Jalapeño — a Compiler-Supported Java^{TM} Virtual Machine for Servers

Bowen Alpern   Anthony Cocchi   Derek Lieber   Mark Mergen   Vivek Sarkar*

IBM Research
Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

Abstract

In this paper, we give an overview of the Jalapeño Java Virtual Machine (JVM) research project at the IBM T. J. Watson Research Center. The goal of Jalapeño is to expand the frontier of JVM technologies for server machines. As reported in the paper, several of the design and implementation decisions in Jalapeño depend heavily on compiler support.

Two noteworthy features of the Jalapeño JVM are as follows. First, the Jalapeño JVM takes a compile-only approach to program execution. Instead of providing both an interpreter and a JIT compiler as in other JVMs, bytecodes are always translated to machine code before they are executed. Second, the Jalapeño JVM is itself implemented in Java! This design choice brings with it several advantages as well as technical challenges.

The Jalapeño project was initiated in January 1998 and is work-in-progress. This paper summarizes our design decisions and early experiences in working towards our goal of building a high-performance JVM for SMP server machines.

1 Introduction

This paper describes work-in-progress on the Jalapeño research project under way at the IBM T. J. Watson Research Center since January 1998. Jalapeño is a research Java Virtual Machine (JVM) that targets server machines. Compared to other JVMs, a key distinguishing feature of Jalapeño is its widespread use of compilers and compiler technologies. Specifically, all of the following JVM functionalities are driven by compiler support:

- *Program execution* — Jalapeño takes a *compile-only* approach to program execution. Instead of providing both an interpreter and a JIT compiler as in other JVMs, bytecodes are always translated to machine code before they are executed. Jalapeño has three different compilers to provide such translation: a highly optimizing compiler [5] for computationally intensive methods, a "quick" compiler that performs low optimization for initial execution of dynamically loaded methods, and a "baseline" compiler that mimics the stack machine of the JVM specification document [17]. The baseline compiler is used for code that executes only once, to validate the other compilers, and for debugging.

- *Adaptive Optimization* — Jalapeño's compile-only approach makes it easier to mix unoptimized and optimized compiled methods compared to mixing interpreted execution and JIT-compiled execution in other JVMs.

*Contact author. Email: vsarkar@us.ibm.com. Phone: +1-914-784-7105. Fax: +1-914-784-6576.

- *Thread Scheduling* — Jalapeño's implementation of Java threads is lightweight and nonpreemptive. Jalapeño multiplexes Java threads on a small number of operating-system threads. Support for nonpreemptive thread scheduling is provided by Jalapeño's compilers, which insert explicit code for "yield points" in each compiled method.

- *Garbage Collection (GC)* — Jalapeño's compiler-directed implementation of nonpreemptive thread scheduling paves the way for our parallel GC algorithms. The presence of yield points ensures that each mutator thread will yield from its execution when so requested by the GC subsystem. As in past systems that supported type-accurate GC (e.g., Self [9]), compiler support such as generation of reference maps and insertion of "write barriers" is also necessary to support the GC algorithms used in Jalapeño.

- *Synchronization* — the Java memory model requires that all data written in a critical section be made visible to other threads on a `monitorexit` instruction. The Jalapeño compilers generate explicit cache management instructions for a monitorexit, so as to ensure correct execution semantics on multiprocessor machines such as PowerPC SMPs. In addition, compiler support is used for Jalapeño's implementation of lightweight locks (an extension of the scheme presented in [4]).

- *Exception handling* — to ensure correct handling of exceptions in the presence of interleaved calls to unoptimized code and optimized code on the stack, all the Jalapeño compilers generate compatible exception tables that match the same interface to the runtime routines used for exception-handling. In addition, the Jalapeño optimizing compiler takes great pains to obey Java exception semantics while constraining code motion as little as possible. (This is accomplished by compiler analysis of catch blocks and by using an intermediate representation in which exception checks are exposed as explicit instructions.)

- *Dynamic linking* — to satisfy Java's semantics for dynamic class loading (*viz.,* a class cannot be loaded until a byte-code instruction is executed that requires the class), Jalapeño compiles an unresolved reference (i.e., a reference to a field or method in an unloaded class) into a call to a special routine in the Jalapeño runtime system. This routine triggers class loading and compilation (as needed), and then dynamically overwrites ("backpatches") the instruction sequence for the runtime routine's call site with a direct reference to the field or method.

