Interfacing Java to the Virtual Interface Architecture

Chi-Chao Chang and Thorsten von Eicken

Department of Computer Science
Cornell University
Ithaca, NY 14853

{chichao,tve}@cs.cornell.edu

ABSTRACT

User-level network interfaces (UNIs) have reduced the overheads of communication by exposing the buffers used by the network interface DMA engine to the applications. This removes the kernel from the critical path of message transmission and reception, and reduces the number of data copies performed on that path. Unfortunately, the fact that UNIs require the application to manage buffers explicitly makes it difficult to provide direct access to a UNI from Java, as the language explicitly prevents programs from controlling the location or layout of objects.

This paper describes Javia, a Java interface to the Virtual Interface Architecture (VIA), an emerging UNI standard in the industry. Javia explores two points in the design space. The first approach manages buffers in C code and requires data copies between the Java heap and native buffers. The second approach introduces a special Java-level buffer abstraction that allows programs to allocate regions of memory outside the Java heap and to use them directly and safely as Java arrays. These buffers eliminate the bottlenecks of the first approach but require modifications to the garbage collector. Simple experiments show that Java programs can achieve bandwidths of 80Mbytes/sec for 8Kbyte messages, which is within 1% of those achieved by C programs.

Keywords

Java, user-level network interfaces, high-performance communication, memory management, garbage collection.

1. INTRODUCTION

User-level network interfaces (UNIs) introduced over the last few years have reduced the overheads of communication within clusters by removing the operating system from the critical path [PLC95, vEBB+95, DBC+98]. Intel, Compaq, and Microsoft have taken input from the numerous academic projects to produce an “industry standard” UNI called the Virtual Interface Architecture (VIA) [VIA97]. At this point, commercial VIA hardware is available for Windows NT/2000 and studies demonstrate the architecture’s potential for high performance [BGC98] as well as for supporting higher level communication abstractions and clustered applications [SPS98, SSP99].

This paper examines how the performance of VIA can be made available to Java cluster applications. To date, communication performance has not been a major focus of Java developers. To begin with, Java programs run more slowly than comparable C or C++ programs, suggesting that the performance bottleneck of networked Java applications may not yet be communication, but computation. Furthermore, Java has been mainly used for applications over wide-area networks (i.e. the Internet), where the kind of performance delivered by user-level networking hardware is not particularly interesting.

We believe that recent advances in Java compilation technology and the growing interest in using Java for cluster applications are making the performance of Java communication an interesting topic. Research in JITs, static Java compilers, locking strategies, and garbage collectors [ACL+98, ADM+98, BKM+98, FKR+98, MGG98] have delivered promising results, gradually reducing the performance gap between Java and C programs. Thus, providing access to VIA from Java may soon become an important building block for Java cluster applications.

The important advances made by UNIs are (i) to enable the DMA engine to move data directly between the network and buffers placed in the application address space, and to (ii) allow the application to manage these buffers explicitly. The DMA access to application buffers eliminates the traditional path through the kernel, which typically involves one or more copies. By managing buffers explicitly, the application can often avoid copies and can use higher-level information to optimize their allocation. Unfortunately, requiring applications to manage the buffers in this manner is ill matched to the foundations of Java. Java prevents the programmer from exerting any control over the layout, location and lifetime of Java objects, which is exactly what is required to take advantage of UNIs.

In this paper, we propose a two-level Java interface to VIA, called Javia. The first level of Javia (Javia-I) manages the buffers used by VIA in native code (i.e. hides them from Java) and adds a copy on the transmission and reception paths to move the data into and out of Java arrays. The copy in the transmission path can be optimized away by pinning the array on the fly. Javia-I can be implemented for any Java VM or system that supports a JNI-like native interface. Benchmarks show that Javia-I achieves a peak bandwidth of 70Mbytes/s, which is 10 to 15% lower than those achieved by C programs over VIA.

The second level of Javia (Javia-II) introduces a special buffer class that, coupled with special features in the garbage collector, eliminates the need for the extra copies. In Javia-II, the application can allocate pinned regions of memory and use these regions as Java arrays. These arrays are genuine Java objects (i.e. can be accessed directly) but are not affected by garbage collection as long as they need to remain accessible by the

network interface DMA engine. This allows Java applications to explicitly manage the buffers used by VIA and to transmit/receive Java arrays directly. Benchmarks show that programs using Javia-II can achieve bandwidths of over 80 Mbytes/s for large messages (> 8Kbytes), which are within 1% (error range) of those achieved by C programs over VIA.

