

USENIX Association

Proceedings of the
Java™ Virtual Machine Research and
Technology Symposium
(JVM '01)

Monterey, California, USA
April 23–24, 2001



© 2001 by The USENIX Association

All Rights Reserved

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>

Rights to individual papers remain with the author or the author's employer.

Permission is granted for noncommercial reproduction of the work for educational or research purposes.

This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

The HotSpot™ Serviceability Agent: An out-of-process high level debugger for a Java™ virtual machine

Kenneth Russell¹ and Lars Bak²
Sun Microsystems
{Kenneth.Russell, Lars.Bak}@eng.sun.com

Abstract

The HotSpot™ Serviceability Agent (SA) is a set of APIs for the Java programming language which model the state of Sun Microsystems' Java™ HotSpot Virtual Machine. Unlike most previous debugging systems for dynamic languages which assume a “cooperative” model in which the target process runs code to assist in the debugging process, the SA requires no code to be run in the target VM. Instead, it uses primitives like symbol lookup and reading of process memory to implement its functionality. The SA can transparently examine either live processes or core files, making it suitable for debugging both the VM itself and Java programs in production. We describe the design and implementation of the SA, comparing it to other debugging systems for both statically compiled and dynamic languages, and illustrate future directions for this architecture.

1 Introduction

The Java HotSpot Virtual Machine implementation (hereafter referred to as the HotSpot JVM) is Sun Microsystems' high-performance VM for the Java platform³. HotSpot's origins are in language research in Beta [18], Smalltalk [6], and Self [12] [13]. The HotSpot JVM uses many advanced techniques to achieve high performance. The system employs a mixed-mode assembly interpreter which shares the stack with both C code and Java programming language methods (Java methods) compiled to machine code. Runtime profiling focuses compilation effort only on “hot” methods. Dynamic deoptimization [14] allows a compiled method to revert back to the interpreted state if invariants under which the method was compiled are compromised by future class loads. The ability to deoptimize allows the compiler to perform aggressive optimization and inlining.

How does one debug the product version of a highly-optimizing JVM which is written largely in C or C++ ?

¹901 San Antonio Road, M/S UCUP02-302, Palo Alto, CA 94303.

²Computer Science Department, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark.

³An arbitrary Java virtual machine implementation is hereafter referred to as a JVM.

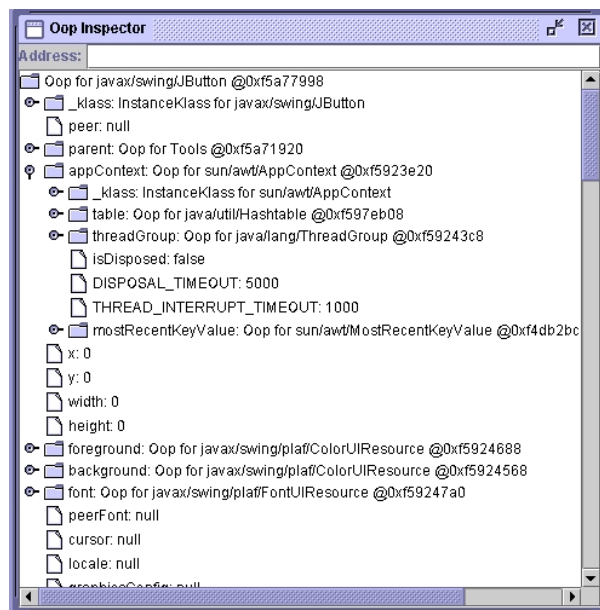


Figure 1. An object inspector built with the SA's APIs.

Such a JVM

- tends to operate with generated machine code, and to merge the C++ and Java virtual machine stacks (Java stacks). Activations on the stack corresponding to invocations of Java methods (Java frames) show up in a C++ debugger as raw program counters with no corresponding symbol.
- may have a highly-optimizing compiler. An activation on the stack may correspond to more than one Java method invocation, because of inlining.
- may encode many of its run-time data structures to save space.
- will not have available debug information for the C++ data structures describing portions of the run-time system such as the layout of the heap.

In short, when one examines the product version of such a JVM with a traditional C++ debugger, one deals with raw bits. All of the high-level abstract data types are gone.

