Jaguar: Enabling Efficient Communication and I/O in Java

Matt Welsh and David Culler
University of California, Berkeley
{mdw,culler}@cs.berkeley.edu

Abstract

Implementing efficient communication and I/O mechanisms in Java requires both fast access to low-level system resources (such as network and raw disk interfaces) and direct manipulation of memory regions external to the Java heap (such as communication and I/O buffers). Java native methods are too expensive to perform these operations and raise serious protection concerns. We present *Jaguar*, a new mechanism which provides Java applications with efficient access to system resources while retaining the protection of the Java environment. This is accomplished through compile-time translation of certain Java bytecodes to inlined machine code segments. We demonstrate the use of Jaguar through a Java interface to the VIA fast communications layer, which achieves nearly identical performance to that of C, and *Pre-Serialized Objects*, a mechanism which greatly reduces the cost of Java object serialization.

1 Introduction

The Java programming environment [9] has made significant headway in support of a wide array of application areas, including mobile agent systems [24], distributed programming models [15], enterprise-wide information processing [12], and scientific and numerical computing [7]. As Java's popularity grows, so will the demands placed upon it to support even more diverse computing platforms, from embedded systems [23] to workstation clusters [28]. If we wish to bring Java to bear on large problems, it seems natural that Java should take advantage of the resources of large-scale servers, including multiprocessors, high-speed networks, and fast I/O.

A great deal of previous work has addressed problems with Java processor performance, namely, efficiency of compiled code, thread synchronization, and garbage collection algorithms [13, 16]. Java compilers, including both static and "just-in-time" (JIT) compilers, are now capable of generating code which rivals lower-level languages such as C++ in performance [20]. However, in a large server environment, high-performance communication and I/O play a dominant role. These aspects of Java performance remain largely unaddressed.

Implementing efficient communication and I/O generally requires the application to invoke operating system calls, perform direct manipulation of memory (e.g., use pointers) to access memory-mapped I/O and network devices, and so forth. Unfortunately, these operations are inexpressible as machine-independent Java bytecodes. Rather than exposing low-level (and hence unsafe) machine operations directly to the Java application, it is desirable to abstract away the *access mechanisms* required for communication and I/O as a Java object replete with type-safe methods and fields. Operations on these Java objects should translate into efficient and direct (as much as possible) access to the low-level machine resources they represent, while retaining the protection of the Java environment.

Java provides so-called "native methods" which enable code implemented in another language (such as C) to be invoked from Java. This is typically used to abstract O/S calls and other functions as Java classes. However, native methods incur a high overhead, requiring that data be copied between Java and native code; in addition, implementing native methods in a low-level language is error-prone and potentially negates the safety guarantees of the Java sandbox. Considering these performance and safety limitations, we believe native methods are ill-suited to enable high-performance communication and I/O in Java.

We present an alternate approach, *Jaguar*,[1] which enables direct, protected access to system resources represented as Java objects. This is accomplished through compile-time code transformation which maps certain Java bytecodes to short, inlined machine code segments. This approach retains the type-safety and protection of the Java environment while allowing applications to directly leverage

[1] Jaguar is an acronym for *Java Access to Generic Underlying Architectural Resources*. For more information, see `http://www.cs.berkeley.edu/~mdw/proj/jaguar`.

server resources such as memory-mapped network interfaces, raw disk I/O, and so forth. We demonstrate Jaguar through two examples: *JaguarVIA*, a Java binding to the VIA [5] fast communications substrate, and *Pre-Serialized Objects*, a mechanism which greatly reduces the cost of rendering Java objects in an externalized form for communication or I/O. We believe that the approach taken in Jaguar is general enough to capture a large range of additional uses.

This paper makes three major contributions. First, we present a novel approach to enabling efficient and safe utilization of server hardware resources from Java. Second, we present *JaguarVIA*, which obtains the same VIA communications performance as C. Third, we present Pre-Serialized Objects, and demonstrate how their use, as enabled by Jaguar, can eliminate the high overhead of Java object serialization.

The organization of the rest of this paper is as follows. Section 2 provides background on the issues faced by Jaguar and related work. Section 3 describes the design and implementation of Jaguar, and Section 4 demonstrates its use through a fast Java binding to VIA. Section 5 describes Pre-Serialized Objects. Section 6 discusses directions for future work and Section 7 concludes.

2 Motivation and Background

In this section, we motivate the approach taken by Jaguar by looking more closely at the problems which it addresses.

A number of performance issues arise when one considers implementing large-scale server applications in Java. These can be roughly divided into two categories: CPU-related issues and I/O-related issues. In terms of the CPU, performance of compiled Java code is the paramount concern, but other factors — including garbage collection and thread synchronization — must be considered as well. Fortunately, a great deal of previous work has investigated this problem domain, for Java [13, 16] as well as other object-oriented languages [2, 6].

Java I/O performance remains largely uninvestigated. A primary goal is to give Java applications efficient access to low-level system resources (such as fast network interfaces, I/O and RAID controllers, and so forth); such access is necessary for implementing high-performance communication and I/O. It is this set of problems which Jaguar intends to solve.

Traditionally, the operating system is responsible for providing applications access to hardware, either through high-level interfaces (such as filesystems and sockets) or lower-level mechanisms (such as raw disk I/O calls). However, in many cases it is desirable to circumvent the operating system to obtain higher performance. In the case of fast networking, user-level network interfaces provide low-overhead communication while allowing multiple processes to safely share the network interface (NI). Applications circumvent the operating system kernel and directly access network interface resources, such as memory-mapped data structures or "protected" NI registers. Doing so eliminates context switch overhead and the cost of copying data between user and kernel space. A large number of of user-level network interface prototypes have demonstrated this principle, such as Active Messages [4] and U-Net [26]; VIA [5] is a recent effort to standardize these interfaces.

A related requirement for communication and I/O 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 [27]. 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.

2.1 Related Work

Efficient I/O and communication in Java has been investigated through two primary avenues: implementing fast object serialization, and binding fast network interfaces to the Java environment. In terms of serialization, [22] describes a more efficient implementation of Java remote method invocation (RMI) which is based on careful coding and a new serialization algorithm, coded entirely in Java. Manta [17] takes the more extreme approach of translating the entire Java application to C, generating specialized per-class serialization code. While this moves much of the run-time overhead of communication to compile time, this necessitates a reengineering of the Java run-time, and the resultant environment is arguably something other than "true" Java.

