University of Waterloo
Faculty of Engineering

Tracing Java Operations in Hotspot

Sun Microsystems Labs
2600 Casey Avenue
Mountain View, CA, 94043
United States of America

Michael Jarrett
Student #99318764
msjarret@engmail.uwaterloo.ca
Computer Engineering - Term 3B

April 25th, 2003

Michael Jarrett
2646 Standish Drive
North Vancouver, B.C.
Canada, V7H 1N1

April 25th, 2003

Dr. Anthony Vannelli, Chair
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1

Re: 3B workterm report for Michael Jarrett (#99 318 764)

Dear Dr. Vannelli,

Attached is the report "*Tracing Java Operations in Hotspot*", submitted as my fourth work report. My last academic term was 3B, after which I began a co-op work term at *Sun Microsystems Labs*, working for the *Mayhem Project*.

The *Mayhem Project* is a project at *Sun Microsystems Labs* designing computer systems that are aware of software object-oriented constructs. This reports outlines the changes required to make a modern Java Virtual Machine output a log of events suitable for simulating such a system.

This report was prepared without assistance. It was prepared for Mario Wolczko, Senior Staff Engineer at Sun Microsystems, and manager of the Mayhem Project.

I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Michael Jarrett
99318764

Contributions

The *Mayhem Project* is a research team of three to six people at *Sun Microsystems Labs* consisting of Mario Wolczko, Greg Wright, and Matthew Seidl, and further assisted by between one and three student interns at any particular time. The goal of the Mayhem Project is to design hardware systems to support the efficient use of object-oriented software constructs. Specific goals include the design of hardware systems, and designing software algorithms to make use of this hardware. Much of the team's efforts are directed towards determining the effectiveness of such systems with realistic workloads.

Currently, the primary focus of the Mayhem group is the design of an object-aware memory system. This system supports hardware-assisted object indirection, in-cache object allocation, and in-cache garbage collection. To determine the feasibility and potential advantages of such a system, a simulator is being built capable of compiling statistics on cache performance and garbage collection. This simulator is very flexible and allows models to be built of conventional memory systems as well as Mayhem's system designs.

The simulator is driven by a specialized 'trace' of the execution of a program written in Java. These traces of Java programs are generated by a modified Java Virtual Machine capable of intercepting and recording all relevant modifications of Java heap and stacks. At the time of my arrival at Sun, the only existing implementation was based on the Exact Java Virtual Machine (JVM), a virtual machine used by Sun for their production JVM for Java 2 (version 1.2) on the Solaris platform. The Exact JVM was incapable of generating traces for many of the latest Java applications, as at least two versions of Java introducing new features have passed since the Exact JVM was in production use. This limited the selection of applications that could be used to test the Mayhem Project designs.

My task during the work term was to implement the features required to generate traces in Sun Microsystem's latest JVM, known as the Hotspot JVM. By successfully implementing these features, the latest Java applications could be traced and analyzed by our simulator. Furthermore, since Hotspot is being actively updated to meet the latest Java specifications, only minimal effort will be needed in the future to keep the modified Hotspot current with the latest Hotspot releases.

My work with Hotspot has given the Mayhem Project the flexibility to use the full range of Java applications for simulations. Also, by fully documenting the changes required to implement this functionality, maintaining and upgrading the tracing features of Hotspot will be much easier than the previous implementation.

This work report documents some of the critical differences found between the Exact JVM and the Hotspot JVM, as well as an analysis of how these affect the unique requirements presented by tracing Java applications It analyzes the solutions chosen for implementation of these features in Hotspot, and draws conclusions about the quality and usefulness of traces produced by the final implementation. It discusses whether the quality is sufficient to be used for simulations. This report is the final document presented to the Mayhem Project for that team to determine how they can make use of the modified Hotspot Java Virtual Machine.

Key to the strategy of *Sun Microsystems* is Java, an object-oriented language supported by a rich set of libraries and a virtual machine specification to support the “write once run anywhere” concept of software. Creating hardware that helps to increase performance of Java application on Sun’s line of *SPARC* processors will lead to great gains both for Java as a platform and for Sun’s extensive line of hardware solutions. The ability to simulate in detail the performance of Java applications on new architectures and new virtual machine designs will allow Sun to produce superior computing solutions.

Summary

The purpose and scope of this report is to describe the issues and trade-offs involved in updating the Hotspot Java Virtual Machine (JVM), a JVM produced by Sun Microsystems, to add the ability to trace a Java application. In order to perform detailed simulations of the workings of a system running Java applications, a record is required of everything that the application does during execution, recorded in a way that is independent of the JVM that originally recorded it.

The major points in this report are that a system exists to generate traces from Java applications, derived from Sun's Exact JVM. This virtual machine does not support the latest versions of Java, so it was decided that Sun's Hotspot JVM would be modified to provide tracing capabilities. Many differences exist between the Exact JVM and the Hotspot JVM; some break assumptions of the tracing specification, and therefore making tracing difficult. The lack of consistent regions, hefty internal use of the Java heap, optimized machine code interpreter, lack of type information in untyped bytecodes, and lack of memory abstraction layer all required changes that could affect the usefulness of the trace.