The HotSpot Serviceability Agent is a set of APIs for the Java programming language which recover this high-level state from a product-mode HotSpot JVM or a core file. Clients of the SA can use its APIs to write application-specific tools, debugging aids, and querying operations which operate directly on a target JVM and in a completely non-intrusive fashion; Figure 1 illustrates an object inspector built on top of the SA's APIs. Unlike most debuggers for dynamic languages, the SA requires no code to be run in the target JVM. It is therefore robust in the face of JVM failures. This property is what allows it to operate as a post-mortem JVM debugger. The SA is also applicable to more situations than just debugging the JVM; for example, end users can use it to write heap profilers which operate on servers in production without taking them down.

The rest of the paper is organized as follows. Section 2 compares the SA to other work in the field of debugging. Section 3 describes its contributions. Section 4 gives a concrete example of walking the JVM's thread list, and describes how the SA's APIs relate to the target JVM. Section 5 describes how the SA obtains type and offset information for C++ data structures in the target JVM. Section 6 details the low-level debugging primitives used by the SA. Section 7 explains the traversal of the remote JVM's heap, and gives an example of an object histogram. Section 8 describes how the SA walks the remote JVM's Java stacks. Section 9 discusses future work.

2 Related Work

We partition related work into two categories: debuggers for statically compiled languages (e.g., C, C++, Fortran, Modula-2) and "dynamic languages", which are typically characterized by the ability to load and/or compile code at run time and which typically have a substantial run-time system including a garbage collector.

The two standard UNIX debuggers, dbx [17] and gdb [25], assume the model of an *uncooperative* debugging target. In this model it is assumed that the target process may be dead, in which case only examination-only operations are allowed. While both debuggers have a "call" facility which allows an arbitrary procedure to be executed in the target process if it is alive, a large percentage of the debugging tools are available for use on core files. These debuggers are language-independent, but have only been developed in the realm of statically-compiled languages like C, C++, Fortran, and Modula-2.

Higher-level debugging tools for statically compiled languages have been developed as well. Sosic [23] describes a set of APIs which were used to implement debuggers in the Dynascope distributed system; these APIs map closely to much of the functionality provided in dbx and gdb. The notion of modeling the contents of the re-

mote process's heap is introduced with an example of copying a linked list from one process to another assuming that the two processes are written in the same language. While language independence is claimed, no language other than C is described. Gough, Ledermann and Elms [11] describe a substantially different type of debugger which is targeted toward fully optimized machine code. Their system works in a "forward" fashion, using a modified compiler to evaluate expressions in the context of optimized code, rather than attempting to annotate the generated code with debugging information. This system also claims language independence, but was developed for statically-compiled languages like C, Pascal and Modula-2 [7]. Both of the systems described in Sosic and Gough et al. use an agent embedded in the target process to implement all of the functionality, thereby requiring that the target process be alive.

Debuggers for dynamic languages generally assume a *cooperative* debugging model; that is, the debugger cooperates with the execution environment (or virtual machine or "VM") for the language in order to allow debugging of programs written in the language. The debugger may be implemented in-process as in Perl [1], Python [29], and Scheme [2] or out-of-process using a wire protocol as in Spinellis' two-process Prolog debugger [24] or as in the Java Platform Debugger Architecture [19]. Some debuggers for safe languages like Tolmach and Appel's Standard ML debugger [27] and the Objective Caml debugger [8] take advantage of the safety of the target language to provide facilities not commonly found in debuggers for statically-compiled languages, such as the ability to step backward through the execution of the program. Gill's post-mortem debugger for Haskell [10] is in the same vein as the SA an examination-only system, but is post-mortem in the sense that the Haskell program has terminated abruptly, not the entire run-time system. All of these systems assume that code can be run in the target VM. Squeak [15] was developed and debugged using itself; the debugging methodology used was to prototype and debug in Smalltalk and then translate to C for high performance. Debugging a crashed Smalltalk VM is not discussed. Lisp systems, like the Allegro Common Lisp environment [9], often have the capability to dump an executable image. However, it appears that it is not possible to debug such a dumped image without running code in it.

The Jalapeño project has developed a JVM written in the Java programming language which is based around compiling Java bytecode to machine code [5]. Because nearly all of the code being executed is generated by the system itself, a traditional (C or C++) debugger is of no help in examining data structures. The debugging system for Jalapeño [21] uses the same mechanisms used by dbx, gdb, and the SA to attach to a target JVM and read its process memory without running code in the tar-

get. A unique aspect of the Jalapeño debugger is that since both the debugger and the JVM being debugged are written in Java, a modified Java interpreter can be used to execute the debugger’s code which allows the JVM’s data structures to debug themselves. This aspect of the Jalapeño debugger goes beyond what the SA can currently do, since the HotSpot JVM is implemented in C++.

