Safe and Efficient Hardware Specialization of Java Applications

Matt Welsh
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776 USA
mdw@cs.berkeley.edu

Abstract

Providing Java applications with access to low-level system resources, including fast network and I/O interfaces, requires functionality not provided by the Java Virtual Machine instruction set. Currently, Java applications obtain this functionality by executing code written in a lower-level language, such as C, through a *native method interface*. However, the overhead of this interface can be very high, and executing arbitrary native code raises serious protection and portability concerns.

Jaguar [37] provides Java applications with efficient access to hardware resources through a bytecode specialization technique which transforms Java bytecode sequences to make use of inlined *Jaguar bytecode* which implements low-level functionality. Jaguar bytecode is portable and type-exact, making it both safer and more efficient than native methods. Jaguar requires that the target JVM or compiler recognizes Jaguar bytecode, which is a superset of the Java bytecode instruction set. We describe two implementations of Jaguar: one based on a static Java compiler, and the other which uses a standard JVM coupled with a Jaguar-enabled just-in-time compiler. The JIT compiler applies *code patches* to the original Java bytecode, stored in the form of annotations in the Java class file. We demonstrate that Jaguar-specialized Java applications perform as well as C for a set of communication benchmarks, and that the code patching technique involves minimal compile-time overhead.

1 Introduction

Java [14] is becoming increasingly popular for large-scale server applications, including Internet services [27, 34], databases [16], and parallel processing [40, 24]. To obtain high performance, these applications need to make use of specialized hardware and O/S interfaces, such as fast cluster networks [32], asynchronous I/O [23], and raw disk access [17]. To date, few Java systems have striven to take full advantage of these interfaces, instead focusing on the other aspects of performance, such as compilation [18], garbage collection [1], and thread performance [2].

Traditionally, Java applications make use of low-level system functionality through the use of *native methods*, which are written in a language such as C. To bind native method code to the Java application, a *native method interface* is used, which has been standardized across most JVMs as Sun Microsystems' Java Native Interface [29]. However, the use of native methods raises two important concerns. The first is performance: the cost of traversing the native method interface can be quite high, especially when a large amount of data must be transferred across the Java-native code boundary. The second is safety: invoking arbitrary native code from a Java application effectively negates the protection guarantees of the Java Virtual Machine. These two problems conflate, as programmers tend to write more application code in the native language to amortize the cost of crossing the native interface.

We have developed a system called Jaguar [37] which provides applications with efficient and safe access to low-level system resources. This is accomplished through a bytecode specialization technique in which certain Java bytecode sequences are translated to low-level code which is capable of performing functions not allowed by the JVM, such as direct memory access. Because this low-level code is inlined into the Java application at compile time, the overhead of the native interface is avoided. Also, aggressive optimizations can be performed on the combined application and inlined Jaguar code.

Our original prototype of Jaguar, described in [37], made use of a Java just-in-time (JIT) compiler which recognized certain bytecode sequences and translated them to x86 machine code directly. This approach is both non-portable and error-prone, requiring that the machine code sequences be tailored for the particular combination of JVM, JIT,

and machine architecture. It also requires non-trivial changes to the JIT compiler to recognize and translate Java bytecode sequences.

This paper presents a new design based on translating Java bytecode to *Jaguar bytecode*, which is a superset of the Java instruction set. Jaguar bytecode is used to represent low-level operations otherwise unsupported by the JVM. Jaguar bytecode contains two additional instructions — *peek* and *poke* — which are used to perform direct memory access. Application programmers are not permitted to make use of these instructions; rather, they are restricted to use within Java-to-Jaguar translation rules. Because Jaguar bytecode is portable, type-exact, and inlined directly into the Java application, it is both safer and more efficient than using native methods for low-level operations.

We present two implementations of Jaguar based upon this design. The first makes use of a static Java compiler to translate Jaguar bytecode to machine code. The second uses a modified JIT compiler which applies a "patch" to produce Jaguar bytecode at compile time; this mechanism requires Jaguar support only in the JIT compiler, and is fully compatible with unmodified JVMs. Both approaches relegate the (relatively expensive) translation from Java to Jaguar bytecode to a portable front-end compiler, minimizing the complexity impact on the back-end compilers.

2 Motivation and Background

Much previous work has addressed the CPU-related aspects of Java performance, including compilation [28, 18, 11], thread synchronization [2], and garbage collection [30]. However, Java I/O performance remains largely uninvestigated. Implementing high-performance communication and I/O in Java requires two classes of operations which the Java Virtual Machines does not directly support. The first is direct access to I/O device interfaces, including user-level network interfaces [35, 32] and raw disk I/O [17]. These mechanisms generally require the use of specialized system calls, or even memory-mapped interfaces which circumvent the O/S kernel entirely. The second is the use of explicitly-managed memory regions. For example, user-level network interfaces often require that communication buffers be pinned to physical memory for direct access by the NI hardware; these pages must be allocated from a special pool or pinned dynamically by the O/S or NI [36]. Memory-mapped files are often used for I/O, and raw disk interfaces usually have special requirements for buffer allocation. However, this requirement runs counter to the existing Java model in which all objects and arrays are allocated from a single heap, managed by the JVM's garbage collector.