The major conclusion in this report is that the tracing features of Hotspot are sufficient for producing traces for JVM simulations. Timing statistics showed that significant overhead had been added to support tracing compared to an unmodified version of Hotspot, but that when tracing, the dominant factor was always that of file output. Measure of inconsistency statistics showed that the average number of trace events between consistent regions was several times worse in the Hotspot JVM as compared to similar traces with the Exact JVM. The use of safepoints to force consistent regions allowed the worst case to be adjusted to a chosen level. The event distribution in each of these cases showed that only marginal trace overhead was added in doing this. Event distributions also revealed a predominance of memory and bytecode operations with a distribution very similar to that of the Exact JVM.

The major recommendations in this report are that Hotspot become the main focus generating traces, that work continue with a full-time employee to improve the quality of trace and track changes to the Hotspot JVM, and that the trace format be updated to be more efficient and more implementation-neutral.

Conclusions

From the analysis in the report body, it was concluded that the trace format is sufficiently independent of a particular Java Virtual Machine (JVM) to be implemented on alternate JVMs to the Exact JVM. With some creative programming, the trace specification can be implemented in the Hotspot JVM, as demonstrated in the several successful tests on real-world Java benchmarks. It was concluded that the tracing specification made implementation difficult on other JVMs for certain events, specifically those dealing with Java Native Interface roots and untyped bytecodes.

It was concluded that the traces produced by the tracing modifications to the Hotspot JVM have sufficient flexibility in the creation of inconsistent regions for garbage collection simulations. Based on the ability to adjust the safepoint interval, most requirements for consistent regions can be satisfied.

It was concluded that the distribution of event types in the trace are close to optimal, and will not produce unnecessary overhead in simulations. The fact that the numbers also are very close to those produced by the Exact JVM give confidence to the belief that the trace is largely correct, but the smaller trace for the Javac benchmark leads to the conclusion that events may be have been missed in the Hotspot JVM.

It was concluded that performance of the Hotspot JVM tracing modifications were bounded mostly by the underlying filesystem output, and therefore quite efficient overall. Overhead without these disk operations was still significant, but not relevant to tracing support.

Overall, it is concluded that the implementation of tracing support in the Hotspot JVM has been successful enough for it to be used as a replacement for the tracing version of the Exact JVM in most applications.

Recommendations

Based on the analysis and conclusions in this report, it is recommended the that primary focus of those working on tracing support in Java Virtual Machines (JVMs) should redirect their focus towards the tracing version of the Hotspot JVM. It is recommended that as the Hotspot JVM's tracing support is refined, that traces start being produced with the Hotspot JVM instead of the Exact JVM.

It is recommended that a permanent employee takes over maintenance of the Tracing Hotspot JVM, with responsibility both for applying future changes from the core Hotspot product, and for seeking out and correcting outstanding issues in the traces produced.

It is recommended that an effort be made to improve the amount of time that the heap is consistent within a trace, concentrating on the average case instead of any particular worst case.

Finally, it is recommended that the transition to the new tracing JVM be used as an opportunity to make changes to the tracing format. The format should update several event structures to make them more implementation-neutral, and also be optimized to reduce the number of events required for tracing of actual bytecode execution. Key areas of focus should include separating JNI root behaviour from the Java stack, and addition of untyped operations to the tracing specification.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Simulating the Execution of Java Programs

Modern computer processors are extremely complex, taking years to develop from the design stage to a final implementation. Java Virtual Machines (JVMs) have also grown extremely complex, taking years to write, and entire teams of programmers to maintain. These realities can make experimentation in the area of JVMs difficult and impractical for small research projects. This is especially relevant for garbage collection research, where there is a great demand for a way to examine the behaviour of a garbage collector without the great effort involved in integrating a new garbage collector into a JVM.

As with many fields of engineering, one solution to this is to perform simulations. A simulation can be written to ignore aspects of a Java program that are not relevant to it, while examining in great deal the interactions of relevant aspects of the program's execution. For example, given a record of every new object created by an executing Java program, a simulation could easily determine the efficiency of a simulated memory allocator without ever needing to understand the logic of the program itself.

The key to performing these simulations is the need for a real JVM to execute the program and record the relevant events. In this context an 'event' describes a particular action that a simulator is interested in for the purpose of the simulation. This real JVM would handle the actual logic of program execution, which makes the implementation of the various simulators much simpler. Furthermore it allows benchmarks from different simulations to be fairly compared, as each can be driven by the exact same record of a program's execution.

1.2 Java Trace Specification

To increase the usefulness of an implementation of a 'Tracing JVM' that could produce sequences of events for simulating various aspects of JVMs, a specification was written consisting of the events that would be needed for a variety of simulations. Due to the dominant need for garbage collection research at the time, this specification concentrated on events relating to the memory operations performed by Java programs.

This specification [1] described the format and contents of a Java program's 'trace', described several types of events:

- Creation, destruction, and manipulation of local 'root structures', which are references to Java objects retained by the virtual machine and therefore uncollectable by a garbage collector.
- Manipulations of the Java stack, including individual reads and writes, as well as manipulation of stack frames.
- Manipulations of memory in the Java heap, in the forms of reads and writes to objects and arrays.
- The allocation of objects, including arrays. This explicitly does not include the destruction of objects due to the fact that garbage collection is the main focus of many of the simulators that will consume these traces.
- Thread execution behaviour, including the starting and stopping of threads, locking of objects and potential blocking on these locks.
- Class loading and definition.

Critical to the utility of this specification is its independence from the JVM that produced it. Simulations require the flexibility to make their own decisions as to how to react to particular events, and any event that relies on the underlying implementation of the JVM that produced the trace will limit the simulator to using this same technique.