3 Contributions

Compared to previous work, the HotSpot Serviceability Agent is closest in design to the Jalapeño debugger [21], which is also constructed around examination-only, out-of-process operations. In addition, the Jalapeño system, by taking advantage of the fact that the debugger and target JVM are implemented in the same language, achieves a level of elegance not currently possible with the SA.

The HotSpot Serviceability Agent was designed to be able to diagnose JVM failures. This requirement informed several design decisions, including that no code is run in the target process. The SA is currently an examination-only system, meaning that it derives all of its information using low-level primitives like symbol lookup and reading memory from a remote process. This allows it to work both by attaching to a running process as well as reading a core file. It can also run its code in an arbitrary JVM.

Compared to the Jalapeño work, this paper contributes a deeper discussion of debugging in the face of JVM failures, especially regarding heap and stack traversal. It describes a new architecture for interacting with remote Java objects which is similarly applicable to a run-time system. It discusses cross-language issues which arise when the target JVM is not implemented in Java. Finally, it analyzes the modeling process used to describe JVM data structures and discusses possible alternative implementations.

Because of the architectural similarities between the SA and the Jalapeño debugger, throughout this paper we explicitly point out some of the differences between the two systems. Chief among these is that the SA has been used to successfully diagnose difficult bugs in the production HotSpot JVM, whereas Jalapeño is a research system.

4 Example: Walking the Thread List

The HotSpot JVM maintains information indicating what kind of code each Java thread is executing: JVM-internal code, “native” code [16], or Java code. Consider the simple example of traversing the target JVM’s thread list and determining this information.

Figure 2 provides a diagram of the interaction between the SA and the target JVM. The JVM uses the C++

data structure in 2A to describe the thread list. A static pointer in class `Threads` points to the head of the list. The memory layout of the resulting objects in the target JVM is shown in 2B. Figure 2C illustrates the analogous code from the SA. The SA does not actually use hard-coded offsets, as is described and illustrated later.

The APIs in the SA mirror the C++ structures. When an object created by the SA’s code models an object in the target JVM, it fetches data from the target using the `Address` abstraction, which contains the illustrated methods as well as those which fetch strongly-typed, Java-sized primitive data like byte `getJByteAt(long offset)` and short `getJShortAt(long offset)`.

5 Describing C++ Types

In order to avoid hardcoded offsets in the SA’s code (as used in Figure 2), it is necessary to model the types and structures of C++ objects in the target JVM, so that fields can be fetched by name. There are several ways that this structural information can be obtained, the most widely used and automatic being to have the C++ compiler generate debugging information such as *stabs* [17] during the build. These annotations are stored in the object file and resulting linked shared object or executable and can be parsed by the C++ debugger on the same platform.

The HotSpot JVM contains some classes which have fields defined to be the same sizes as some of the Java primitive types; for example, the C type `jshort`, defined by the JNI specification [16], is identical in size and signedness to the Java type `short`. In order to reduce the possibility of accidental errors in the SA’s code, it was necessary to understand which fields in C++ types were actually “Java fields” and expose their contents as the correct Java primitive types in the SA.

The nature of the process by which *stabs* and other debug information are generated typically loses such type information; for example, a `jshort` may be typedefed to a C `short int` or similar, and the debug information for a data structure containing a field of that type will identify the given field as a `short int`, not a `jshort`. This information loss was deemed unacceptable in the type modeling process for the SA.

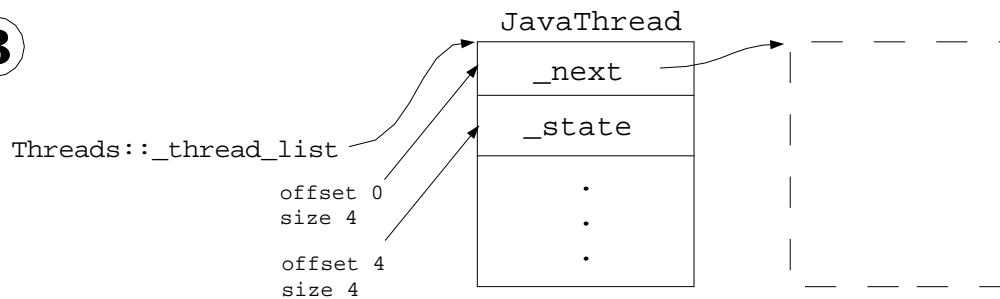
For this reason, the structural information provided by the target JVM to the SA is presented in the form of three tables which are generated by C++ preprocessor macros and compiled into the target JVM. We illustrate only the first of these tables in Figure 3A. This table contains name, type, and offset or address information for nonstatic and static fields, respectively. The second table models the C++ inheritance hierarchy, and the third provides named integer constants needed by the SA to properly traverse data structures. The SA uses a general symbol lookup mechanism to locate these tables and