- *Bootstrapping* — a unique characteristic of the Jalapeño JVM is that it is implemented in Java! Thus, compiler support is essential for boostrapping the Jalapeño JVM. A Jalapeño boot image is created by writing out compiled code for the core VM methods, along with some key data structures.

The rest of the paper is organized as follows. Section 2 provides background on JVM requirements for server machines, and outlines how Jalapeño addresses these requirements. Section 3 contains an overview of the Jalapeño JVM. Section 4 summarizes the current status of the Jalapeño project. Section 5 briefly discusses related work, and section 6 contains our conclusions.

2 JVM Requirements on Servers

Over the last three years, Java [2] has been rapidly gaining importance as a programming language for the client side of networked applications, and more recently, for the server side as well. Java programs are executed on a Java

Virtual Machine (JVM) [17]. An important source of Java's appeal is its *universality* i.e., the fact that it is available as a common programming language across a heterogeneous mix of client and server platforms.

However, the fact that Java is a universal language does not imply that a single JVM implementation is suitable across all tiers of network computing — one size does not fit all! There is already an established need for a special class of JVMs that are tailored to obey the unique resource constraints of embedded devices. Our position is that server machines also have unique requirements that motivate the need for a special class of JVMs for servers. In particular, we believe that the following critical requirements for servers are not being addressed effectively by current JVMs:

1. *Exploitation of high-performance processors in servers* — the performance gap between interpreted execution and optimized-compiled execution is much larger on servers than on clients. However, JIT compilers in current JVMs do not perform the extensive optimizations for exploiting modern hardware features (memory hierarchy, instruction-level parallelism, multiprocessor parallelism, etc.) that are necessary to obtain performance comparable with statically-compiled languages such as C++.

2. *SMP scalability* — SMP configurations are very popular for server machines. However, current JVMs do a 1:1 mapping of Java threads onto heavyweight O/S threads, thus leading to poor scalability of multithreaded Java programs (when the numbers of threads is increased on an SMP machine).

3. *Thread limits* — many server applications need to create new threads for each incoming request. However, due to O/S constraints, current JVMs are unable to create a large number of threads and hence can only deal with a limited number of simultaneous requests. These constraints are severely limiting for applications such as chat servers that need to support one or two thousand users simultaneously.

4. *Continuously-running JVMs* — server applications must be able to satisfy incoming requests while running continuously for long durations (e.g., several months). However, current JVMs on servers are rarely able to run continuously for more than 4 or 5 days.

5. *GC pause times* — most server applications have stringent response-time requirements e.g., at least 90% of requests must be served in < 1 second. However, most current JVMs perform non-incremental GC thus leading to severe response-time failures.

6. *Use of libraries* — server applications written in Java are typically based on existing libraries/frameworks/components, rather than being written from scratch. However, since the libraries and frameworks are written to handle generic cases, they usually perform very poorly on current JVMs.

We now briefly summarize how the Jalapeño JVM addresses these requirements for servers. More details on the Jalapeño JVM can be found in section 3.

Requirement 1 is addressed by the optimizing compiler in Jalapeño's two-compiler strategy. (More details on the optimizing compiler can be found in [5].) Requirements 2 and 3 are addressed by the implementation of lightweight threads in Jalapeño. Requirement 4 is satisfied by our decision to implement Jalapeño in Java, thus avoiding storage leaks that might have appeared if Jalapeño had been implemented in a language that does not support GC. We expect requirement 5 to be satisfied by the parallel and incremental GC algorithms that are currently being designed and implemented for Jalapeño. Requirement 6 will be satisfied by specialization transformations in the Jalapeño optimizing compiler that tailor the dynamically compiled code for a library (say) to the calling context of the server application.