It should be clear that Javia solely provides efficient data transfer between hosts in a cluster and does not implement or replace a complete message passing, RPC, or RMI interface. However, it is intended as a building block for the construction of such complete communication libraries.

Section 2 provides background on the Virtual Interface Architecture and on the Marmot Java system used in the paper. Sections 3 and 4 describe the Javia-I and Javia-II architectures, respectively. Section 5 relates Javia to other efforts in improving Java's communication performance and Section 6 concludes.

2. BACKGROUND

2.1 Virtual Interface Architecture

The Virtual Interface Architecture (VIA) defines a standard interface between the network interface hardware and applications. The key feature of VIA is that the applications manage buffers explicitly and that the network interface DMA engine transfers data directly into and out of the application buffers located in user-space. The target application area of VIA is cluster communication: VIA is connection-oriented and assumes that the links have high reliability. The rest of this subsection focuses on the aspects of VIA that are relevant to the communication critical path.

To access the network, an application opens a virtual interface (VI), which forms the endpoint of the connection to a remote VI. Each VI has two associated queues—a send queue and a receive queue—that are thread-safe and implemented as linked lists of message descriptors, each of which points to one or multiple buffer descriptors. To send a message, an application composes the message in a buffer, builds a buffer descriptor, and adds it to the end of the send queue. The network interface fetches the descriptor, transmits the message using DMA, and sets a bit in the descriptor to signal completion. An application eventually checks the descriptors for completion (e.g. by polling) and dequeues them.

Similarly, for reception, an application adds descriptors for free buffers to the end of the receive queue, and checks (polls) the descriptors for completion. The network interface fills these buffers as messages arrive and sets completion bits. Incoming packets that arrive at an empty receive queue are discarded. VIA permits a single-threaded application to poll more than one receive queue at a time. Although VIA allows for interrupt-driven reception, the commercial implementation used in this paper only supports poll-based reception. This paper does not use VIA's remote DMA operations (direct remote memory access).

Protection is enforced by the operating system and by the virtual memory system. All buffers and descriptors used by an application are located in memory mapped into that application's address space. Other applications cannot interfere with communication because they do not have the buffers and descriptors mapped into their address space.

A major difficulty in the design of user-level network interfaces is handling virtual to physical address translations in the network interface. This is required because pointers (e.g. to descriptors or buffers) are specified as virtual addresses by the applications yet the network interface must use physical addresses to access main memory with DMA. In VIA this is handled by placing all buffers and descriptors into memory regions that are registered with the network interface before they are used. A memory region is a virtually contiguous memory segment that an application allocates and registers with VIA. The registration is performed by the operating system, which pins the pages underlying the region and communicates the physical addresses to the network interface. The latter stores the translation in a table indexed by a region number. While all addresses in descriptors are virtual, the application is required to indicate the number of the region with each address (in effect all addresses are 64 bits consisting of a 32-bit region number and a 32-bit virtual address) so that the network interface can translate the addresses using its mapping table.

2.2 Marmot

Marmot [FKR+98] is a Java system developed at Microsoft Research that consists of a static, optimizing, bytecode to native code compiler and a runtime system. The compiler applies standard optimizations (e.g. array bounds check elimination, common sub-expression elimination, and constant folding), object-oriented optimizations (e.g. method inlining and type cast elimination), and Java-specific optimizations such as array-store-check elimination. The compiler currently generates x86 code and does not rely on any external compiler or back-end. Java programs compiled by Marmot run roughly 1.5x to 5x faster than using Microsoft's Visual J++ VM (5.0).

Most of Marmot's runtime support is implemented in Java, including casts, *instanceof*, array store checks, thread synchronization, and interface call lookup. Synchronization monitors are implemented as Java objects, which are updated in critical sections written in C. Threads are also Java objects that are mapped onto Win32 threads. Marmot supports most of JDK1.1. The garbage collector used in this paper is a semi-space copying collector based on the Cheney scanning algorithm. All objects are allocated in the garbage-collected heap.

Marmot's native code interface is not JNI-compliant but possesses all the features that are needed for interfacing with VIA. It passes all Java objects by reference to native code, where they are accessed as C structures. Garbage collection is automatically disabled when any thread is running native code, but can be enabled upon request. Marmot's native interface is fast: a call of a null native method costs only 0.3µs on a 450Mhz Pentium-II.