Native methods are the traditional mechanism by which these operations are implemented in Java. Native methods permit the binding of arbitrary code written in a lower-level language (such as C) to the Java application. Apart from raising serious protection and safety concerns, native methods are not well-suited for providing fine-grained, efficient access to system resources. As we demonstrated in [37], the Java Native Interface itself imposes a large performance penalty, making them suitable only for relatively long operations which pass only a small amount of data across the Java-native code boundary. Furthermore, native methods limit expressiveness, as native code can only be bound to method invocation. Field access, operators, and other Java operations cannot hook into the native code interface in any way.

Several projects have attempted to bind fast communication layers into the Java environment through the use of native methods. Native method bindings to MPI [13] and PVM [33] have been described, however, neither of these have considered performance issues with respect to obtaining low latency or high bandwidth. Little work has been done to support other types of I/O, although Sun Microsystems is currently considering new APIs supporting asynchronous I/O interfaces [25].

Another approach is to extend the JVM or Java compiler to provide special support for certain classes, such as communication buffers [9], bulk I/O interfaces [4], or JVM-internal data structures [8]. While this technique can provide a point solution for particular communication and I/O interfaces, it lacks generality and requires that significant new functionality be incorporated into each JVM implementation. In addition, this functionality must itself be trusted, much in the same that native methods are.

The Jaguar approach is motivated by the observation that the sort of low-level operations required for enabling high-performance communication and I/O are generally short and easily expressed as a sequence of simple instructions (e.g., accessing a particular memory address, or invoking a system call). This suggests that such operations can be *inlined* into the compiled Java bytecode stream for performance, and that some form of static analysis could be performed to guarantee safety or type-exactness.

Jaguar is based on the concept of *bytecode spe-*

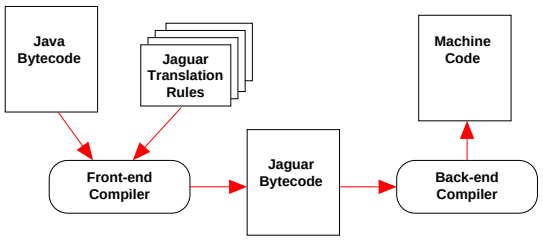

Figure 1: *The design of the Jaguar system.* A *front-end compiler* applies a set of translation rules to a Java classfile to produce a Jaguar classfile, which is then compiled by a *back-end* compiler to machine code.

cialization in which Java bytecode is specialized at compile time to perform low-level operations. Our original prototype of Jaguar, described in [37], implemented this specialization through a set of bytecode-to-machine code translation rules within a JIT compiler. While this approach produces highly efficient code, it is not very flexible. Because the machine code is tailored for a particular JIT and machine architecture, it is non-portable, making it difficult to support Jaguar on another system. Also, the bytecode translation rules themselves are complex, making it difficult to add rules to support new functionality. Also, implementing specializations in low-level machine code does not allow that code to easily verified for correctness.

3 Design and Implementation

Our original prototype of Jaguar [37] directly translated Java bytecode sequences to specialized x86 machine code within a JIT compiler. While this approach performs well, it is desirable to represent Jaguar specializations in a form that is portable and verifiable for safety. Also, we wish to make it easy to add Jaguar support to an existing JVM or Java compiler, thereby reducing implementation complexity and allowing Jaguar to be widely deployed.

Our new design is based on *Jaguar bytecode*, which is a superset of the Java bytecode instruction set containing additional instructions which enable access to low-level system resources. Bytecode specializations are represented as a set of *translation rules* from Java bytecode to Jaguar bytecode. These translation rules are applied by a *front-end compiler* which takes standard Java classfile as input, and produces a Jaguar-specialized classfile as output. The *back-end compiler*, which can be a static or just-in-time compiler, translates Jaguar bytecode

Instruction	Stack effect
`<type>peek`	..., `int` *address* ⇒ ..., `<type>` *value*
`<type>poke`	..., `int` *address*, `<type>` *value* ⇒ ...

Figure 2: *Effect of Jaguar instructions on the Java stack.* `<type>` represents one of the Java primitive types `byte`, `short`, `int`, or `long`.

to machine code. Our design is illustrated in Figure 1.

An example of a Jaguar translation rule is to convert a call to a particular method (represented in Java bytecode using the `invokevirtual`, `invokestatic`, or `invokespecial` instructions) to a sequence of Jaguar bytecodes implementing some low-level functionality, such as access to an I/O interface. In this case, the original method is never invoked; the call is replaced by inlined Jaguar bytecode performing the low-level operation. Translation rules are not restricted to use on method calls: object field accesses, operators, or any other sequence of Java bytecodes can be specialized as well. For example, field accesses on an instance of a particular class could be translated into access to a memory-mapped I/O device.

