Implementing Jalapeño in Java

Bowen Alpern alpern@watson.ibm.com *
C. R. Attanasio dick@watson.ibm.com*
John J. Barton john_barton@hpl.hp.com †
Anthony Cocchi tony@watson.ibm.com*
Susan Flynn Hummel hummel@watson.ibm.com*
Derek Lieber derek@watson.ibm.com*
Ton Ngo ton@us.ibm.com*
Mark Mergen mergen@us.ibm.com*
Janice C. Shepherd janshep@us.ibm.com*
Stephen Smith steve@watson.ibm.com*

www.research.ibm.com/jalapeno

Abstract

Jalapeño is a virtual machine for Java^{TM} servers written in Java.

A running Java program involves four layers of functionality: the user code, the virtual-machine, the operating system, and the hardware. By drawing the Java / non-Java boundary below the virtual machine rather than above it, Jalapeño reduces the boundary-crossing overhead and opens up more opportunities for optimization.

To get Jalapeño started, a *boot image* of a working Jalapeño virtual machine is concocted and written to a file. Later, this file can be loaded into memory and executed. Because the boot image consists entirely of Java objects, it can be concocted by a Java program that runs in *any* JVM. This program uses reflection to convert the boot image into Jalapeño's object format.

A special Magic class allows unsafe casts and direct access to the hardware. Methods of this class are recognized by Jalapeño's three compilers, which ignore their bytecodes and emit special-purpose machine code. User code will not be allowed to call Magic methods so Java's integrity is preserved.

A small non-Java program is used to start up a boot image and as an interface to the operating system.

Java's programming features — object orientation, type safety, automatic memory management — greatly facilitated development of Jalapeño. However, we also discovered some of the langauge's limitations.

1 INTRODUCTION

In December 1997, a group at IBM's T. J. Watson Research Center began a three month pilot study to determine the feasibility of writing a virtual machine in Java^{TM} [8]. At the end of that period, we had a minimal Java virtual machine (JVM) [14] running, convincing us that such a JVM was both feasible and desirable.

This led to a full-scale project with the goal to invent, construct, and evaluate the best technology for Java virtual machines on large computational servers. (A server JVM's requirements — high-performance, scalability in an SMP environment, continuous availability, rapid response, graceful degradation under load, and support for a very large number of threads — that are less important for client, embedded, or personal systems JVM's. On the other hand, the real memory constraints on a server JVM are not as tight.) A subsidiary goal is to create a flexible research platform on which to investigate and evaluate novel virtual machine ideas. Portability is explicitly *not* a design goal — we feel obliged to exploit any performance advantage available on our target architecture: PowerPC [15] multiprocessors running the AIX operating system [10]. Realizing, however, that we may someday want to port Jalapeño to a different architecture, we strive to make Jalapeño as portable as possible without compromising performance.

There are advantages and challenges to building a JVM in Java. The major development advantages are those that follow from using a modern, object-oriented, type-safe, and memory-safe programming language. In addition, we hope to realize two kinds of performance advantages. First, we need no extra code to adapt the call stack between user code and frequently called runtime services: conventional JVM's implement these services using "native" methods (typically, written in C, C++, or assembler) incurring an overhead adapting Java state to native state for each call. Second, Jalapeño's seamless operation allows simultaneous dynamic optimization of user code and runtime services. Finally, we hoped that implementing Jalapeño in Java would give us more experience with the language, help us to identify some of its problematic features, and give some insight into how to implement them efficiently.

The primary challenges to building Jalapeño in Java involve: creating a minimal JVM that can be expanded to a complete system, bootstrapping the system into operation, bending restrictions of the Java language without compromising its integrity, and exploiting — and sometimes carefully skirting — the advanced features of the language by the very code that implements them.

Development of Jalapeño is guided by a performance driven methodology that seeks to avoid premature optimization. Initially, simple mechanisms are used to implement required functionality. As these mechanisms are measured and identified as performance bottlenecks, they are replaced with more sophisticated techniques. This methodology, while generally followed, is not rigorously adhered to: some forethought is used to avoid pervasively poor performance. (Also,

*IBM T. J. Watson Research Center, PO box 218, Yorktown Heights, NY 10598.
†Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304-1126.

skilled programmers do not always resist the temptation to take some methodological shortcuts.)

The design of Jalapeño is presented elsewhere [3, 1]. This paper relates our experience building Jalapeño in Java. An appendix presents a brief overview of Jalapeño's design, motivating its data layout and its compilation, memory management, and multithreading strategies.

Section 2 describes our minimal JVM more or less as it existed at the end of our pilot study, including a baseline compiler, a class loader with dynamic loading capability, two primitive garbage collectors, and support for bootstrapping and debugging. This minimal JVM is the seed from which Jalapeño evolved.

Section 3 explains the process of bootstrapping Jalapeño into operation. This is done by building the image of a working system in *another* JVM, writing it to a file, loading it into memory, and branching to the start of a static `boot()` method.

Section 4 outlines the mechanism that allows Jalapeño to evade some of Java's constraints while protecting Jalapeño's users from these evasions. Implementation of Jalapeño's runtime, memory management, and multithreading subsystems require circumventing Java's type system and memory model. Jalapeño's compilers employ a novel approach to effect these transgressions and to limit their use.

Section 5 reviews those few components of Jalapeño that are *not* implemented in Java, and explains why we believe they cannot, or should not, be.

Section 6 considers related work, including two other JVM's that have been written in Java [6, 18]. These run on top of other virtual machines. Jalapeño runs on bare metal (with modest operating system support).

Section 7 discusses the limitations we found in Java as a language for writing Java virtual machines.

Section 8 reviews our experience writing Jalapeño in Java.

2 AN INITIAL JVM

Our implementation philosophy was to get something running as soon as possible and then to extend and enhance it. We knew this would involve considerable rework and reorganization as the system evolved, but we felt that design and architectural issues would surface sooner and at a point when rework was possible and not too costly.

The initial design for Jalapeño is depicted in figure 1. This figure dates from December 1997. If you squint a little, the outlines of the mature Jalapeño are clearly visible. The "POOF!" cloud indicates the output of the boot-image writer discussed in section 3. Method invocation stacks are now allocated as ordinary objects (arrays of `int`s). Code generation is performed by Jalapeño's three compilers. The "*Executor*" has become Jalapeño's virtual processors. And, we have yet to tackle JNI.

We started writing code from the beginning and the first "proto-JVM" was running a few weeks into the project. This first system could only generate a boot image and boot itself (see section 3), execute a few simple bytecodes, and return. It quickly evolved into a single-threaded system with a simple class loader, an object allocator (no garbage collection), a baseline compiler that translated an ever enlarging subset of Java's bytecodes, and a primitive debugger. The debugger's functionality was sufficient to allow us to verify that the machine code produced by the baseline compiler for each bytecode performed as intended.