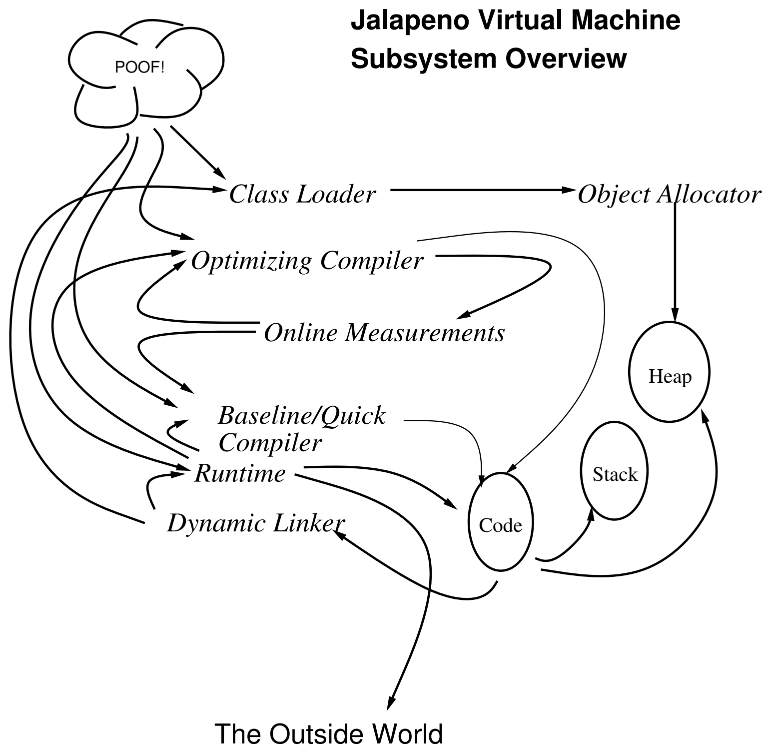

Figure 1: Subsystems of the Jalapeño Virtual Machine

3 An Overview of the Jalapeño Virtual Machine

As shown in figure 1, the subsystems of the Jalapeño JVM include a dynamic class loader, dynamic linker, object allocator, garbage collector, thread scheduler, profiler (on-line measurements system), three dynamic compilers, and a runtime system[1]. The Jalapeño JVM supports dynamic class loading (including backpatching of compiled code to link with classes loaded after compilation), exception handling, type testing, and online performance measurement. Memory management in Jalapeño consists of an object allocator and a garbage collector. The Java library currently used by Jalapeño is Sun's JDK 1.1.4 classes.zip file; the key modification that we had to make was to translate some native methods in the library from C to Java!

The rest of this section is organized as follows. Section 3.1 describes the challenges that we had to overcome to implement Jalapeño in Java. Section 3.3 describes how we support dynamic class loading with a compile-only execution strategy. Section 3.4 outlines our approach for supporting threads and synchronization. Jalapeño relies on compiler support for implementing all these JVM functionalities.

3.1 Implementation in Java

Other Java Virtual Machines have been written in Java (e.g., [21, 10]), but these run on top of other JVM's. Jalapeño runs on bare metal. There are advantages and disadvantages to building a substantial systems project 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 performance advantages as well: firstly, by eliminating the "bridge" that must be crossed between user code and frequently accessed system services, and secondly, by

[1] The "POOF" in figure 1 refers to the bootstrapping of the Jalapeño Virtual Machine, which is described in section 3.1.

applying to those system services the same dynamic optimizations that we apply to user code. The disadvantages to building Jalapeño in Java are those entailed by the delicate dance involved in exploiting the advanced features of the language by the very code that implements them. Of particular concern, is how to get the whole JVM in motion to begin with. Another issue that must be faced is the necessity of a virtual machine computing with the raw addresses. How else, for instance, could a copying garbage collector written in Java copy an object?[2]

Our first challenge was the bootstrap problem; how to get started? This was accomplished by constructing a *boot image*, a snap-shot of a working Jalapeño system. A short program (about two hundred lines of C and a dozen of Assembler) called the *boot image runner* reads a Jalapeño boot image from a file, writes it into memory, starts it running. (The boot-image-runner program effectively serves as the "main program" for the Jalapeño Virtual Machine, and intercepts all system interrupts and passes them to the exception delivery mechanism written in Java.

A boot image is constructed by a *boot image writer*, a Java program that runs on an existing JVM (which is Sun's JDK 1.1.4 in our current implementation). A special class-loader reads the classes needed for a minimal Jalapeño system i.e., selected Jalapeño classes and classes from the standard Java Library. The methods of these classes are compiled by one of the Jalapeño compilers; the baseline and the optimizing compilers have the same interface and can be used interchangeably for this purpose. Note that a Jalapeño compiler is itself a Java method. In the boot image writer, the compiler is interpreted by the JDK. One of the methods it compiles is itself, and so the "compiled-compiler" gets included in the boot image too. Later, when Jalapeño is running, this compiled-compiler compiles methods that are dynamically loaded. The Jalapeño class images and their compiled methods are copied to an array of **int**s which is then written to a file as the boot image.

