Interfacing Java with the Virtual Interface Architecture

Chi-Chao Chang and Thorsten von Eicken
Department of Computer Science
Cornell University

{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 it 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 implements a special buffer abstraction that allows Java programs to allocate arrays in pinned memory and use them as communication buffers without copy. The location and lifetime of these arrays are controlled through small modifications to the garbage collector. Simple experiments show that Java programs can achieve round-trip times of 21µs for small messages and bandwidths of 95Mbytes/sec for 4Kbyte messages.

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, CMC98]. 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 (e.g. from GigaNet [GNN]) 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. 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] 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 all copies and it 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 and location of Java objects, which is exactly what is required to use UNIs directly.

In this paper, we propose a two-level Java interface to VIA, called Javia, which overcomes these problems. 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 byte arrays. Javia-I uses native code and can be implemented in any JVM that provides a JNI-like native interface.

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 create pinned regions of memory and then allocate Java arrays in those regions. 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. Simple performance benchmarks show that programs using Javia-II can achieve round-trip latencies of 21μs for small messages and bandwidths of over 95 Mbytes/sec for large messages.

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 discusses the results.

2 Background

This section provides background information on the VIA architecture and on the Marmot Java system.

2.1 The 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 DMAs message 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 it assumes that the links have high reliability.

To access the network, an application opens virtual interfaces (VIs) which form the endpoints of connections to VIs at the remote ends. Each VI has two associated queues—a send queue and a receive queue—that are 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. Eventually the application dequeues the descriptor. For reception, the application adds descriptors for free buffers to the end of the receive queue. The network interface fills these buffers as messages arrive and sets completion bits. Incoming packets that arrive at an empty receive queue are discarded.

Protection is enforced by the operating system and by the virtual memory system. The key idea is that all the buffers and descriptors used by an application are located in memory mapped in 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 to handle 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.

VIA also supports direct access to remote memory using so-called RDMA operations, however, Javia does not support them.

A note on reliability: VIA assumes that the underlying network is basically reliable and that errors indicate network failures. It nevertheless allows applications to choose between three levels of reliability: *unreliable*, *reliable delivery*, and *reliable reception*, which correspond to the different stages at which failures are signaled. In all these cases, a failure is catastrophic and all subsequent operations on that VI are discarded and the VI is placed in an error state.

2.2 The Marmot Java System

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 native 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), as well as Java-specific optimizations such as array store check elimination. The multi-stage compilation process includes converting bytecode to a custom intermediate representation in SSA form for high-level optimizations, and then to a low-level representation for optimization prior to code generation. 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 native critical sections. Threads are also Java objects that are mapped onto Win32 native threads. Marmot supports JDK1.1-compliant Java libraries, including most of `java.lang`, `java.util`. `java.io`, `java.awt`, `java.net`, and `java.lang.reflect`. The garbage collector used in this paper is a semi-space copying collector based on the Cheney scanning algorithm.

The Marmot native code interface is not JNI-compliant but possesses most of the features that are needed for interfacing with VIA. In particular, it allows native code to enable and disable garbage collection during the native call, provided that the thread is put in a "safe state" prior to garbage collection. Native code is not allowed to retain Java pointers across garbage collections, but the per-thread state includes fields where native code may place pointers that must be updated by the collector. This is essential for tracking byte arrays posted in the receive queue across garbage collection occurrences, as explained in the next section. Marmot's native interface is fast: a call of a null native method costs only 0.27µs on a 300Mhz Pentium-II. Unfortunately acquiring and releasing a lock costs 1.2µs compared to 0.4µs with the Visual J++ VM.

3 Javia-I

Javia-I consists of a set of Java classes and a native library that allow Java applications to send and receive byte arrays using VIA. All the VIA data structures are managed in the native code and are beyond the control of the Java programmer. To send a Java byte array, Javia-I normally copies the data into a VIA buffer and enqueues that on the send queue. For byte arrays over 8Kbytes, Javia-I makes the pages holding the byte array into a VIA region, pins that, and sends the data without copy. In all cases, `send` is blocking and returns when VIA indicates that the message has left the network interface. We experimented with asynchronous sends but found that in virtually all cases the message is transmitted instantaneously and that the extra completion check is more expensive than blocking in the native library.