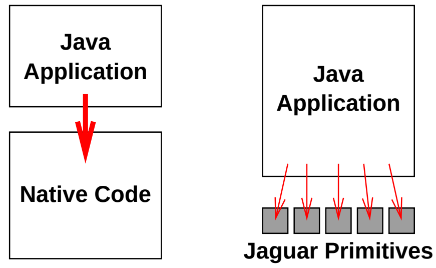
Several projects have attempted to bind fast communication layers into the Java environment through the use of native methods. Native method bindings to MPI [8] and PVM [25] have been described, however, neither of these have considered performance issues with respect to obtaining low latency or high bandwidth. The approach taken by Javia [3] is closest to that in Jaguar, which describes modifications to a static Java compiler to enable efficient bindings to a commercial VIA implementation. Here, native methods were used to invoke the C-based VIA library, while communication buffers were exposed to Java through specially-generated code from a modified compiler. While this addresses most of the performance issues with implementing a fast Java VIA interface, the approach does not cover efficient access to hardware resources in general.

2.2 Are native methods adequate?

Java native methods can provide access to low-level system functions, albeit at high cost: the overhead of invoking native methods, and transferring data between Java and native code, often outweighs their utility. This is of particular concern for fine-grained operations such as manipulation of network interface data structures. Such operations are performance-critical and should incur as little overhead as possible. Additionally, native code requires that data be copied between specially-managed memory regions (such as network buffers) and the Java heap, again resulting in high overhead.

Figure 1 details the overhead of native code invocation from Java. These measurements were performed on a 350 MHz Pentium II running Linux 2.2.5 using Sun JVM 1.1.7. Here, the standard Java Native Interface (JNI) [14] was employed, which abstracts away details of the JVM structure from native code; the intent is to allow native code to be ported across different JVM architectures. For comparison, similar tests conducted in C are shown; all compiler optimizations were disabled for the C benchmark. As the results show, use of JNI is quite expensive, requiring nearly a microsecond just to perform a native method call and return. More serious is the array-copy overhead which would surely limit the performance of any fast communication or I/O system implemented using native methods.

Regardless of performance, however, native code is a blunt instrument with which to enable low-level operations in Java. Native code must be as trustworthy as the JVM and compiler, yet its power is effectively unlimited: a native method can spin in an infinite loop, access any memory location, and crash the virtual machine. It is up to programmers to exercise proper discipline when implementing native methods, but this discipline cannot be enforced by the system in any way. Likewise, because native code is generally implemented in a low-level language such as C, it is both error-prone and non-portable; it is difficult to convince oneself that a piece of native code will work as advertised. The problem is exacerbated by the fact that native methods must generally do a large amount of work to amortize the cost of their invocation. This concern is a serious one, as it is the robustness of the Java environment which makes the language attractive in the first place.



(a) Using Native Code (b) Using Jaguar

Figure 2: *Native code and Jaguar compared.*

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. Such an approach is tantamount to extending the Java runtime with new, safe primitives which perform specialized operations on behalf of an application.

This situation is depicted in Figure 2. Rather than binding a large amount of native code to Java, Jaguar allows the Java runtime to be extended with a set of new, simple primitives. Because use of these primitives is inexpensive, nearly all functionality, including complex system software, can be implemented in Java, leading to more robust applications. The use of Jaguar lends itself to a programming style which uses mostly Java and a small amount of native code, rather than the converse.

3 Jaguar Design and Implementation

Jaguar allows the Java runtime to be extended with new primitive operations which enable effi-

Benchmark	Java Native Interface	Comparable C code	Slowdown factor
void arg, void return native method call	.909 μsec	0.038 μsec	23.9
void arg, `int` return native method call	.932 μsec	0.042 μsec	22.2
`int` arg, `int` return native method call	.985 μsec	0.049 μsec	20.1
4-`int` arg, `int` return native method call	1.31 μsec	0.072 μsec	18.2
10-byte C-to-Java array copy	3.0 μsec	0.354 μsec (`memcpy`)	8.47
1024-byte C-to-Java array copy	18.0 μsec	1.68 μsec (`memcpy`)	10.7
102400-byte C-to-Java array copy	1706.0 μsec	432.5 μsec (`memcpy`)	3.94
10-byte Java-to-C array copy	7.0 μsec	0.354 μsec (`memcpy`)	19.8
1024-byte Java-to-C array copy	272.0 μsec	1.68 μsec (`memcpy`)	161.9
102400-byte Java-to-C array copy	27274.0 μsec	432.5 μsec (`memcpy`)	63.1

Figure 1: *A comparison between Java Native Interface and C overheads.*

cient access to hardware resources. These primitives are specified as short machine code segments which are directly inlined into the Java bytecode as it is compiled. The fundamental operation of a Java compiler is to translate sequences of Java bytecodes (which manipulate Java objects) into native machine code (which manipulate analogues of those objects on the actual hardware). Jaguar builds upon this concept by introducing an additional set of bytecode-to-machine code translation rules into the compiler, transforming certain bytecode sequences directly to operations on low-level hardware resources.

There are two primary concepts embodied in Jaguar: *code mappings* and *External Objects*.

3.1 Code mappings

Specifying new Java bytecodes to represent low-level machine operations would necessitate modifications to the `javac` bytecode compiler and perhaps to the Java language itself. Rather, we have chosen to apply the concept of *code mappings* which describe transformations from Java bytecode sequences to inlined machine code. In this way, preexisting bytecodes (say, method calls or field accesses) are translated to specialized operations at compile time. This approach affords a very natural programming model: low-level machine operations are expressed as operations on regular Java objects. Accessing a field or calling a method may transparently trigger an alternative sequence of machine events.

While Jaguar code mappings are similar in nature to native methods, there are two major differences:

- Jaguar code mappings may be applied to virtually any bytecode sequence (such as field accesses, operators, and so forth) while native code is limited to method invocation. As such, Jaguar enables much greater expressiveness: machine resources can be represented by Java objects, with methods, fields, or operators being used as appropriate to describe low-level operations.
- Jaguar primitives consist of short, limited sequences of machine code rather than C functions of arbitrary complexity. This property makes it easier to verify that the implementation of a Jaguar primitive is correct. It also inherently limits Jaguar code mappings to provide basic, low-level operations rather than extensive functionality. In this way, applications can be written almost entirely in Java, aided by a few simple primitives provided by Jaguar.