The second challenge posed by implementing a JVM in Java is the support of standard Java libraries. Many of the classes in these libraries are written entirely in Java and Jalapeño can use those classes unchanged. However, several methods in these libraries are written in native code (i.e., C). Native methods that use Sun's JNI (Java Native Interface) will be supported unchanged when JNI support is implemented in Jalapeño. But native methods in the core JDK libraries that do not use Sun's JNI must be rewritten to be usable by the Jalapeño JVM (or any JVM that differs from Sun's JDK), because they make assumptions about the implementation of the JVM that hold for Sun's JDK but don't hold for Jalapeño. Since these native methods need to be rewritten anyway, our preference is to rewrite them in Java. We have completed this rewrite for commonly used methods in the java.lang, java.util and java.io libraries. Other native methods will be rewritten on an as-needed basis. (Given Jalapeño's focus on server platforms, we currently have no plans to rewrite client-side user-interface libraries such as AWT.)

The third challenge that we faced was the need to bypass Java's type system in controlled ways. The solution adopted by the JDK to address this challenge is to use native methods e.g., the doubleToLongBits method of the Double class. Our solution is to introduce a special class called "VM_Magic" whose methods can be used (only by the Jalapeño JVM[3]) to bypass Java's type system in special cases. Of particular importance are the methods VM_Magic.objectAsAddress() and VM_Magic.addressAsObject() which cast an object reference as an **int** and *vice versa*.[4] All VM_Magic methods are treated in a special way by the Jalapeño compilers, analogous to intrinsic functions in other language implementations. In addition to allowing our JVM to bypass Java's type system when needed, VM_Magic methods are also used for two other purposes: to access (read and, sometimes, write) the memory

[2] The squeamish reader is assured that the existence of mechanism that allow the implementation of the JVM to violate the Java type discipline does not imply that such mechanisms will be accessible (as they must not be) to users of the JVM.

[3] The default Jalapeño class loader will create a separate name space for user applications that will disallow them access to VM_Magic methods.

[4] Garbage collection must be inhibited during computations involving raw addresses.

and registers of the underlying hardware, and to enable reflective method invocation.

3.2 Jalapeño Runtime Data Structures

In this section, we give a brief summary of the memory layout and runtime data structures used by the Jalapeño JVM. All the internal data structures in the Jalapeño JVM are accessible as Java objects. As shown in figure 2, each scalar (non-array) object has a two word header: a pointer to a *type information block* (TIB), and a *status word* for hashing, locking, and garbage collection. An array object has an additional *length* field in its header. Note that arrays grow up from the object reference (with the array length at a fixed negative offset), while scalar objects grow down from the object reference with all fields at a negative offset. This design was chosen because any attempt to access a field at a negative offset with a null pointer will result in a hardware trap (so long as the high end of virtual memory is unused), thus providing a null pointer check for free.

The first word in a scalar/array object header is a reference to the object's TIB, which is declared to be of type `Object[]` i.e., an array of object references. Element 0 of the TIB describes the object's class (including its superclass, the interfaces it implements, offsets of any object reference fields, etc.). The remaining elements are references to compiled method bodies (executable code) for the virtual methods of the class. Thus, the TIB serves as Jalapeño's virtual method table. Compiled method bodies are arrays of machine instructions i.e., they have type `int[]`. As shown in figure 3, a virtual method dispatch entails loading the TIB pointer at a fixed offset off the object reference, loading the address of the method body at a given offset off the TIB pointer, and branching to the address to perform the call.

Finally, all static fields and references to static methods are stored in a single array called the *Jalapeño Table Of Contents* (JTOC). For efficiency, a reference to this array is maintained in a dedicated machine register (referred to as the JTOC register). All of Jalapeño's global data structures are accessible through the JTOC. Literals, numeric constants and references to String constants are also stored in the JTOC. To enable fast common-case dynamic type checking, the JTOC also contains references to the TIB for each class in the system. The JTOC is depicted in figure 4.