A

```

class JavaThread {
    JavaThread* _next;
    // _in_vm, _in_native, _in_Java
    JavaThreadState _state;
public:
    JavaThread* next()      { return _next; }
    JavaThreadState state() { return _state; }
};

```

B**C**

```

public class JavaThread {
    private Address addr;
    public JavaThread(Address addr) { this.addr = addr; }
    public int state() {
        // offset, size, isUnsigned
        return (int) addr.getCIntegerAt(4, 4, false);
    }
    public JavaThread next() {
        Address nextAddr = addr.getAddressAt(0);
        if (nextAddr == null) return null;
        else return new JavaThread(nextAddr);
    }
}

```

Figure 2. Illustration of the mirroring of the JVM's data structures in the SA. (A) shows a subset of the JVM's `JavaThread` code, including the state of the thread (in JVM code, in native code, or in Java code) and the structure of the thread list. (B) illustrates the memory layout of this data structure in the JVM's address space; starting with the global thread list, `JavaThread` objects are linked together. (C) shows the SA code which accesses this data structure. Access to the start of the list is not shown.

```

A typedef struct {
    const char* typeName; // Type name containing field (Example: "JavaThread")
    const char* fieldName; // Field name within type (Example: "_state")
    const char* typeString; // Quoted type of field (Example: "JavaThreadState")
    int32_t isStatic; // Effectively a boolean
    uint64_t offset; // Used for nonstatic fields
    void* address; // Used for static fields
} VMStructEntry;

extern "C" VMStructEntry* gHotSpotVMStructs; // The table itself
// The following offsets describe the layout of the table for bootstrapping
extern "C" uint64_t gHotSpotVMStructEntryTypeNameOffset;
extern "C" uint64_t gHotSpotVMStructEntryFieldNameOffset;
...

B public interface TypeDataBase {
    public Type lookupType(String name);
}

public interface Type {
    // Java fields:
    public JByteField getJByteField(String fieldName)
        throws WrongTypeException;
    ...
    // C fields:
    public CIntegerField getCIntegerField ... ;
    public AddressField getAddressField ... ;
}

C public class JavaThread {
    private Address addr;
    private CIntegerField stateField;
    private AddressField nextField;
    static {
        TypeDataBase db = VM.getVM().getTypeDataBase();
        Type type = db.lookupType("JavaThread");
        stateField = type.getCIntegerField("_state");
        nextField = type.getAddressField("_next");
    }

    public JavaThread(Address addr) { this.addr = addr; }
    public int state() {
        return (int) stateField.getValue(addr);
    }
    public JavaThread next() {
        Address nextAddr = nextField.getValue(addr);
        if (nextAddr == null) return null;
        else return new JavaThread(nextAddr);
    }
}

```

Figure 3. Modeling of C++ types in the SA. (A) shows the structure of one of the JVM's three type tables and the global symbols used to access and describe it. (B) highlights the salient aspects of the TypeDataBase and Type interfaces. (C) shows a portion of the actual SA code; compare to the hardcoded offsets in Figure 2.

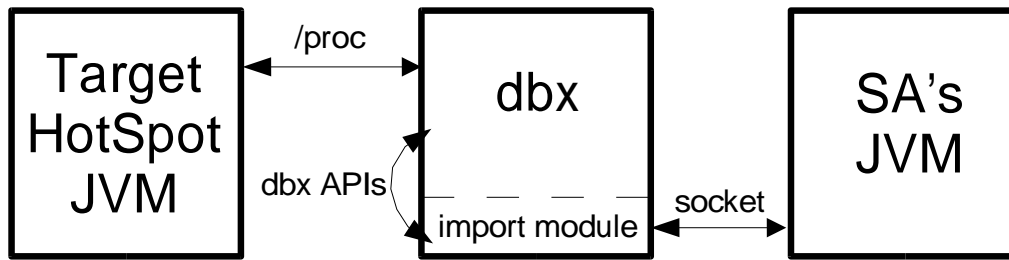


Figure 4. Block diagram of connections between the SA and the target HotSpot JVM on Solaris.

parses them upon attaching to the target JVM or core file, effectively downloading the information it needs to analyze the remote JVM.