3. JAVIA-I

The general Javia architecture consists of a set of Java classes and a native library. The Java classes are used by applications and interface with a commercial VIA implementation through the native library. The core Javia classes are shown in Figure 1. The class `Vi` represents a connection to a remote VI and borrows the connection set-up model from the JDK sockets API. When an instance of `Vi` is created a connection request is sent to the remote machine (specified by `ViAddress`) with a tag. A call to `ViServer.accept` (not shown) accepts the connection and returns a new `Vi` on the remote end. If there is no matching accept, the `Vi` constructor throws an exception.

```
1   package cornell.slk.javia;
2
3   public class Vi { /* connection to a remote VI */
4     public Vi(ViAddress mach, ViAttributes attr) { … }
5
6     /* async send */
7     public void sendPost(ViBATicket t);
8     public ViBATicket sendWait(int millisecs);
9
10    /* async recv */
11    public void recvPost(ViBATicket t);
12    public ViBATicket recvWait(int millisecs);
13
14    /* sync send */
15    public void send(byte[] b,int len,int off,int tag);
16
17    /* sync recv */
18    public ViBATicket recv(int millisecs);
19  }
20
21  public class ViBATicket {
22    private byte[] data; private int len, off, tag;
23    private boolean status;
24    /* public methods to access fields ommited */
25  }
```

Figure 1. Core Javia classes and the Javia-I interface.

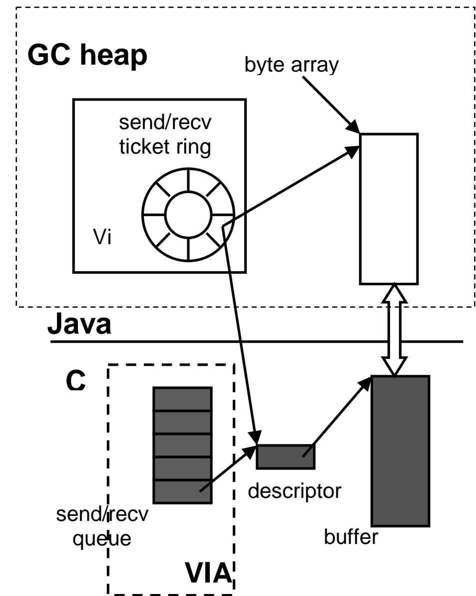

Figure 2. Javia-I endpoint architecture. Solid arrow indicates data copying.

Javia-I is an interface with methods to send and receive Java byte arrays[1]. The asynchronous calls (lines 7-12) use a Java-level descriptor (ViBATicket) to hold a reference to the byte array being sent or received and other information such as the completion status, the transmission length, offset, and a 32-bit tag. Figure 2 shows the Java and native data structures involved during asynchronous send/recv. Buffers and descriptors are managed (pre-allocated and pre-pinned) in native code and a pair of send/recv ticket rings is maintained in Java and used to mirror the VI queues. To send a Java byte array, Javia-I gets a free ticket from the ring, copies the data from the byte array into a buffer and enqueues that on the VI send queue. To receive into a byte array, Javia-I obtains the ticket that corresponds to the head of the VI receive queue, and copies the data from the buffer into the byte array. The ticket ring is updated upon the completion of the operation.

Javia-I provides a blocking send call (line 15) because in virtually all cases the message is transmitted instantaneously—the extra completion check in an asynchronous send is more expensive than blocking in the native library. It also avoids accessing the ticket ring, which requires locks and Java array checks and enables two send variations. The first one (*send-copy*) copies the data from the Java array to the buffer whereas the second (*send-pin*) pins the array on the fly, avoiding the copy[2]. The blocking receive call (line 18) polls the reception queue for a message, allocates a Java byte array of the right size, copies the data into it, and returns a ticket. While the blocking receive is more "natural", it requires an allocation for every message received and eventual garbage collection. Pinning the byte array for reception is unacceptable because it would require the garbage collector to be disabled indefinitely.

3.1 Performance

Two simple benchmark programs are used to evaluate the round-trip latency and the bandwidth achieved between two hosts using Javia-I. The experimental set-up consists of two 450Mhz Pentium-II systems with a 100Mhz system bus and running Windows 2000 beta3. The VIA cards are two Giganet 1.25Gbps GNN1000 interfaces connected through a Giganet GNX5000 (version A) switch.

Figures 3(a) and 3(b) show the round-trip latency and the bandwidth for a C program using VIA library directly (*raw*), a Java program using Javia-I send-copy and send-pin with asynchronous receive (*copy* and *pin* respectively) and with blocking receive (*copy+alloc* and *pin+alloc* respectively). Table 1 shows the 4-byte r/t latencies and the per-byte cost.

