Tigris: A Java-based Cluster I/O System

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

Abstract

We present *Tigris*, a high-performance computation and I/O substrate for clusters of workstations, implemented entirely in Java. Tigris automatically balances resource load across the cluster as a whole, shielding applications from asymmetries in CPU, I/O, and network performance. This is accomplished through the use of a *dataflow programming model* coupled with a work-balancing *distributued queue*. To meet the performance challenges of implementing such a system in Java, we present *Jaguar*, a system which enables direct, protected access to hardware resources (such as fast network interfaces and disk I/O) from Java. Jaguar yields an order-of-magnitude performance boost over the Java Native Interface for Java bindings to system resources. We demonstrate the applicability of Tigris through *Tigris-Sort*, a one-pass, parallel, disk-to-disk sort exhibiting high performance.

1 Introduction

Java is emerging as an attractive platform allowing heterogeneous resources to be harnessed for large-scale computation and I/O. Increasingly, Java is becoming pervasive as a core technology supporting applications as diverse as large parallel and distributed databases, high-performance numerical computing, Internet portals, and electronic commerce. Java's object orientation, type and reference safety, exception handling model, code mobility, and distributed computing primitives all contribute to its popularity as a system upon which novel, component-based applications can be readily deployed. Systems such as Enterprise Java Beans [14], ObjectSpace Voyager [13], and Jini [17] are demonstrative of the momentum behind Java in the server computing landscape.

Nevertheless, there are a number of outstanding issues inherent in the use of Java within server architectures. The first of these is inherent performance limitations in current Java runtime environments. While compilation techniques have greatly enhanced Java performance, they do not address the entire issue: garbage collection, I/O, and exploitation of low-level system resources remain outstanding performance problems. Other Java issues include memory footprint, the binding between Java and operating system threads, and resource accounting.

Still, the benefits of Java seem to outweigh the limits of existing implementations, and various projects have considered the use of Java as a server computing platform. MultiSpace [12] employs a cluster of commodity workstations, each running a Java Virtual Machine, as a scalable, fault-tolerant architecture for novel Internet services, while Javelin [7] harnesses the spare cycles of workstations running Java to perform large-scale computation. The promise of Java's "write once, run anywhere" philosophy has been employed in a number of Internet agent systems, such as Ninflet [25].

Investigating further the applicability of Java to server computing environments, we present *Tigris*, a cluster-based computation and I/O system implemented in Java. The goal of Tigris is to facilitate development of applications which can dynamically utilize workstation cluster resources for high-performance computing and I/O, by automatically balancing resource load across the cluster as a whole. Tigris borrows many of its concepts from River [3], a cluster I/O system implemented on C++ on the Berkeley Network of Workstations [24]. By exploting the use of Java as the native execution and control environment in Tigris, we believe

This research was sponsored by the Advanced Research Projects Agency under grant DABT63-98-C-0038, and an equipment grant from Intel Corporation.

that cluster application development is greatly simplified, and that applications can take advantage of code mobility, strong typing, and other features present in the Java milieu.

In order for such an approach to be feasible, several important issues must be resolved. Most important is the use of high-speed communication and I/O facilities from Java. To address this concern, we present *Jaguar*, a system enabling efficient, direct Java access to underlying hardware resources while maintaining the safety of the JVM sandbox. Jaguar overcomes the high overhead present in the Java Native Interface [16], which is commonly used for performing such "native" machine operations. This is accomplished through the use of a specially modified Just-in-Time (JIT) compiler which transforms Java bytecodes into machine code segments which perform native operations, such as low-overhead communication.

2 The Tigris System

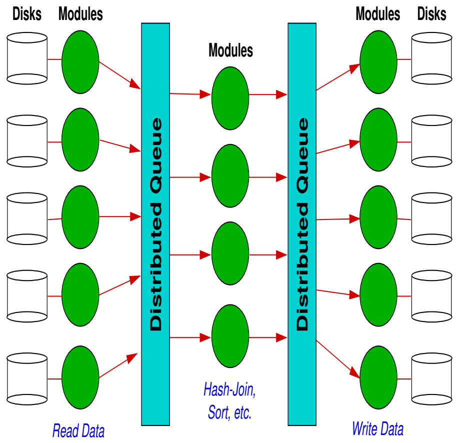


Figure 1: *A sample River application.*

The key ideas in Tigris are borrowed from River [3], a system supporting cluster-based applications which automatically balance CPU, network, and disk I/O load across the cluster as a whole. River employs a dataflow programming model wherein applications are expressed as a series of *modules* each supporting a very simple input/output interface. Modules communicate through the use of *reservoirs*, channels upon which data packets can be pushed into or pulled out of. A simple data-transformation application might consist of three distinct modules: one which reads data from a disk file and streams it out to a reservoir; one which reads packets from a reservoir and performs some transformation on that data; and one which writes data from a reservoir back onto disk. Figure 1 depicts this scenario. By running multiple copies of these modules across many nodes of a cluster, the overall throughput of the data transformation can be scaled.

The goal of River is to automatically overcome cluster resource imbalance and mask this behavior from the application. For example, if one node in the cluster is more heavily loaded than others, without some form of work redistribution the application may run at the rate of the slowest node. The larger and more heterogeneous a cluster is, the more evident this problem will be; often, performance imbalance is difficult to prevent (for example, the location of bad blocks on a disk can seriously affect its bandwidth). This is especially true of clusters which utilize nodes of varying CPU, network, and disk capabilities. Apart from hardware issues, software can cause performance asymmetry within a cluster as well; for example, "hot spots" may arise based on the data and computation distribution of the application.