For reception, Javia-I provides two interfaces for the sake of experimentation. The first interface conforms to the Java mind-set: `recv` checks the reception queue for a message, allocates a Java byte array of the right size, copies the data into it, and returns the array. While this interface is "natural", it requires an allocation and eventual garbage collection for every message received. The second interface provided in Javia-I is closer to traditional C-style asynchronous message passing interfaces and avoids allocating within Javia-I. `recvPost` adds a byte array to the receive queue and `recvWait` returns the next received byte array. This second interface uses a descriptor (`ViBATicket`) to hold a reference to the byte array being filled, the completion status, and the length of the received message.

The core Javia-I classes are shown below. The class Vi represents a connection to a remote VI and borrows the connection set-up model from Java sockets (`java.net.Socket`). 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 here) with a matching tag accepts the connection and returns a new `Vi` on the remote end. If there is no matching accept, the `Vi` constructor throws an exception.

```
public class Vi { /* connection to a remote VI, methods to send/receive */
  public Vi(ViAddress mach, ViAttributes attr,
      ViCompletionQ sendQ, ViCompletionQ recvQ, int tag) throws ViException {…}

  /* send: send a sub-byte array, return when transmission completes */
  public void send (byte[] b, int off, int len) throws ViException {…}

  /* recv: wait for a message with timeout, return it in an allocated byte[] */
  public byte[] recv (int microseconds) throws ViException, ViTimeout {…}

  /* asynchronous recv: post adds a buffer to the receive queue and wait
   * returns the next received message */
  public void recvPost(ViBATicket t) throws ViException {…}
  public ViBATicket recvWait(int microsecs) throws ViException, ViTimeout {…}

  public void close() throws ViException {…}
}

public class ViBATicket { /* descriptor for a byte array buffer */
  public ViBATicket(byte[] b, int off) {…}
  public boolean isDone() {…} /* true indicates send/recv completed */
  public int getLength() {…} /* for recv: length of message received */
  public byte[] getByteArray() {…}
}
```

Send:	Recv:

`Vi vi = new Vi(addr, attr, null,`
`null, 123);`
`byte[] b = new byte[100];`
`/* send b */`
`vi.send (b, 0, 100);`
`. . .`
`vi.close();`
	`ServerVi sv = new ServerVi(123);`
	`Byte[] b = new byte[100];`
	`Vi vi = sv.accept();`
	`ViBATicket t = new ViBATicket(b,0);`
	`vi.recvPost(t);`
	`t = vi.recvWait(Vi.INFINITE);`
	`b = t.getByteArray();`
	`/* read b */`
	`vi.close();`

Table 1. Sample code for message transmission and reception using Javia-I

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 300Mhz Pentium-II systems with a 66Mhz system bus and running Windows NT 4.0 SP3. The VIA cards are two GigaNet's 1.25Gbps GNN1000 interfaces connected back-to-back.

Figure 1 shows the round-trip time for (i) a C program using the raw VIA library interface, (ii) Javia-I when the messages are copied between the VIA buffers and the Java byte arrays, and (iii) Javia-I when the Java byte arrays are pinned by Javia on every transmission. In both Javia programs the asynchronous receive is used. The round-trip latency for a 4-byte message is 18μs with C and 22μs with Javia-I with the copy, indicating a Javia-I overhead of 4µs. Pinning the transmit buffer increases the latency to 32µs, thus pinning costs about 10µs.

Figure 2 shows the bandwidth using the same three interfaces when sending a total of 15Mbytes of data. Evidently pinning the transmission buffer is only advantageous when the message length exceeds 7000 bytes.

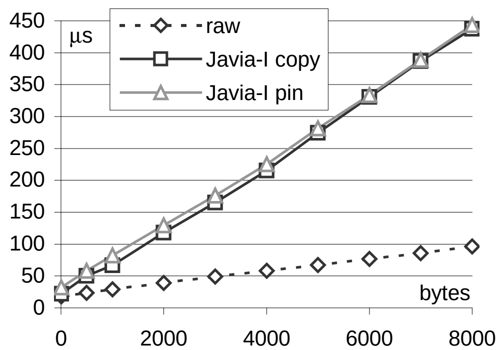


Figure 1. round-trip latency using the raw C VIA library, using Javia-I with message copy, and Javia-I with buffer pinning on the sending side.