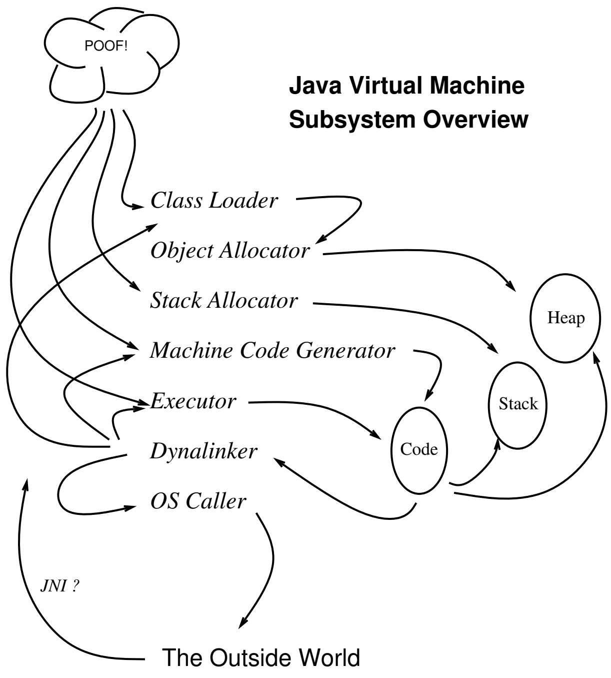

Figure 1: The initial design of Jalapeño.

The remainder of this section describes our JVM as it existed about the beginning of March 1998.

2.1 The baseline compiler

The baseline compiler translated bytecode to machine code by religiously implementing the stack-based definition of the Java virtual machine [14]. (Compilation is performed in a single-pass using an pseudo-assembler containing a mechanism for resolving forward branches and emitter methods for each PowerPC instruction we required.)

This approach has much to commend it. It was very easy to implement. This greatly contributed to early development of our minimal JVM. The compiled code was simple enough that we were reasonably confident that we could "get it right." It also allowed graceful integration and validation of Jalapeño's other compilers. When one of these encounters a situation it can't handle yet, it throws an exception and the offending method gets compiled by the baseline compiler. The close correspondence between machine code and the original bytecode allowed Jalapeño's debugger to exploit information in Java's classfiles. Finally, since the baseline compiler (like all of Jalapeño) is written in Java, it runs on either a traditional JVM or on Jalapeño. This was crucial for our bootstrapping strategy.

Unfortunately, the code produced by the baseline compiler performs poorly. The simple assignment `a = b + c` gets translated as follows. Local variable `b` is loaded into a register, then pushed on the stack. Local variable `c` is loaded into a register, then pushed on the stack. The values of `b` and `c` are loaded from the stack back into registers. Their sum is formed in a register and then stored on the stack (adjusting the stack pointer). This value is popped from the stack into a register, then stored in local variable `a`.

This poor code quality motivated development of Jalapeño's optimizing and quick compilers. These compilers make much more effective use of registers than the baseline compilers. Following our development methodology, we were forced to change our calling conventions [1] to pass parameters in registers rather than on a stack in May and June of 1998. This was a painful transition. Particularly tricky to implement was reflective method invocation.

2.2 A class loader

The first version of the system had only static linking: all classes had to be loaded before the JVM started executing. A primitive class loader ran in the boot-image writer that loaded these classes from a given list before any code was compiled. Since all the necessary class information was available during compilation, no dynamic linking code was required.

When the baseline compiler had progressed to the point that this class loader could run in the booted image, we added support for dynamic linking. When the baseline compiler encountered a bytecode (e.g. **getfield**) that mentioned a class that had not yet been loaded, the compiler emitted code to call a runtime method that would do the linking. This method would first verify that the class had been loaded. (If not, it would load it.) Then, it would overwrite its call site with code that performed the desired operation (so that the linking operation would only be performed once). Finally, it branched to the newly back-patched code. We tripped on a common problem of dynamic linking at this point: we failed to flush the instruction cache properly. Further intricacies of dynamic linking in a multi-threaded multiprocessor environment are addressed in a separate paper [2]. Suffice it to say here that this code was severely error prone and that the resulting bugs were difficult to track down. (In ironic mockery of our professed development methodology, a simpler, but less efficient, table-based scheme for dynamic linking was developed for the optimizing compiler which is less tolerant of register usage by the back-patching code.)

2.3 Minimal memory managers

The initial object allocator was quickly replaced by a memory manager with a simple copying garbage collector. This collector was neither type accurate nor conservative. It could perhaps be charitably characterized as *optimistic*: any value that could be interpreted as an address within the JVM's image was treated as an object reference. It would happily move an "object" pointed to by a large integer ("updating" the value of the integer in the process).

We briefly considered modifying the baseline compiler to emit code for integer operations that checked that the result was not a valid address. However, at this stage Jalapeño didn't produce any large integers anyway and the copying manager was soon joined by a conservative non-copying memory manager. By the end of March, reference maps for the baseline compiler allowed both managers to be made type accurate.

2.4 The debugger

We recognized that a customized debugger would be necessary, especially in the early phases of the project. Jalapeño's debugger is a Java program. It runs on a conventional JVM as a separate AIX process. It connects to Jalapeño through a JNI interface to AIX's `ptrace` facility. The initial version supported only register and memory display and breakpoints at absolute addresses.

Stackframe and traceback displays were implemented. Support was added to set breakpoints on entry to specified methods and at line numbers in source files. The baseline compiler supplied a simple map of bytecode indices into an array of corresponding machine instructions. The debugger exploits this and the debug information in Java's classfiles to support symbolic names for classes, methods, fields, and local variables.

To provide similar functionality for dynamically loaded classes, a customized interpreter now executes the debugger. Although this interpreter runs on a different JVM, it operates on Jalapeño's internal data and methods. This allows the debugger to use Jalapeño's own methods to interrogate the contents of its data structures.

3 GETTING STARTED

A fairly substantial set of services — a class loader, an object allocator, a compiler — must exist before a JVM can load all remaining services required for normal operation. The initial services for a JVM written in native code or one that runs on top of another JVM are available from an underlying runtime. Jalapeño is not written in native code and it has no underlying runtime. Therefore, we assemble the essential core services into an executable *boot image* prior to running the JVM. This boot image is a snap-shot of a Jalapeño virtual machine written into a file. Later, this file is loaded into memory and executed.

