScannerInputReader.read
   7  1.13% 15.78%    2864   404 sun/io/ByteToCharSingleByte.convert
   8  1.11% 16.89%    2800   307 java/lang/String.length
   9  1.00% 17.89%    2516  4028 java/lang/Integer.toString
  10  0.95% 18.84%    2403   162 java/lang/System.arraycopy
...
CPU SAMPLES END
```

Figure 4: HPROF Profile of CPU Usage Hot Spots

3 CPU Time Profiling

A CPU time profiler collects data about how much CPU time is spent in different parts of the program. Equipped with this information, programmers can find ways to reduce the total execution time.

3.1 Design Choices

We considered the following design choices when building the support for CPU time profilers: the granularity of profiling information and whether to use statistical sampling or code instrumentation.

3.1.1 Granularity

Shall we present information at the method call level, or at a finer granularity such as basic blocks or different execution paths inside a method? Based on our experience with tuning Java applications, we believe that there is little reason to attribute cost to a finer granularity than methods. Programmers typically have a good understanding of cost distribution inside a method; methods in Java applications tend to be smaller than, for example, C/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 information to Java applications is to report the dynamic stack traces that lead to the resource consumption. Dynamic stack traces become less informative in some 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.

3.1.2 Statistical Sampling vs. Code Instrumentation

There are two ways to obtain profiling information: either statistical sampling or code instrumentation. Statistical sampling is less disruptive to program execution, but cannot provide completely accurate information. Code instrumentation, on the other hand, may be more disruptive, but allows the profiler to record all the events it is interested in. Specifically in CPU time profiling, statistical sampling may reveal, for example, the relative percentage of time spent in frequently-called methods, whereas code instrumentation can report the exact number of time each method is invoked.

Our framework supports both statistical sampling and code instrumentation. Through the JVMPI, the profiler agent can periodically sample the stack of all running threads, thus discovering the most frequently active stack traces. Alternatively, the profiler agent may ask the virtual machine to send events on entering and exiting methods. Naturally the latter approach introduces additional C function call overhead to each profiled method.

A less disruptive way to implement code instrumentation is to inject profiling code directly into the profiled program. This type of code instrumentation is easier on the Java platform 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. The profiler agent may, for example, insert custom byte code sequence that records method invocations, control flow among the basic blocks, or other operations (such as object creation or monitor 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.

3.2 Thread-Aware CPU Time Sampling

The Java virtual machine is a multi-threaded 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 only when the method actually runs on the CPU, not when it is unscheduled and waiting to run. The basic CPU time sampling algorithm is as follows:

```
while (true) {
 - sleep for a short interval;
 - suspend all threads;
 - record the stack traces of all threads
   that have run in the last interval;
 - attribute a cost unit to these stack
   traces;
 - resume all threads;
}
```

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 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 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 `GetThreadTimes` 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% 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.

The cost of suspending all threads and collecting their stack traces is roughly proportional of 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 significantly reduces the profiling overhead for many multi-threaded programs in which most threads are blocked most of the time. Our experience shows that, for typical programs, the total overhead of our sampling-based CPU time profiler with a sampling interval of 1 millisecond is less than 20%.

4 Heap Profiling

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

4.1 Excessive Heap Allocation

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 follows the following steps to pinpoint the part of the program that performs excessive heap allocation:

- Enable the event notification for object allocation, so that the virtual machine issues a function call to the profiler agent when the current thread performs heap allocation.
- Obtain the current stack trace from the virtual machine when object allocation event arrives. The stack trace serves as a good identification of the heap allocation site. The programmer should concentrate on optimizing busy heap allocation sites.
- Enable the event notification for object reclamation, so that the profiler can keep track of how many objects allocated from a given site are being kept live.

4.2 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 comprehensive 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 covers all garbage collection algorithms we are currently concerned with. We introduce the abstract notion of an *arena*, in which objects are allocated. The virtual machine issues the following set of events:

- NEW_ARENA(arena 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 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 anther, a MOVE_OBJECT event is issued. All objects in an arena are implicitly freed when 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 support a wide variety of garbage collection algorithms.

4.3 Unnecessary Object Retention

Unnecessary object retention occurs when an object is no longer useful, but 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 garbage collected, 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. The heap dump contains information about all the garbage collection roots, all live objects, and how objects refer to each other.

Our profiling interface allows the profiler agent to request the entire heap dump, which can in turn be sent to the profiler front-end for further processing and analysis.