3.3 Dynamic Linking

The key challenge in implementing a compile-only strategy in Jalapeño is dealing with the dynamic linking semantics of Java. When the BootImageWriter code runs in the source JVM it prepares the classes of our target JVM in three stages: load, meaning reading the class file data, resolve, meaning looking up symbols to determine field and meta-data sizes, and instantiate, meaning compiling methods and setting global data values. All classes are loaded before any are resolved and all classes are resolved before any are instantiated. As a result when we invoke the compiler during instantiation the the compiler can lookup the address of called static methods and type information block (our method dispatch tables) and place these addresses directly into the machine instructions.

When the target JVM runs, class loading must happen piece-meal by Java's late binding rules. To be a valid JVM we cannot load a class until we "execute" a byte-code that requires the class. For example, if a method of class `A` calls a static method of an as-yet-unloaded class `B`, we cannot load `B` until the call executes.

To achieve this in compiled machine instructions we issue code for "dynamic linking". As we compile a method and encounter a call to a method of a class that is not loaded, we issue machine instructions to call our runtime function. The instruction stream we issue contains an index for the method's name and descriptor. At execution time this instruction sequence enters our runtime method which examines the call stack to find the saved instruction

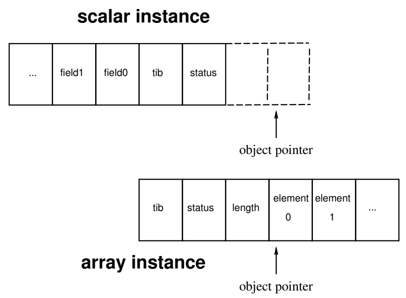

Figure 2: Memory layout for scalar objects and array objects

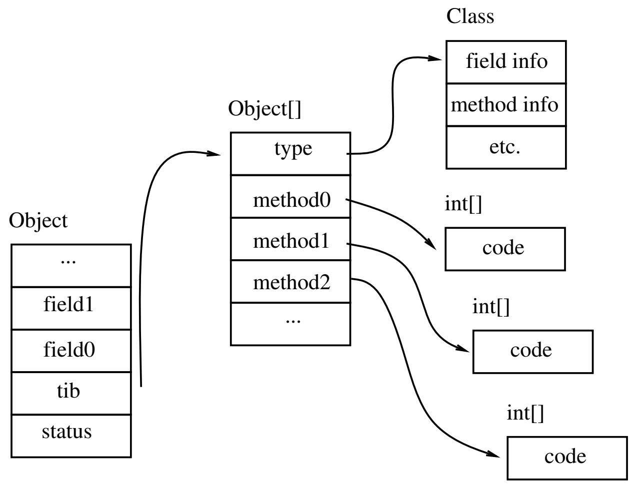

Figure 3: Virtual method dispatch

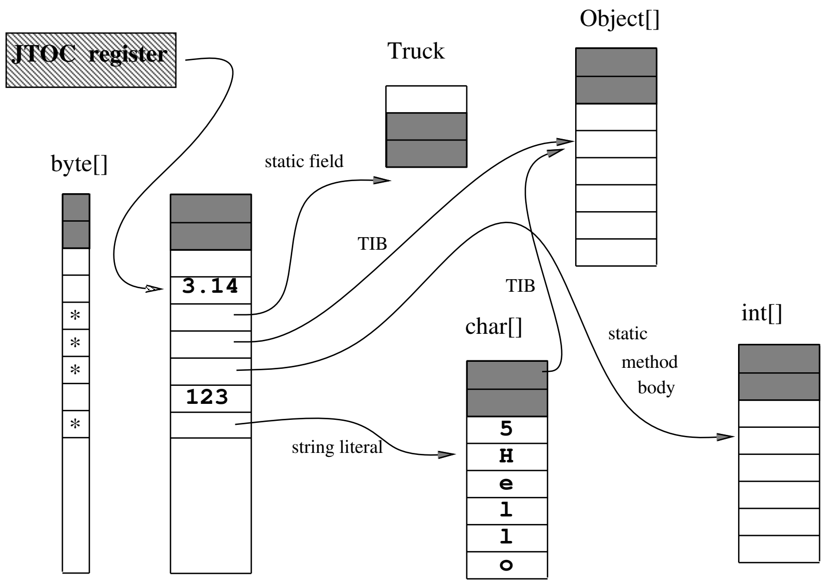

Figure 4: The Jalapeño Table Of Contents

pointer and from that locates the index where it was stored in the instructions. Then the method name is used to load the class and compile the method. The runtime method then overwrites the instruction stream to place the address of the just-compiled method into the spot where the runtime method was called and adjusts its callers instruction pointer to re-execute this portion of the instructions. When the runtime method returns, the dynamically loaded and compiled method is executed. Subsequent calls use the "back-patched" instruction stream and call the dynamically loaded method directly.