The boot image is created by a Java program called a *boot-image writer*. It constructs a mockup of a running Jalapeño virtual machine and then packages it into a boot image. The boot-image writer is an ordinary Java program and it can run on any JVM. The JVM that runs the boot-image writer will be called the *source* JVM, and the resulting Jalapeño virtual machine, the *target* JVM.

The boot-image writer resembles a cross-compiler and linker: it compiles bytecodes to machine code and rewrites machine addresses to bind program components into a runnable image. However, since Jalapeño's compilers, class loaders, and runtime data structures are all in Java, it, unlike most compilers, must also bind live objects into the boot image.

The boot-image writer instantiates, in the source JVM, Java objects that represent the target JVM. Then it uses Java's built-in reflection facility to translate these mockup objects from the object model of the source JVM to Jalapeño's object model. This self-referencing aspect of the boot-image writer makes it relatively simple — it's really just an object-model translator.

Since Jalapeño is a Java program, each of its components is a Java object and the boot-image writer can construct the mockup by executing special `init` methods in each of Jalapeño's major subsystems. A custom classloader makes sure that any classes needed to execute this code are loaded into the mockup as well as into the source JVM. As a class is loaded, its methods are compiled (by the baseline compiler running in the source JVM) and included in the mockup.

This strategy of loading classes into both the source JVM and its mockup of the target JVM requires a complete class list to succeed. If, when Jalapeño starts running, a method of the core runtime references any class not in the boot image, an endless recursion results: the runtime needs to load part of itself in order to load part of itself ... and so on.

The problem of determining the minimal set of classes needed in the mockup to prevent this was solved using a combination of careful planning and trial and error. All of Jalapeño's core classes were named with a `VM_` prefix. These are the classes needed to provide enough machinery to allow the virtual machine to perform compilation, memory management, and dynamic class loading. The special prefix is recognized by Jalapeño's compilers and used to suppress normal dynamic linking rules: they never generate dynamic linking code between methods whose classes have this prefix. The core classes were also carefully written to avoid unnecessary use of Java library classes. The fundamental classes — `java.lang.Object`, `java.lang.Class`, `java.lang.String`, and a few I/O classes — were unavoidable exceptions. Together, the `VM_` classes and fundamental Java classes formed a starting set of classes that we thought needed to appear in the boot image.

A small number of additional dependencies (for example, `Integer`, `Float`, `Double`, and various array and exception classes) were then identified by trial and error. We built a boot image and attempted to execute it. If it crashed trying to (recursively) load class `X`, then we added `X` to the list of classes written into the boot image and repeated the exercise. This process converged with a small number of retries and did not prove to be a maintenance problem once the implementation of the core `VM_` classes stabilized.

When the mockup is complete it is transformed into a boot image. This involves finding all the objects in the mockup, converting them to Jalapeño's object format, and storing them in a *boot-image* array. All components of a running Jalapeño virtual machine can be reached from a single JTOC array (see appendix A.2). This is true of the mockup as well. The structure rooted in the JTOC array is walked recursively and the values, both reference and primitive, encountered are translated into the boot-image array. Since the Type Information Block (again, see appendix A.2) for each loaded class is referenced from the JTOC, all necessary compiled method bodies will be included in the boot image.

The translation process uses reflection. The boot-image writer obtains the `java.lang.Class` object for each object in the mockup and iterates over the fields returned by the `getFields` method. For each field, it extracts the field value from the source object and extracts the target field offset from Jalapeño's class description for the object. Then, it writes the value at that offset from the index of the object in the boot image. When object references are encountered, the we cannot use any value from the mockup. The references in the mockup are converted to boot-image addresses using a hash-table maintained as boot-image space is allocated. (An array containing the addresses of all references in the boot image can be included in the boot image to support relocation of the image at boot time.)

Overall, the boot-image writer copies Java objects field-by-field from the mockup into the boot image, simultaneously translating from the source JVM's to the target JVM's object model. Relying on Java's reflection capability, we ran into one inconvenience: Sun's JDK 1.1.4 did not permit reflective access to `private` fields. This is not a problem in Java 1.2 which allows such access. We solved the problem in the earlier version by preprocessing the classfiles, turning the `private` bits off.

In addition to the objects reachable from the JTOC array, two other objects are needed in the boot image: an initial thread object containing an empty stack ready to run the first instruction of the `boot()` method when Jalapeño starts up and a "boot record" to interface the boot image with the boot-image runner (see below). This boot record contains the start, end, and last-used addresses in the image, four register values used to start Jalapeño, the address of the `boot()` method, and the addresses for AIX's system calls. When these values are stored in the boot-image array, it is written to disk.

A short program called a *boot-image runner* starts Jalapeño running. It reads the boot image into memory, sets the four registers to the indicated values, and branches to the `boot()` method. The boot-image runner is written in C (with a little assembler to set the registers and perform the final branch) not Java so *it* does not require a JVM to run on.

When the `boot()` method starts executing, the virtual machine is in a fragile state: it can run a single thread of machine instructions, but it has not yet created the external operating-system resources it needs to support its own execution. These operating-system resources cannot be created by the boot image writer, because they refer to external state that will not exist until the boot image is executed. Thus, Jalapeño must perform additional initialization.

At boot time, the virtual machine initializes hardware specific addresses (for example, it will eventually establish a hardware guard page on its own stack), opens files corresponding to the Java library's `System.in`, `System.out`, and `System.error` stream objects, parses command line arguments, and creates a `System.Properties` object corresponding to the current execution environment. Then, the multithreading subsystem is initialized by creating operating-system threads to serve as the virtual processors upon which Java threads are multiplexed. Finally, timer interrupts are enabled to support thread preemption and a Java thread is spawned to run the application program specified on the command line.

Jalapeño runs until the last (non-daemon) Java thread terminates or `System.exit()` is called.

4 INTERFACE TO THE HARDWARE

To implement Jalapeño, a number of special facilities needed to be provided. These facilities were low-level virtual-machine operations outside the Java program model. In particular, a JVM must be able to:

- access machine registers and memory,
- use architecture-specific machine instructions,
- transfer execution to an arbitrary address,
- coerce object references to raw addresses and *vice versa*, and
- invoke operating system services.

Jalapeño needs these capabilities to effectively utilize the underlying hardware (and operating system), but, in order to preserve the integrity of the language, the users of the JVM must be prevented from accessing them.