This approach has two additional advantages over using the compiler's raw debug information: first, since the symbols being referenced in the SA are explicitly exported by the JVM, changes to the JVM code which are incompatible with the SA's code are often encountered at build time as a failure during compilation of the side tables. Second, it reduces porting time by eliminating the need to understand the platform-specific debugging information generated by the C++ compiler.

The structural information for C++ types is stored in a `TypeDataBase` (Figure 3B), which supports looking up a `Type` by name. From the `Type`, `Fields` can be fetched by name. The `Field` mechanism provides type checking between the SA's code and the target JVM, in addition to ensuring that field offsets are always kept synchronized as in Figure 3C.

Because the SA models C++ types as first-class Java objects, the relatively simple current implementation of the type system can be straightforwardly extended in the future.

6 Access to the Remote Process

The SA is built on top of a few very simple debugging primitives, including

- attach to remote process or core file
- lookup symbol in remote process
- read remote process memory

The Solaris™ Operating Environment (hereafter Solaris) version of the SA uses the native debugger, `dbx`, to obtain this functionality, encapsulating it in a small interface called `Debugger`.

When the SA is launched, it uses `java.lang.Runtime.exec()` to launch a subordinate `dbx` process. It sends commands to `dbx` to load and execute a small piece of self-contained C++ code, called an *import module* (Figure 4), and either to attach to the target HotSpot JVM (causing it to be suspended, as with any program being debugged) or to load a core file. The

`import module` communicates with `dbx` via a small set of internal APIs and with the SA using a custom text-based protocol over a socket, and provides the SA with the above debugging primitives. The SA itself is written entirely in Java.

The only significant additional mechanism that has been added to the SA's `Debugger` interface as the system has evolved is the ability to query the CPU register set for a given thread. The need for this is described in Section 8.

Compared to the Jalapeño debugger, the SA explicitly models the interaction with the remote address space. This was a requirement since the target JVM was not implemented in Java and, therefore, JVM data structures could not be directly repurposed (and still write the SA in Java). Explicitly describing the interaction with the remote address space conveys two additional advantages: first, it describes failures using the Java language (for example, by throwing an `UnmappedAddressException`), and thereby allows the debugging system to become more robust where necessary: see Section 7. Second, it opens up the possibility for language-independent debuggers written in Java; for example, the SA's modeling of C++ types is the first small step toward writing a C++ debugger. This is discussed further in Section 9.

7 Traversing the Heap

The SA exposes HotSpot's generational [28] garbage collection framework in an abstraction called the `ObjectHeap`. The most significant operation provided by this abstraction is the ability to uniformly visit all fields in all objects in the heap. These objects include both Java objects as well as JVM-internal reflective objects like methods and classes. For this reason objects are termed `Oops`, a term borrowed from the Smalltalk community.

Figure 5 illustrates the iteration mechanism provided to clients. Implementing debugging and profiling tools as shown in Figures 1 and 6 requires very little additional code.

The iteration mechanism can be made robust in the face of JVM failures. Consider the situation where a bad

```

// An OopVisitor can be used to inspect
// all fields within an object.
// Fields include vm fields, java
// fields, indexable fields.

public interface OopVisitor {
    // Called before visiting an object
    public void prologue();

    // Called after visiting an object
    public void epilogue();

    public void setObj(Oop obj);

    // Returns the object being visited
    public Oop getObj();

    // Callback methods for each field type
    // in an object
    public void doOop(OopField field);
    public void doByte(ByteField field);
    public void doChar(CharField field);
    public void doBoolean(BooleanField field);
    public void doShort(ShortField field);
    public void doInt(IntField field);
    public void doLong(LongField field);
    public void doFloat(FloatField field);
    public void doDouble(DoubleField field);
    public void doCInt(CIntField field);
}

```

Figure 5. The OopVisitor interface, which provides uniform iteration over all objects in the heap.

pointer has been stored in an object field of a Java object. Constructing an Oop for the value stored in that field will likely result in an `UnmappedAddressException` or `UnalignedAddressException`. Client code, for example in an object inspector, can explicitly catch these exceptions and raise a red flag in the user interface if one arises. It is not clear from the description of the Jalapeño debugger how similar functionality is provided, since it seems that such exceptions are not modeled explicitly in the Java language, but instead handled somehow by the underlying remote-aware interpreter.