3.2 External Objects

Jaguar allows Java applications to directly manipulate memory outside of the Java heap, such as specially-allocated buffers for communication and I/O. Jaguar code mappings are used to rewrite field accesses on certain Java objects to directly manipulate this "external" memory; we call the result *External Objects*. External Objects are treated by the application as regular Java objects, the memory storage for which happens to be located outside of the Java heap. This eliminates the expense of copying data between Java and external memory as required by native code.

External Objects have numerous uses. They can be used to map Java object references onto shared-memory segments, memory-mapped files, communication and I/O buffers, and even memory-mapped hardware devices. Because field accesses are processed by Jaguar using knowledge of both field name and type, different behaviors can be implemented for different fields. For example, one field in an object may reference a communication buffer while another references the network interface with which it is associated.

3.3 Implementation

Our prototype of Jaguar is implemented as a Java just-in-time compiler which has been augmented with a set of transformation rules implementing Jaguar code mappings. Each such mapping describes a particular bytecode sequence and a corresponding machine code sequence which should be generated when this bytecode is encountered during compilation. An example of such a mapping might be to transform the bytecode `invokevirtual SomeClass.someMethod()` into a specialized machine code fragment which directly manipulates a hardware resource in some way.

Our prototype JIT compiles Java bytecode to machine code by performing a straightforward translation from each bytecode to a particular machine code template. Jaguar code mappings are implemented by rewriting certain Java bytecodes as Jaguar-specific "meta-bytecodes" during the first pass of the compiler; machine code templates for each such meta-bytecode are provided which implement new Jaguar primitives. For example, the operation `invokevirtual SomeClass.someMethod` might be translated to the meta-bytecode `opc_do_somemethod`, and the machine code template for `opc_do_somemethod` will be inlined into the compiled code sequence during the compiler's second pass.

Jaguar code mappings can be applied to virtually any bytecode sequence; however, they are limited in two fundamental ways:

- The system must have enough information to determine whether the mapping should be applied at compile time. This has an impact on the use of bytecode transformation for virtual methods (see below).
- Recognizing the application of certain mappings is easier than others. For example, mapping a complex sequence of `arrayref` and `add` bytecodes to, say, a fast vector-add instruction would certainly be more difficult than recognizing a method call to a particular object.

In our current prototype, these transformation rules must be compiled into the JIT compiler itself; however, we are currently working on a new implementation (based on the OpenJIT [19] compiler) which allows new code mappings to be specified at runtime. Such an approach presents numerous opportunities for dynamic code specialization beyond the scope of this paper.

Jaguar runs on the Intel x86 platform under Linux 2.2.5 and Sun JDK 1.1.7.

3.4 Discussion

Apart from the mechanisms employed by Jaguar, by far the most important aspect of this approach is the programming model which it enables. By extending the Java environment with the minimal set of necessary primitives, it is possible to implement complex system software entirely in Java. For example, high-level messaging protocols or disk buffer allocation strategies can be implemented in Java, with only the lowest-level system functions aided by Jaguar code mappings. This helps to ensure the safety and robustness of such system software, and is preferable to wrapping a complex, unwieldy piece of C code up as a set of Java native methods.

Because our prototype specifies code mappings as machine code segments, it is necessary to trust these code mappings as one would trust the compiler or JVM. In some sense, this is more viable than trusting native methods; it is far easier to convince oneself that a short piece of machine code will behave correctly and maintain protection than a complex set of functions coded in C. Using machine code also has the beneficial side-effect that it is difficult for programmers to implement overly complex functionality as a single Jaguar primitive. We are currently investigating the use of a higher-level language in which to represent code mappings, which may allow automatic type-checking and verification.

There is an issue with respect to applying code mappings to virtual method invocations. Normally, the Java runtime resolves virtual method calls at run time, dispatching them to the correct implementation based on the type of the object being invoked. Jaguar currently does not perform any run-time type checks for virtual method code mappings, meaning that an "incorrect" code transformation may be applied to an object if it is cast to one of its superclasses at runtime. While it is feasible to incorporate code transformations into the run-time "jump table" used by the JVM for virtual method resolution, a workaround in the current prototype is to limit transformations to virtual methods which are marked as `private` or `final`, which prohibit overloading. Use of static methods is unproblematic.

Quite similar to Jaguar code mappings is *semantic inlining* [29], a technique which extends the compiler to treat certain operators and method calls as new Java primitives which are inlined. Semantic inlining has been used to implement fast complex arithmetic (by inlining operators on objects of type `Complex`) as well as efficient multidimensional arrays. While the mechanism has much in common with Jaguar, its focus has been on the needs of numerical computing rather than enabling fast communication and I/O. As such, Jaguar raises issues with safely exposing low-level resources to Java applications which semantic inlining alone does not address.

The next two sections evaluate Jaguar through two applications: a fast Java binding to the VIA communications architecture, as well as Pre-Serialized Objects, a mechanism which eliminates Java object serialization overhead for communication and I/O.

4 JaguarVIA

As an example use of Jaguar enabling efficient access to low-level resources, we have implemented *JaguarVIA*, a Java interface to the Berkeley Virtual Interface Architecture (VIA) communications layer [1]. VIA [5] is an emerging standard for user-level network interfaces which enable high-bandwidth and low-latency communication for workstation clusters over both specialized and commodity interconnects. This is accomplished by eliminating data copies on the critical path and circumventing the operating system for direct access to the network interface hardware; VIA defines a standard API for applications to interact with the network layer. Berkeley VIA is implemented over the Myrinet system area network, which provides raw link speeds of 1.2 Gbps; generally, the effective bandwidth to applications is limited by I/O bus bandwidth. The Myrinet network interface used in Berkeley VIA has a programmable on-board controller, the LanAI, and 1 megabyte of SRAM which is used for program storage and packet staging. The implementation described here employs the PCI Myrinet interface board on dual 450 MHz Pentium II systems running Linux 2.2.5.