Manipulation of one's own call stack and overwriting existing machine code requires care. In addition to the extra challenge of debugging, we have to insure that the CPU's instruction and data cache are properly synchronized after the back-patch is installed. We have not yet studied the performance impact of these cache updates.

3.4 Threads and Synchronization

In Jalapeño, Java threads are multiplexed by virtual CPU's called *system threads*. These system threads are implemented as AIX pThreads. Currently, there is one system thread for each physical processor of a SMP. Eventually, additional system threads may be added to mask I/O latencies. Currently, each system thread has it's own ready queue of user threads available for execution. In addition, there is a global ready queue for load balancing.

User threads are preemptable but only at specified safe-points. Currently, method entries are the only save-points. Backward branch sites are may eventually be safe-points at well. When execution reaches a safe-point, the thread switch bit of the processors control register is checked. (It gets set one hundred times a second (per AIX process) by a system interrupt.) If it is *not* set, normal execution continues. Otherwise, a special method is called. This method performs some timer related bookkeeping. Then, it saves the current state of the thread (saves its registers), places the thread on one of the ready queues, removes a thread from a ready queue, restores its state, and resumes execution of the new thread. These operations are somewhat delicate, since the method that performs them is running (as a phantom) on the stack of the old thread. Care must be taken to prevent another system thread from dispatching this thread before the transfer of execution to the new thread is complete. Voluntary user thread yields are treated similarly.

Jalapeño uses a variant of lightweight locks [4] to implement Java synchronization. A novel feature of the Jalapeño approach is that even contended locks are handled without recourse to operating system services.

4 Current Status

Our initial implementation target for the Jalapeño JVM is PowerPC SMPs running AIX. As mentioned earlier, work on the Jalapeño project began in January 1998. The current status is that almost all the entire JVM specification for JDK 1.1.6 has been implemented in Jalapeño, including support for exceptions, garbage collection, threads, and synchronization. The key omissions are lack of support for finalizers and for suspend/resume methods. We have successfully run a large variety of Java programs on the Jalapeño JVM. This level of functionality has demonstrated the feasibility of our approach of using a compile-only strategy and of implementing the Jalapeño JVM in Java.

Apart from the core JVM, the key functional deficiency lies in incomplete support for libraries in JDK 1.1.6. Since JNI support has not been implemented as yet, Jalapeño currently does not support sockets, AWT, JDBC, RMI, or other libraries that call native methods through JNI. However, we have rewritten selected native methods from Sun's **java.lang** package into Java versions on an as-needed basis for the programs that we're interested in running on Jalapeño.

Preliminary performance measurements of single-threaded micro-benchmarks suggest that the quality of code being generated by the Jalapeño Optimizing Compiler is comparable to that of the best available JIT compilers, despite the fact that only a small set of optimizations has been implemented in the Jalapeño Optimizing Compiler thus far. This is demonstrated by the performance measurements reported in [5] for the Symantec micro-benchmarks, in which the Jalapeño delivers better performance on three of the nine benchmarks compared to the JDK with the best available JIT compiler for AIX/PowerPC.

For larger benchmark programs such as those in SPECjvm98 [12], the results reported in [5] show that Jalapeño runs these programs 1.2× to 3.3× slower than the JDK. The slowdown is not a surprise to us because the performance methodology followed in the Jalapeño project is one of avoiding premature optimizations. Initially, all required functionality is implemented using simple mechanisms. Only as these mechanisms are measured to be performance bottlenecks are they replaced with more sophisticated mechanisms. Currently, there is work in progress on more efficient implementations of the synchronization and allocation runtime routines in Jalapeño. In addition, the extra optimizations currently being implemented in the Jalapeño Optimizing Compiler should improve the performance of application code as well as Jalapeño's runtime routines. Finally, we have been evaluating the scalability of the Jalapeño JVM on an IBM-internal Java benchmark program for transaction processing on an SMP. For this benchmark, Jalapeño's "m-to-n" implementation of Java threads leads to superior scalability and speedups, compared to the JDK implementation of threads. In summary, we believe that our results thus far suggest that a JVM written entirely in Java can deliver performance that is comparable to, and even superior than, a state-of-the-art JVM implemented in C.