An example of this is the layout of fields in an object. A JVM may choose to place fields of an object anywhere in memory. Therefore, reads and writes to object fields must be traced in a way that is at very least independent of their actual location of memory, and even independent of their relative position from the start of an object. This is particularly relevant for simulators experimenting with the reordering of fields in an object.

1.3 The Tracing Exact JVM

The Tracing Exact JVM [2] was the only known complete implementation of a JVM capable of generating all the events of the documented trace format, at the time that the work described in this report was started. It is a virtual machine derived from the Exact JVM, a JVM implementation by Sun Microsystems used for the production release of Java 1.2 on the Solaris operating system. Designed as a replacement for Sun Microsystem's original JVM, now known as the Classic JVM, this virtual machine implementation is clean and structured C code. The structure of the Exact JVM was designed with the idea of making it easily modifiable and extensible. Many operations, especially memory operations, pass through an intermediate layer where they are easily intercepted for tracing purposes.

The Exact JVM predates the specification of the tracing format, so the structure of Exact JVM did somewhat influence the structure of trace events, contrary to the stated goals of the trace format.

With Sun's release of Java version 1.3, the Exact JVM was replaced by a competing virtual machine project named Hotspot. While the Exact JVM is still maintained, it has not been updated to the latest Java specifications. The current specification of Java is version 1.4, and Sun's implementation of this is also based on the Hotspot JVM.

1.4 The Need for a Modern Tracing JVM

In May 2000, the “Java 2 Standard Edition version 1.3” specification, a major update to the Java platform, was released. This added a significant number of classes and packages to the Java Development Kit (JDK), and made some key changes that would cause applications written for this newest release of Java incompatible with older JVMs. A large internal change with Sun's implementation of Java 1.3 was the inclusion of the Hotspot JVM, replacing the Classic JVM on most platforms, and replacing the Exact JVM on Solaris.

The current version of Java as released by Sun is version 1.4, which also uses the Hotspot JVM. New classes and packages were again provided making it impossible for users of older JVMs to run new applications for this platform.

The majority of Java software currently in use, with the exception of web applets[1], make use of at least Java version 1.3 features. Many software developers for Java software have already switched to the features of Java 1.4. This causes a significant problem for the collection and simulation of "real-world" applications, because the only implementation of a Java virtual machine capable of generating traces is the Exact JVM; a JVM which only implements Java 1.2, which was replaced three years ago.

Due to these factors, a JVM was required capable of successfully tracing Java code compatible with at least Java 1.3. It was decided this would be accomplished by the modification of the Hotspot JVM to include these features. This decision will allow practically any Java application written to be traced. Changes to the Hotspot JVM in the future would also be relatively simple to apply to a modified tracing version, in a process known as 'backporting'.

[1]Web applet developers have avoided later Java versions, as key browser vendors have tended to include old or incomplete JVMs with their products.

2 Tracing Support in Hotspot versus Exact

There are some key architectural features in the Hotspot JVM which differ significantly from the Exact JVM. Each major architectural difference indicates a potential area where problems could lessen the utility of a trace.

2.1 Inconsistent Regions and Safepoints

Most forms of garbage collection (GC) require the ability to stop the execution of a program, known as "stopping the world". Once all threads have been halted, the garbage collector may run without interference from the program, then restart program execution when it is complete.

This is made more difficult by the fact that not all threads can be stopped at an arbitrary point. For example, if a thread is in the process of creating an object, and was halted between the allocation of the object and storing the pointer to this object somewhere visible to the garbage collector, this object could be collected, and the memory overwritten, despite still being needed.

The Exact JVM solves this problem by creating "inconsistent regions", as shown in figure 1. In carefully marked regions of code, the heap is considered to be in an inconsistent state, and therefore unsuitable for garbage collection. When a garbage collection is requested, all threads not in an inconsistent region are suspended. Threads in an inconsistent region are allowed to continue running until they leave the region, and are then suspended.

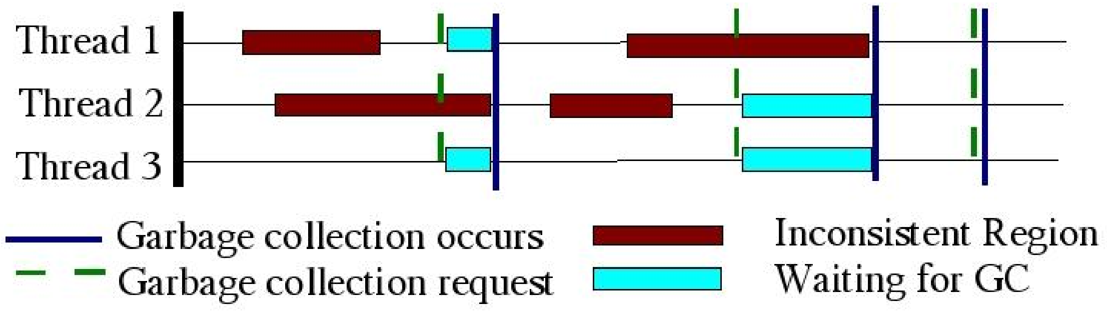

Figure 1: Inconsistent Regions in the Exact JVM

Hotspot takes a different approach, as shown in figure 2, where each thread is assumed to be inconsistent except at specific instants in the execution. When a garbage collection is requested, threads continue to execute until reaching one of these points, where they will suspend themselves. Once all threads have suspended, Hotspot has reached a "safepoint", and may perform operations requiring all threads to be stopped, such as garbage collection.