4.1 The Berkeley VIA architecture

The Berkeley VIA architecture is shown in Figure 3. Each user process may contain a number of Virtual Interfaces (VIs), each of which corresponds to a peer-to-peer communications link. Each VI has a pair of transmit and receive descriptor queues as

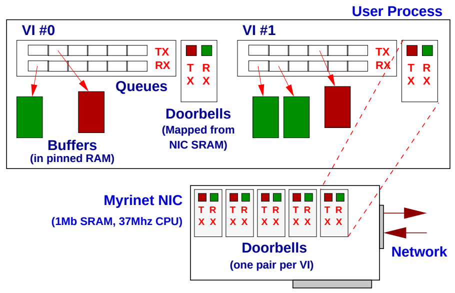


Figure 3: *Berkeley VIA Architecture.*

well as a transmit and receive *doorbell* corresponding to each queue.

To transmit data, the user builds a descriptor on the appropriate transmit queue, indicating the location and size of the message to send, and "rings" the transmit doorbell by writing a pointer to the new transmit queue entry. In order to receive data, the user pushes a descriptor to a free buffer in host memory onto the receive queue and similarly rings the receive doorbell.

The LanAI processor on the NI is responsible for polling the doorbells and taking appropriate action to transmit or receive data on behalf of the (potentially) multiple user processes sharing the network interface.

Transmit and free packet buffers must be first *registered* with the network interface before they are used; this operation, performed by a kernel system call, pins them to physical memory. The network interface performs virtual-to-physical address translation by consulting page maps in host memory, using an on-board translation lookaside buffer to cache address mappings.

The C API provided by VIA includes routines such as the following:

- `VipPostSend()`, post a buffer on the transmit queue;
- `VipPostRecv()`, post a buffer on the receive queue;
- `VipSendWait()`, wait for a packet to be sent;
- `VipRecvWait()`, wait for a packet to be received;

as well as routines to handle VI setup/teardown, memory registration, and so forth.

4.2 JaguarVIA Implementation

Implementing an efficient Java binding to VIA, then, relies upon two major requirements:

1. The ability to efficiently manipulate VIA doorbells and queues; and

2. The ability to directly access registered VIA data buffers, without a copy.

Exposing the JaguarVIA API to Java could be implemented through the Java native code interface, however, copying data between C and Java is expensive, and the high overhead of native method invocation would dominate the cost of issuing VIA API calls; most of these functions do little more than manipulate a couple of pointers, or write small values to the doorbells. Because CPU overhead can be the dominant factor when considering application sensitivity to network interface performance [18], maintaining minimal host overhead for VIA operations is desirable.

JaguarVIA is implemented using two components: first, a Java library duplicating the functionality of the C-based library which provides the VIA API; and second, a set of Jaguar code mappings which translate low-level operations on VIA descriptor queues, doorbells, and data buffers into fast machine code segments. Thus, the majority of JaguarVIA is implemented in Java itself, and only the barest essentials are handled through Jaguar code transformations.

Let us consider the operation of the `VipPostSend` method, contained in the `VIA_VI` class. Here is the Java source code:

```
public int VipPostSend(VIA_Descr descr) {
  /* Queue management omitted ... */
  while (TxDoorbell.isBusy()) /* spin */;
  TxDoorbell.set(descr);  return VIP_SUCCESS;
}
```

Its essential function is to poll the transmit doorbell until it is ready to be written, and then set its value to point to the transmit descriptor specifying the data to be sent.[2] Here, `TxDoorbell` is a private field in the `VIA_VI` class representing the transmit doorbell for this VI, and `Descr` is an object of the type `VIA_Descr` representing the descriptor-queue entry for the packet to be sent.

The layout of the doorbell structure, as mapped from the SRAM of the network interface, is two 32-bit words: the first is a pointer to the transmit descriptor itself, and the second is a *memory handle*, an opaque value which is associated with the registered memory region in which the descriptor is contained. To poll the doorbell it is sufficient to test whether the first word is non-zero. To update the doorbell, both values must be written (first the memory handle, then the descriptor pointer) as virtual addresses in the process address space; however, the Java application has no means by which to generate or use virtual addresses directly. In fact, we wish to prevent the application from specifying an arbitrary address as a transmit or receive descriptor (say), as this would allow the application to access or corrupt any virtual memory address, including memory internal to the JVM.

The methods `VIA_Doorbell.isBusy` and `VIA_Doorbell.set` are implemented through Jaguar code mappings, as shown in Figure 4. Jaguar recognizes the bytecode sequence `invokevirtual VIA_Doorbell.isBusy` (as well as for `VIA_Doorbell.set`) and inlines machine code which performs the doorbell polling and write functions, respectively. In the case of `isBusy`, the machine code segment simply tests the first word of the doorbell for a non-zero value, and pushes a `true` or `false` value onto the Java stack as appropriate. In the case of `set`, the machine code segment writes the two words of the doorbell in the appropriate order. The address of the doorbell itself (as mapped from the LanAI SRAM) is stored in a private field within the doorbell class, and is extracted from the doorbell object by the generated machine code. Similarly, the address and memory handle of the `VIA_Descriptor` object are stored in private fields of that class. The use of private fields ensures that only trusted code is capable of accessing those values — in this case, constructors which create doorbell and descriptor objects, and the Jaguar code mappings which operate on them.

VIA packet buffers are an example of Jaguar External Objects at work. They are implemented as the class `VIA_Databuffer`, which represents a region of registered virtual memory. The data buffer may be manipulated in a manner similar to a Java array, through the methods `readByte`/`writeByte`, `readInt`/`writeInt`, and so forth. These methods are implemented through Jaguar code mappings which directly manipulate the contents of the buffer in virtual memory. The class contains the private fields `vaddr`, `size`, and `memhandle` which keep track of the buffer's address, size, and VIA memory handle, respectively. A `VIA_Databuffer` is created through a special constructor which allocates a memory region outside of the JVM heap and registers (pins) it through the appropriate system call; this memory is not managed directly by the JVM. The class can also be used as a "container" for Jaguar Pre-Serialized Objects, as described in Section 5.

[2] Additional code to maintain a linked list of outstanding transmit descriptors has been omitted for space reasons.