River addresses resource imbalance in a cluster through two mechanisms: a *distributed queue* (DQ) which balances work across consumers in the system, and *graduated declustering* (GD), mechanism which adjusts load across producers. The DQ allows data to flow at autonomously adaptive rates from producers to consumers, thereby causing data to "flow to where the computation is." GD is a data layout and access mechanism which allows producers to share the production of data being read from multiple disks. By mirroring data sets on several disks, disk I/O imbalance is automatically managed by the GD implementation.

Tigris is an implementation of the River system in Java. This was motivated for several reasons. First, Java is a natural platform upon which to build cluster-based applications, for reasons described in the introduction. Second, River is attractive as a programming paradigm for cluster-based Internet service architectures being investigated by the Ninja project [23] at UC Berkeley. Because Ninja relies heavily upon the use of the Java runtime environment (as in MultiSpace [12]), mapping the concepts in River to an implementation in Java presented an opportunity to address issues with the use of Java, the Ninja service platform, and the River programming model all at once. Finally, we felt that River could benefit greatly from the integration of Java, both in terms of code simplification and added flexibility. For example, the use of Java Remote Method

Invocation (RMI) for control of Tigris components is more expressive and simpler to program than a lower-level control mechanism.

2.1 Implementation overview

Here, we focus on the details of the Tigris system as they differ from the original C++ implementation of River (Euphrates) described in [3].

Tigris is implemented entirely in Java. Each cluster node runs a Java Virtual Machine which is bootstrapped with a receptive execution environment called the *iSpace* [12]. iSpace allows new Java classes to be "pushed into" the JVM remotely through Java Remote Method invocation. A Security Manager is loaded into the iSpace to limit the behavior of untrusted Java classes uploaded into the JVM; for example, an untrusted component should not be allowed to access the filesystem directly. This allows a flexible security infrastructure to be constructed wherein Java classes running on cluster nodes can be given more or fewer capabilities to access system resources based on trust.

```
public interface ModuleIF {
  public String getName();
  public void init(ModuleConfig config);
  public void destroy();
  public void doOperation(Water inWater,
    Reservoir outRes);
}
```

Figure 2: *Tigris Module interface.*

Tigris modules are implemented as Java classes which implement the `ModuleIF` interface, which is shown in Figure 2. This interface provides a small set of methods which each module must implement. `init` and `destroy` are used for module initialization and cleanup, and `getName` allows a module to provide a unique name for itself. The `doOperation` method is the core of the module's functionality: it is called whenever there is new incoming data for the module to process, and is responsible for generating any outgoing data and pushing it down the dataflow path which the module is on.

Communication is managed by two classes: `Reservoir` and `Water`. The `Reservoir` class represents a communications channel between two or more modules; it provides two methods, `Get` and `Put`, which allow data items to be read from and written to the communications channel. The `Water` class represents the unit of data which can be read from or written to a `Reservoir`; this is the same unit of work which is processed by the module `doOperation` method. A `Water` can be thought of as containing one or more data buffers which can be accessed directly (similarly to a Java array) or out of which other Java objects can be allocated from. This allows the contents of a `Water` to represent a structure with typed fields which have meaning to the Java application, rather than as an untyped collection of bytes or integers.

By subclassing `Reservoir` and `Water`, different communication mechanisms can be implemented in Tigris. A particular `Water` implementation can be associated with a particular `Reservoir`; for example, in case the communications channel requires special handling for the data buffers which can be sent over it. Our prototype implementation includes three reservoir implementations:

- `ViaReservoir` provides reliable communications over Berkeley VIA [4], a fast communications layer implemented on the Myrinet system area network. The VIA-to-Java binding used in Tigris is described in Section 3.3.
- `MemoryReservoir` implements communications between modules on the same JVM, passing the data through a FIFO queue in memory.
- `FileReservoir` associates the `Get` and `Put` reservoir operations with data read from and written to a file, respectively. This is a convenient way to abstract file I/O.

`Waters` are initially created by a `Spring`, an interface which contains a single method: `createWater(int size)`. Every `Reservoir` has associated with it a `Spring` implementation which is capable of creating `Waters` which can be sent over that `Reservoir`. This allows a `Reservoir` implementation to manage allocation of `Waters` which will be eventually transmitted over them; for example, a reservoir may wish to initialize data fields in the `Water` to implement a particular communications protocol (e.g., sequence numbers). The implementation of `Water` can ensure that a module is unable to modify these "hidden" fields once the `Water` is created, by limiting the range of data items which can be accessed by the application.