Jalapeño's compilers enable such transgressions with the help of a special MAGIC class. The methods of this class correspond to the illegal operations we wish to perform. The bodies of these methods are empty. Java's source (Java to bytecode) compilers can compile them. However, Jalapeño's (bytecode to machine code) compilers ignore the resulting bytecodes. Rather, they recognize the name of the MAGIC class and inline the necessary machine code. To make sure that user code does not evade Java's restrictions, Jalapeño's compilers will verify that the method they are compiling is an authorized part of the JVM, when they encounter a call to a MAGIC method.

4.1 Uses of MAGIC

A number of services — object allocation, garbage collection, dynamic linking, dynamic type checking, exception handling, reflection, and I/O — require the above mentioned special facilities. The following are examples:

- Object Allocation — To allocate an object, Jalapeño's memory managers must transform a piece of raw memory into a object. They obtain a chunk of available space of the required size and initialize the new object's header and body. To achieve this initialization, the manager must be able to compute (load, store and manipulate) with memory addresses.
- Garbage Collection — The collectors must access object headers to mark objects during garbage collection. They must walk the thread stacks to identify object references in the stackframes. In addition, memory allocation and work queues must be maintained. Parallel collector's must synchronize. And, of course, a copying manager must access raw memory to copy an object.
- Dynamic Linking — To perform dynamic linking it is necessary to overwrite (back-patch) the original site and to do a transfer of control to the back-patched site. To synchronize the data and instruction cache and (on multiprocessors) to clear the instruction prefetch buffer, it is necessary to execute special hardware operations.
- Exception Handling — The exception handler (like the garbage collector) uses the memory access facilities to walk the thread stack. It also uses special services to restore the register state and transfer control whenever a catch block of the proper type is encountered.
- Reflection — In order to pass parameters to and from the caller, Jalapeño constructs additional frames on the thread stack. The construction and removal of these frames require the use of extra-Java memory access services.
- Input/Output — Jalapeño uses special method calls to communicate I/O requests to the operating system. The primary purpose of the calls are to pass parameters and return the necessary responses.

Most conventional JVM's provide these services with runtime routines written in native code. To request these services, a Java program must call a native routine. One advantage to this approach is that frequently used runtime services can be heavily optimized using mature (static) optimizing compiler technology. However, often the calling conventions of the two languages differ, and a bridging stackframe in the native convention must be concocted. Either two different call stacks (per thread) must be maintained or every Java method invocation must pay a price to support host operating system and native language functionality that Java does not require. The inter-language boundary is also a barrier to optimization: runtime services cannot easily be inlined into user code.

Except in order to handle I/O requests, Jalapeño avoids these difficulties by providing most runtime services in Java. The overhead of crossing from one language to another is eliminated. An optimizing compiler is free to inline runtime services. The exposure of this approach is that dynamic optimization of Java must be competitive with the older technology.

There is also a challenge to implement each service in a subset of Java that does not use the service it implements. Obviously, the memory management subsystem can't use `new` to create its objects. Similarly, dynamic type-checking code cannot use casts, and the code that implements `aastore` cannot store object references into arrays. On the other hand, the baseline compile can translate the `ldiv` bytecode into a call on a Java method that implements long division as a series of shifts, adds, and subtracts.

4.2 Computing with raw addresses

Even with the ability to inline machine instructions, some operations require systemic considerations. Consider, for example, computing with raw addresses. Casts from object reference to raw address are merely identity transformations needed to fool Java's type system: Jalapeño's compilers simply ignore the corresponding method calls. This functionality is needed, for instance, to perform dynamic linking. It is, however, problematic.

Jalapeño's copying memory managers update object references when they move the referenced object, but raw addresses are not updated. Care must be taken to avoid garbage collection when computing with raw addresses lest a copying collector invalidate them. This is done by calling a method that disables garbage collection.

A thread that has disabled garbage collection cannot try to create an object because the system would hang if there weren't enough memory. (Other threads are free to request memory; if it is unavailable, these threads are delayed and a collection will be initiated as soon as garbage collection is re-enabled.) Similarly, if the thread were to synchronize on a shared object currently owned by a thread waiting for a garbage collection, the system would deadlock.

There are subtle implications of the restriction that a thread that has disabled garbage collection may not create objects. Classes cannot be loaded, since objects are created during class loading. This means dynamic linking must be avoided. It follows also that casts (and stores into object arrays) cannot be allowed either. Thus, a thread must operate in a tightly restricted subset of Java when computing with raw addresses.

Assertions are used to detect attempts to create objects or to lock shared objects when collection is disabled even if there is sufficient memory available to service all pending requests. It is vitally important to catch code that might lead to a disaster before the disaster occurs.

One difficulty remains: what if execution reaches a yield point with garbage collection disabled? If the thread were to yield control to another thread, a needed collection could be arbitrarily delayed until the yielding thread gets another opportunity and finish its (supposedly short) critical section. To prevent this, thread-switching is postponed on a virtual processor anytime garbage collection is disabled for that processor.

Fortunately, only a small number of JVM methods need MAGIC. These methods already require extraordinary care. The restrictions imposed by its use do not add significant problems to the overall JVM development.

5 INTERFACE TO THE OPERATING SYSTEM

The Jalapeño virtual machine was designed to run as a user-level AIX process. As such, it must interface with the host operating system to access the underlying file system, network, and processor resources. To access these resources, we were faced with a choice: we could call the AIX kernel directly, using low level system calling conventions, or we could access kernel services via the standard C library. We chose the latter path to isolate ourselves from release-specific operating system kernel dependencies. This required that we write a small portion of Jalapeño in C rather than Java.

To date, the amount of C code required has been small ($\sim$ 1000 lines). About half of this code consists of simple "glue" functions that relay calls between Java methods and the C library. The only purpose of this code is to convert parameters and return values between Java format and C format. For example, filenames are represented as counted arrays of 16-bit unicode characters in Java, but as null-terminated arrays of 8-bit characters in C.

The other half of the C code consists of the boot-image runner (see section 3) and two signal handlers. The first signal handler captures hardware traps (generated by null pointer dereferences) and software traps (generated by Jalapeño's compilers for array-bounds and divide-by-zero checks), and relays these into the virtual machine, along with a snapshot of the register state. The other signal handler captures timer tick interrupts (generated every 100ms), and sets a global flag. This flag is periodically checked in method prologues and on the back-edge of loops. These checks will eventually cause the current thread to relinquish control to the Jalapeño scheduler, allowing another thread to be dispatched.

6 RELATED WORK

We are aware of two other JVM's written in Java [6, 18]. Both were written to run on top of conventional JVM's. IBM's VisualAge for Java's JVM [7] is written in Smalltalk. Performance is not a critical design goal of these projects. Other JVM's [12, 9] are all written in native code. Runtime services are provided by C routines.