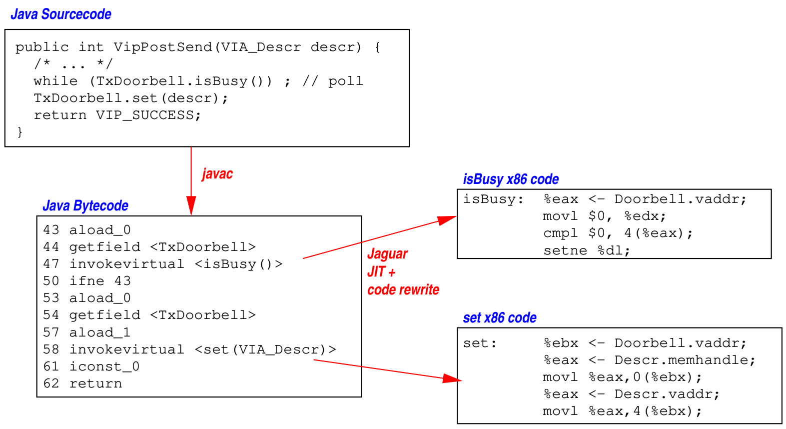


Figure 4: *Jaguar VIA Code Transformations.*

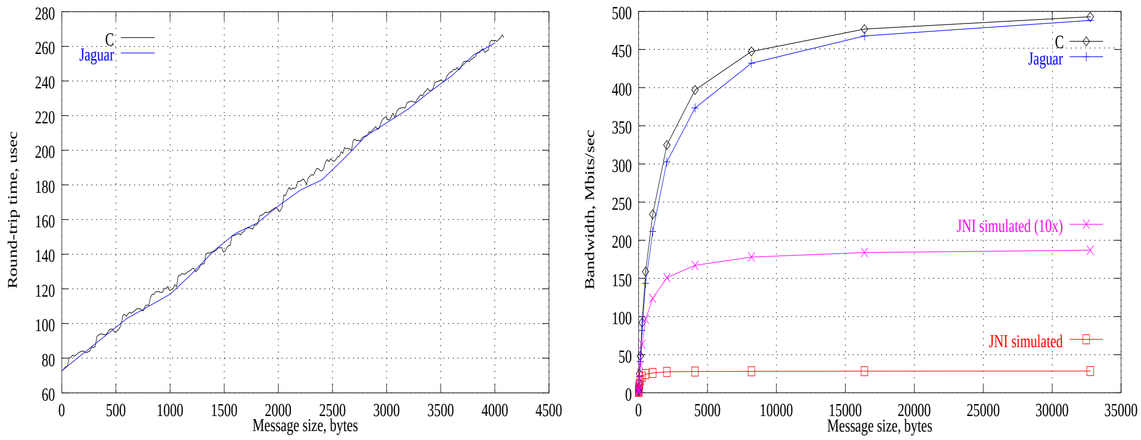


Figure 5: *JaguarVIA microbenchmark results.*

4.3 JaguarVIA Performance

To demonstrate the efficiency of this approach to mapping VIA resources into Java, we implemented two standard VIA microbenchmarks: `pingpong`, which measures round-trip latency for messages of varying sizes, and `bandwidth`, which measures the bandwidth obtained when streaming packets through the network interfaces at the maximum rate.[3]

The results of these microbenchmarks for C and Java are shown in Figure 5. The Java and C `pingpong` benchmarks obtain identical performance with a minimal round-trip time of 70 microseconds for small messages. `bandwidth` in Java obtains 99% of the bandwidth achieved by C, peaking at 488 megabits/sec for 32Kb packets. The lost efficiency is due to higher loop and method-call overheads in Java. More aggressive optimization in the JIT compiler used by Jaguar should be able to overcome these issues.

To highlight the advantage of using Jaguar over the Java native code interface, we have estimated the performance of the `bandwidth` benchmark if the Java Native Interface (JNI) were used to provide VIA functionality in Java. In the estimation, the overhead of using JNI (from Figure 1) 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

[3] Note that Berkeley VIA itself does not implement flow control or reliable delivery; applications are expected to implement their own protocols over the raw transport mechanisms provided. Therefore, the bandwidth benchmark makes no presumption about the flow-control protocol used, and assumes that data is received as rapidly as it is transmitted.

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.

These results clearly show the performance benefit of the Jaguar approach. VIA communication requires several fine-grained manipulations of NI resources (doorbells and descriptor queues) per message, for which the cost of the native code interface would be prohibitive. Furthermore, the use of Jaguar External Objects provides a thin interface to VIA packet buffers, enabling zero-copy communication.

5 Pre-Serialized Objects

JaguarVIA allows arbitrary sequences of bytes to be transferred over the network, using the `VIA_Databuffer` class to represent a registered communication buffer. The methods on this class treat the buffer as a simple array; however, it is desirable to allow more structured Java objects to be communicated over VIA.

The traditional approach to communicating or storing Java objects is to use Java object serialization, which writes out the state of a set of Java objects as a string of bytes. Java objects may be later *recovered* from this string of bytes, meaning that the bytes are retrieved from the disk or network and converted into a set of new Java objects.

Standard implementations of Java serialization are quite costly, although alternatives have been developed [22]. These alternatives, however, rely upon making a copy of the data contained within a Java object and all objects referred to by it. Efficient serialization is the key problem to overcome in implementing high-performance communication and persistence models in Java, such as Remote Method Invocation [17].

A special use of Jaguar code mappings is to implement *Pre-serialized Objects*, or PSOs. Abstractly, a PSO can be thought of a Java object for which the memory representation is already serialized. PSOs eliminate the copy and reference-traversal steps in serialization and de-serialization by requiring that the object be stored in a "pre-packaged" form, ready for storage or communication. Sending the PSO over a communications link, therefore, requires nothing more than directly transmitting the pre-serialized object buffer in memory. On the receiver, the buffer into which data was received need only be pointed to by a new PSO reference.

5.1 PSO Implementation

PSOs are implemented through specialized Jaguar code mappings which recognize `putfield` and `getfield` accesses to the object in question, marshalling object data into and out of its pre-serialized form. Atomic fields (`byte`, `long`, and so forth) are stored using a simple machine-independent representation. The position of each field within the PSO buffer region is determined in a manner similar to that of a C `struct`: each field is stored at a location which maintains alignment constraints on common architectures (for example, that a 32-bit value must be stored on a 32-bit word boundary).

Figure 6 shows code for a simple user-defined PSO type and the memory layout of three such PSOs with references between them.