The Jaguar bytecode instruction set is machine-independent and type-exact, allowing it to be efficiently compiled on many architectures, and verified for safety. These characteristics are mainly derived from the fact that Jaguar bytecode is a superset of Java bytecode. The current version of the Jaguar instruction set includes just two additional instructions — *peek* and *poke* — which allow direct (virtual) memory access by specialized bytecode. Each instruction represents a memory address using a 32-bit *int* on the Java stack.[1] These instructions are strongly typed, with variants for the Java primitive types *byte*, *short*, *int*, and *long*. Therefore the *peek* or *poke* instructions only accept and produce objects of the given type on the Java stack; for example, the *ipeek* instruction only produces a value of type *int*. The effect of these instructions on the Java stack is shown in Figure 2.

This typing does not of course guarantee that the values read or written by *peek* or *poke* will conform to any particular type in the underlying memory representation. For example, a given memory address could be subsequently accessed as either a *byte* or an *int*. However, it does guarantee that spe-

[1] A later version of the instruction set will use a 64-bit *long*, in anticipation of the use of Jaguar on 64-bit architectures.

cializations produced using these instructions will be well-formed with respect to the Java stack; the verifier can determine whether a given instruction is being used appropriately, thereby avoiding a large class of buggy translation rules.

Because these instructions provide unguarded memory access, they should not be exposed directly to application programmers. Therefore, Jaguar bytecodes can only be used within translation rules which define specializations to application code. Because these translation rules can emit instructions which bypass Java's protection model, they must be trusted. However, we believe that placing this trust in Jaguar translation rules is more desirable than allowing arbitrary native code, for several reasons. First, because Jaguar bytecode is machine-independent and type-exact, it is more readily verified for correctness than code written in a low-level language such as C. Second, translation rules can be restricted in a number of ways; for example, the front-end compiler could reject translation rules which made use of certain instructions, or produce sequences of Jaguar bytecode longer than a certain length.

The Jaguar front-end compiler and translation rules are themselves implemented in Java. When the front-end compiler starts, it loads classes corresponding to the translation rules specified in a user-supplied configuration file. It then reads a Java class file and iterates over the instructions in each method, calling a method in each translation rule class. The rule inspects the current instruction and can ignore it or replace it with zero or more Jaguar instructions. In addition, translation rules can add entries to the target class' constant pool, create additional fields or methods, and modify exception handler table entries.

The Jaguar back-end compiler can be a straightforward modification to an existing Java compiler; adding support for the additional Jaguar instructions is relatively easy. The following sections present two implementations of this design: one based on a static Java compiler and the other based on a JIT compiler.

The process of compiling a Java method using Jaguar translation rules and a back-end compiler is illustrated in Figure 3. The method `VipPostSend` is used to transmit data using the Berkeley VIA user-level network interface [7]. It makes use of a special object, the `VIA_Doorbell`, which represents a register on the network interface hardware. The method calls on the doorbell object (`set` and `isBusy` are specialized by the Jaguar front-end compiler to directly access the doorbell's value using the Jaguar

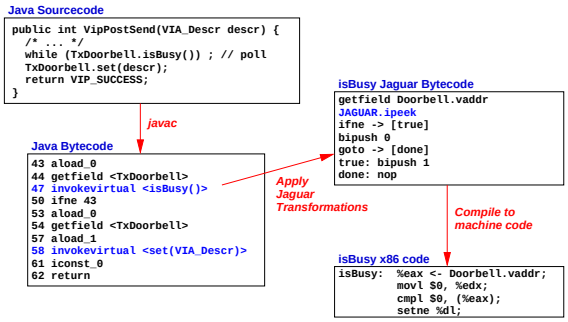

Figure 3: *Example of Jaguar bytecode specialization.* The top left shows the Java source code for the `VipPostMethod`, which transmits data using the VIA communication interface. The corresponding Java bytecode is shown below. The two method calls (`isBusy` and `set`) that are specialized by the Jaguar front-end compiler are highlighted. The Jaguar bytecode produced by the translation rule for the `isBusy` call is shown at right, with the Jaguar *ipeek* instruction highlighted. The Intel x86 code corresponding to the Jaguar bytecode is shown below. This method reads the value of the VIA doorbell device register and returns a boolean value based on whether the doorbell's value is non-zero.

ipeek instruction. Note that the Jaguar bytecode for these calls is inlined into the original application code. The resulting Jaguar bytecode sequence is rendered by the back-end compiler as four x86 machine code instructions.