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.

5 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*. With the recent advances in monitor implementation [4] [21], non-contended monitor operations are no longer a performance issue. A non-contended monitor enter operation, for example, takes only 4 machine instructions on the x86 CPUs [21]. In properly tuned programs, vast majority of monitor operations are non-contended. For example, Table 1 shows the ratio of contended monitor operations in a number of programs. The first 8 applications are from the SPECjvm98 benchmark. The last two applications are GUI-rich programs. The monitor contention rate is extremely low in all programs. In fact, all but one program (mtrt) in the SPECjvm98 benchmark suite are single-threaded.

program	# non-contended	# contended	percent contended
compress	14627	0	0.00%
jess	4826524	0	0.00%
raytrace	377921	0	0.00%
db	53417611	0	0.00%
javac	17337221	0	0.00%
mpeg	14546	0	0.00%
mtrt	715233	11	0.002%
jack	11929729	0	0.00%
HotJava	2277113	564	0.02%
SwingSet	1587585	1332	0.08%

Table 1: Monitor Contention Rate of Benchmark Programs

5.1 Monitor Contention

Monitor contention is the primary cause of lack of scalability in multi-processor systems. Monitor contention is typically caused by multiple threads holding global locks too frequently or too long. To detect these scenarios, the profiler may enable the following three types of event notifications:

- A thread waiting to enter a monitor that is already owned by another thread issues a `MONITOR_CONTENDED_ENTER` event. This event indicates possible performance bottlenecks caused by frequently-contended monitors.
- After a thread finishes waiting to enter a monitor and acquires the monitor, it issues a `MONITOR_CONTENDED_ENTERED` event. This event indicates the amount of elapsed time the current thread has been blocked before it enters the monitor.
- When a thread exits a monitor, and discovers that another thread is waiting to enter the same monitor, the current thread issues a `MONITOR_CONTENDED_EXIT` event. This event indicates possible performance problems caused by the current thread holding the monitor for too long.

In all these three cases, 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.

5.2 Deadlocks

If every thread is waiting to enter monitors that are owned by another thread, the system runs into a deadlock situation. A thread/monitor dump is what programmers need to find the cause of this kind of deadlocks.[1] A thread/monitor dump includes the following information:

- The stack trace of all threads.
- The owner of each monitor and the list of threads that are waiting to enter the monitor.

To obtain a consistent view of all threads and all monitors, we suspend all threads when collecting thread/monitor dumps. The JDK has historically provided support for thread/monitor dumps triggered by special key sequences (such as Ctrl-Break on Win32). The JVMPI now allows the profiler agent to obtain the same information programmatically.

6 Support for Interactive Low-Overhead Profiling

The profiling support we built into the Java virtual machine achieves the following two desirable goals:

- We must be able to support *interactive profiler front-ends*. An approach that only supports collecting profiling events into trace files does not meet the needs of programmers and tools vendors. The

[1] Deadlocks may also be caused by implicit locking and ordering in libraries and system calls, such as I/O operations.

user must to enable and disable profiling events during program execution in order to pinpoint performance problems in different stages of running an application.

- The profiling support must incur *low overhead* so that programmers can run the application at full speed when profiling events are disabled, and only pay for the overhead of generating the type of events specifically requested by the profiler front-end. An approach that requires the use of a less optimized virtual machine implementation for profiling leads to additional discrepancies between the profiled environment and real-world scenarios.

Because of the low overhead of our approach, we are able to provide full profiling support in the standard deployed version of the Java virtual machine implementation. It is possible to start an application normally, and enable the necessary profiling events later without restarting the application.

6.1 Overhead of Disabled Profiling Events

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

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 JNI global reference creation and deletion. We can easily support interactive low-overhead profiling by placing checks in the corresponding code paths without having a performance impact in normal program execution.

Heap profiling events, in particular `NEW_OBJECT`, `DELETE_OBJECT`, and `MOVE_OBJECT` introduced in Section 4.2, could be quite frequent. An added check in every object allocation may have a noticeable performance impact in program execution, especially if the check is inserted in the allocation fast path that typically is inlined into the code generated by the Just-In-Time (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.

Method enter and exit events are another kind of events that may be generated frequently. They can be easily supported by the JIT compilers that can dynamically patch the generated code and the virtual method dispatch tables.