Object references are handled by requiring that each PSO have an associated *container*, which is a Jaguar External Object acting as the backing store for the object's pre-serialized form. Multiple PSOs may share the same container, and containers can be nested. (The JaguarVIA `VIA_Databuffer` class implements a PSO container, allowing PSOs to be stored within VIA communication buffers.) Each PSO can be thought of as occupying a certain location in its container, with an associated offset and size. The PSO's container and offset are stored as private fields in the PSO itself, and are accessed by the Jaguar code mappings which implement PSOs.

When a reference to another PSO is stored using the `putfield` bytecode, if the two PSOs are within the same container, then the referenced object's offset into that container is stored. Otherwise, a special null value is stored, indicating that the object reference cannot be recovered externally to this JVM. Note, however, that references to PSOs outside of the container and to non-PSO objects are permitted; such references are stored within the field slot of the Java object corresponding to the PSO. However, these references are unrecoverable outside of this JVM (e.g., by the receiver of a PSO sent over a communications channel).

The first time an object reference is read (using the `getfield` bytecode), a new Java PSO object is created which maps onto the container at the given offset. If the stored offset is null or outside of the range of the container, the special Java `null` value is returned. Subsequent `getfield` accesses will yield the PSO reference created during the original access, which is stored in the actual Java object corre-

```
public class MyObject extends PSO {
  public static int getPSOSize() {
    /* Jaguar redirected */
  }
  /* Fields */
  public int someInt;        // Offset 0
  public byte someByte;      // Offset 4
  public MyObject someRef;   // Offset 8
}
```

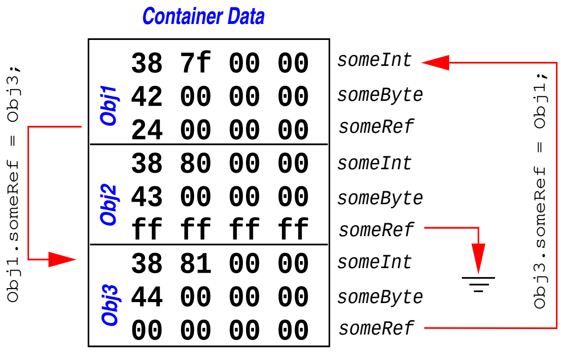


Figure 6: *An example PSO and its memory layout.*

sponding to the PSO. Thus, object references within a PSO are resolved "lazily," that is, only upon their first use. This has the advantage that if a reference within a received PSO container is never traversed, a Java object reference will never be created for it.

5.2 Limitations

Pre-serialized Objects have several limitations. The first is that arbitrary Java objects cannot be represented as PSOs; the implementation depends upon the use of Jaguar code mappings for `putfield/getfield` operations on particular classes (in this case, any subclasses of `Jaguar.PSO`). It would be possible, however, to integrate the use of a standard Java object serializer with PSOs, allowing those portions of the object not pre-serialized by Jaguar to be serialized and deserialized in the standard way (albeit at higher cost).

A second limitation is that only atomic types and references to other PSOs within the same container are recoverable from a PSO's memory representation. This is not as limiting as it might seem. Java arrays are simulated through a generic `PSOArray` class which permits array-like operations on a container using method calls such as `readByte` and `writeInt`.

Further, we believe that the efficiency afforded by PSOs will make it worthwhile for programmers to manage PSO cross-references within the same container. Finding the right balance of programming generality and efficiency in this case is an open research issue.[4]

[4] Supporting cross-container PSO references is feasible, but unsupported by our current prototype. We have decided to retain the simplicity and performance of this design rather than building a more general, and less efficient, implementation.

5.3 PSO Performance

We measure the performance of Pre-Serialized Objects in three ways: a set of microbenchmarks showing basic performance, a benchmark comparing PSOs to object serialization for communication over VIA, and a benchmark demonstrating the use of PSOs for high-performance disk I/O.

Benchmark	Time
Create PSO object	9.24 μsec
Recover PSO reference	8.9 μsec
Follow recovered PSO reference	0.305 μsec
Assign PSO `int` field	0.033 μsec
Assign Java `int` field	0.023 μsec
Write `int PSOArray` element	0.053 μsec
Read `int PSOArray` element	0.049 μsec
Write `int` array element	0.041 μsec
Read `int` array element	0.043 μsec

Figure 7: *Pre-Serialized Object microbenchmarks.*

Figure 7 shows results for a simple microbenchmark of Pre-serialized Object performance. This benchmark creates a linked list of objects, with the same structure as `MyObject` shown in Figure 6, filling a one-megabyte container.

First, the benchmark creates each object in the list and fills in each field in the object. Next, recovery of the list from the pre-serialized buffer is simulated by "forgetting" the original list head and mapping a new PSO onto the beginning of the buffer. Each list entry is traversed by following the `someRef` reference to the next list element; this requires that the next list element be recovered, creating a new PSO instance mapping onto the container at the appropriate offset.Next, the benchmark traverses each list element a second time, which uses the cached PSO references created during the first traversal.

Times are shown for creating a PSO, for recovering a PSO from its pre-serialized form, and for reading a pre-recovered PSO reference. Also shown is the time to write a PSO field of type `int`, which is compared to writing an `int` field to a regular Java object.

Also shown in Figure 7 are timings for reading and writing every element of the one-megabyte container as a `PSOArray`, which treats the container as a simple array of `int`. These values are compared to accessing elements of a Java `int` array. `PSOArrays` are only slightly more expensive than regular Java arrays, due to higher bounds-checking cost in the Jaguar code mappings implementing `PSOArrays`.

Pre-Serialized Objects eliminate the high cost of Java object serialization for communication and I/O. To demonstrate this, we have augmented the original JaguarVIA `pingpong` benchmark (from Figure 5) to transmit a linked list of simple Java objects consisting of nine fields: four `bytes`, four `ints`, and a reference to the next object in the list.

There are two variations on this benchmark. The first uses Pre-Serialized Objects to store object data directly into a VIA communications buffer. The second uses standard Java object serialization to write the linked list into the buffer. This was accomplished by implementing a simple class, `ViaOutputStream`, which writes bytes into a VIA communications buffer. A standard Java `ObjectOutputStream`, which performs object serialization, is created which writes serialized data to the `ViaOutputStream`.