8 Traversing the Stacks

A debugger is hardly useful if it does not provide detailed information about where the program has failed. In the context of debugging a static program, for example from a core file, this means providing a stack backtrace of all threads. Figure 7 shows the trace for one thread.

The HotSpot JVM necessarily has internal abstractions for describing and walking the stack; it performs

type-precise garbage collection [4], and must therefore be able to walk the stacks of all Java threads at GC time, precisely locating all pointers to live Java objects and updating those pointers if the object is moved during GC.

The HotSpot JVM uses a safepointing mechanism [3] to halt execution of interpreted or compiled Java code only when the locations of all objects are known precisely. (Recent work has described how to eliminate the need for safepoints for the purposes of garbage collection [26], but HotSpot also uses safepoints for other system-wide operations such as deoptimization.) Metadata generated by HotSpot’s compilers describes the state of compiled code in great detail at safepoints, including any inlining the compiler may have performed.

The code in HotSpot’s run-time system was designed to operate at safepoints. In a debugging scenario, however, the JVM will in general not be at a safepoint; it may be suspended, or may have crashed, at an arbitrary point in its execution.

The SA’s stackwalking code was ported from the JVM and modified over time to handle problems which arise when the JVM is not at a safepoint:

- the JVM may not have metadata for a program counter in compiled Java code. In this case the metadata associated with the closest possible PC is used, unless there is no such information (as in a leaf method), in which case the iteration code assumes that there is, for example, no inlining.
- an interpreter frame may not have been set up yet, yielding an incorrect bytecode index. In this case the iteration code skips the topmost frame on the stack.
- a thread executing generated machine code may have been interrupted by a signal.
- the topmost Java frame may not be available to the run-time system at all.

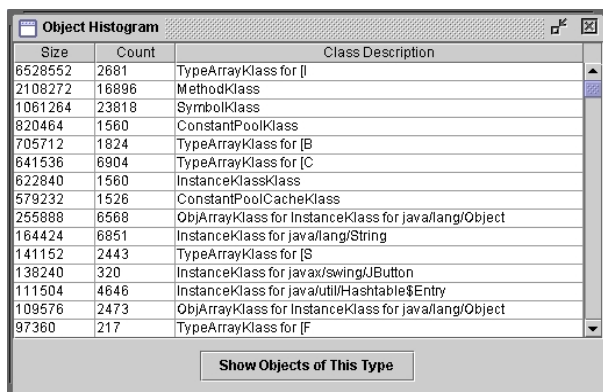


Figure 6. Histogram of objects in the heap.