Each `Module` has an associated `ModuleThread` which is responsible for repeatedly issuing `Get` from the module's "upstream" reservoir and invoking `doOperation` with two arguments: the input `Water`, and a handle to the "downstream" reservoir to which any new data should be sent. A single `ModuleThread` may have mulitple upstream and downstream reservoirs associated with it; for example, to implement one-to-many or many-to-one

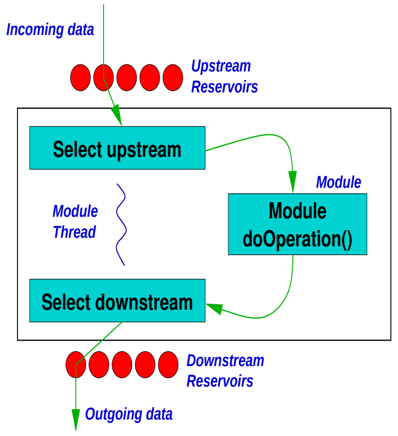


Figure 3: `ModuleThread` *operation.*

communication topologies in the dataflow graph of the application. (This is also the cornerstone of the Distributed Queue implementation in Tigris, as we will see later.) Different implementations of `ModuleThread` can implement different policies for selecting the reservoir which should be used for each invocation of the module's `doOperation` method. For example, `RRModuleThread` implements a round-robin scheme for selection of both the upstream and downstream reservoir on each iteration.[1] Figure 3 depicts the operation of the `ModuleThread` main loop.

2.2 Distributed Queue implementation

In Tigris, the DQ is implemented as a subclass of `ModuleThread` which balances load across multiple downstream reservoirs. In this way, *all* reservoirs in Tigris are maintained by `ModuleThreads`, and modules themselves are unaware of the connectivity of the dataflow graph.

There are three `ModuleThread` implementations included in our prototype:

`RRModuleThread` selects the upstream and downstream reservoir for each iteration in a round-robin fashion.

`RandomModuleThread` selects the upstream reservoir for each iteration using round-robin, and the downstream reservoir using a randomized scheme. The algorithm maintains a credit count for each downstream reservoir. The credit count is decremented for each `Water` sent on a reservoir, and is incremented when the `Water` has been processed by the destination (e.g., through an acknowledgement). On each iteration, a random reservoir R is chosen from the list of downstream reservoirs. If that reservoir has a zero credit count, another reservoir is chosen. This is the DQ implementation used in the original River implementation [3].

`LotteryModuleThread` selects the upstream reservoir for each iteration using round-robin, and the downstream reservoir using a "lottery" scheme. The algorithm maintains a credit count for each downstream reservoir. On each iteration, a random number r is chosen in the range $(0..N)$ where N is the total number of downstream reservoirs. The choice of r is weighted by the value $w = (c_R/C)$ where c_R is the number of credits belonging to reservoir R and $C = \sum c_R$. The intuition is that reservoirs with more credits are more likely to be chosen, allowing bandwidth to be naturally balanced across multiple reservoirs.

2.3 Initialization and control

A Tigris application is controlled by an external agent which contacts the iSpace of each cluster node through Java RMI, and communicates with the `RiverMgr` service running on that node. `RiverMgr` provides methods to create a `ModuleThread`, to create a reservoir, to add a reservoir as an upstream or downstream reservoir of a given `ModuleThread`, and to start and stop a given `ModuleThread`. In this way the Tigris application and module connectivity graph is "grown" at runtime on top of the receptive iSpace environment rather than hardcoded *a priori*. Each cluster node need only be running iSpace with the `RiverMgr` service preloaded.

Execution begins when the control agent issues the `moduleStart` command to each module, and ends when one of two conditions occur:

- The control agent issues `moduleStop` to every module; or,
- Every module reaches the "End of River" condition.

"End of River" (EOR) is indicated by a module receiving a null `Water` as input. This can be triggered by a producer pushing a null `Water` down a reservoir towards a consumer, or by some other event (such as the `ModuleThread` itself declaring

[1] By passing a handle to the current downstram reservoir to `doOperation`, the module is capable of emitting zero or more `Waters` on each iteration. Also, this permits the module to obtain a handle to the reservoir's `Spring` to create new `Waters` to be transmitted. Note that the module may decide to re-transmit the same `Water` which it took as input; because a reservoir may not be capable of directly transmitting an arbitrary `Water` (for example, a `ViaReservoir` cannot transmit a `FileWater`), the reservoir is responsible for transforming the `Water` if necessary, e.g., by making a copy.

an EOR condition). A module may indicate to its surrounding `ModuleThread` that EOR has been reached by throwing an `EndOfRiverException` from its `doOperation` method; this obviates the need for an additional status value to be passed between a module and its controlling thread.

2.4 Distributed Queue Performance

Figure 4 shows performance of the Tigris Distributed Queue implementations under scaling and perturbation. The first benchmark demonstrates the three DQ implementations (round-robin, randomized, and lottery) as the number of nodes passing data through the DQ is scaled up. The `ViaReservoir` reservoir type is used, which implements a simple credit-based flow-control scheme over the VIA fast communications layer. End-to-end peak bandwidth through a `ViaReservoir` is 46 MByte/sec.

In each case an equal number of nodes are sending and receiving data through the DQ. The results show a 12% bandwidth loss (from the optimal case) in the 8-node case. This is partially due to the DQ implementation itself; in each case, the receiving node selects the upstream Reservoir from which to receive data in a round-robin manner. Although the receive operation is non-blocking it does require the receiver to test for incoming data on each upstream Reservoir until a packet arrives. We also believe that a portion of this bandwidth loss is due the VIA implementation being used; as the number of active VIs increases, the network interface must poll additional queues to test for incoming or outgoing packets.

The second benchmark tests the performance of the lottery DQ implementation as receiving nodes are artificially loaded by adding a fixed delay to each iteration of the receiving module's `doOperation()` method. The total bandwidth in the unperturbed case is 1181.58 MByte/second (4 nodes sending 8Kb packets at the maxiumum rate to 4 receivers through the DQ), or 295.39 MByte/sec per node. Perturbation of a node limits its receive bandwidth to 34.27 MByte/sec. The lottery DQ balances bandwidth automatically to nodes which are receiving at a higher rate, so that when 3 out of 4 nodes are perturbed, 56% of the total bandwidth can be achieved. Over 90% of the total bandwidth is obtained with half of the nodes perturbed.