	4-byte(us)	per-byte(ns)
raw	16.5	25
copy	21.5	42
pin	38.0	38
copy+alloc	18.0	55

Table 1. 4-byte and per-byte round-trip latencies.

Pin has the highest 4-byte latency because of the pinning costs (20µs to pin and unpin a page) and has a per-byte cost that is closest to that of *raw* (the difference is because data is still being copied at the receive end). *Copy+alloc*'s 4-byte latency is only 1.5µs above that of *raw* because it bypasses the ticket ring on both send and receive ends. Its per-byte cost, however, is significantly higher than that of *copy* due to garbage collection overhead. *Pin*'s effective bandwidth is about 85% of that of *raw* for messages larger than 6Kbytes. Due to the high pinning costs, *copy* achieves an effective bandwidth (within 70-75% of *raw*) that is higher than that of *pin* for messages smaller than 6Kbytes.

[1] The complete Javia-I interface provides send/recv calls for all primitive-typed arrays.

[2] The garbage collector is disabled during the operation.

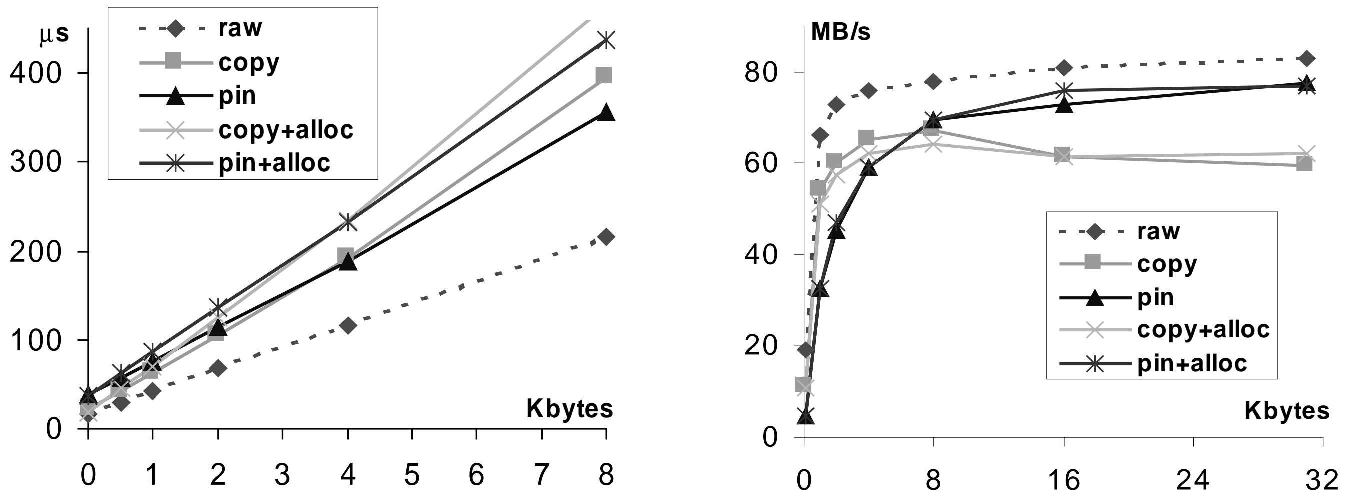

Figure 3. (a) round-trip latencies and (b) effective bandwidth of four variants of Javia-I compared with C over VIA (*raw*).

3.2 Summary

Javia-I provides a simple interface to VIA by hiding all the VIA data structures in a native library and copying all data between the VIA buffers and Java arrays. The *pin* variant on the sending side replaces the copy costs with those of pinning and unpinning an array in the critical path, which can be quite high. While C applications can amortize this cost by re-using buffers, Java programmers cannot because of the lack of explicit control over object location and lifetime. Moreover, as mentioned before, pinning on the fly cannot be applied to the receiving end.

While this approach does not achieve the best performance with large messages, it is attractive for small messages and can be implemented on any off-the-shelf Java system that supports a native interface similar to Sun's Java Native Interface specification (JNI). The latest version of JNI (version 2) provides functions that allow native code to obtain a direct pointer to array elements (through `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` calls), enabling the implementation of the *pin* variant. As the JVM may or may not actually make a copy of the array, we expect the best case performance to closely match the one reported in this paper.