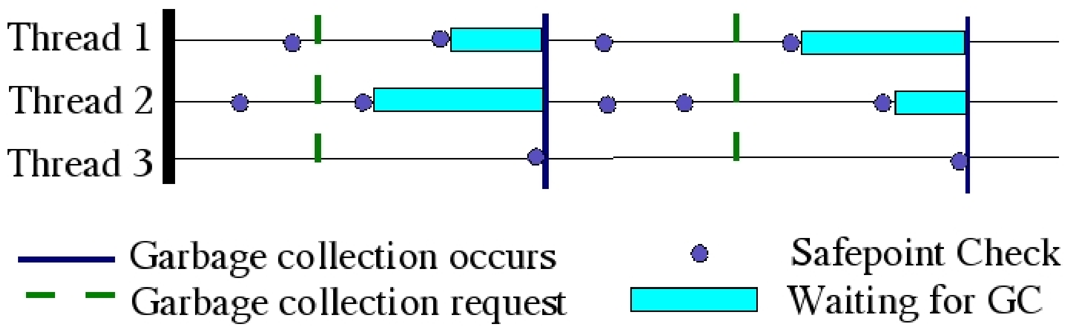

Figure 2: Inconsistent Regions in the Hotspot JVM

Inconsistency is of particular interest to a simulation, since it is unable to control execution of a thread. It must rely on all threads becoming consistent at the same time within a reasonable period of time after it has requested a garbage collection. While both the Exact JVM and the Hotspot JVM can guarantee this at any time it chose to garbage collect, the Exact JVM can only make a best effort attempt to keep threads consistent at any other time. Hotspot is even worse, where it is guaranteed that the only time all threads are consistent are exactly when the Hotspot JVM chose to request a safepoint.

2.2 Interpreter Loop

In its simplest form, a bytecode interpreter is essentially a state machine, taking action based on the current bytecode and current JVM state, updating the JVM state, and producing outputs.

The interpreter in the Exact JVM is written in C as a loop and a switch statement. This approach is straightforward and easy to manipulate.

Hotspot takes a different approach, in an attempt to deliver improved efficiency. Each bytecode is generated as a "template" at runtime in machine code. Separate versions of these templates are generated based on the state of various caches upon entry to the template. All of these are collected and stored in a table called the "dispatch table". Using the next bytecode as an index, each template has an instruction appended to jump to a location stored in this table, which will immediately begin execution of the next bytecode.

The most significant implication of this is that it is much more difficult to manipulate complex structures from the interpreter. In fact, even a basic call to a C++ method requires a complex set of storage operations to prevent the fragile state of the interpreter from being disturbed from compiled C++ code. This becomes more difficult with the widespread assumptions in the interpreter of exactly where certain pieces of code are allowed to call C++ at all, and if they may block while there. It is very difficult to modify the interpreter while not disturbing program state.

2.3 Java Memory Accesses

To allow for later modification and extension, the Exact JVM included the "memory subsystem layer" - a common interface for all code that modifies the Java heap or stack. This abstraction can be implemented using C preprocessor macros, and can therefore be optimized out at compile time. However, in the case of tracing support, all accesses to memory can be easily intercepted, and trace events generated if required.

Hotspot does the exact opposite of this - not only does it have no memory layer, there is little or no shared interface for structure access at all! The interpreter itself literally uses load and store instructions at the machine code level to manipulate both the stack and objects. From C++ code, there are very few rules, with hefty manipulation of pointers for array access, and at least four unique ways in which one can manipulate static fields in a class.

2.4 JNI Roots

The Java Native Interface (JNI) is a method by which Java programs may execute compiled machine code specific to a platform within the execution content of the JVM. This code must store references to Java objects through a level of indirection, so that the references themselves are in a location known to garbage collectors. These are known as JNI roots.

A common approach, used by the Exact JVM, is to store JNI roots on the Java stack. This approach allows the JNI roots to be processed by the same code that scans the stack for references.

The Hotspot JVM allocates these roots in special-purpose blocks of memory, with one such block allocated per thread. Should one become full, more roots can be allocated in a new block, which is then chained to the first in a singly linked list fashion.

The trace format expects, and requires, that JNI roots are allocated on the stack, and not interfere with the objects already there. Commands to request new frames of JNI roots require the push of a new native stack frame. This can significantly complicate management of JNI roots, since roots may conflict with objects on the stack if they are not actually allocated there. Furthermore, efforts must be taken to ensure incorrect manipulations of JNI root frames in native code do not negatively affect the stack for the application.

2.5 Heap Objects

Java does not have a concept of a pointer that can be directly manipulated. Objects and arrays are manipulated in a Java program through means of references, with the underlying implementation of these references a decision left to the JVM designer. Based on early Smalltalk research [3] in the 1980's, most JVMs how use pointers directly to a heap of Java objects to implement references for efficiency reasons.

However, maintaining both a C++ heap for internal objects and a Java heap for objects, and allowing both to grow to arbitrary sizes, can be difficult to manage. Hotspot solves this by having a common heap for both Java objects and objects internal to the virtual machine. Furthermore, Java objects, arrays, and several other forms of object that cannot be directly manipulated by a Java program, all derive from common base of functionality. This allows all objects derived from this base, including internal ones, to be garbage collected.