3 Jaguar: Implementing Tigris Efficiently

While the design of Tigris is greatly simplified through the use of Java, this raises a number of issues, the most important of which is performance. The original River system was implemented on the Berkeley Network of Workstations in C++, using Active Messages II [8] as a fast communication substrate and Solaris `mmap` and `directio` features to perform disk I/O. Several important performance limitations in the Java runtime environment must be overcome in order for Tigris to rival its C++ predecessor. Our approach is to enable direct but protected Java access to hardware resources through *Jaguar*.

3.1 The Java Native Interface

While programs expressed as Java bytecodes are very expressive (incorporating notions of object-orientation, strong typing, exception handling, and thread synchronization), the machine-independent nature of this representation restricts the set of actions that can be efficiently performed through direct bytecode transformation. Java compiler technology is advancing rapidly to address this concern for general-purpose computation, performing optimizations such as loop unrolling and efficient usage of machine registers, cache, and memory. However, certain operations require a tigher binding between the Java bytecode and its machine representation; generic compilation techniques cannot apply here.

Issues arise when one wishes to make hardware resources — such as fast network interfaces, disk I/O, and specialized machine instructions — directly available to Java applications while maintaining the protection guarantees of the Java environment. Traditionally, Java runtime environments have enabled the use of such resources through a *native code interface*, which allows *native methods* to be implemented in a lower-level programming language, such as C, which is capable of directly accessing these resources (say, by manipulating virtual memory addresses, issuing system call traps, and the like). The native code interface ensures that protection is maintained within Java, assuming that the native code itself can be trusted.

However, the native code interface employed by most Java runtimes incurs a high overhead for each native method call, and sharing of data between the Java runtime and the native code environment is often costly. For example, in the Java Native Interface (JNI) provided in JVMs from Sun Microsystems,

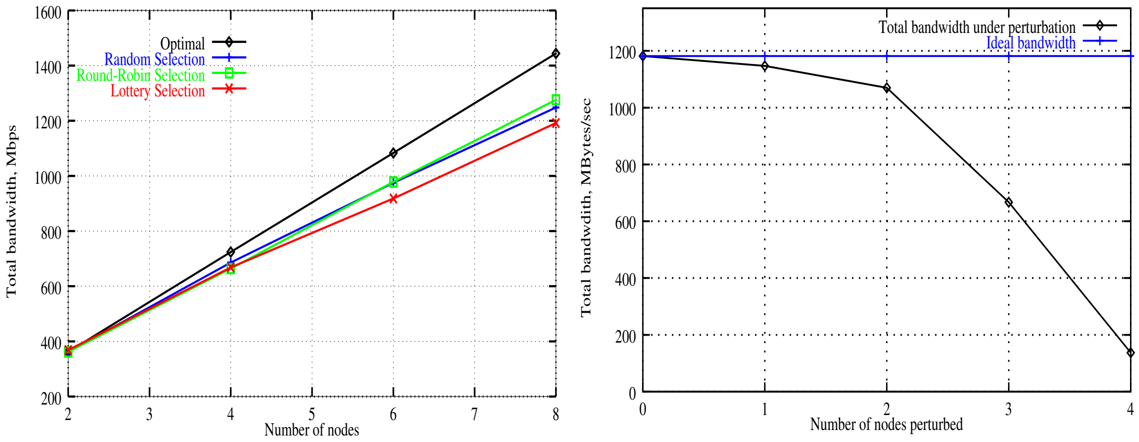


Figure 4: *Distribued Queue performance.*

Benchmark	**Java Native Interface**	**Comparable C code**
void arg, void return native method call	.909 μsec	0.038 μsec
void arg, `int` return native method call	.932 μsec	0.042 μsec
`int` arg, `int` return native method call	.985 μsec	0.049 μsec
4-`int` arg, `int` return native method call	1.31 μsec	0.072 μsec
10-byte C-to-Java array copy	3.0 μsec	0.354 μsec (`memcpy` only)
1024-byte C-to-Java array copy	18.0 μsec	1.68 μsec (`memcpy` only)
102400-byte C-to-Java array copy	1706.0 μsec	432.5 μsec (`memcpy` only)
10-byte Java-to-C array copy	7.0 μsec	*n/a*
1024-byte Java-to-C array copy	272.0 μsec	*n/a*
102400-byte Java-to-C array copy	27274.0 μsec	*n/a*

Figure 5: *A comparison between Java Native Interface and C overheads.*

there is no way to export a region of "native" virtual memory as, say, a Java array. Rather, a new array must be allocated and the data copied into the JVM's object heap. Figure 5 gives results for a simple Java Native Interface benchmark on a 450 MHz Pentium II running Linux 2.2.5 and JDK 1.1.7. For comparison, similar tests conducted in C are shown; all optimizations were disabled when compiling the C benchmark.