4. JAVIA-II

Javia-II addresses the shortcomings of Javia-I by exposing the communication buffers used by VIA to Java applications. The idea is to give Java programmers the same flexibility that C programmers have for managing buffers while maintaining Java's type and storage safety. An application can manage buffers explicitly, access them efficiently, and re-use them with the cooperation of the garbage collector.

A buffer is abstracted by the `ViBuffer` class, shown in Figure 4. Allocating a `ViBuffer` with the static `alloc` method (line 5) causes Javia-II to allocate a buffer of the specified size *outside* the Java heap. The `register` method (line 9) pins the buffer to physical memory (so it can be used by VIA), associates it with a VI, and obtains a descriptor to the buffer, which is represented by a `ViBufferTicket`. At that point, the buffer can be directly accessed by VIA for communication. Note that the buffer can be unregistered (line 10), which unpins it, and later re-registered with the same or a different VI. An application can access the buffer (e.g. perform read and write operations) as a Java primitive-typed array. For example, an invocation of `toByteArray` (line 13) returns a reference to a genuine Java byte array that is located in the buffer.

For transmission and reception of buffers, Javia-II provides only asynchronous primitives, as shown in lines 26-33. Javia-II differs from Javia-I in that the VIA descriptors point directly to the Java-level buffers instead of native buffers (Figure 5). The application composes a message in the buffer (through array write operations) and enqueues the buffer for transmission using the `sendBufPost` method. `sendBufPost` is asynchronous and takes a `ViBufferTicket`, which is later used to signal completion. After the send completes, the application can compose a new message in the same buffer and enqueue it again for transmission. Reception is handled similarly—the application posts buffers for reception with `recvBufPost` and uses `recvBufWait` to retrieve received messages. For each message, it extracts the data through array read operations and can choose to post the buffer again.

An application can manifest its intention to re-use or de-allocate a buffer by invoking its `unRef` method (line 17). This call makes the buffer visible by the garbage collector, enabling it to track the array references into the buffer and to notify the application when such references no longer exist. At this point, an application can obtain new array references into the buffer, possibly of some other primitive type, or de-allocate the buffer.

4.1 Explicit Management and Safety

Javia-II allows the programmer to manage buffers explicitly and safely by detaching the their lifetime from the lifetime of their Java references. In other words, the allocation of a `ViBuffer` does not result in a Java array reference to it (for example, a programmer has to call `toByteArray` to obtain that reference), and a `ViBuffer` is not automatically freed when there are no more Java array references to it. To maintain the safety properties of Java, Javia-II guarantees that referenced buffers will never be de-allocated.

Safety is enforced using runtime checks with the cooperation of the garbage collector. A buffer can be in three states:

```
1   package cornell.slk.javia;
2
3   public class ViBuffer {
4     /* explicit allocation and free */
5     public static ViBuffer alloc(int bytes);
6     public void free();
7
8     /* pinning and unpinning */
9     public ViBufferTicket register(Vi vi);
10    public void deregister(ViBufferTicket t);
11
12    /* handing out references */
13    public synchronized byte[] toByteArray();
14    public synchronized int[] toIntArray();
15
16    /* getting rid of references */
17    public void unRef(CallBack cb);
18  }
19
20  public class ViBufferTicket {
21    /* no public constructor */
22    ViBuffer buf; private int len, off, tag;
23    /* public methods to access fields ommited */
24  }
25
26  public class Vi {
27    /* async send */
28    public void sendBufPost(ViBufferTicket t);
29    public void sendBufWait(int millisecs);
30
31    /* async recv */
32    public void recvBufPost(ViBufferTicket t);
33    public void recvBufWait(int millisecs);
34  }
```

Figure 4. ViBuffer class and the Javia-II interface.

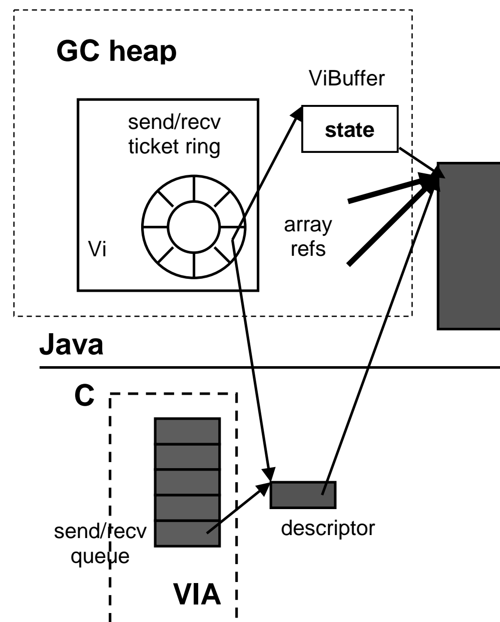

Figure 5. Javia-II endpoint architecture. Java array references point directly into the region outside the heap.