To simplify both benchmarks, serialization is performed only by the transmitter; no de-serialization (or mapping of PSO objects onto the received packet) is performed by the receiver. In the PSO version of the benchmark, the time to assign values to each field of each PSO in the linked list is included in the measurement, which represents the worst-case performance: a real application may be able to eliminate this overhead by re-using a PSO multiple times. In the object serialization benchmark, only the time to create a new `ObjectOutputStream`, and call `writeObject` with the head of the linked list as an argument, are included. This is the minimum amount of work required to serialize a set of objects.[5]

Figure 8 shows the round-trip time as a function of the message size for the use of PSOs over VIA, object serialization over VIA, and the raw JaguarVIA timings(from Figure 5). The right-hand plot does not include the object serialization times, as these dwarf the PSO and raw VIA measurements. It is clear that Pre-Serialized Objects eliminate the high overhead of object serialization: transmitting a linked list of 128 PSOs filling a four-kilobyte buffer has a round-trip latency of 341 μsec, while using Java object serialization costs 26843 μsec, a factor of 78 higher.

For comparison, transmitting an empty 4 Kb buffer using JaguarVIA has a round-trip latency of 262 μsec; filling the buffer using PSOs adds only 39.5 μsec each way, or (39.5 μsec / 128 objects) = 0.30 μsec per object. If the buffer were filled using the `PSOArray` mechanism described above, the cost would be (0.053 μsec per word / 1024 words) = 54.2 μsec. Note that accessing fields of a PSO does not require any bounds-checking to be done: the check is performed when the PSO is created (and mapped onto an underlying container). However, the `PSOArray` must bounds-check each access.

Benchmark	**Time**	**MByte/sec**
`DataInputStream`	4910 ms	0.203
`DataInputStream` (buffered)	488 ms	0.672
Jaguar `PSOArray`	28 ms	35.71
C (unbuffered `read`)	771 ms	1.32
C (`mmap`)	23 ms	43.47

Figure 9: *File-scan benchmarks.*

To evaluate the use of Pre-Serialized Object arrays to implement efficient disk I/O in Java, Figure 9 shows results for a simple benchmark which scans a one-megabyte file of random integers for the maximum value. This not only stresses the I/O component of the system but brings the data into the application to perform simple analysis.

There are several variations on the benchmark. The first two use the standard Java `DataInputStream` class, both with and without an underlying `BufferedInputStream`. The third uses the Jaguar `PSOArray` class to treat a memory-mapped file as an array of bytes or integers. The final two results show the same benchmark in C, using unbuffered `read` system calls as well as `mmap` to access the file.

As the results show, only the Jaguar `PSOArray` and C-based `mmap` benchmarks obtain good performance (23 and 28 milliseconds, respectively). Both of these operate on memory-mapped files, so we should expect performance to be higher than the use of file I/O. The additional cost of the `PSOArray` over direct use of `mmap` from C is due to several

[5] It is necessary to create a new `ObjectOutputStream` for each message; otherwise, the stream will serialize the linked list just once, for the first packet, and for subsequent packets will store a reference to the previously-serialized state. Because each packet is independent, this is unacceptable.

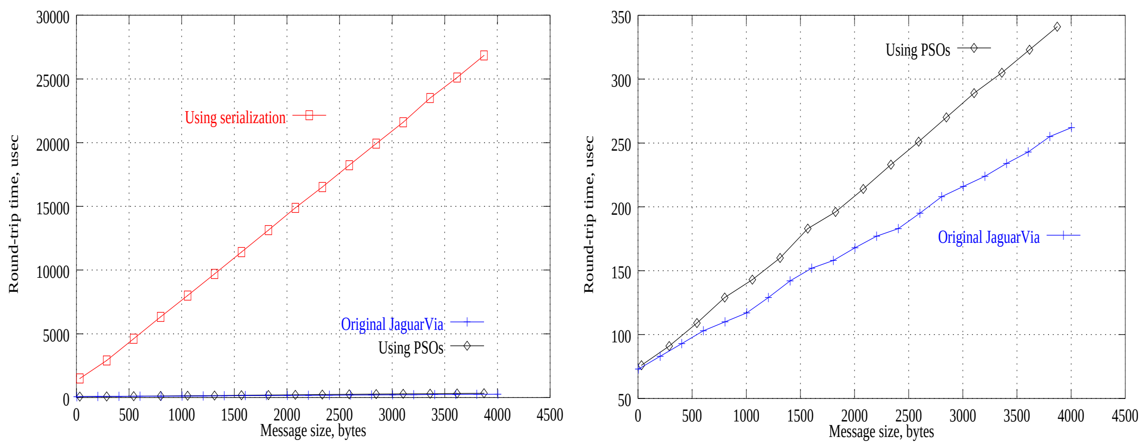


Figure 8: *PSO-over-VIA benchmark results.*

factors: the `PSOArray` methods perform bounds-checking while the C code does not, and the optimizations in our prototype JIT compiler are not as advanced as in the C compiler.

External Objects and PSOs are a powerful means of enabling efficient I/O in Java. They provide direct access to memory-mapped files and a means to reduce the cost of object serialization. We believe these results indicate that higher-level I/O and communication mechanisms (such as persistent data structures and RPC) can be efficiently implemented using Jaguar.

6 Issues and Future Work

Our initial experience with Jaguar has indicated a number of possible avenues for further research. While our prototype has provided encouraging results, we are interested in the extension of the Jaguar approach to other application areas.

One major concern is protection. Currently, the user must trust Jaguar code extensions (built-in to the JIT compiler as a set of bytecode-to-machine code transformation rules) as much the JVM and the compiler itself. As discussed previously, however, this is perhaps better than the use of arbitrary native methods, which have the same trust requirements but far greater complexity in general.

However, it is still desirable to express extensions to the Java environment in a way which enables certain properties to be verified, such as type-safety, bounded execution time, and limited impact on the Java protection model. One approach would be to use a higher-level language to represent Jaguar code mappings; typed assembly language [21] is one candidate, but other languages are possible. The use of such a language should make it possible to statically verify important properties about Jaguar's code mappings — while this may not permit entirely untrusted Java extensions, the goal is rather to raise the degree of trustworthiness such that new code mappings do not have unexpected behavior.

Use of a limited extension language may have the secondary effect that it inherently limits the set of actions which can be implemented as Jaguar code mappings. For example, loops, unbounded branches, and ill-formed Java stack and object operations may be restricted by the semantics of the language. This is desirable as it prevents the abuse of the Jaguar code mapping technique to inline large amounts of low-level code as a single Java primitive; the philosophy of Jaguar is to build in only the minimal set of extensions needed to provide efficient Java access to some server resource.