In addition, the native code interface applies only to method invocations; other operations (such as object field references, use of the `new` operator, and so forth) cannot be delegated to native code. Transforming these operations into native code could be very useful; for example, a Java object could be thought of as mapping onto a particular region in virtual memory (such as a memory-mapped file or I/O device), and field read and write operations could affect that memory region directly. More generally, one might desire that a Java object has a particular representation in memory, for reasons of efficiency, or sharing data between Java and hardware devices and native libraries. Maintaining an externalized representation of a Java object can also be used for data persistence or communication. Such an approach has deep ramifications for the binding between Java bytecode and underlying hardware resources, as well as the maintenance of type and reference safety within the Java "sandbox."

We believe the limitations discussed above are not fundamental to Java; rather, they arise as the result of a desire to maintain platform-independence for the Java runtime environment itself, allowing both the JVM and hopefully any native code which it uses to be easily ported to other systems. A straightforward JVM implementation transforms Java bytecode using only generic machine instructions, and relegates all other actions to native code. The Java Native Interface is relatively portable as well, and maintains a strong separation between the native code and data internal to the JVM; the result is a higher overhead when moving data or execution between the Java/native code boundary.

On the opposite end of the spectrum, *static* Java compilers are emerging which transform Java bytecodes directly into native machine code. These compilers do a good job of machine code optimization, eschewing portability for performance. To our knowledge, no static Java compiler addresses the high overhead for crossing the native code interface, nor do they perform special transformation of Java bytecode into native code (for example, to implement "native fields" as described above).

In order to address these issues, we introduce a new system, *Jaguar*,[2] which bridges the gap between Java bytecode and efficient acccess to underlying hardware resources. This is accomplished through *Just-in-Time code transformation* which translates Java bytecode into machine code segments which directly manipulate system resources while maintaining type and reference safety. Jaguar is implemented in the context of a standard Java Just-in-Time compiler, rather than through reengineering of the JVM, allowing seamless interoperation with a complete Java runtime environment.

3.2 Jaguar concepts

The fundamental concept embodied in Jaguar is that of *code mappings* between Java bytecode and native machine code. 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 for "`invokevirtual SomeClass.someMethod()`" into a specialized machine code fragment which directly manipulates a hardware resource in some way.

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 `add` and `mult` bytecodes to, say, a fast matrix-multiply instruction would certainly be more difficult than recognizing a method call to a particular object.

[2]Jaguar is an acronym for *Java Access to Generic Underlying Architectural Resources.*

Using Jaguar code mappings, operations which would normally be handled through native method calls can be inlined directly into the compiled bytecode. The performance improvement can be very impressive: for example, invoking an `int`-argument, `int`-return value method as machine code inlined by Jaguar costs 0.066 μsec on a 450MHz Pentium II, while the same operation through JNI costs 0.985 μsec.

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 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. 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 `final`, which prohibits overloading.

3.3 An example: 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 [4]. VIA [9] 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.

The Berkeley VIA architecture is shown in Figure 6. Each user process may contain a number of Virtual Interfaces (VIs), each of which corresponds

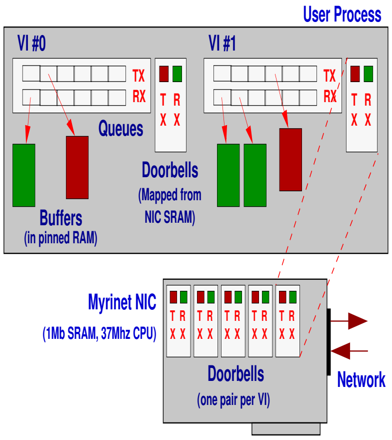


Figure 6: *Berkeley VIA Architecture.*

to a peer-to-peer communications link. Each VI has a pair of transmit and receive descriptor queues as well as a transmit and receive *doorbell* corresponding to each queue. The doorbells are mapped from the SRAM of the network interface, and are polled by the LanAI processor. 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. 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/tear-down, memory registration, and so forth.

Exposing this API to Java could be implemented using the Java Native Interface, however, for the reasons given above this would incurr unnecessary costs. 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 [19], maintaining minimal host overhead for VIA operations is desirable.

Rather, JaguarVIA is implemented using two major components: first, a Java library duplicating the functionality of the C-based `libvia`; and second, a set of Jaguar code mappings which translate low-level operations on VIA descriptor queues and doorbells into fast machine code segments. Thus, the majority of JaguarVIA is in fact 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.[3]

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*, a 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 nonzero. To update the doorbell, both values must be written (first the memory handle, then the 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), in case that memory is internal to the JVM itself.

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

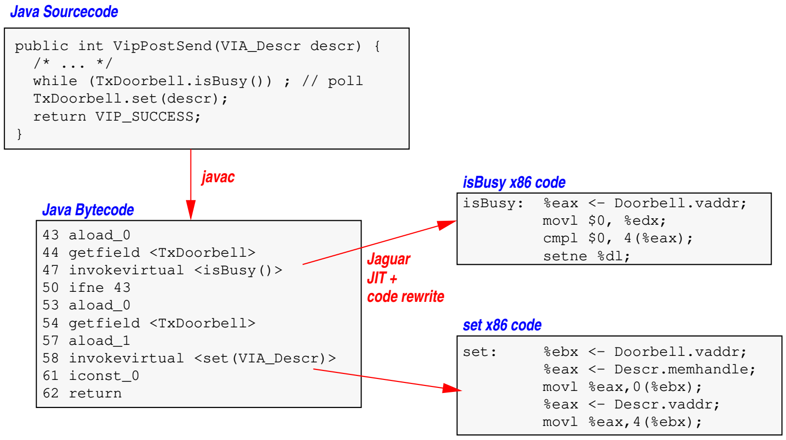


Figure 7: *Jaguar VIA Code Transformations.*

The methods `VIA_Doorbell.isBusy` and `VIA_Doorbell.set` are implemented through Jaguar code mappings, as shown in Figure 7. Jaguar recognizes the bytecode sequence `invokevirtual VIA_Doorbell.isBusy` (as well as for `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 the private field `int vaddr` 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.