1. unreferenced (*unref*), meaning that there are no Java references into the buffer;
2. referenced (*ref<p>*), meaning that there is at least one Java array reference (of primitive type *p*) to the buffer;
3. to-be-unreferenced (*2b-unref<p>*), meaning that the application claims the buffer has no array references of type *p* and waits for the garbage collector to verify that claim.

A `ViBuffer` starts in *unref* and makes a transition to *ref<p>* upon an invocation of `to<p>Array`. The state is parameterized by a primitive type *p* to enforce type safety. After an `unRef` invocation, the buffer goes to the *2b-unref<p>* state and becomes "collectable." It then returns to *unref* once the garbage collector verifies that the buffer is indeed no longer referenced and invokes the callback. A buffer can only be de-allocated if it is in the *unref* state and can be posted for transmission and reception as long as it is not in the *2b-unref<p>* state.

Exactly when the transition back to the *unref* state will occur depends on the type of the collector. A non-copying collector will only invoke the callback after the programmer has dropped all the array references to the buffer. A copying collector, however, ensures that the transition will always occur at the next collection since it will move the array out of the buffer and into the Java heap. This means that, for example, the application can continue using the data received in the array without keeping the buffer occupied and without performing an explicit copy.

4.2 Implementation

In order to support `ViBuffers`, the garbage collector must be able to change the scope of its collected heap dynamically. When a buffer is `unRefed`, the collector should integrate the buffer's region of memory into the heap, and later remove it from the heap after asserting that it is no longer referenced from Java.

We made minor modifications to the Cheney scanning copying garbage collector used in the Marmot system. When following a reference, the collector copies the referenced object to *to-space* if the object lies in the *from-space*. The augmented Marmot collector keeps a list of memory regions that form each of the semi-spaces and for each referenced object, it traverses the *from-space* list to determine whether to copy the object or not. When Javia allocates a communication buffer, it does not yet place it into the collector's *from-space* list, with the effect that the array residing in the buffer is not moved. When the application calls `unRef`, however, Javia adds the buffer to the list. Upon collection termination, Javia removes the buffer from the list and invokes the associated callback method.

4.3 Performance

Table 2 and Figure 6(a) compare the round-trip latency obtained by Javia-II (*buffer*) with *raw* and two variants of Javia-I (*pin* and *copy*). The 4-byte round-trip latency of Javia-II is 20.5μs and the per-byte cost is 25ns, which is the same as that of *raw* because no