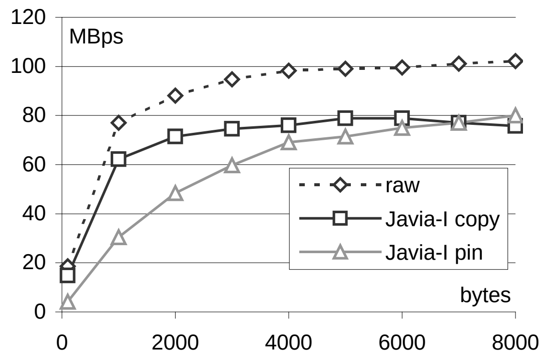


Figure 2. bandwidth using the raw C VIA library, using Javia-I with message copy, and Javia-I with buffer pinning on the sending side.

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 byte arrays. While this approach does not achieve the best performance with large messages, it is attractive for small messages and it can be implemented on any off-the-shelf Java system that supports a native interface similar to Java Native Interface 2 (JNI2).

4 Javia-II

Javia-II exposes the pinned buffers used by VIA to the Java application in order to eliminate the copies performed by Javia-I. The pinned VIA buffers are abstracted by the `ViBuffer` class, which allows Java applications to allocate, pin, use, and deallocate these buffers explicitly. Behind the scenes, a modified garbage collector is aware of these pinned buffers and handles them specially.

Allocating a `ViBuffer` with the standard `new` operator causes the Javia-II native library to allocate a communication buffer of the specified size outside the normal Java heap. The `ViBuffer` object is a descriptor to that communication buffer, which itself is still hidden from the Java application. The `register` method pins the buffer to physical memory (so it can be used by VIA), associates it with a VI, and sets its VIA attributes (e.g., read-only, RDMA enable). At that point the buffer is “initialized” and can be used for communication. Note that the buffer can be unregistered, which unpins it, and later re-registered with the same or a different VI. This means the application has direct control over the pinning and VI association of memory.

```
public class ViBuffer {
  /* constructor and finalizer */
  public ViBuffer(int sizeInBytes) {…}
  public void finalize() {…}

  /* primitive-typed arrays */
  public synchronized byte[] toByteArray() {…}
  public synchronized boolean[] toBooleanArray() {…}
```

```
  public synchronized int[] toIntArray() {…}
  . . .
  /* freeing a buffer */
  public void free(BufferCallBack cb) {…}

  public ViBufferTicket register(Vi vi, ViBufferAttributes attr)
    throws IOException {…}
  public void unregister(ViBufferTicket ticket) throws IOException {…}

  private int baseAddr;
  private int size;
  private int handle;
}

public class ViBufferTicket {
  ViBufferTicket (ViBuffer b) {…}
  public boolean isDone() {…}
  public int getLength() {…}
}

public class Vi {
  /* Javia-I methods */
  ...
  /* Javia-II methods */
  public void sendBufPost(ViBufferTicket t, int off, int len)
                                                    throws ViException {…}
  public ViBufferTicket sendBufWait(int microseconds) throws ViException {…}

  public void recvBufPost(ViBufferTicket t, int off) throws ViException {…}
  public ViBufferTicket recvBufWait(int microseconds) throws ViException {…}
}
```

<table>
<tr>
<td><pre>Send:
  Vi vi = new Vi(addr, attr, null,
                 null, 123);
  ViBuffer vb = new ViBuffer(100);
  ViBufferTicket t =
            vb.register(vi, attr);
  byte[] b = vb.toByteArray();
  /* write to b here */
  vi.sendBufPost(t, 0, 100);
  . . .
  t = vi.sendBufWait(Vi.INFINITE);
  vb.unregister(t);
  vi.close();</pre></td>
<td><pre>Recv:
  ServerVi sv = new ServerVi(123);
  ViBuffer vb = new ViBuffer(100);
  byte[] b = vb.toByteArray();
  Vi vi = sv.accept();
  ViBufferTicket t =
             vb.register(vi, attr);
  vi.recvBufPost(t, 0);
  t = vi.recvBufWait(Vi.INFINITE);
  /* read b */
  vb.unregister(t);
  vi.close();</pre></td>
</tr>
</table>

Table 2. Sample code for message transmission and reception using Javia-II

For transmission and reception the application has a number of options. In the simplest form of sending, the application creates a primitive type array (`byte[]`, `char[]`, `int[]`) in the communication buffer by calling the `toByteArray`, `toCharArray`, or `toIntArray` methods of the `ViBuffer`. This returns a handle to a genuine Java array located in the communication buffer accessible to VIA. The application then composes the message in this array and enqueues it for transmission using the `sendBufPost` method. `sendBufPost` is asynchronous and takes a `ViBufferTicket`, which is used to signal completion. After the send completes, the application can compose a new message in the Java array and enqueue it again for transmission.