The JavaInJava virtual machine [18] was written to be a clean, extensible platform and to serve as a reference platform for Java VMs. It interprets bytecodes and implements a Java virtual machine stack. As a result its performance is poor. Jalapeño on the other hand compiles code and runs directly on the hardware. (The Jalapeño baseline compiler implements a Java virtual machine stack). Both systems do their own multithreading without operating system assistance. JavaInJava uses the underlying virtual machine for storage allocation and garbage collection while Jalapeño does its own (since there is no underlying virtual machine).

The Rivet virtual machine [6] is written as a platform to develop and make available new advanced debugging and and analysis tools for the Java environment. Like JavaInJava it relies on the underlying JVM for allocation and garbage collection. Since debugging and analysis are its main objectives, performance is not a high priority.

Perhaps the most exciting of the conventional JVM is HotSpot [9]. Hotspot is conventional in the sense that it is written in native code. HotSpot initially interprets bytecodes, compiling (and inlining) heavily executed methods. Jalapeño's quick compiler will play a role similar to HotSpot's interpreter. Java threads are implemented as operating-system threads in HotSpot. Its per-thread method activation stacks conform to host operating-system calling conventions. This should give Jalapeño a minor space and performance advantage. Both HotSpot and Jalapeño support type-accurate garbage collection. Jalapeño supports a family of memory managers. None of Jalapeño's collectors is as sophisticated as HotSpot's, but on an SMP Jalapeño's collectors run in parallel using all available CPU's.

Jalapeño shares a similar approach with the Squeak system [11]. Squeak is a Smalltalk virtual machine that is written in Smalltalk. It however, produces a production version by translating the virtual machine Smalltalk code to C code for compilation and linking. (This translator is also written in Smalltalk.) The C language was chosen as the intermediate language for performance and portability. Jalapeño differs from the Squeak approach in that it uses its own bytecode to machine code compilers (running in a different virtual machine) to generate the JVM machine code. The Jalapeño compilers are written in Java and as such can run in any JVM.

7 DISCUSSION

While our use of Java as a systems programming language has been, for the most part, pleasant and productive, it has also exposed a number of pitfalls and weaknesses in the language. One source of perennial headaches is the impedance mismatch between Java's compilation strategy and the **make** facility. There seems to be no way to ensure that class files are up-to-date short of erasing them all and rebuilding the whole system.

Another annoying deficiency in Java is the limited protection facility provided by the Java `package` mechanism: methods of one package may only be accessed from another package if those methods are declared "public". There is no notion of collaborative packages which can communicate among themselves (in the sense of `C++` "friend" functions) to provide a service, while at the same time being hidden from other unrelated classes. This deficiency has caused us much grief: for reasons of clarity, maintainability, and flexibility, we would like to divide our virtual machine implementation classes into various packages: MemoryManager, ClassLoader, DynamicLinker, BaselineCompiler, etc.. But to do so would require that we expose our implementation to other, user-level, classes. This is unacceptable since it would allow user code to escape Java semantics (e.g. a user-level method that happened to call an internal Jalapeño method which used MAGIC to write machine registers or memory might corrupt the whole system). Thus, we are forced to place all of our virtual machine classes into a single package name-space. Furthermore, because of Java's "package-name equals directory-name" approach to class lookup, we would be forced to place all of our virtual machine's source code and class files (several hundred of them) into a single file-system directory, if we were to put Jalapeño in a named package. We find this alternative so repugnant that we have, for the moment, placed all of our classes in the "unnamed" package, so their sources can be distributed across multiple directories, and have postponed the security issues.

This package problem has an impact on flexibility as well. Jalapeño is, in fact, a family of JVM's (parameterized by the choice of memory manager, for example). There are four mechanisms in Java that support developing and maintaining such a family: `interfaces`, `abstract` classes, `static final` fields, and the `-CLASSPATH` option to Java's source compilers.

Perhaps we should have made greater use of Java `inter-`

`faces` than we did. Our initial impression was that the interface invocation overhead would necessarily be prohibitively expensive to use for frequently executed methods. This assumption may have led to premature optimization: we now believe this overhead can be reduced to the point it can be ignored except for the most frequent of operations. (However, we do not yet see how to inline interface calls.)

There is another reason for neglecting Java `interfaces`: they do not allow static methods. In many instances where `interfaces` would have been useful, memory managers for instance, a running system has exactly one class that would implement the interface and that class has static methods. Java has no way of associating these methods with the interface.

(We do make use of one idiom involving interfaces. An empty interface is used to communicate to Jalapeño's compilers that a method is to be treated in a special way. Thus, if a method belongs to a class that implements `uninterruptable`, the compilers will omit the test for thread-switching from its prologue. This facility is quite useful but the granularity is slightly off: it would be nice to be able to designate individual methods as requiring special attention rather than so designating all the methods in a class.)

We make heavy use of Java's other features that support flexibility. Each compiler has a different mechanism for reporting the object references in one of its stack frames. These mechanisms are different realizations of the same abstract class. A static boolean field distinguishes development builds from performance builds. We use the idiom:
`if (VM.VerifyAssertions) VM.assert(`*boolean expression*`);`
to check conditions in development builds that are assumed to hold in performance builds.

Each of our memory managers provide the same functionality. (They conform to the same interface in the general sense, if not in the Java sense). Each is in a separate directory, we use the `-CLASSPATH` option to specify which to include in a specific build. Notice that this would not be possible if we were to combine Jalapeño into a named package, since all of the files in a package must be in the same directory. Nor, would it work to make a memory manager its own package, since the memory managers must have access to methods of Jalapeño's runtime that should not be available to Jalapeño's users.

Yet another deficiency in Java is the lack of preprocessing support, in particular the inability to conditionally include or exclude fields and methods of a class definition. This has had a profound detrimental impact on the flexibility of Jalapeño. Consider, for example, a basic issue in object layout: whether objects are accessed directly or through a table of *handles*. Arguments can be made for either approach. For example, handles may be particularly appropriate in a paging context where the added cost of accessing an object pales in comparison to the benefit of being able to move the object without needing to touch every page containing a reference to it. We would have liked to be able to support multiple object models. However, to do so would entail making every object access through an interface method call. We reluctantly concluded that the computational impact of such an approach was prohibitive. (Until we find a way to effectively inline interface calls, this assessment will be valid no matter how cheap the interface *dispatch* becomes.) With a suitable Java preprocessor, access could be made using static methods of an object-model class. A preprocess-time variable would determine whether or not handles were used. The methods of this class could be inlined by Jalapeño's compilers.