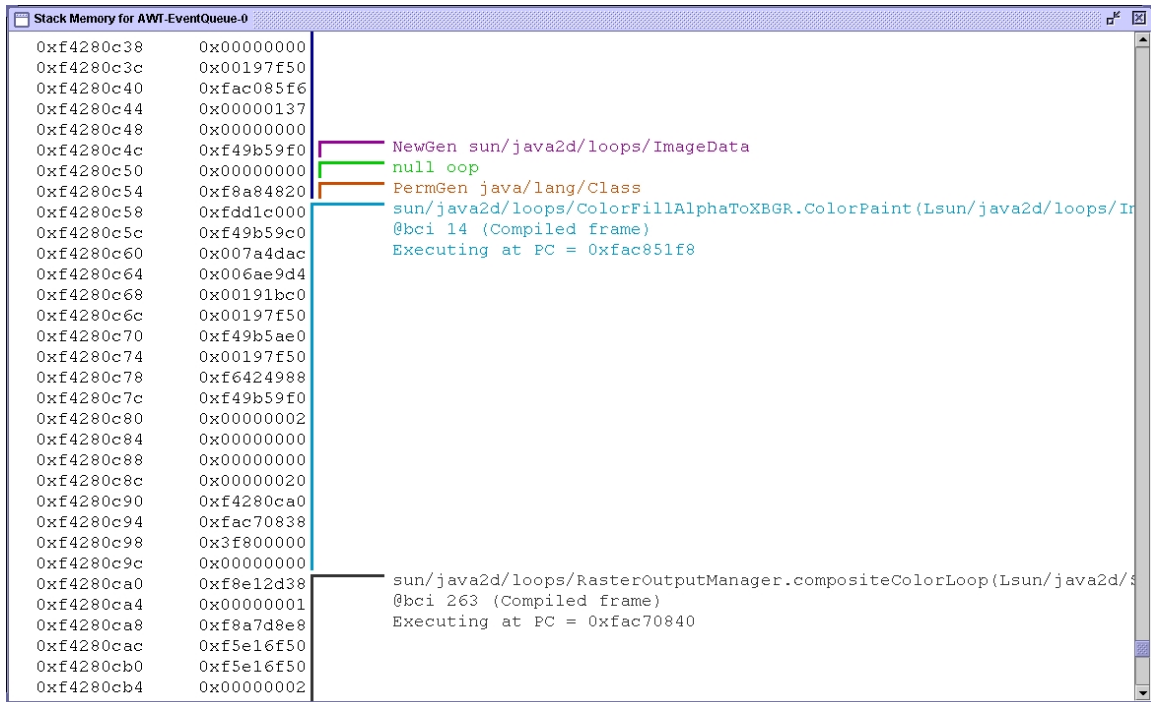


Figure 7. Stack memory annotated with Java frames and live objects.

The latter problem is the most significant. When the JVM reaches a safepoint, each thread currently executing Java code enters the run-time system, storing its last known Java stack pointer into thread-local storage in preparation for stack traversal. The JVM's run-time code traverses only Java frames, skipping around sequences of C frames. In a debugging scenario the entire topmost sequence of Java frames for a given thread will be unreachable with the JVM's built-in stackwalking code if the thread was suspended (or crashed) while executing either interpreted or compiled Java code.

The SA solves this problem by using the Debugger interface to fetch the last known stack pointer for a given thread (Section 6). Given that stack pointer, it must be able to walk backward to find the first Java frame on the stack, proceeding as usual from that point on. We have found that the JVM's run-time code is basically adequate to handle this technique on the SPARC architecture, but expect that in a forthcoming port to the x86 we will have to implement a more general stackwalking mechanism to handle cases where the frame pointer has been eliminated, as described by Linton [17].

A full description of the technique for recognizing and traversing signal handler frames on the stack is beyond the scope of this paper. We briefly note that we have implemented a not-quite-satisfactory mechanism on the SPARC architecture as an extension to the SA's port of the JVM's run-time code.

Our experience has been that implementing stackwalking in the context of a debugging system has been

non-trivial. The Jalapeño debugger rightly emphasizes its reuse of JVM code in examining the remote JVM's data structures. We have found, however, that in order to handle and diagnose JVM failures, significant divergence from the JVM's code is required in the area of stackwalking.

9 Discussion and Future Work

The Serviceability Agent is similar in architecture to the Jalapeño debugger. The most interesting aspect of the latter is that the same code can both implement and debug the JVM's run-time system. A valid criticism of the SA is that it requires duplication of JVM code, since the SA and JVM are implemented in different languages (and necessarily with different underlying architectures). We have found a few unexpected instances of version skew which silently broke the SA. For the most part, however, the table-based symbol export mechanism described in Section 5 provides early warning when the SA will break. Roughly 13,000 lines of the SA's 35,000 lines of code must track the VM with varying degrees of closeness.

Compared to other previous work, the SA appears to be the first system which has the ability to debug in a high-level manner data structures from both static languages like C++ and dynamic languages like Java. Other dynamic languages could be targeted by the SA by using its debugging primitives to model the run-time system of the target VM, as has been done for HotSpot. This is

currently a labor-intensive process but has the advantage of providing a true post-mortem debugger.

The SA operates either on a JVM that has been suspended by platform-specific debugging mechanisms or on a core file. Solaris contains a program called `gcore` which takes a core file snapshot of a running process without terminating that process; it is frequently used in conjunction with `dbx` to examine the state of production systems written in C or C++ rather than attaching a debugger directly [22]. The SA allows `gcore` to be used for Java programs as well. The speed of the SA's access to the target JVM's heap is substantially slower than that enabled by the current standard API, the Java Virtual Machine Profiling Interface [20]. However, JVMPI imposes a significant performance overhead to the normal running of the application. Used in conjunction with `gcore`, the SA only has the impact of suspending the process while taking the snapshot, which takes a few seconds for programs of normal heap size.

An early design decision was to use only examination-only debugging primitives, to make the goal of the system clear and to explore what was possible to build using them. If the target JVM is alive (not a core file) and running properly, being able to write to the process memory would facilitate more rich interactions when debugging. Some of the possible features, like setting fields, would be straightforward to implement and would not require JVM code changes, while others, like setting breakpoints, would likely require a substantial amount of supporting code in the JVM. An advantage of the SA's architecture is that it could degrade gracefully back to an examination-only system if the JVM crashed.