Reception can be handled similarly. The application posts buffers for reception with `recvBufPost` and uses `recvBufWait` to retrieve received messages. For each message, it extracts the data from the Java array and posts the buffer again.

These transmission and reception models have a number of limitations. First, once a Java array is allocated in a communication buffer, it can never be changed to another array type as that would violate the type safety of the language. Second, it is unclear how the memory allocated outside of the normal heap can ever be freed. Finally, while no copy takes place between the Java array and the communication buffer, the data still needs to be copied between the application data structures and the Java array.

The two first limitations can be overcome by involving the garbage collector in the buffer management. The idea is to explicitly control when the Java array located in the communication buffer is collectable and when it is not. As long as the Java array is used for transmission or reception, it must not be freed (or moved in the case of a copying collector). But later, the array can be marked as collectable. When the garbage collector determines that the Java array is unreferenced the communication buffer can be reinitialized for reuse.

In the case of a non-copying garbage collector, only buffers holding unreferenced Java arrays can be reused or freed. To do this, the application needs to drop all references to the Java array located in the buffer and then call `free` on the `ViBuffer`. This flags the Java array as collectable and causes a call-back to be invoked when the collection actually occurs. At that point the application can unregister and free or re-register the `ViBuffer`.

In the case of a copying garbage collector, the application does not even need to drop all references to the Java array as the garbage collector will copy the array into *to-space* in the normal heap at the next collection. This means that, for example, the application can continue using the data received in the byte array without keeping the buffer occupied and without performing an explicit copy.

While it may seem that this simply moves the copy into the garbage collector without eliminating it, this is not necessarily the case. If the message data cannot be processed immediately by the application, it has to allocate a second array to copy the data into. At the next garbage collection, the data in this second array is copied again by the collector. The difficulty is that if the application forces a garbage collection because it is running out of communication buffers, then this may be counter-productive.

4.1 Garbage collector modifications

The modifications made to the Cheney scanning copying garbage collector used in the Marmot system are relatively minor. 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 searches the *from-space* list to determine whether to copy the object or not.

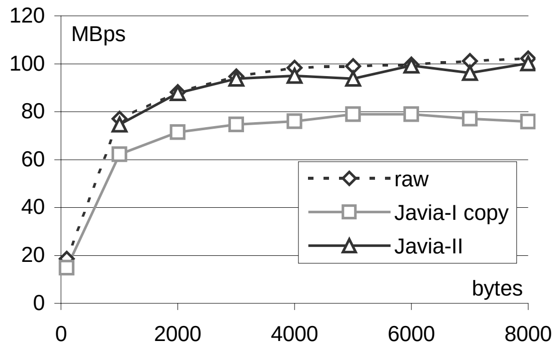
When Javia allocates communication buffers it does not place them into the collector's *from-space* list, with the effect that objects allocated in the buffers are not copied. When the application calls `free`, however, Javia adds the buffer to the list. (Implementation details ensure that the collector does not follow a long list when looking-up objects that are already in *to-space*.)

4.2 Performance

Figures 3 and 4 show the round-trip latencies and bandwidths obtained with Javia-II. The performance is very close to that of the raw C VIA library and significantly better than with Javia-I. The differences between Javia-II and the raw library are due to native code crossings and to the overheads of managing the `ViBufferTickets`, which involves acquiring and releasing locks.

5 Related Work

An alternative to developing Javia would have been to implement a minimal JNI2 wrapper library around the VIA C library calls (in the style the MPI wrapper used in [GFH+98]) or to call that library directly using the Microsoft VM JDirect feature. At best such an approach would have achieved performance similar to Javia-I, with perhaps the advantage of reducing the size of the native code required to interface to VIA. Using JNI could not have eliminated the copy that Javia-I performs. Microsoft's JDirect could, however, eliminate the copy (C arrays can be passed into Java without copy), but all Java accesses to the buffer arrays would have undergone a level of indirection which is expensive. JDirect relies on source annotations, which it propagates through the byte code to the JIT for special code generation.