5 Related Work

Dynamic compilation, also called dynamic translation or just-in-time compilation, has been a key ingredient in a number of previous implementations of object-oriented languages. Deutsch and Schiffman's high performance implementation of Smalltalk-80 dynamically translated Smalltalk bytecodes to native code [13]; their compiler was quite similar to our baseline compiler. Implementations of the Self language also relied on dynamic compilation to achieve high performance [6]. All three generations of Self compilers utilized register-based intermediate representations that are roughly equivalent to the one used by the Jalapeño Optimizing Compiler. Recently, a number of just-in-time compilers have been developed for the Java language [1]. Some of these compilers translate bytecodes to a three-address code, perform simple optimizations and register allocation, and then generate target machine code.

A number of previous systems have utilized more specialized forms of dynamic compilation to selectively optimize program hot spots by exploiting "run-time constants" [11, 3, 19, 14]. In general, these systems emphasize extremely fast dynamic compilation, often performing extensive off-line precomputations to avoid constructing any explicit representation of the program fragment being compiled at dynamic compile-time.

Implementing a Java virtual machine and its related subsystem (including the optimizer) in Java opens several challenges. Taivalsaari [21] also describes a "Java in Java" implementation to examine the feasibility of a high quality virtual machine written in Java. One drawback of this approach is that it runs on another Java virtual machine, which adds performance overhead because of the two-level interpretation process. Our approach avoids the need for another JVM by bootstrapping the system. Compared to Taivalsaari's system we have also implemented several optimizations to improve the performance of the overall system and Java applications.

A large collection of work addresses optimizations specific to object-oriented languages, such as class analysis, both intraprocedural [8] and interprocedural (see related work in [15]), class hierarchy analysis and optimizations [22, 20], receiver class prediction [13, 16, 7], method specialization [22], and call graph construction (see related work in [15]). Other optimizations relevant to Java include bounds check elimination [18] and semantic inlining [23].

6 Conclusions

The use of Java in many important server applications brings with it several requirements that are specific to executing Java on server machines. The Jalapeño project was initiated to address these requirements. In this paper, we gave an overview of the Jalapeño JVM, and described the widespread use of compiler technologies in the design and implementation of the Jalapeño JVM. To the best of our knowledge, Jalapeño is the first JVM for servers that is written in Java and that has a compile-only execution strategy.

Acknowledgments

We would like to thank past and present members of the Jalapeño team — Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Susan Hummel, Vassily Litvinov, Mark Mergen, Ton Ngo, Igor Pechtchanski, Jim Russell, Mauricio Serrano, Janice Shepherd, Steve Smith, V.C. Sreedhar, Harini Srinivasan, John Whaley — for all their contributions to the design and implementation of the Jalapeño JVM.

References

[1] Ali-Reza Ald-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.

[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] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, March 1992. Published as technical report STAN-CS-92-1420.

[7] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, University of Washington, Department of Computer Science and Engineering, June 1996.

[8] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizating dynamically-typed object-oriented programs. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–164, 1990.

[9] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self – a dynamically-typed object-oriented language based on prototypes. In *Proceedings OOPSLA '89*, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

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

[11] Charles Consel and Franccois Noël. A general approach for run-time specialization and its application to C. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145–156, January 1996.

[12] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[13] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Annual ACM Symposium on the Principles of Programming Languages*, pages 297–302, January 1984.

[14] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, to appear.

[15] Dave Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, October 1997.

[16] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994. *SIGPLAN Notices, 29(6).*

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

[18] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[19] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 109–121, June 1997.

[20] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 324–332, June 1998. *SIGPLAN Notices, 33(5).*

[21] Antero Taivalsaari. Implementing a Java virtual machine in the java programming language. Technical Report SMLI TR-98-64, Sun Microsystems, March 1998.

[22] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1997.

[23] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *ACM Java Grande Conference*, San Fransisco, CA, June 1999.