In the future, we plan to complete support for Solaris/x86, Linux, and Win32. We will continue to expand the SA's knowledge of JVM data structures. We plan to write more tools using the SA's APIs, and to explore the addition of debugging primitives which affect the state of the target JVM. We will investigate if it would make sense from a performance or footprint standpoint to rewrite portions of the JVM's run-time system in Java.

10 Acknowledgments

We thank Ivan Soleimanipour and Siva Annamalai for their help with the system's use of `dbx`; our managers, Jerry Driscoll and Tricia Jordan, for their support of the project, especially in its early stages; the entire HotSpot group for encouragement; and Robert Griesemer in particular for reviewing early drafts of this paper. We also thank the anonymous reviewers for their helpful comments.

References

- [1] CPAN: comprehensive perl archive network, 2000.
<http://www.cpan.org/>.
- [2] The Scheme programming language, 2000.
<http://www.swiss.ai.mit.edu/projects/scheme/index.html>.
- [3] O. Agesen. GC points in a threaded environment. Technical Report SMLI-TR-98-70, Sun Microsystems Laboratories, December 1998.
- [4] O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. *ACM SIGPLAN Notices*, 33(5):269–279, 1998.
- [5] B. Alpern, D. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, November 1 1999.
<http://www.research.ibm.com/jalapeno/publication.html#oopsla99-jvm>.
- [6] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference record of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.
- [7] K. Elms. Debugging optimised code using function interpretation. In Mariam Kamkar, editor, *AADEBUG '97, Proceedings of the Third International Workshop on Automatic Debugging*, pages 27–36, Linköping, Sweden, May 26-27 1997.
- [8] X. Leroy et al. The Objective Caml system release 3.00, 2000.
<http://caml.inria.fr/ocaml/htmlman/>.
- [9] Franz Incorporated. Online documentation in html as part of the Allegro Common Lisp 5, July 1998.
- [10] A. Gill. Haskell object observation debugger, 2000.
<http://www.haskell.org/hood/>.
- [11] K. Gough, J. Ledermann, and K. Elms. Interpretive debugging of optimised code, 1994.
- [12] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP'91*, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

- [13] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94 Conference Proceedings*, pages 326–335, Orlando, FL, June 1994.
- [14] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimised code with dynamic deoptimization. In *Proceedings of the SIGPLAN '92 Conference on Programming Languages Design and Implementation, SIGPLAN Notices*, pages 32–43, San Francisco, CA, June 1992. ACM Press.
- [15] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*, pages 318–326, 1997.
- [16] Sheng Liang. *The Java Native Interface*. Addison-Wesley, Reading, MA, 1999.
- [17] M. A. Linton. The evolution of dbx. In *Proceedings of the 1990 Usenix Summer Conference, Anaheim, CA*, 1990.
- [18] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, 1993.
- [19] Sun Microsystems. Java platform debugger architecture, 2000.
<http://java.sun.com/j2se/1.3/docs/guide/jpda/>.
- [20] Sun Microsystems. Java virtual machine profiling interface, 2000.
<http://java.sun.com/j2se/1.3/docs/guide/jvmpi/>.
- [21] T. Ngo and J. Barton. Debugging by remote reflection. In *Proceedings of Euro-Par 2000*, Munich, Germany, August 27-September 1 2000.
<http://www.research.ibm.com/jalapeno/publication.html#remotereflect>.
- [22] I. Soleimanipour and S. Annamalai. Personal communication, 2000.
- [23] R. Sosic. A procedural interface for program directing. *Software: Practice and Experience*, 25(7):767–787, 1995.
- [24] D. Spinellis. The design and implementation of a two process Prolog debugger. Technical Report IR-LP-31-21, ECRC, September 1989.
- [25] R. Stallman and R. Pesch. Debugging with GDB: The GNU source-level debugger, 1999.
- [26] J. Stichnoth, G. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler, 1999.
- [27] A. P. Tolmach and A. W. Appel. Debugging Standard ML without reverse engineering. In *ACM Conference on Lisp and Functional Programming*, pages 1–12, 1995.
- [28] D. M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, 1984.
- [29] G. van Rossum et al. Python library reference: The Python debugger, 2000.
<http://www.python.org/doc/current/lib/module-pdb.html>.