8 CONCLUSIONS

The viability of our decision to build Jalapeño in Java hinged on the early availability of a rudimentary working JVM. Key components of that minimal JVM were: a dynamic class loader (with back-patching capability), a baseline compiler, a simple memory manager, a boot-image writer and runner, and a debugger. All of these components were in place within three months of the initial decision to explore building a JVM. From that point on, components could be incrementally enhanced or added to a working system. From this modest beginning, Jalapeño has grown into a full-fledged multi-threaded Java virtual machine with a family of type-accurate parallel memory managers, multiple compilers to provide different levels of dynamic optimization for methods with different computational intensities, and a fairly sophisticated debugger. Jalapeño's primary limitation is an inability to execute library code not written in Java. The initial and subsequent productivity of Jalapeño's development was due in large part to the Java language's simplicity, type-safety, and memory management features.

We are convinced that building Jalapeño in Java was much more fun than it would have been if a conventional systems programming language had been used. Of course, part of the fun was figuring out how to implement the language in itself. For the most part this turned out to be surprisingly straightforward, although there were a few areas that caused difficulty.

We had to be careful implementing portions of the Java runtime in which the full language facilities could not be used at the source code level.

Bootstrapping Jalapeño is a complex operation. In a non-Jalapeño JVM, we created a mock-up of a running Jalapeño JVM. This was fairly straight forward since Jalapeño is composed entirely of Java objects. We then translate from the object model of the source JVM into Jalapeño's object model. This step exploits and, we believe, fully justifies Java's reflection capabilities.

Dealing with raw memory addresses, unsafe casts, and special machine instructions turned out to be easy: we wrote a special MAGIC class containing declarations for the methods that we wished to implement using inlined machine code. Jalapeño's compilers recognize the MAGIC class name and ignore the byte-codes for these methods, generating instead special-purpose machine code. This approach allows Jalapeño to bend Java's rules in order to implement its own core functionality while preserving the integrity of Java in user applications.

To interact with the operating system, we wrote a short C program that passes kernel calls from Jalapeño to AIX and passes interrupts from AIX to Jalapeño.

If we were to start over tomorrow, there is much that we might do differently. We would probably make greater use of Java `interfaces`. We might consider building an object-oriented Java preprocessor to allow greater flexibility in the family of JVM's we support. We might try to devise a better method for protecting against unsafe behavior while garbage collection is disabled. We might implement a common assembler for Jalapeño's three compilers. We would pass parameters in registers right from the beginning. We would probably at least start with a simpler dynamic linking strategy than back-patching.

Although, building Jalapeño in Java was not without drawbacks, we are well pleased with our decision to build Jalapeño in Java and would happily do so again.

APPENDIX: OVERVIEW OF JALAPEÑO

The overall shape of a Java virtual machine can be determined by five questions: Where is the boundary between Java and non-Java code? How are data laid out? How are bytecodes executed? How is memory managed? And, how are threads scheduled and synchronized?

A.1 The Boundary

We decided early on that we wanted to write as much of Jalapeño as possible in Java. A JVM has four layers: user applications, runtime code, the underlying operating system, and the base hardware. The Jalapeño runtime is written in Java and is, in most respects, treated just like user code. Special mechanisms are required to get Jalapeño started and for interacting with the hardware and operating-system layers. These were treated in sections 3, 5, and 4 respectively.

A.2 Data Layout

Data layout issues include: How are object fields and array components accessed? How are virtual methods dispatched? How is information about the class of an object obtained? And, how are static fields accessed and static methods dispatched?

The initial impetus for Jalapeño's object layout was a desire for efficient array access. To accomplish this, a reference to an array object is represented as the machine address of the first (zeroth) component of the array. The ith component is stored at an offset of i times the component size off this reference. The size of the array is stored in the word preceeding the first component (at offset -4 from the reference).

There is an interesting side-effect to this arrangement. Java requires checking that array indices are within the bounds of the array. This entails loading the array length for each array access. If the array reference were `null` (that is, address 0), this would mean loading the word at address 0xFFFFFFFC. Normally, AIX generates a hardware interrupt whenever an attempt is made to read (or write) high memory. Thus, the null-pointer check (also required by Java) on the array reference is done by the hardware. Unless the reference is `null`, this check is free. On the other hand, catching a null-pointer this way is expensive, involving a couple of operating system calls.

The layout of scalar (non-array) objects was chosen to achieve a similar hardware null-pointer check for field accesses: all fields are allocated at negative offsets from their object reference.

(In AIX it is theoretically possible for another application to load a shared library into the last segment of memory. While this is not a concern for a research system, it would constitute a security hole in a commercial system unless at least the last page of addressable memory were read and write protected. If an object has more than a page worth of fields, accesses to those that don't fit on the first page can be checked explicitly with negligible impact on performance.)

One potential draw-back of this approach is that arrays and scalar objects "grow" in different directions. This makes it difficult to scan memory for objects. To date this has not been a problem. There are two obvious approaches to enabling memory scans: segregating scalar and array objects into different heaps and appending a pseudo-header to the high end of arrays or the low end of scalars.

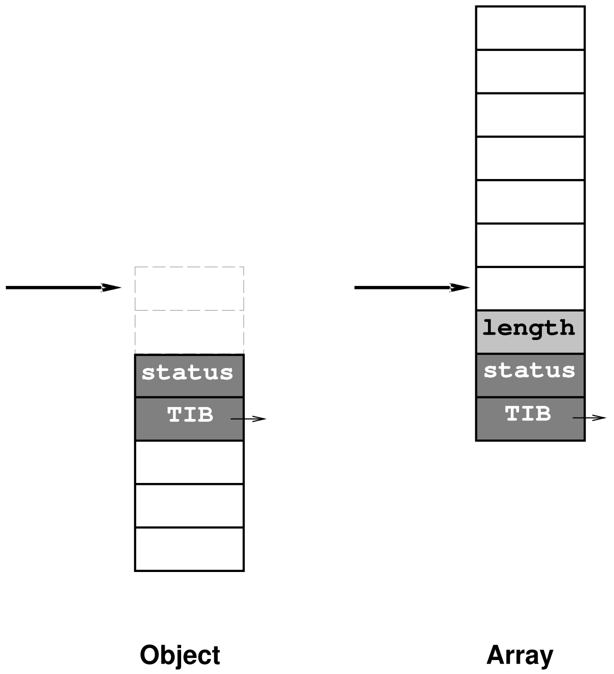

Figure 2: Jalapeño object layout

We initially planned to keep a reference to a virtual method table in a header adjacent to each object. Since method bodies (the compiled code for methods) are arrays of `int`s, the type of such a table would be array of array of `int`. We soon realized it would be convenient to be able to get quickly from any object to another object describing the class of the first. We adopted the expedient of including a reference to this *class object* in the virtual method table. The resulting structure, now an array of objects, was rechristened a *Type Information Block* (TIB). This inverts the, perhaps more natural, arrangement of having an object header point to a class object and the class object point to a virtual method table. The inverted arrangement saves one level of indirection in virtual method dispatch at the cost of an extra level of indirection to access class information.