Pre-Serialized Objects present several untapped opportunities. The first is to exploit PSOs to implement an efficient RPC and data-persistence mechanism for Java; combining the use of JaguarVIA and PSOs should enable a high-performance RPC mechanism for workstation clusters. We are also investigating the use of PSOs to implement distributed data structures for cluster-based Internet services [10] and databases [11].

The prototype implementation of Pre-Serialized Objects has several important limitations. The fact that cross-PSO references are only recoverable if both PSOs are within the same "container" implies an informed programming model which makes this limitation explicit. Currently, it is up to the programmer to arrange for multiple PSOs to coreside in a single container if their object references are to be recovered. While it is possible to remove this limitation, doing so would involve considerable complexity. We believe that programmers who require the performance afforded by PSOs are willing to go to the trouble to carefully maintain PSO relationships; we intend to test this claim by developing applications which use this feature.

Jaguar is a general solution for efficiently binding Java application code to hardware resources. There are myriad potential uses for this mechanism, of which we have yet explored but a few. Other interesting uses include:

- Fast access to devices such as raw disk I/O, framebuffers, and NUMA-style memory-bus network interfaces;
- Transparent data persistence, using a mechanism similar to Pre-Serialized Objects. Certain Java objects could be tagged as "persistent" — Jaguar code mappings could directly implement retention of such objects' state.
- Use of Jaguar code mappings to access shared memory segments in a multiprocessor machine, or to implement distributed shared objects across a network.

Because Jaguar can be applied so generally, it is important to strike the right balance between development of new Java primitives and applications which utilize those primitives. Our claim is that it is undesirable to extend the Java environment arbitrarily; just what the limits are should be brought out by further experimentation.

7 Conclusion

Jaguar bridges the gap between Java applications and the underlying server resources which they wish to exploit. This is accomplished by translating Java bytecodes to inlined machine code sequences at compile time; this ability to abstract system resources as a Java object in this way provides both safety and high performance. The programming model presented by Jaguar allows low-level system software to be coded almost entirely in Java, aided by the minimal set of additional primitives required for direct access to hardware resources.

Jaguar addresses two primary concerns which are essential for enabling high-performance communications and I/O from Java:

1. Efficient, protected access to low-level system resources; and
2. Direct manipulation of memory regions external to the Java heap.

We have described *JaguarVIA*, an efficient Java binding to the VIA communications architecture using Jaguar code mappings to provide fast access to VIA queues, doorbell registers, and specially-pinned data buffers. JaguarVIA obtains identical communication performance to VIA as accessed from C. Pre-Serialized Objects are another application of Jaguar code mappings which reduce the cost of Java object serialization by rewriting object field references to directly access an externalized form of the object's state.

We believe that the approach taken by Jaguar can be extended in a number of ways, both in terms of applications (such as applying Pre-Serialized Objects to implement a fast RPC layer) as well as protection (by expressing Jaguar code transformations in a higher-level language). Jaguar is a general solution which covers a wide range of application demands on the Java environment. As such, it is important to consider the performance and complexity tradeoffs of extending Java with new primitive operations in this way.

Acknowledgements

We are indebted to Kazuyuki Shudo of Waseda University, Japan, for providing us with the *ShuJIT* just-in-time compiler upon which Jaguar is based. design. Philip Buonadonna provided the original implementation of Berkeley VIA used in our measurements. Steve Gribble provided valuable comments on this paper, and Joe Hellerstein contributed feedback during the early stages of Jaguar's design. This work is supported by DARPA grant DABT63-98-C-0038, NSF grant EIA-9802069, and an equipment donation by Intel Corporation. Matt Welsh is supported by a NSF Graduate Student Fellowship.

References

[1] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. In *Proceedings of SC'98*, November 1998.

[2] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, June 1989.

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

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

[5] The VIA Consortium. The virtual interface architecture. `http://www.viarch.org`.

[6] Jeffrey Dean. Whole-program optimization of object-oriented languages. In *PhD thesis, University of Washington, Seattle, Washington*, 1996.

[7] The Java Grande Forum. The java grande forum charter. `http://www.javagrande.org/jgfcharter.html`.

[8] 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.

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

[10] Steven Gribble. Simplifying cluster-based internet service construction with scalable distributed data structures. `http://www.cs.berkeley.edu/~gribble/papers/quals/sdds-cluster.ppt`.

[11] Joe Hellerstein, Eric Brewer, and Mike Franklin. Telegraph: A universal system for information. `http://db.cs.berkeley.edu/telegraph/`.

[12] Sun Microsystems Inc. Enterprise java beans technology. `http://java.sun.com/products/ejb/`.

[13] Sun Microsystems Inc. Java hotspot performance engine. `http://java.sun.com/products/hotspot/index.html`.

[14] Sun Microsystems Inc. Java native interface specification. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html`.

[15] Sun Microsystems Inc. Jini connection technology. `http://www.sun.com/jini/`.

[16] Sun Microsystems Labs. The exact virtual machine (evm). `http://www.sunlabs.com/research/java-topics/`.

[17] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of java's remote method invocation. In *Proceedings of PPoPP'99*, May 1999.

[18] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of ISCA'97*, June 1997.

[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] José Moreira, Sam Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. In *Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing (LCPC'98)*, 1998. `http://www.research.ibm.com/ninja/`.

[21] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, , and Steve Zdancewic. Talx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, May 1999.

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

[23] Inc. Sun Microsystems. The k virtual machine (kvm). `http://java.sun.com/products/kvm/`.

[24] Hiromitsu Takagi, Satoshi Matsuoka, Hidemoto Nakada, Satoshi Sekiguchi, Mitsuhisa Satoh, and Umpei Nagashima. Ninflet: a migratable parallel objects framework using java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998. `http://www.cs.ucsb.edu/conferences/java98/papers/ninflet.pdf`.

[25] D.A. Thurman. jpvm: The java to pvm interface. `http://www.isye.gatech.edu/chmsr/JavaPVM`.

[26] 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.

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

[28] A. Woo, Z. Mao, and H. So. The berkeley jaws project. `http://www.cs.berkeley.edu/~awoo/cs262/jaws.html`.

[29] Peng Wu, Sam Midkiff, José Moreira, and Manish Gupta. Improving java performance through semantic inlining. `http://www.research.ibm.com/ninja/`.