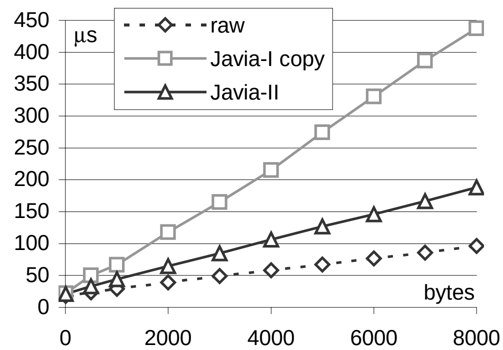


Figure 3. round-trip latency using the raw C VIA library, using Javia-I with message copy, and Javia-II.

Figure 4. bandwidth using the raw C VIA library, using Javia-I with message copy, and Javia-II.

The JNI2 specification further complicates attempts to interface Java to user-level network interfaces such as VIA. The JNI2 specification isolates the details of the VM from native code to allow copying to be performed at will by the JNI2 implementation. Even in cases where the VM might support pinning of arrays, the JNI2 specifies that accesses to such pinned arrays have to be wrapped with calls to `GetPrimitiveArrayCriticalSection` and `ReleasePrimitiveArray- CriticalSection`. The understanding is that one should not run "for too long" in such a critical section and certainly not call any system function that would block the current thread. These restrictions are plainly incompatible with the pinned communication buffers used in user-level network interfaces.

A number of projects have provided interfaces to native code message passing libraries [GFH+98, GS98] or have implemented traditional message passing standards in Java [F98]. The Javia interface presented here should not be viewed as an attempt to compete with these libraries. Rather, it provides a building block that offers high-performance data transfer and can be used to implement a number of different message passing paradigms.

6 Summary

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 implementation focuses on the Virtual Interface Architecture standard and proposes a two-level Java interface to VIA, called Javia-I and Javia-II. Javia-I hides the management of the communication buffers and descriptors entirely in a native code library and it copies all data sent or received between Java byte arrays and the buffers used by VIA.

Javia-II exposes the management of the communication buffers. To do so, we modified a copying garbage collector slightly to allow Java arrays to be allocated directly in the communication buffers and their collection to be controlled explicitly. While the buffers are pinned, the Java arrays are not collected and thus do not move. To reuse or free the buffers the arrays can be marked as collectable, in which case they are copied into the normal heap (or freed altogether) during the next collection.

The performance achieved with Javia in simple host-to-host communication benchmarks is encouraging: the overhead of the Javia interface is small compared to the round-trip latency and the peak bandwidth using 4Kbyte messages is essentially the same as that achieved using the raw C VIA library. Of course this is not all that surprising, given that Javia-II essentially provides a direct interface to VIA. However, the results indicate it is worth eliminating the copies of Javia-I.

It is clear that Javia is not an end goal, but a building block towards a user-friendly high-performance communication library for Java. It is fairly straight-forward to implement a library with MPI, PVM, or similar interface. It is also possible to use object serialization and Javia to implement JavaRMI, although the current overhead of serialization implementations would negate any performance benefits gained with Javia. We are currently extending the flexibility of Javia's `ViBuffers` to allow arbitrary objects to be allocated in the communication buffers.

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 *Proc. of 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 *Proc. of ACM Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

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

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

[CMC98] Chun, B. N., A. M. Mainwaring, and D. E. Culler. Virtual Network Transport Protocols for Myrinet. *IEEE Micro*, Vol. 18, No. 1, January/February 1998.

[CvE98] Chang, C-C., and von Eicken, T. A Software Architecture for Zero-Copy RPC in Java. Cornell CS TR 98-1708, September 1998.

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

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

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

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

[GS98] Gray, P.A., V. S. Sunderam, IceT: Distributed Computing and Java. Concurrency, Practice and Experience, Vol. 9, No. 11, John Wiley & Sons, Nov. 1997.

[GS97] Gokhale, A., and Schmidt, D. Measuring the Performance of Communication Middleware on High-Speed Networks. *Computer Communication Review 26(4),* October 1996.

[Java] JavaSoft. *Java Object Serialization Specification*. http://java.sun.com.

[PLC95] Pakin, S., M. Lauria, and A. Chien. *High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet*. In Proceedings of Supercomputing '95, San Diego, California, 1995.

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

[SSP99] Shah, H. V., R. M Sankaran, and C. Pu, High performance sockets and RPC over virtual interface architecture, 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 Proceedings of the 15th Annual Symposium on Operating System Principles, p. 40-53, Copper Mountain Resort, Colorado, Dec. 1995.

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