3.1 Static Compiler Implementation

Our first implementation is based on a modified version of GCJ [5], an static Java compiler based on GCC. GCJ compiles Java source or class files to standard object files and links them together with a library implementing runtime support for threads and garbage collection into an executable. GCJ produces very efficient code and runs across multiple architectures and operating systems. The garbage collector is based on Boehm's incremental copying collector [3]. Many standard Java libraries are supported, and the runtime includes a Java bytecode interpreter for executing dynamically-loaded classes. GCJ supports two native code interfaces: Sun Microsystems' JNI [29], as well as the Cygnus Native Interface (CNI) [26], an efficient Java binding to C++.

The Jaguar front-end compiler takes a Java classfile and produces a Jaguar classfile containing Jaguar bytecode (which may be the same as the Java bytecode, if no translation rules were applied).

The Jaguar classfile is distinguished by the filename extension `.jagc` and a magic number in the header; otherwise, it is identical in format to a Java classfile. GCJ was modified to accept Jaguar instructions if a Jaguar classfile is being compiled. This required adding only 98 lines of code to GCJ. 36 of these lines generate intermediate representation structures for the additional Jaguar instructions, and 33 lines add verification support. The modified GCJ has been tested on Intel x86 systems running Linux, but should work across all architectures and operating systems supported by GCJ.

3.2 JIT Compiler Implementation

Our second implementation makes use of a modified just-in-time compiler which works in tandem with a standard JVM. This is useful as it gives Jaguar-based applications access to the wide range of APIs supported by standard Java implementations, as well as dynamic class loading, albeit at the expense of somewhat lower performance than GCJ. We have added Jaguar support to OpenJIT [19], a portable JIT compiler for x86 and SPARC platforms which supports several operating systems. OpenJIT is unique in that it is itself implemented in Java, making it particularly easy to add support for Jaguar. We have experimented with OpenJIT on x86 Linux running Sun JDK 1.1.7v3.

Implementing Jaguar support in a JIT compiler raises two new concerns. The first is that we wish to avoid modifying the JVM itself, to minimize the impact on existing Java implementations. Unfortunately, modifications to the JVM are necessary if Jaguar instructions are to pass by the bytecode verifier (which is implemented within the JVM, not the JIT). In our case, the JVM source code is unavailable, so modifying the JVM is not an option anyway. The second concern is that the compilation overhead of Jaguar bytecode support should be minimal, since any overhead within the JIT compiler is reflected in the runtime of the application.

One approach to adding Jaguar support to a JIT compiler is to do the bytecode specialization as late as possible; that is, to move the Jaguar translation rules into the JIT compiler itself (and hence, after the original Java bytecode has passed verification by the JVM). This is problematic as our current translation mechanism is not particularly efficient, and would have a noticeable performance impact if implemented in the JIT. This problem could be alleviated by using a more streamlined approach to bytecode translation, such as precompiled pattern matching.

To avoid these problems, the Jaguar front-end compiler performs bytecode translation and stores a *code patch* in the Java classfile describing the set of changes to the Java bytecode produced by the translation rules. The JIT compiler then applies this patch before compiling the code for a method. The patch is stored as an optional attribute in the method descriptor of the classfile, and is passed to the JIT compiler using a standard interface to the JVM [20]; the JVM otherwise ignores this attribute. Methods to which no translation rules have been applied have an empty code patch and incur no storage or compile-time overhead.

We implemented two code patch techniques. The first simply stores the expanded Jaguar bytecode for a method; while this can significantly increase the size of a patched classfile, it is trivial to apply the "patch" at compile time. The second stores a set of differences between the Java and Jaguar bytecode based on a longest-common-subsequences analysis of the two [21], which reduces patch size but requires more overhead to apply.

Code patching raises the concern that the patch itself may produce invalid Jaguar bytecode. This concern can be alleviated by reverifying the Jaguar bytecode after the patch has been applied. While our current prototype does not perform reverification, the OpenJIT compiler will catch many invalid bytecode sequences during compilation. To reduce the overhead of reverification, it may be possible to perform a "fast verification" of only those bytecodes affected by the patch; however, this becomes more complicated if the patch changes the control flow of the method in question.

Another issue is whether the code patch should be trusted by the application (that is, that it correctly implements the specializations required by the application). Both the verification and trust issues can be addressed using techniques such as code signing, which authenticate the source of the code patch (say, as being produced by a trusted set of translation rules). Our prototype currently assumes that patched classfiles are trusted. Note, however, that the mechanism used to deliver the patch to the JIT (placing it within a code attribute of the classfile) only works for classfiles loaded from the local filesystem; the JVM will not pass along this attribute for classfiles loaded over the network or from another source. This provides one level of protection, requiring that code patches be generated by a Jaguar front-end compiler running locally.

Adding Jaguar support to OpenJIT required changes to 4 lines of code, and the addition of 164 new lines of code. Of these, 70 lines implement diff

patch support, 7 lines implement the flat patch, and 43 lines construct the intermediate representation for the additional Jaguar bytecode instructions.