The obvious way to address static fields and static method bodies is at fixed absolute addresses. However, the PowerPC instruction set does not provide for convenient access to absolute addresses. Another natural approach would be to include static fields with the appropriate class object and its static methods in this object's TIB. There are several draw-backs to this approach. It imposes two additional levels of indirection to get from an object to a static field (or method) of its class. There must be some mechanism for getting to the class object where there isn't an instance of the class at hand. Finally, it requires that the class objects be of different types (since different classes have static fields of different types).

The alternative which we adopted was to put all static data — static fields, references to static method bodies, constants, and the TIB for each class — into a single array, the *Jalapeño Table Of Contents* (JTOC). (The name derives from the *Table Of Contents* (TOC) register used by AIX to achieve addressability to different load modules.) All of the objects in Jalapeño are reachable from this array. This facilitates bootstrapping (section 3).

The JTOC is one of two Jalapeño constructs that breaks

Java's type discipline (method invocation stacks being the other). This does not present a barrier to accessing data through the JTOC, since such data is not accessed by Java code treating the JTOC as a Java object. Rather, Jalapeño's compilers emit code that access the slots of the JTOC directly at a fixed offset from a dedicated machine register. A parallel descriptor array identifies those slots in the JTOC that are references.

We considered factoring the JTOC into homogeneous arrays. However, this would have entailed either using an excessive number of registers (one for object references and one for each primitive type) or an added level of indirection to some static accesses. An alternative, that would eliminate the need for a descriptor array, would grow the JTOC from both ends: static object references would go on one end, primitive static fields on the other.

A.3 Compilation

The *raison d'etre* of a JVM is to execute Java bytecodes. There are two basic approaches: bytecodes are either interpreted or compiled. The initial JVMs were interpreters. Later JVMs employed Just-In-Time (JIT) compilers to compile heavily executed code. Code that was only occasionally executed was still interpreted. Excepting its debugger, Jalapeño never interprets, it always compiles.

There are three quantities that can be optimized by a compiler: execution-time of the compiled code, compile-time of this code, or development-time of the compiler itself. Jalapeño has three compilers. The *baseline* compiler is designed to be as easy as possible to implement. It turns out to be the fastest of the three. However, its code quality is competitive with an interpreter (that is to say, poor). The *optimizing* compiler [5] is designed to produce high-quality code for methods that are computationally intensive and/or often executed. The *quick* compiler is designed to carefully balance compile-time and execution-time: it performs a small number of inexpensive high-impact optimizations, principally register allocation.

Jalapeño's compile-only strategy runs into a difficulty with dynamically loaded classes. Certain bytecodes can refer to classes that may not get loaded until the bytecode is executed. Such bytecodes present no problem to an interpreter, but confront a compiler with a dilemma: load the class preemptively or defer class loading until execution. Preemptive class loading is attractive because it simplifies code generation, but it has two draw-backs. First, class loading is expensive. If the bytecode is on an execution path that is rarely if ever taken (e.g. handling a disk-full condition) then this work will almost always have been done in vain. Second, Java requires that if the class isn't available that this exception not be reported until the referring bytecode is executed. For these reasons, Jalapeño defers class loading when its compilers encounter a bytecode referring to a class that has not yet been loaded. Executing each such bytecode requires an offset value that only becomes available when the class is loaded.

We are exploring two different approaches to obtaining that value at runtime: getting it from a table or calling a method for it. In the first case, if the slot of the table has not been filled in then a method is called to load the class. To avoid repeated method call overhead in the other case, the method overwrites it call site with code the compiler would have emitted had the class been available at compilation.

A.4 Memory Management

Perhaps the most useful innovative feature of Java is its provisions for automatic memory management. Besides sparing programmers the necessity of implementing their own memory management facility, Java has eliminated a notorious source of pernicious and insidious bugs. A pointer into memory that has been freed and reallocated can cause untold havoc that may only become evident long after the damage has been done. Detecting the culprit in such cases is often very difficult because the evidence is cold by the time the crime is discovered.

As useful a feature as automatic memory management is, it presents many challenges to JVM implementors. There is a wide variety of approaches to automatic memory management [13]. No one approach is clearly superior to all others in all circumstances. We decided early on to pursue a number of approaches in tandem with a view toward understanding the trade-offs in each and toward perhaps designing a hybrid suited to the specific needs of a particular server environment. To this end, Jalapeño supports a family of memory managers each consisting of an object allocator and a garbage collector.

Most garbage collectors are either conservative or type accurate. A conservative collector treats any value that "appears to be" an object reference as one. A type-accurate collector must identify exactly those values that are object references. It is possible that an `int` might happen to be interpretable as a reference to an object that is garbage. A type-accurate collector would collect the object, a conservative collector would not. This situation rarely occurs, so the space impact of this limitation on conservative collectors is slight. However, there is another implication: conservative collectors cannot move an object if an apparent reference to it could, in fact, be an `int`.

Another anomaly of conservative collectors can have a noticeable space impact: phantom references in uninitialized variables on method invocation stacks. Suppose method `A` calls method `B` which has a local variable `x` containing the only reference to an object. If `B` returns, the object is garbage. But, suppose `A` then calls `C`. The slot for `x` is now assigned to some local variable `y` of `C`. If `y` is not initialized to a different value, a conservative collector will mistake `y` for a reference to the object. Although we have at times had conservative collectors to support early versions of our compilers, we decided in the beginning that Jalapeño would provide type-accurate garbage collection.

Java's strong typing greatly simplifies the task of identifying references. However, the two places where we break Java's type system — the JTOC and method invocation stacks — are problematic. A descriptor array parallel to the JTOC identifies which components are references. References on method invocation stacks are located with the help of *stack maps* provided by Jalapeño's compilers. Each compiler must provide for each method it compiles a map that shows where references are in the stack frame at each place, called a *safe point*, that garbage collection might occur. Each compiler must also provide a register map for each safe point, to track references in registers. The *reference maps* for a method are the combination of its stack and register maps.