However, this makes the separation of internal JVM data structures from objects allocated by a Java program difficult, since many of the mechanisms are shared. The most common offender are arrays, where the Java array mechanisms are used extensively both in Java code and in internal JVM structures.

2.6 Untyped Operations

Java is a typed language, meaning that every operation on a variable must be valid for the type of the variable. This variable type must be known at compile time, as Java bytecodes change based on the type of the variable manipulated.

There are some exceptions to this, the most important being a set of Java bytecodes described by the Java Virtual Machine Specification [4], which do not depend on type. Several of these bytecodes, including the 'dup' series of bytecodes, rely on manipulating fixed chunks of memory (normally in 32-bit words) rather than by Java types. The trace represents all memory operations as typed operations, so this information needs to be preserved.

One way to escape this problem is to track at all times what the type of the operands on the Java stack are. Since the maximum size of the Java operand stack for a particular method is fixed at compile-time, this can be tracked in structures allocated on method entry.

A more efficient way to handle this would be to allow the representation of untyped stack operations in the trace format itself. Since untyped operations are part of the definition of a JVM, there is no reason to abstract these away in the trace format.

3 Implementation Decisions

3.1 Safepoint Intervals

Hotspot has a debugging feature that forces safepoints to be triggered at a fixed time intervals, even when no garbage collection is required. By enabling this feature, and then recording events to make it appear that all threads become consistent at this point, the heap will become briefly consistent in a predictable, periodic fashion. While there will never be sequences of events that are consistent, each instant of consistency will be a possible point where a simulated garbage collector could collect.

This has the advantage of providing a guarantee as to the worst case of the number of events that a simulated garbage collector could have to wait before beginning a garbage collection. Furthermore, this parameter can be adjusted to suit a certain situation. However, the worst case and the average case will be the same, unless other techniques are adopted.

3.2 Consistency Through Thread State

Hotspot makes some assumptions about thread consistency when attempting to reach a safepoint.

- A thread is only allowed (by programmer convention) to block with a consistent heap. Therefore, if the thread is prevented from resuming execution, it may be considered to be at a safepoint.
- A thread executing native code (excluding code internal to the Java virtual machine) cannot directly modify Java state, and is therefore always consistent. It may be considered to be at a safepoint as long as it is prevented from leaving native code.

Both of these assumptions can be used to create larger consistent regions. No events can occur while in native code (since the JVM has no control over it), again leading to isolated points of consistency. The blocking is significant however, since threads often spend long periods of time blocking. This keeps threads which are not executing from keeping the state of the heap from appearing inconsistent in a trace.

3.3 Object Tracing Levels

Determining whether or not an event is internal to the JVM or an action triggered by a Java program is a significant problem since many of the mechanisms are shared between Java code and the JVM, including the use of object locks for synchronization, and heap objects in general. A fine line has to be drawn, especially in the case of internal synchronization on structures, as these, while being dependent on the JVM implementation, can directly affect the execution of the Java program.

To this effect, each object in the Java heap is given a score at the time of creation. This score represents how relevant an object is, and is affected by the context in which the object is created and the type of the object. High scores are given to objects that are created as the result of an explicit request by a Java program, while low scores are given to objects which are allocated by the JVM independently and stored in internal data structures.

All the events which can appear for both JVM actions and Java program actions involve at least one object. This object's trace level can then be used as a filter to ensure only objects relevant to the Java program's execution end up in the final trace.

3.4 Untyped Operations

To represent as accurately as possible what the Hotspot JVM actually does in this case, these are treated as a sequence of integer copies. This may cause problems with some simulators, but will work with those that represent stacks in a contiguous block of memory.

Special attention was paid to that of objects, since the data written in these cases in abstracted to object identifiers. For these cases, the code checks a potential pointer to see if it points to an area within the heap and is correctly aligned for an object. If these conditions are met, the object ID is read from the structure pointed to, and if valid, written to the trace.

While this can fail for certain bytecode sequences, the uses of unsigned operations with Sun's Java compiler are limited to certain sequences of operations, in which the assumptions used by Hotspot actually result in correct type information in all but pathological cases.

Regardless, a more flexible solution would involve modifying the trace specification to represent these operations.

4 Implementation Results

Several measurements were used to determine the success of the Hotspot tracing modifications. Key factors that may affect the usefulness of the new tracing version of the Hotspot JVM are the execution performance of the JVM while tracing, the distribution and number of consistent regions in the trace, and the distribution of different event types within the trace.

The benchmarks were all run on a Sun 440R machine with four 450MHz processors, 4 GB of RAM, and running the Solaris 9 operating system. The following benchmarks were used.

- A basic "Hello, World!" Java application. This benchmark will always be dominated by the JVM initialization process.
- The SPECJVM[5] benchmark suite, using the "_213_javac" benchmark at size 10 from the suite. This is a benchmark of an extended computation without requiring a large number of JDK libraries.
- Jedit[6] version 4.0.3. This is a graphical text editor, with numbers being drawn from executing and then immediately leaving the editor. This application demonstrates successful execution of Java 1.3 version programs, and gives statistics for an application heavy on JDK class usage.

4.1 Execution Performance

Tracing support in the Hotspot JVM comes at a steep performance cost. Even discounting the actual time taken for writing traces out to disk, several factors cause the performance of Hotspot to decrease.