VIA packet buffers 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 object's address, size, and VIA memory handle, respectively. A `VIA_Databuffer` is allocated through a special constructor which allocates and registers a memory buffer; this memory is not managed by the JVM. The class can also be used as a "container" for other Java objects, as described in Section 4.

3.4 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.[4]

The results of these microbenchmarks for C and Java are shown in Figure 8. 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 61 MByte/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.

The `ViaReservoir` reservoir type used in Tigris is based on JaguarVIA and implements a simple credit-based flow control scheme (coded entirely in Java). It obtains a peak end-to-end bandwidth of 46 MByte/sec, 75% of raw JaguarVIA performance.

[4]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.

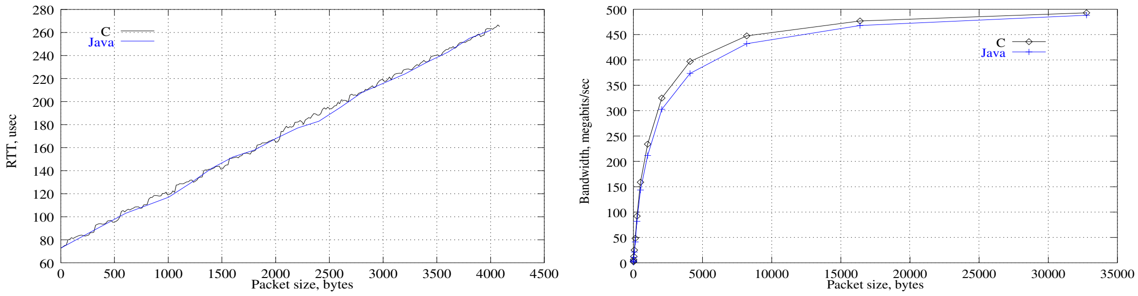


Figure 8: *JaguarVIA microbenchmark results.*

4 Pre-Serialized Objects

JaguarVIA allows a registered virtual memory region to be accessed using the `VIA_Databuffer` class, through methods which treat this region as an array of values. However, if more structured objects are to be transmitted and received through JaguarVIA, the traditional approach would be to use Java object serialization. Standard implementations of Java serialization are quote costly, although alternatives have been developed [20]. These alternatives, however, rely upon making a copy of the 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, such as Java Remote Method Invocation [18].

A special use of Jaguar code mappings is to implement *Pre-serialized Objects*, or PSO's. Abstractly, a PSO can be thought of a Java object for which the memory representation is already serialized. PSO's 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 PSO reference.

PSO's are implemented through 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 an 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 9 shows code for a simple user-defined PSO type and the memory layout of three such PSO's with references between them.

Object references are handled by requiring that each PSO have an associated *container*, a memory buffer which acts 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.) Each PSO can be thought of as occupying a certain location in the container, with an associated offset and size. 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; these references are simply 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 (through `getfield`), a new PSO reference 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, `null` is returned. Subsequent `getfield` accesses will yield the PSO reference created during the original access. 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 is never traversed, a Java object reference will never be created for it.

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. It would be possible, however, to integrate

```
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
}
```

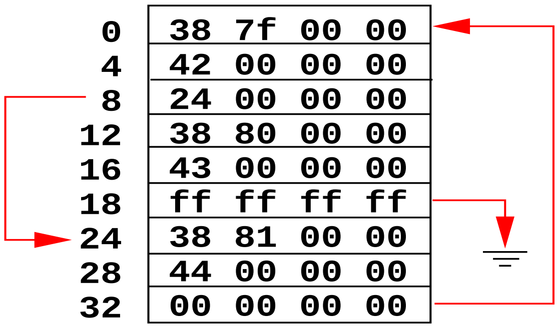


Figure 9: *An example PSO and its memory layout.*

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 as needed (albeit at higher cost).

A second limitation is that only atomic types and references to other PSO's within the same container are recoverable from a PSO's memory representation. This is not as limiting as it might seem. Java arrays can be simulated through a generic `PSOArray` class which permits array-like operations (such as `readByte` or `writeLong`). 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 balanace of programming generality and efficiency in this case is an open research issue.[5]

4.1 PSO Microbenchmarks

Benchmark	**Time**
Create list element	12.2 μsec
Read back list element, first pass	11.6 μsec
Read back list element, second pass	0.488 μsec
Write `PSOArray` element	0.067 μsec
Read `PSOArray` element	0.086 μsec

Figure 10: *Pre-serialized Object microbenchmarks.*

Figure 10 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 9, within a one-megabyte container. Times are shown (per list element) for creating the list elements, traversing each element in the list a first time (which requires that a new Java object reference be created for each element) and a second time (which simply returns the cached Java object reference). Also shown are timings for reading and writing every element of the one-megabyte container as a `PSOArray`.