4 Analysis

In this section we analyze the two implementations of Jaguar described above. First, we measure the performance of several benchmark applications using both implementations. Our results show that GCJ produces marginally faster code for various microbenchmarks; larger applications will probably see a more pronounced performance difference. Second, we measure the overhead of the two code patch techniques in the Jaguar-enhanced JIT compiler, demonstrating that the difference in storage and compilation time overhead between the two techniques is minimal.

4.1 Benchmark Performance

All of the measurements below were taken on a 2-way 500 MHz Pentium III SMP system running Linux 2.2.13 and 512 MB of memory. Jaguar support was added to GCJ v2.95.2 and OpenJIT v1.1.10; the JVM used with OpenJIT is the Blackdown port of Sun JDK v1.1.7v3 [31] using green threads.

Figure 4 shows the round-trip latency and bandwidth of the Berkeley VIA user-level network interface [7] as accessed from C code, as well as in Java using both Jaguar/GCJ and Jaguar/OpenJIT. These measurements were taken using a PCI Myrinet interface card with a LANai 4.3 chipset and 1 megabyte of SRAM, with a single Myrinet switch between the two benchmark systems.

VIA communication performance is a good measurement of the performance of Jaguar, as it makes heavy use of bytecode specialization for access to low-level device registers as well as to network buffers outside of the Java heap. The Jaguar interface to VIA is described in detail in [37]. As the results show, both Jaguar implementations match the performance of the C-based benchmark exactly. The round-trip latency for small messages is 80 microseconds, and peak bandwidth is 480 megabits/sec for 100 KB packets.[2]

Figure 4 also shows estimated communication bandwidth if the Java Native Interface (JNI) were used to provide VIA functionality in Java. In the estimation, the overhead of using JNI (as measured in [37]) was added to the measured per-message cost and the resulting bandwidth recalculated. We assume that each native method call costs 1.0 μsec and that copying data from Java to native code costs 270 μsec per kilobyte. Four native method calls are required per message transmitted. The estimated bandwidth peaks at 28.55 megabits/sec, a factor of 17 less than JaguarVIA. Even if the performance of the native interface were a factor of 10 faster, the peak bandwidth would be only 187 megabits/sec, far below that obtained with Jaguar.

Benchmark	**Jaguar/ GCJ**	**Jaguar/ OpenJIT**
Create PSO object	3.5 μsec	4.2 μsec
Recover PSO reference	3.3 μsec	3.1 μsec
Follow PSO reference	0.16 μsec	0.29 μsec
Assign PSO `int` field	0.014 μsec	0.035 μsec
Assign Java `int` field	0.008 μsec	0.018 μsec
Write `int PSOArray` element	0.022 μsec	0.057 μsec
Read `int PSOArray` element	0.014 μsec	0.049 μsec
Write `int` array element	0.017 μse	0.023 μsec
Read `int` array element	0.008 μsec	0.035 μsec

Figure 5: *Pre-Serialized Object microbenchmarks.* The time for several primitive operations on Pre-Serialized Objects is shown for both Jaguar implementations.

Figure 5 shows the performance of a benchmark testing *Pre-Serialized Object* (PSO) operations under both implementations of Jaguar. Pre-Serialized Objects, described in detail in [37], are a mechanism used to reduce the cost of Java object serialization, which is commonly used to transmit Java objects over a network, or to store object data on disk. Serialization has a strong impact on the performance of Java Remote Method Invocation (RMI) implementations [22].

Pre-Serialized Objects make use of Jaguar bytecode specializations to affect the layout of Java object fields; the idea is to store objects in memory in a form that is "pre-serialized", ready for storage or communication. PSOs are implemented through bytecode specializations which recognize `putfield` and `getfield` accesses to objects of some class, marshaling object data into and out of its pre-serialized form. A `PSOArray` is a PSO which provides methods (such as `readByte` and `writeByte`) allowing it to be treated as an array.

The benchmark measured in Figure 5 first creates a linked list of PSO objects and assigns values to the object fields. Next, the benchmark simulates

[2]The measurements here use a newer version of the VIA software than that described in [37]; the new version has slightly lower performance as additional features have been added, including support for remote DMA operations.