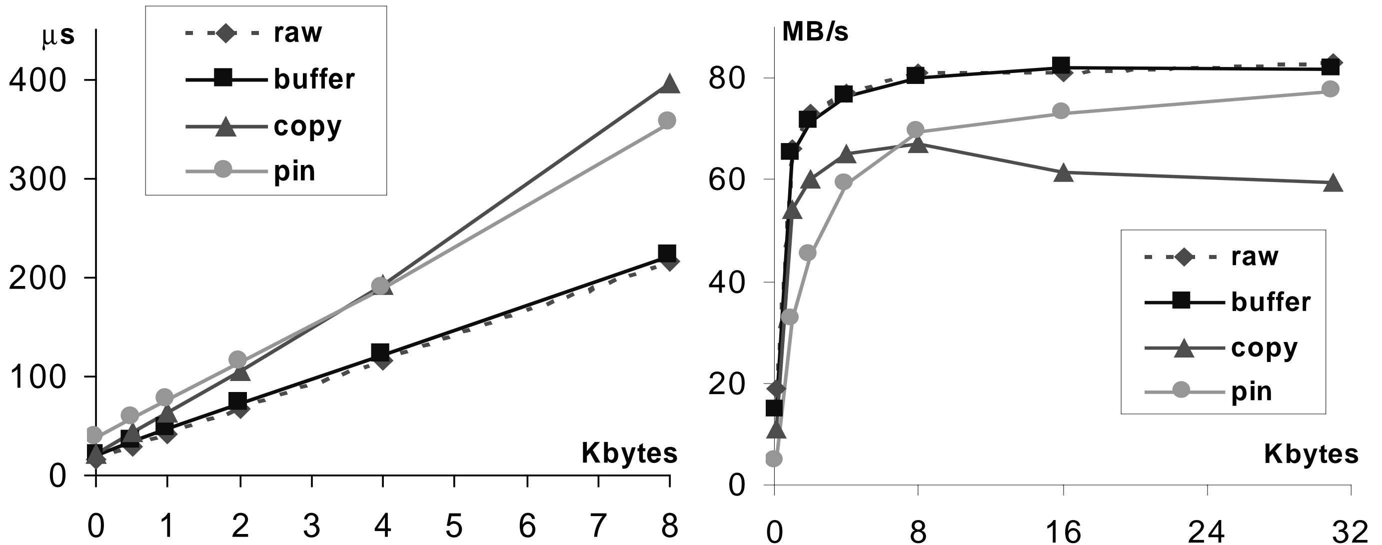

Figure 6. (a) round-trip latencies and (b) effective bandwidth of Javia-II (*buffer*), compared with C over VIA (*raw*), and two variants of Javia-I (*copy* and *pin*).

	4-byte (us)	per-byte(ns)
raw	16.5	25
buffer	20.5	25
pin	38.0	38
copy	21.5	42

Table 2. 4-byte and per-byte round-trip latencies.

data copying is performed in the critical path. The effective bandwidth achieved by Javia-II (Figure 6(b)) is within 1 to 3% of that of *raw*, which is in the margin of error.

4.4 Summary

Javia-II exposes the communication buffers to the Java application so the programmer can manage them efficiently using application-specific information. These buffers can be efficiently accessed from Java (through array read and write operations) and can be accessed directly by VIA, eliminating the need for additional buffers in native code. As a result, data copies are removed from the critical path and deferred to garbage collection only if they are needed.

It may seem that buffers would violate Java's storage safety because an application can leak memory by not de-allocating buffers. However, the language itself does not prevent a programmer from consuming unlimited memory (e.g. by deliberately growing an unused linked-list indefinitely), which in many ways is equivalent to leaking memory.

One difficulty is that an application or communication library may have to force a garbage collection whenever it needs to reclaim its communication buffers, which could be counter-productive. A programmer, however, can tune that application for maximum performance—our goal is to provide the flexibility to doing so.

5. RELATED WORK

One design alternative for Javia-I would have been to implement a JNI wrapper library around the VIA C library calls (in the style of the MPI wrapper used in [GFH+98]). The JNI specification isolates the details of the VM from native code for portability purposes. Implementing Javia-I using JNI would have been a straightforward exercise although its effectiveness would depend on the underlying JVM implementation. At best, the JNI approach would have achieved performance similar to Javia-I presented in this paper.

Another design alternative for Javia would have been to use the JDirect technology available in the Microsoft JVM. JDirect enables a Java programmer to embed a primitive-typed array into a pinned (non-collectable) object using source-level annotations, which propagates through the bytecode to the JIT for special code generation. Although a pinned array can be passed between Java and C without copying, accesses to a pinned array from Java would have to undergo a level of indirection, which would be prohibitively expensive. In addition, managing pinned arrays instead of Javia buffers for communication would likely be a more arduous and expensive task because the programmer cannot rely on the garbage collector to track references and to copy data out of the buffers.

A number of projects have adopted a "front-end" approach to developing communication software for Java applications: given a particular abstraction (e.g. sockets, RMI, MPI), they provide "glue-code" for interfacing with legacy libraries in native code. For example, [GFH+98] makes the MPI communication library available to Java applications by providing automatic tools for generating Java-native interface stubs. [BDV+98] deals with interoperability issues between Java RMI and HPC++, and [F98] presents a simple Java front-end to PVM. These projects do not address the performance penalty incurred when interfacing Java with native code using conventional techniques.

Recently, several projects have focused on making the performance of Java RMI suitable for parallel computing on clusters of workstations. Manta [MNV+99] implements Java RMI efficiently over a Panda, a custom communication system. Manta relies on compiler-support for generating marshaling and unmarshaling code in C, thereby avoiding type checking at runtime. It communicates using both JDK's serialization protocol (for compatibility) as well as a custom protocol (for performance). Manta is able to avoid array copying in the critical path by relying

on a non-copying collector and scatter/gather primitives in Panda. The authors report a RMI latency of 35µs and a throughput of 51.3 Mbytes/s on a PII-200/Myrinet cluster, which is within 15% of the throughput achieved by Panda.