5 TigrisSort: A Sample Application

In order to evaluate the performance and flexibility of the low-level mechanisms embodied in Jaguar, as well as the design of the Tigris system as a whole, we have implemented *TigrisSort*, a parallel, disk-to-disk sorting benchmark. As with Datamation [1] and NOWSort [2], external sorting is a good way to measure the memory, I/O, and communication performance of the complete system. While the existence of previous sorting results on other systems yields a yardstick by which the Tigris system can be compared, we were also interested in understanding the functional properties of the Tigris and Jaguar mechanisms in the context of a "real application."

5.1 TigrisSort structure

The structure of TigrisSort is shown in Figure 11. Tigris sort implements a one-pass, disk-to-disk parallel sort of 100-byte records each of which contain a 10-byte key. Data is initially striped across the input disks with 5 megabytes of random data per node. The application consists of two sets of nodes: *partitioners* and *sorters*. Partitioning nodes are responsible for reading data from their local disk and partitioning it based on the record key; the partition "buckets" from each node are transmitted to the sorting nodes which sort the entire data set and write it to the local disk. This results in the sorted dataset being striped by increasing key value across the sorting node disks.

[5] 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.

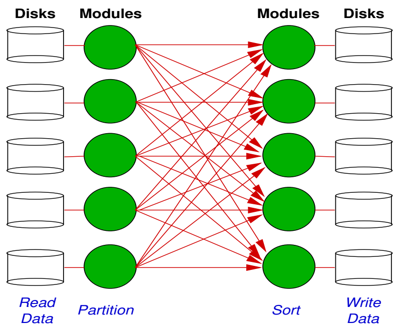


Figure 11: *TigrisSort structure.*

Partitioning is implemented as a `PartitionModuleThread` class which continually reads a buffer of data from a local disk file, partitions the buffer into a number of buckets based on the key, and writes full buckets to the reservoir corresponding to that bucket. All buckets are flushed to the communications channel when file data has been exhausted. The actual partitioning of records is accomplished through a native method, `bucketize()`, which takes as arguments a PSO container corresponding to the input file buffer and an array of PSO containers corresponding to each bucket. No data copying between Java and C is necessary as both Java and native code can directly manipulate the contents of the PSO containers.

Sorting is implemented as a `SortModule` contained within a `LotteryModuleThread`. The module's `doOperation` method saves a copy of each packet received until End-of-River is signalled; at that time is coallates every packet into a single buffer, sorts that buffer, and writes the sorted data out to disk. Sorting is accomplished by a native method, `doSort()`, which takes an array of PSO containers corresponding to the packets received during the lifetime of the module. Again, no data copy between Java and C is required due to the use of PSO containers.

File I/O is implemented as a class, `FilePSOBuffer`, which causes a file to be memory mapped (through the `mmap` system call) into the address space of the JVM and exposed to the Java application as a PSO container. PSO operations on that container correspond to disk reads and writes through the memory-mapped file. A special method is provided, `flush()`, which causes the contents of the memory-mapped file to be flushed to disk.

This approach has several limitations. One is that the operating system being used (Linux 2.2.5) does not allow the buffer cache to be circumvented using memory-mapped files. Another is that a particular write ordering cannot be enforced. Currently, Linux does not provide a "raw disk" mechanism which provides these features. Rather than concerning ourselves with these details, we assume that performance differences arising because of them will be negligible. This seems to be reasonable: first, disk I/O is just one component of the TigrisSort application, which does not appear to be disk-bandwidth limited. Secondly, double-buffering of sort data in memory is not problematic with the small (5 megabyte) per-node data partitions being dealt with. Third, write ordering is not important for implementing parallel sort; it is sufficient to ensure that all disk writes have completed.

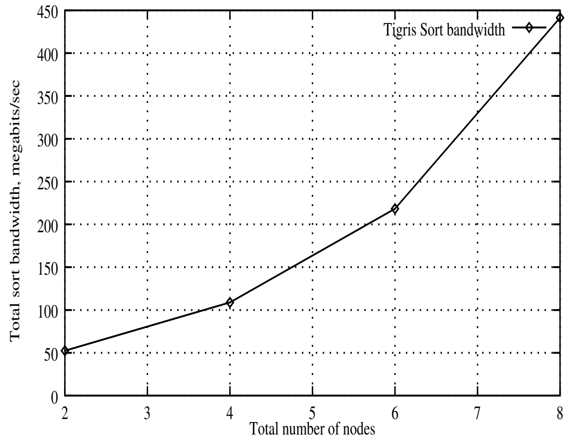
5.2 TigrisSort performance

Figure 12 shows the performance of TigrisSort as the benchmark is scaled up from 2 to 8 nodes. In each case, half of the nodes are configured as partitioners, and half as sorters; 5 megabytes of data are partitioned or sorted per node. The total time to complete the sort averaged 738 milliseconds; this implies that as more nodes are added to the application, more data can be sorted in constant time. Given this result we feel that with cafeful tuning, TigrisSort can compete with the current world-record holder of the Datamation sort record, Millennium Sort [5], which was implemented on the same hardware platform using Windows NT and C++. However, the dominant cost of Datamation sort on modern systems is application startup; the results above do not include the cost of starting the Tigris system itself. There is some question as to what should be included in the startup measurements (e.g., for traditional implementations, the cost of cold-booting the operating system is not measured).

6 Related Work

Related work in the area of high-performance Java computation and I/O environments addresses three major issues: efficient communication mechanisms, language and runtime extensions for parallelism, and compilation techniques. Several projects fall into more than one of these categories.