- To escape the need to add tracing support to the two just-in-time (JIT) compilers in Hotspot, both of these are disabled. The interpreter, while quite efficient, cannot match the speed of compiled code.

- Each tracing event is guarded at runtime by a check of a configuration flag, which will be performed quite frequently, and can actually be a significant fraction of the time spent in a bytecode template.
- Even when tracing no events, all objects, stack frames, classes, and roots must be assigned unique identifiers. These will also have to be copied when objects are relocated during garbage collection.

The effect of safepoints on performance is of particular concern, since this may allow for the adjustment of inconsistency statistics. Figure 3 shows how the execution of Javac is affected by safepoint interval. From a period from 25ms to 6400ms, no measurable change in performance is visible beyond measurement noise. This means that the safepoint interval can be adjusted for ideal consistency and event distribution conditions without concern for performance.

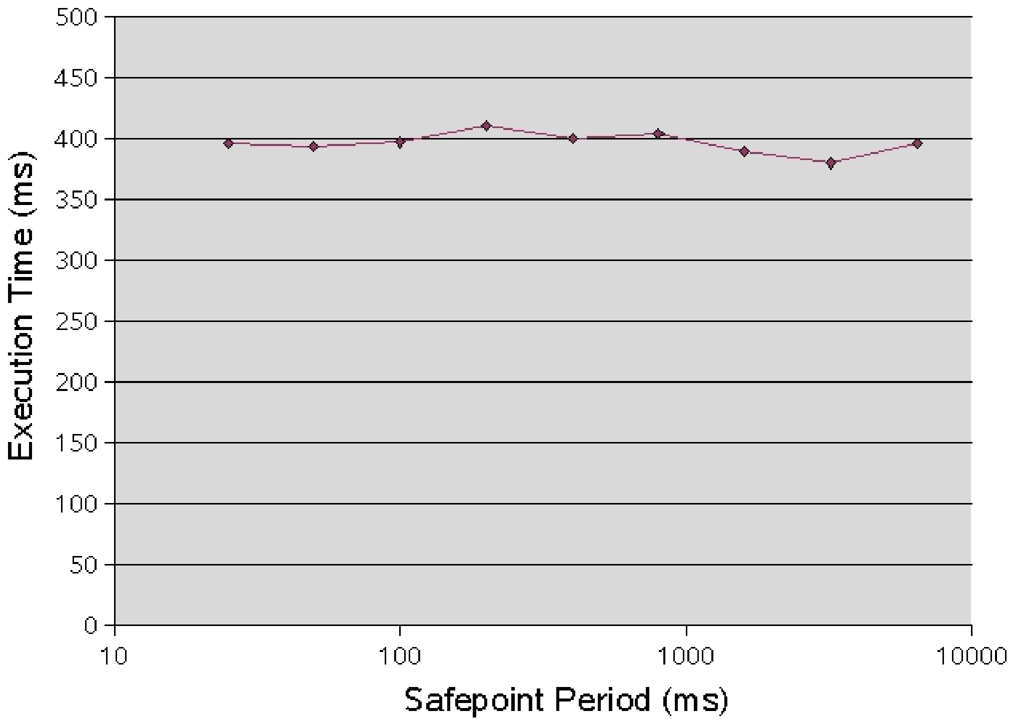

Figure 3: Graph of Execution Time versus Safepoint Interval

Table 1 shows a comparison of performance between the Exact JVM while tracing and the Hotspot JVM while tracing. Comparing the numbers between the modified and unmodified Hotspot, we can see that tracing support does indeed pose a overhead, even when disabled, but that this can vary widely depending on the application. The Hotspot

JVM seems to significantly lag in performance compared to the Exact JVM in Hello World, but excels at Javac. However, by examining the output trace file, we see that Hello World file is also significantly larger when produced by the Hotspot JVM. This is most likely due to a larger initialization cost for the additional libraries supported. The javac trace was, conversely, somewhat smaller when produced by the Hotspot JVM, indicating that the Exact JVM may be producing excess events, or the Hotspot JVM not enough. The exact sizes are show in section 4.3, which shows the number of events generated for each JVM. It shows that performance depends strongly on the underlying attempts to write to disk. Jedit was not run in the Exact JVM, as it cannot trace Java 1.3 applications.

Table 1: Execution Times

Benchmark	**Exact (Tracing)**	**Unmodified Hotspot**	**Tracing Hotspot (Not tracing)**	**Tracing Hotspot (Tracing)**
Hello, World	5 s	< 1 s	< 1s	13 s
Javac size 10	13 min	4 s	12 s	11 min
Jedit	-	20 s	24 s	29 min

Overall, the numbers give a sense that the Hotspot JVM tracing support is somewhat slower than that of the Exact JVM's, but that both are limited to the speed of file output. This performance is quite acceptable for tracing, and can be improved by using machines with faster disk drives.

4.2 Inconsistent Regions

The amount of time spent in a consistent state determines the flexibility of the simulator to choose points to garbage collect. While a real JVM can manipulate threads to begin a GC soon after it desires one, the simulator does not have any control on the order of events, and therefore must rely on all threads becoming consistent at the same time.

Table 2 shows the inconsistency statistics for both the Exact JVM and the Hotspot JVM, with Hotspot using the default tracing safepoint interval (SPI) of one second. The application traced is a standard "Hello, World" application, which even with a one-second safepoint interval, will have several safepoints over the execution of the program.