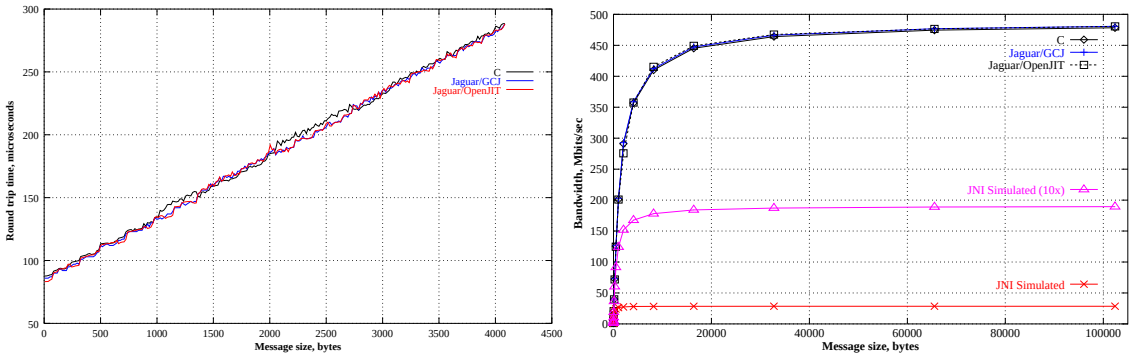

Figure 4: *JaguarVIA microbenchmark results.* The graph on the left shows the round-trip latency of messages transmitted over Berkeley VIA as a function of the message size. The graph on the right shows the bandwidth as a function of message size for C and Jaguar, as well as an estimation of the bandwidth obtained if JNI were used to access VIA functions as native methods. Also shown is the bandwidth if JNI performance were improved tenfold.

recovery of the objects from the pre-serialized form by mapping a new PSO object list onto the data. Recovering references to PSO objects requires demarshalling a pointer and creating a new object to map onto the pointer's location.

As the results show, GCJ has a significant performance advantage over OpenJIT for these primitive operations. In particular, GCJ has highly-optimized routines for accessing object fields and arrays; OpenJIT must deal with the Java object representation within the JVM, and does not optimize code as heavily as the static compiler. Accessing PSOs is slower than accessing standard Java objects in both cases, because these primitives are constructed using multiple Java object field accesses (as well as Jaguar *peek* and *poke* instructions).

4.2 Code Patch Overhead

To assess the overhead of Jaguar code patches, we measured the size difference between original Java classfiles and those patched using both the two patch mechanisms. We also measured the size of Jaguar classfiles generated for use with the static Jaguar compiler (which contain only expanded Jaguar bytecode, and no patch). The results are shown in Figure 6. 27 classes in the Jaguar code tree were processed by the Jaguar front-end compiler; only 7 of these classes had Jaguar translation rules applied to them. As the figure shows, several classes decreased in size when processed by the front-end compiler; others increased in size even if no translation rules were applied. This is because of the way in which the Jaguar front-end compiler parses and writes out classfile data. Ideally, a class file with no translation rules applied to it would be exactly the same size after processing.

The average classfile size increase was 698 bytes using the diff patch method, and 738 bytes using the flat method. The average increase for Jaguar classfiles was 494 bytes. Note that using the diff approach does not have significant savings over simply storing the fully-expanded Jaguar bytecodes; this is because the number of instructions added by a patch tends to be relatively large compared to the number of instructions in the original method. (Most Jaguar translation rules take a single Java bytecode instruction and expand it out to 5-20 new instructions.)

The time to apply Jaguar code patches and compile methods using both patch techniques is shown in Figure 7. These results show that applying a diff patch takes about twice as long as applying a 'flat patch. However, the number of methods patched is very small, less than 2% in most cases. Although the time to apply a patch is about 10% of the average method compilation time, because the number of patched methods is small the total patch time for an application is negligible. Although these are small benchmark applications, we expect that the use of Jaguar specializations (and hence, code patches) will be restricted to a small set of low-level libraries in a given application. In the case of the VIA benchmarks, for example, only the core VIA library makes use of code patches.

5 Issues and Future Work

Our approach to bytecode specialization raises a number of issues for future investigation. The first is whether the class of operations required for im-

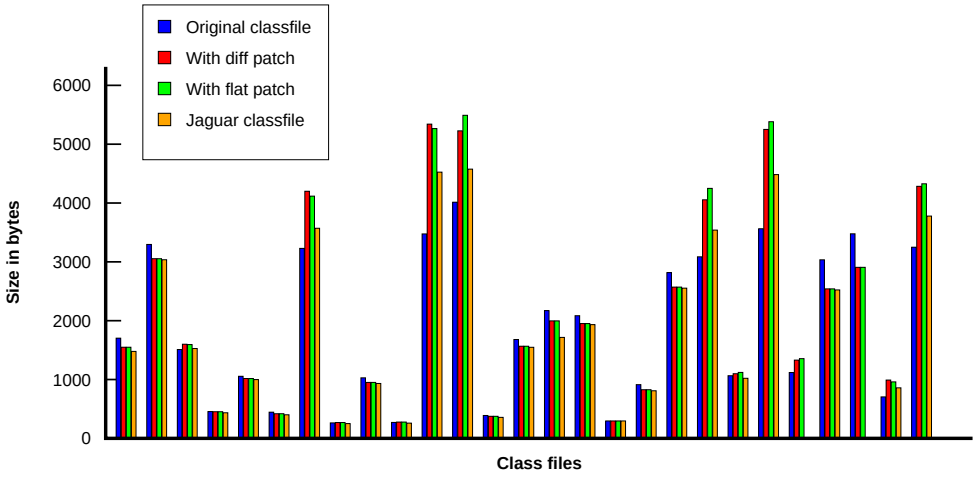

Figure 6: *Classfile sizes using Jaguar.* The size in bytes of 27 Java classfiles, before and after processing by the Jaguar front-end compiler, are shown. Patched classfiles are used with the JIT-based Jaguar implementation, and Jaguar classfiles are used with the static compiler implementation. The decrease in size for some classfiles is attributed to removal of extraneous information during processing, and differing data layouts.