6.2 The Partial Profiling Problem

A problem that arises when profiler events can be enabled and disabled is that the profiler agent receives incomplete, or partial, profiling information. This has been characterized as the *partial profiling problem* [16]. For example, if the profiler agent enables the thread start and end events after a thread has been started, it will receive an unknown thread ID that has not been defined in any thread start event. Similarly, if the profiler agent enables the class load event 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 ID.

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 (such as class IDs) that can be sent with profiling events. The virtual machine need not keep track of outdated entities (such as a class that has been loaded and unloaded) because they will not not appear in profiling events. When the profiler agent receives an unknown entity (such as an unknown class ID), the entity is still valid, and thus the agent can immediately obtain all the relevant information from the virtual machine. We introduce a JVMPI function that allows the profiling agent to request information about unknown entities received as part of a profiling event. For example, when the profiler agent encounters an unknown class ID, 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 pro-

filing 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, for example, calling one of the JVMPI functions). 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.

7 Related Work

Extensive work has been done in CPU time profiling. The `gprof` tool [11], for example, is an sample-based profiler that records call graphs, instead of flat profiles. Recent research [7] [18] [19] 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 [5] [1]. 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 [12]. Sansom et al [20] investigated how to properly attribute costs in profiling higher-order lazy functional programs. Appel et al [2] 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 multi-threaded programs, however.

Issues similar to profiling multi-threaded programs arise in parallel programs [3] [13], 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 [22], and Modula-3 [9]. 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 [8] that target specific compiler optimizations.

There have been numerous earlier experiments (for example, [6]) on building interactive profiling tools for Java applications. These approaches are typically based on placing custom instrumentation in the Java virtual machine implementation.

8 Conclusions

We have presented a profiling architecture that provides comprehensive profiling support in the Java virtual machine. The scope of profiling information includes multi-threaded CPU usage hot spots, heap allocation, garbage collection, and monitor contention. Our framework supports interactive profiling, and carries extremely low run-time overhead.

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

Acknowledgements

We wish to thank a number of colleagues at IBM, in particular Robert Berry and members of the Jinsight team, for numerous comments, proposals, and discussions that led to many improvements in the JVMPI. We also appreciate the comments given by the anonymous reviewers of the COOTS'99 conference.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[2] Andrew W. Appel, Bruce F. Duba, David B. MacQueen, and Andrew P. Tolmach. Profling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, 1988.

[3] Ziya Aral and Ilya Gernter. Non-intrusive and interactive profiling in parasight. In *Proceedings of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems*, pages 21–30, July 1988.

[4] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 258–268, June 1998.

[5] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[6] John J. Barton and John Whaley. A real-time performance visualizer for Java. *Dr. Dobb's Journal*, pages 44–48, March 1998.

[7] Matt Bishop. Profiling under UNIX by patching. *Software—Practice and Experience*, 17(10):729–739, October 1987.

[8] Michal Cierniak and Suresh Srinivas. Java and scientific programming: Portable browsers for performance programming. In *Java for Computational Science and Engineering – Simulation and Modeling II*, June 1997.

[9] David L. Detlefs and Bill Kalsow. Debugging storage management problems in garbage-collected environments. In *USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 69–82, June 1995.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[11] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler˙ In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.

[12] Robert J. Hall and Aaron J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of Summer 1993 USENIX Technical Conference*, pages 1–13, June 1993.

[13] Jonathan M.D. Hill, Stephen A. Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool: a case study in optimizing SQL. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing – PDP'98*, pages 286–294, January 1998.

[14] Dan Ingalls. The execution profile as a programming tool. In R. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[15] Java Software, Sun Microsystems, Inc. *Java Development Kit (JDK) 1.2*, 1998. Available at `http://java.sun.com/products/jdk/1.2`.

[16] The Jinsight team (IBM Corporation). Private communication, July 1998.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[18] Carl Ponder and Richard J. Fateman. Inaccuracies in program profilers. *Software—Practice and Experience*, 18(5):459–467, May 1988.

[19] John F. Reiser and Joseph P. Skudiarek. Program profiling problems, and a solution via machine language rewriting. *ACM SIGPLAN Notices*, 29(1):37–45, January 1994.

[20] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 355–366, New York, January 1995. ACM Press.

[21] Hong Zhang and Sheng Liang. Fast monitor implementation for the Java virtual machine. Submitted for publication.

[22] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for c and lisp programs. In *Proceedings of Summer USENIX'88 Conference Proceedings*, pages 223–237, June 1988.