KaRMI [NPH99] presents a ground-up implementation of object serialization and RMI entirely in Java. Unlike Manta, the authors seek to provide a portable RMI package that runs on any JVM. On an Alpha500/ParaStation cluster, they report a point-to-point latency of 117µs and a throughput of 2.3 Mbytes/s (compared to a raw throughput of 50 Mbytes/s). The low bandwidth is attributed to the several data copies in the critical path: on each end, data is copied between objects and byte arrays in Java and then again between arrays and message buffers. The copying overhead is so critical that their serialization improvements over JDK1.4 vanish quickly as transfer size increases. Not surprising, both Manta and KaRMI identify data transfer, in particular object serialization, as the major bottleneck in Java communication.

6. CONCLUDING REMARKS

The research presented here pursues the simple goal of exposing user-level network interfaces to Java applications. Unfortunately, the automatic memory management of Java makes it difficult to expose the UNI concepts in a straightforward manner as explicit memory management by the application is at the core of the UNI concept.

The main contribution of this paper is to show that, with adequate support from the garbage collector, Java programs can interface efficiently with the networking hardware. By first exploiting native buffers, Javia-I motivates the need for explicit management of buffers and Javia-II. The performance achieved with Javia-II is encouraging: the overhead of the interface is small compared to the round-trip latency and the peak bandwidth is essentially the same as that achieved using the raw C VIA library. This is a clear indication that it is worth removing the copies from the critical path. Although Javia-II cannot be ported to any off-the-shelf JVM, we believe the support required from the garbage collector can be easily implemented.

The essential motivation for Javia is to provide programmers with an alternative to the traditional, "front-end" approach for Java communication support. Instead of performing ad-hoc optimizations in C libraries and in glue-code, Javia tackles the fundamental inefficiency by exposing communication buffers to Java programmers. It is intended to serve as a basic building block for writing applications and higher-level communication paradigms *entirely* in Java. It is also possible to use object serialization and Javia to implement Java RMI, although the current overhead of serialization implementations would negate any performance benefits gained with Javia. We are currently extending the flexibility of Javia buffers for high-performance streaming of arbitrary Java objects.

7. REFERENCES

[ACL+98] Adl-Tabatabai, A., Cierniak, M., Lueh G-Y., Parikh, V., and Stichnoth, J. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *ACM Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

[ADM98] Agesen, O., Detlefts, D., and Moss, E., Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *ACM Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

[BDV+98] Breg, F., Diwan, S., Villacis, J., Balasubramanian, J., Akman, E., Gannon, D. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, February 1998.

[BGC98] P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Supercomputing '98*, Orlando, FL, November 1998.

[BKM+98] Bacon, D., Konuru, R., Murthy, C., Serrano, M. Thin Locks: Featherweight Synchronization in Java. In *ACM Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

[DBC+98] Dubnicki, C., A. Bilas, Y. Chen, S. Damianakis, and K. Li. *Shrimp Project Update: Myrinet Communication.* IEEE Micro, Vol. 18, No. 1, January/February 1998.

[F98] Ferrari, J. A., JPVM: Network Parallel Computing in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, February 1998.

[FKR+98] Fitzgerald, R., Knoblock, T., Ruf, E., Steensgard, B., and Tarditi, D. Marmot: An Optimizing Compiler for Jav*a*. Submitted for publication, October 1998.

[GFH+98] Getov, V., S. Flynn-Hummel, and S. Mintchev, High-Performance Parallel Programming in Java: Exploiting Native Librarie*s*. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, February 1998.

[GNN] Giganet, Inc. http://www.giga-net.com.

[MMG98] Moreira, J., Midkiff, S., and Gupta, M. From Flop to MegaFlops: Java for Technical Computing. *IBM Research Report RC 21166 (94954)*, April 1998.

[MNV+99] Maassen, J., Nieuwpoort, R., Veldema, R., Bal, H., and Plaat, A. An Efficient Implementation of Java's Remote Method Invocation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1999.

[PLC95] Pakin, S., M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrine*t*. In *Supercomputing '95*, San Diego, CA, 1995.

[SPS98] R. Sankaran, C. Pu, and H. Shah. Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks. In *USENIX NT Symposium*, Seattle, WA, August 1998.

[SSP99] Shah, H. V., R. M Sankaran, and C. Pu, High performance sockets and RPC over virtual interface architecture, In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing CANPC '99*, Orlando, FL, January 1999.

[vEBB+95] von Eicken, T., A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *15th Annual Symposium on Operating System Principles*, p. 40-53, Copper Mountain, CO, December 1995.

[VIA97] The Virtual Interface Architecture. http://www.via.org