The computational impact of automatic memory management can be felt in throughput — the time devoted to allocation and collection — and latency — the pause time associated with a collection. It is important to keep both as small as possible. It may be that in a server environment pause times are most important, but this certainly does not mean that throughput impact can be ignored. Garbage collectors can be categorized as concurrent (running at the same time as mutators) or stop-the-world (running only when mutators are not). There may be throughput advantages to stop-the-world collectors, but concurrent collectors reduce pause times to their logical minimums. It is too early to tell which of the two approaches is best. As a practical matter, we felt stop-the-world collectors would be easiest to implement first and we chose to do so. However, we have tried not to make choices that would preclude going to concurrent collection later.

Collectors can be characterized as copying or non-copying depending on whether or not they move objects. Copying collectors have an advantage in allocating objects but need to do more work during collection. We suspect that a non-copying concurrent collector will be easier to implement and possibly more efficient than a copying concurrent collectors. We decided to support both copying and non-copying collectors in Jalapeño.

It has been observed that most objects become garbage very soon after they are allocated. *Generational* collectors try to take advantage of this observation by categorizing objects as old or young. Typically, an object is old if it has survived a collection. Minor collections assume all old objects are live and just collect new garbage. Major collections collect all garbage. Minor collections have shorter pause times but free less memory and thus may be required more often. Usually, *remembered set* of old objects that may point to young ones must be maintained. This slows down mutation, but greatly speeds up collection. Jalapeño supports both generational and non-generational collectors.

Since Jalapeño is intended to run on multiprocessors, memory management must not be a serial bottleneck. Objects being allocated at the same time by different threads are obtained from different regions of memory (conceptually they reside in the same heap however). Jalapeño's collectors are also designed to run in parallel.

A.5 Threads and Synchronization

Memory management considerations played a determining role in shaping Jalapeño's multithreading strategy. Transitions between mutation and collection must be accomplished efficiently. Jalapeño must be able to handle thousands of user threads. Type-accurate stop-the-world garbage collection requires that each of these threads be stopped at a safe point before collection can proceed. If each user thread is a separate operating system thread, this can be an arduous task. (We initially assumed, perhaps prematurely [17], that making every machine instruction a safe point would not be feasible because of the space required to store the reference maps. However, even if it were, merely stopping thousands of system threads is non-trivial.)

This consideration was a major motivation for multiplexing user threads on *virtual processors* implemented as operating-system threads. By requiring that the points a thread could lose control of a virtual processor be safe points, we are assured that all threads, *not* currently running, are stopped at safe points. It is only necessary to stop (at safe points) those threads currently executing before beginning garbage collection. Other considerations also supported the choice to multiplex Java threads on virtual processors: it helped minimize dependence on system services, it enabled fast thread switching, and it allowed tight integration of synchronization support with thread switching.

Our initial implementation of *quasi-preemptive* thread switching proved to be relatively straight forward. Jalapeño's compilers generate (as part of every method prologue and, eventually, on backward branches as well) a test of a reserved condition code bit. On a PowerPC this test can usually be overlapped with other instructions and therfore has "zero" cost. The condition code bit is set by a periodic timer interrupt in the condition register of the processor that handles the interrupt. When the set condition code bit is detected, the method prologue transfers control to the scheduler function. There the thread state is saved (using Magic functions) and the next thread is dispatched (again, using Magic functions).

This is an efficient and elegant solution to the problem of how to interrupt Java threads in a timely manner but only at safe points, and we are justifiably proud of it. Unfortunately, it doesn't work. In AIX, timer ticks are delivered *at most* ten times a second. This is a long time for a thread to execute before it gets context switched. On a SMP, only one processor gets interrupted each timer tick. So, if there are P processors, a thread gets interrupted *on the average* only every P timer ticks. Furthermore, AIX makes no fairness guarantees about how often any particular processor will get interrupted. We observed significant runs of the same processor being interrupted. Thus, our initial solution did not guarantee sufficiently frequent thread-switching.

The revised solution was to have all processors perform a thread switch at every timer tick. This seems to require storing the thread switch flag in memory. This, in turn, requires that the compiled code periodically load and test the flag. These are not extremely costly operations by any means, but they are no longer free.

When a garbage collection is required, the scheduler saves the state of the running thread and dispatches a collector thread. In a system with multiple virtual processors, the collector threads wait until all the virtual processors have dispatched collector threads; then the garbage collection begins. Since there is one collector thread running on each virtual processor, all the mutator threads must be at safe points. When garbage collection completes, the collector threads return to the scheduler and normal dispatching resumes.

Synchronization in Jalapeño is based on bimodal locks [4, 16]. Uncontended-for objects may be locked with *thin locks* in the object headers. When two or more threads try to lock the same object, its thin lock gets promoted to a *thick* lock. The protocol for obtaining a thick lock is more expensive than that for obtaining a thin one. Jalapeño's locking mechanism differs from previous bimodal locking work in that thick locks are ordinary Java objects that can be obtained without assistance from the operating system. The mechanism requires the use of special RS/6000 instructions that ensure mutually exclusive operations. Jalapeño issues these instructions by calling the designated Magic routines and the compiler generates them in line.

ACKNOWLEDGMENTS

Our work benefited greatly from consultations with other people involved in the Jalapeño project. Jong-Deok Choi contributed to the initial design. Vassily Litvinov implemented a sophisticated dynamic type-checking algorithm. Peter Sweeney and Brian Cooper did preliminary work on dynamic profiling. Perry Cheng helped develop Jalapeño's quick compiler. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J.

Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley built its optimizing compiler.

Craig Chambers kindly reviewed an earlier version of this paper and offered much helpful criticism.

References

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio J. Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalepeño virtual machine. *IBM Systems Journal*, 2000.

[2] Bowen Alpern, Mark Charney, Jong-Deok Choi, Anthony Cocchi, and Derek Lieber. Dynamic linking on a shared-memory microprocessor. In *International conference on Parallel Architectures and Compiler Techniques*, October 1999.

[3] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a Compiler-Supported Java Virtual Machine for Servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999. (Also available as INRIA report No. 0228, March 1999.).

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

[5] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[6] John Chapin. Personal communication re. the Rivet project at MIT. See http://sdg.lcs.mit.edu/rivet.html for further information.

[7] John Duimovich. Personal communication.

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

[9] The Java Hotspot Performance Engine Architecture. White paper available at http://java.sun.com/products/hotspot/whitepaper.html.

[10] IBM Corporation. *AIX Version 4.3 Technical References*, 1998.

[11] Dan Ingels, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 318–326, October 1997.

[12] Java development kit 1.1, see http://java.sun.com/marketing/collateral/jdk_sc.html.

[13] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[15] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[16] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[17] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a java compiler. In *SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 326–336, May 1999.

[18] Antero Taivalsaari. Implementing a Java Virtual Machine in the Java programming language. Technical Report SMLI TR-98-64, Sun Microsystems, March 1998.