Table 2: Inconsistency Statistics for Hello World

Trace	**% Events Consistent**	**Avg. Events Inconsistent**	**Max. Events Inconsistent**	**% Consistency Events**
Exact - Hello, World	0.945%	232	358,838	0.854%
Hotspot (SPI: 1s) - Hello World	0.141%	1256	342,998	0.160%

The data is not promising: both the average number of events between consistent regions and the percentage of events consistent are several times worse in the Hotspot JVM than the Exact JVM. The only advantage is in the number of consistency events, where smaller number of consistent regions reduced the number of consistency events.

Hotspot has the ability to change the period of forced safepoints in tracing mode. These can significantly affect the results, at the cost of more inconsistency events.

Figure 4 shows the results of shrinking the safepoint interval on the average number of events between consistent regions. Even at 1ms (the minimum forced safepoint period), the average is still several times that of the Exact JVM, and quickly stops growing by 500ms. In higher cases, consistency from other events, such as blocking operations, likely dominate, preventing the average from growing indefinitely.

Figure 5 shows the longest measured time to reach a consistent region. This varies linearly with the safepoint interval. At 800ms, the Hotspot JVM is already significantly ahead of the Exact JVM, and continues to drop to approximately ten times the average case for 1ms. The forced safepoints break up regions that would otherwise have nothing to force them consistent, and ensures that all threads do this at exactly the same time. In the worst case, Hotspot can always make guarantees to the number of events between safepoints, while there are no such guarantees with inconsistent regions in the Exact JVM.

Clearly, while the average case for consistency events may have positive implications for the effectiveness of the simulator of simulating GC, the simulator must be able to handle the worst case if it is to operate. Therefore limiting the number of events in the longest inconsistent region is essential to generating useful traces. Here, the Hotspot JVM has