Efficient communication in Java has been investigated in terms of direct access to VIA [6], bindings to message-passing libraries such as MPI [11], and fast implementations of Java RMI [20, 18]. Several of these projects [11] build JNI bindings to na-



# nodes	Amount sorted	Avg time/node	Avg sort bandwidth/node	Total sort bandwidth
2	5 MBytes	762 msec	52.49 Mbps	52.49 Mbps
4	10 MBytes	734.5 msec	51.725 Mbps	108.91 Mbps
6	20 MBytes	733 msec	51.82 Mbps	218.28 Mbps
8	40 MBytes	725 msec	52.40 Mbps	441.37 Mbps

Figure 12: *TigrisSort performance results.*

tive libraries to perform communication. [20] describes a more efficient Java RMI implementation accomplished through careful coding and a fast serializer, coded entirely in Java. [18] takes the more extreme approach of compiling the Java application down to a language such as C, moving some 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, rather, a static executable representing what was once Java sourcecode. The approach in [6] is closest to that in Jaguar, which makes modifications to a *static* Java compiler to build efficient bindings to hardware resources (in this case, a commercial VIA implementation). Jaguar takes a more generalized approach to building mappings between Java bytecode and native machine code, and does so in the context of a JIT compiler running alongside a standard JVM implementation.

Various projects have considered language extensions for parallel and distributed computing. JavaParty [22] is an extension to Java providing transparent remote object access and object mobility. Titanium [27] is a dialect of Java for large-scale scientific computing; it is focused on static compilation and automated techniques for optimization of parallel programs coded expressed in the Titanium language. Other models for Java-based parallel computing, such as work stealing in JAWS [26] and agents in Ninflet [25], have also been considered. Tigris is the first dataflow and I/O-centric programming model to our knowledge to have been explored in the Java environment.

Java compilation for high-performance applications has focused primarily on static compilation. Toba [21] was one of the first Java bytecode-to-C translators; several commercial products, such as IBM VisualAge for Java [10], now incorporate static Java compilers. Sun's HotSpot [15] is a *dynamic* JIT compiler which iteratively optimizes compiled Java bytecodes based on execution patterns. All Java compilers to our knowledge focus on straightforward optimizations for CPU and memory usage; many of these provide more efficient garbage collection and threads in their runtime system. Jaguar takes the novel approach of applying Java bytecode transformation to enable protected access to hardware resources such as communications and I/O. We believe that the fundamental concepts in Jaguar can be incorporated into other compilation techniques.

7 Conclusions

The challenges presented by employing Java for high-performance computation, I/O, and communication are myriad. On one hand, there is a clear mismatch between existing Java Virtual Machine implementations and the capabilities of modern computer systems; on the other, the potential benefits of Java in terms of software engineering, flexibility, and interoperability provide a strong incentive to make ends meet. Java presents an opportunity to explore new design techniques and to re-evaluate the goals of high-performance system architecture.

We have presented Tigris, a cluster I/O substrate coded entirely in Java which enables utilization of resources cluster-wide to achieve high performance in light of CPU, I/O, and network imbalance. To overcome the performance gap between the Java runtime and Tigris' communication and I/O needs, we have developed Jaguar, a system which provides a tight binding of Java bytecode to native machine code used to access low-level system resources. JaguarVIA and Pre-serialized Objects are two uses of Jaguar which provide important performance benefits for Tigris (namely, fast communication and fast object serialization). Finally, we have demonstrated TigrisSort, a resource-intensive application implemented using the Tigris dataflow programming model. The initial performance analysis of TigrisSort indicates that a Java-based environment is in fact feasible for applications requiring large-scale parallel I/O.

References

[1] et. al. Anon. A measure of transaction processing power. In *Datamation, 31(7): 112-118*, February 1985.

[2] Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the sorting record: Experiences in tuning now-sort. In *Proceedings of the 1998 Symposium on Parallel and Distributed Tools (SPDT '98)*, 1998.

[3] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster i/o with river: Making the fast case common. In *IOPADS '99*, 1999. `http://www.cs.berkeley.edu/~remzi/Postscript/river.ps`.

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

[5] Philip Buonadonna, Joshua Coates, Spencer Low, and David E. Culler. Millennium sort: A cluster-based application for windows nt using dcom, river primitives and the virtual interface architecture. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

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

[7] B. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based parallel computing using java. In *1997 ACM Workshop on Java for Science and Engineering Computation*, 1997. `http://www.cs.ucsb.edu/~schauser/papers/97-javelin.pdf`.

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

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

[10] IBM Corporation. Ibm visualage for java. `http://www.software.ibm.com/ad/vajava/`.

[11] 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.

[12] S. Gribble, M. Welsh, D. Culler, and E. Brewer. Multispace: An evolutionary platform for infrastructural services. In *Proceedings of the 16th USENIX Annual Technical Conference*, Monterey, California, 1999.

[13] ObjectSpace Inc. Objectspace voyager. `http://www.objectspace.com/Products/voyager1.htm`.

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

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

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

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

[18] 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.

[19] 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.

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

[21] University of Arizona Sumatra Project. Toba: A java-to-c compiler. `http://www.cs.arizona.edu/sumatra/toba/`.

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

[23] UC Berkeley Ninja Project. The uc berkeley ninja project. http://ninja.cs.berkeley.edu.

[24] UC Berkeley NOW Project. The uc berkeley network of workstations project. http://now.cs.berkeley.edu.

[25] 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.

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

[27] 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.