plementing efficient communication and I/O in Java can be captured by just two additional bytecode instructions (*peek* and *poke*). Direct memory access is a requirement for memory-mapped I/O interfaces, and for manipulation of other memory regions (such as I/O buffers) outside of the Java heap. However, other communication layers, such as MPI [12] and Active Messages [10], operate primarily through a function-call interface. In addition, most operating system features can only be accessed through the use of system calls.

This suggests that the Jaguar instruction set should provide a binding to arbitrary library and system calls; however, this is already provided (indirectly) through the use of Java native methods. The main performance limitation of the Java Native Interface is in the way that native code accesses Java data, and vice versa. Jaguar External Objects (described in [37]) can be used to avoid this bottleneck, as Jaguar specializations allow Java and native code to directly share data external to the Java heap. Sharing data on the Java heap is more difficult, as the garbage collector must be informed when objects are being accessed outside of Java code. The remaining concern with native methods is portability and safety. One solution to these problems would be to introduce a restricted form of native methods which map only onto standardized APIs, such as POSIX calls, and are responsible only for marshaling data between Java and the library or system call in question.

Another set of optimizations that Jaguar could enable is fast floating-point operations and complex arithmetic using standard Java operators. These optimizations have been investigated using other techniques, including class-specific compiler optimizations [39] and dialects of the Java language [6]. While augmenting the Jaguar instruction set with support for these optimizations fits nicely into the goal of hardware specialization of Java applications, it is not clear whether these instructions would be portable across many architectures. Our desire has been to strike a balance between performance, portability, and generality.

The real opportunity presented by Jaguar is to build novel systems in Java. Apart from the benefits of increased portability, modularity, and robustness, there is the potential to leverage Java's protection model to achieve higher performance than can be afforded by languages such as C. This is possible because code that was once protected from the application by placing it in the kernel can now be moved into a Java library, and co-optimized with application code at compile time. We have investigated the benefit of moving the protection code of the Berkeley VIA implementation from the slow network interface co-processor (which runs at 37 MHz) into a Java library on the Pentium-based host, re-

Benchmark	Methods Patched	Methods Total	Average patch size	Average patch time	Average compile time
JaguarVIA Pingpong	6	215			
Diff patch			470 bytes	1.5 ms	11.92 ms
Flat patch			438 bytes	1.16 ms	11.92 ms
JaguarVIA Bandwidth	4	214			
Diff patch			460 bytes	3.0 ms	11.93 ms
Flat patch			505 bytes	1.5 ms	11.93 ms
PSO Benchmark	3	221			
Diff patch			341 bytes	2.66 ms	11.34 ms
Flat patch			343 bytes	1.33 ms	11.34 ms
PSO Test	3	194			
Diff patch			442 bytes	2.66 ms	12.25 ms
Flat patch			510 bytes	1.33 ms	12.25 ms

Figure 7: *Patch application and compile time for various microbenchmarks.* Also shown are the number of methods compiled in the application, and the number of methods which were patched. Note that the number of patches applied is very low compared to the total number of methods used in the application.

sulting in a 15-20% increase in peak bandwidth and a 25% reduction in round-trip latency. Using Java's protection model in place of the traditional system-call boundary could improve other aspects of application performance, including support for high I/O concurrency and throughput [38, 15].

6 Conclusion

Java server applications must necessarily make use of functionality not directly provided by the JVM. These applications require both access to low-level system resources, such as fast communication and I/O interfaces, as well as management of memory external to the Java heap. While native methods provide a general-purpose way to invoke code written in a lower-level language from Java, they are generally non-portable, do not perform well, and raise important safety concerns.

Jaguar takes an alternate approach, that of specializing Java bytecode at compile time to make use of low-level operations expressed as type-exact, portable Jaguar bytecode. Jaguar bytecode is a superset of Java bytecode with just two additional instructions enabling direct memory access for specializations. Adding Jaguar support to an existing JVM is straightforward, and does not require the addition of significant new functionality. Because Jaguar bytecode is inlined into the application, the compiler can also perform aggressive optimizations against the combined application and low-level code.