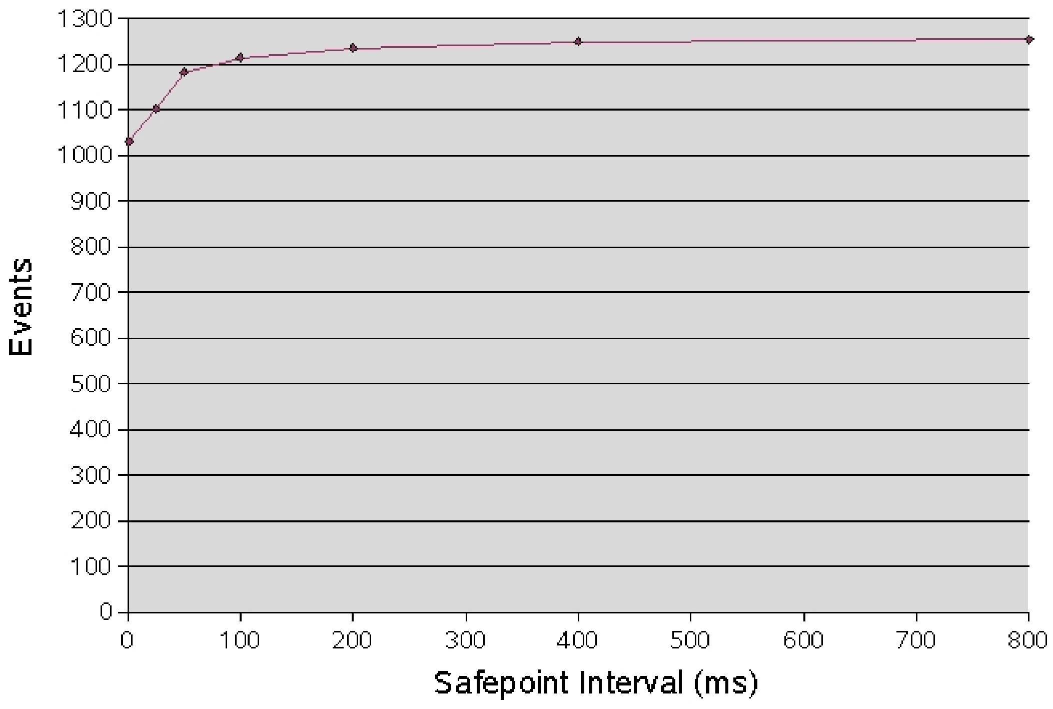

Figure 4: Average Events Between Consistent Points

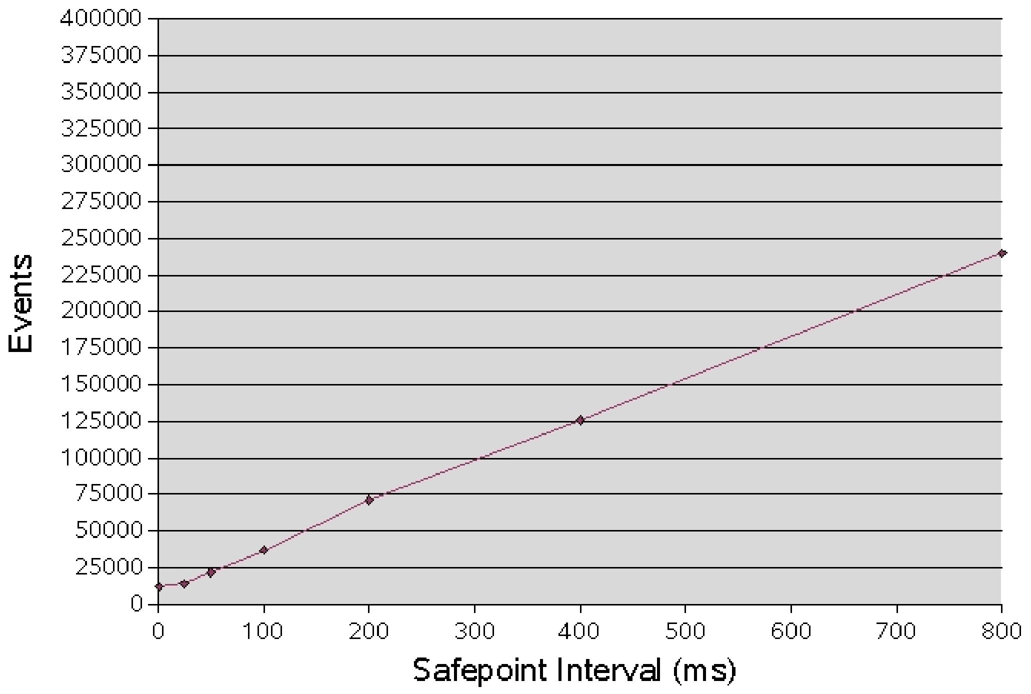

Figure 5: Maximum Events Between Consistent Points

proven itself successful.

4.3 Event Distribution

Distribution of event types can indicate the relative efficiency of a trace. Generally, a predominance of root management or inconsistency events will mean a trace can take longer to produce, and longer for a simulator to consume it. Conversely, a trace consisting of mostly allocation and memory events will tend to derive useful information quickly.

Table 3 shows the distribution of events for the Exact JVM and the Hotspot JVM for the "Hello, World application". Table 4 shows the same statistics for the Javac benchmark. The first point of note is that the event distributions are within a couple of percent across applications and even across virtual machines. This is most likely due to the nature of Java itself, where each bytecode will likely be followed by a number of stack and field events. This also indicates that, assuming that most memory events are stack and field events, that the vast majority of events in each case (over 90%) are the direct result of bytecode execution.

Table 3: Event Distributions for Hello, World

Category	**Exact Events**	**Hotspot Events**
Memory Events	1,027,196 (68.4%)	2,375,207 (65.6%)
Bytecode Events	425,364 (28.3%)	1,123,177 (31.0%)
Root Events	3797 (0.253%)	10,257 (0.283%)
Consistency Events	12,815 (0.854%)	5792 (0.160%)
Other Events	32,287 (2.15%)	104,659 (2.89%)
TOTAL	1,501,459	3,619,092

Table 4: Event Distributions for Javac

Category	**Exact Events**	**Hotspot Events**
Memory Events	155,077,867 (68.4%)	127,545,610 (64.5%)
Bytecode Events	63,146,851 (27.8%)	62,090,312 (31.4%)
Root Events	63,576 (0.0280%)	64,858 (0.0.0328%)
Consistency Events	603,490 (0.266%)	514,894 (0.260%)
Other Events	7,864,512 (3.47%)	7,534,005 (3.81%)
TOTAL	226,756,296	197,749,679

We also notice that the Hotspot JVM produces a great deal more events on the Hello, World benchmark, but significantly less on Javac. Hello, World is easily explained by the larger set of classes that need to be loaded for the Java 1.4 JDK. No explanation can be found for the smaller Javac trace. There may be some gains made by optimizations made to bytecodes by the JVM at runtime, but it is also possible that there are events missing. This would be a serious problem, and merits further investigation.

Glossary

bytecodes The native instructions of a Java implementation. These instructions are very similar to the machine instructions found in many processors, but without the ability to directly manipulate memory through pointers.

GC *garbage collection* The process by which the memory used by objects that can no longer be found by the system is freed for later use.

JDK *Java Development Kit* Equivalent to a software development kit (SDK) for other platforms, the JDK is a set of class libraries that provide critical classes to a Java program's operation. The JDK provides a rich set of classes for a variety of tasks from thread management to 3d rendering.

JIT *Just-In-Time compiler* A technique used by many modern virtual machines, where bytecodes can be translated directly into equivalent operations in machine instructions of the underlying hardware, and thus executed much more efficiently.

JNI *Java Native Interface* A description of a method by which Java bytecode and machine code may interact. It specifies a set of interfaces and structures that can be used to interact with a Java Virtual Machine when machine code is running.

JVM *Java Virtual Machine* A piece of software that emulates a machine that executes Java bytecodes, and provides services such as class loading and thread scheduling.

References

[1] M. Wolczko, *Using a Tracing Java^{TM} Virtual Machine to gather data on the behavior of Java programs*, tech. report SML 98-0154, Sun Microsystems Laboratories, Mountain View, CA, 1999.

[2] M. Wolczko, "Tracing VM Description," *Welcome to www.ExperimentalStuff.com*, http://www.experimentalstuff.com/Technologies/TracingJVM/index.html (current April 2003).

[3] D. Ungar and D. Patterson, "What Price Smalltalk," *IEEE Computer*, vol. 20, no. 1, Jan. 1987, pp. 67-74.

[4] T.Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Specification Second Edition*, Addison-Wesley, Reading, MA, 1999.

[5] "SPEC JVM98," *Standard Performance Evaluation Corporation*, http://www.spec.org/jvm98/ (current April 2003).

[6] "jEdit - Open Source programmer's text editor," *jEdit - Open Source programmer's text editor*, http://www.jedit.org/ (current April 2003).