We have presented two implementations of the Jaguar design, one based on a static Java compiler, and the other based on a JIT compiler. The static compiler makes use of a Jaguar classfile containing specialized bytecodes; the JIT solution uses a code patch embedded in a standard Java classfile. We have demonstrated both implementations to perform well against a set of communication and I/O benchmarks, and that the overhead of the code patch technique is minimal. Our design represents an efficient, safe means of extending the Java environment to support large-scale server applications that can be readily employed in existing Java implementations.

References

[1] Ole Agesen. GC Points in a Threaded Environment. Technical Report SMLI-TR-98-70, Sun Microsystems Laboratories, December 1998.

[2] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of ACM Conference on Programming Language Design and Implementation*, June 1998.

[3] Hans Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.

[4] Dan Bonachea. Bulk File I/O Extensions to Java. In *Proceedings of the ACM 2000 Java Grande Conference*, June 2000.

[5] Per Bothner. A GCC-based Java Implementation. In *Proceddings of the 42nd IEEE International Compuer Conference*, Spring 1997.

[6] John Brophy. Design of the visual numerics complex class. http://www.vni.com/corner/garage/grande/.

[7] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the Virtual Interface Architecture. In *Proceedings of SC'98*, November 1998.

[8] Michael Burk, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V.C. Sreedhar, and Harini Srinivasan. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[9] Chi-Chao Chang and Thorsten von Eicken. Interfacing Java with the virtual interface architecture. In *ACM Java Grande Conference 1999*, June 1999.

[10] B. Chun, A. Mainwaring, and D. Culler. Virtual network transport protocols for Myrinet. In *Proceedings of Hot Interconncts V*, August 1997.

[11] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgard, and David Tarditi. Marmot: An Optimizing Compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

[12] Message Passing Interface Forum. The MPI Message Passing Interface Standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, April 1994.

[13] Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[15] Steven Gribble. Simplifying Cluster-Based Internet Service Construction with Scalable Distributed Data Structures. http://www.cs.berkeley.edu/~gribble/papers/quals/sdds-cluster.ppt.

[16] Joe Hellerstein, Eric Brewer, and Mike Franklin. Telegraph: A Universal System for Information. http://db.cs.berkeley.edu/telegraph/.

[17] Silicon Graphics Inc. SGI Releases Technology to Form Foundation for Enterprise-Class Linux Applications. http://www.sgi.com/newsroom/press_releases/1999/december/linux_oss.html.

[18] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 Java Grande Conference*, June 1999.

[19] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT: A Reflective Java JIT Compiler. In *Proc. of OOPSLA '98, Workshop on Reflective Programming in C++ and Java*. http://openjit.is.titech.ac.jp/.

[20] Sun Microsystems. The JIT Compiler Interface Specification. http://java.sun.com/docs/jit_interface.html.

[21] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica 1*, 2:251–266, 1986.

[22] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *ACM Java Grande Conference 1999*, June 1999.

[23] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[24] Michael Philippsen and Matthias Zenger. JavaParty - transparent remote objects in Java. In *Concurrency: Practice and Experience, 9(11):1225-1242*, November 1997.

[25] Mark Reinhold. New I/O APIs for the Java Platform. Sun Microsystems Java Community Process Specification JSR-000051, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_051_ioapis.html.

[26] Cygnus Solutions. The Cygnus Native Interface for C++/Java Integration. http://sourceware.cygnus.com/java/papers/cni/t1.html.

[27] Sun Microsystems Inc. Enterprise Java Beans Technology. http://java.sun.com/products/ejb/.

[28] Sun Microsystems Inc. Java HotSpot Performance Engine. http://java.sun.com/products/hotspot/index.html.

[29] Sun Microsystems Inc. Java Native Interface Specification. http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html.

[30] Sun Microsystems Labs. The Exact Virtual Machine (EVM). http://www.sunlabs.com/research/java-topics/.

[31] The Blackdown Java Linux Porting Team. Java on linux. http://java.blackdown.org/.

[32] The VIA Consortium. The Virtual Interface Architecture. http://www.viarch.org.

[33] D.A. Thurman. jPVM: The Java to PVM interface. http://www.isye.gatech.edu/chmsr/JavaPVM.

[34] UC Berkeley Ninja Project. The UC Berkeley Ninja Project. http://ninja.cs.berkeley.edu.

[35] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, December 1995.

[36] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects V*, August 1997.

[37] Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O from Java. *Concurrency: Practice and Experience*, 2000. Special Issue on Java for High-Performance Network Computing, To appear, `http://www.cs.berkeley.edu/~mdw/papers/jaguar-journal.ps.gz`.

[38] Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A design framework for highly-concurrent systems. October 2000.

[39] Peng Wu, Sam Midkiff, Jose Moreira, and Manish Gupta. Efficient support for complex numbers in java. In *Proceedings of the ACM 1999 JavaGrande Conference*, June 1999.

[40] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998.