2 Interfacing Java to Network Interfaces

This chapter focuses on the performance issues that arise when interfacing Java to the underlying network devices. The chapter starts with an introduction to the Virtual Interface architecture [Via97], the de-facto standard of user-level network interfaces. It points out the features of the architecture that are critical to high performance and the language requirements (or lack of thereof) for applications to capitalize on these features. As these requirements—in particular, the ability to manage buffers explicitly—are ill matched to the foundations of Java, direct access to hardware resources must be carried out in native code. This results in a hard separation between Java's garbage-collected heap and the native, non-collected heap.

Existing Java-native interfaces cope with this separation in different ways. A common theme in their design is the tension between efficiency and portability[5], a perennial issue in computer system design. On one hand, Java can interact with native modules through a custom interface that is efficient

[5] This is best exemplified by the ongoing lawsuit between Microsoft Corporation and Sun Microsystems regarding Microsoft's Java-native interface technologies, namely the Raw Native Interface (RNI) and J/Direct. On November 17, 1998, the district court ordered, among other things, that Microsoft implement Sun's JNI in *jview*, its publicly available JVM [Mic98].

but not portable across different Java platforms, as in the case of the Marmot [FKR+99] and J/Direct [Jd97]. On the other hand, this interaction can take place through a well-defined, standardized interface such as the Java Native Interface (JNI), which sacrifices performance for native code portability. The second part of this chapter takes a deep look at these three points in the design spectrum, and provides a qualitative evaluation of the tradeoffs through micro-benchmarks. Not surprisingly, the costs of copying data between the Java and native heaps are a significant factor across the efficiency/portability spectrum. This suggests that the heap separation imposed by garbage collection is an inherent performance bottleneck.

The last part of this chapter presents Javia-I [CvE99b], a Java interface to the VI Architecture that respects the hard separation between Java and native heaps. Javia-I provides a front-end API to Java programmers that closely resembles the one proposed by the VI architecture specification and manipulates key data structures (user-level buffers and VI control structures) in native code. The performance impact of Java/native crossings in Javia-I is studied in the context of Marmot and JNI. Even in the case where the native interface is highly tuned for performance, results show that the overheads incurred by the hard heap separation still manifests itself in the overall performance of Javia-I.

2.1 Background

2.1.1 The Virtual Interface Architecture

The Virtual Interface (VI) architecture defines a standard interface between the network interface hardware and applications. The specification of the VI architecture is a joint effort between Microsoft, Intel and Compaq, and encompasses ideas that have appeared in various prototype implementations of user-level network interfaces [vEBB+95, PLC95, DBC+98, CMC98]. The target application space is cluster computing in system-area networks (SAN).

The VI architecture is connection-oriented. To access the network, an application opens a *virtual interface* (VI), which forms the endpoint of the connection to a remote VI. In each VI, the main data structures are user-level buffers, their corresponding descriptors, and a pair of message queues (Figure 2.1). User-level buffers are located in the application's virtual memory space and used to compose messages. Descriptors store information about the message, such as its base virtual address and length, and can be linked to other descriptors to form composite messages. The in-memory layout of the descriptors is completely exposed to the application. Each VI has two associated queues—a send queue and a receive queue—that are thread-safe. The imple-

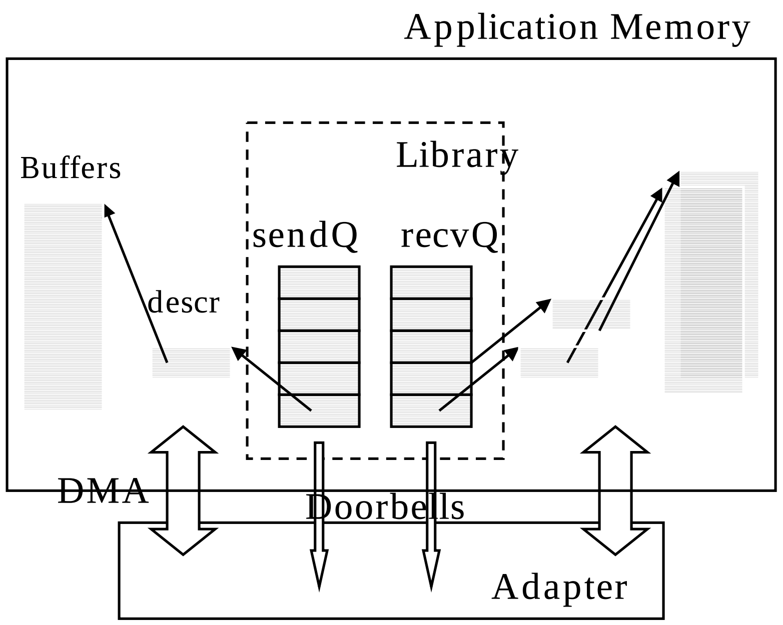


Figure 2.1 Virtual Interface data structures. Shaded structures must be pinned onto the physical memory so they can be accessed by DMA engines.

mentation of enqueue and dequeue operations is not exposed to the application, and thus must take place through API calls.

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. An application is permitted to poll at multiple receive queues at a time using VI completion queues. Apart from polling, the architecture also supports for interrupt-driven reception by posting notification handlers on completion queues.

The VI architecture also provides support for remote, direct memory access (RDMA) operations. A RDMA send descriptor specifies a virtual address at the remote end to which data will be written (RDMA-write) or from which data will be read (RDMA-read). Completion of RDMA-read operations may not respect the FIFO order imposed by the send queue. In addition, RDMA-reads do not consume receive descriptors at the remote end while RDMA-writes can if explicitly asked to. RDMA requires that the sender and receive exchange information about the virtual address prior to communication.

2.1.2 Giganet cLAN™ GNN-1000 Cluster

The network interface used throughout this thesis is the commercially available cLAN™ GNN-1000 adapter from Giganet [Gig98] for Windows2000™ beta 3. The GNN-1000 can have up to 1024 virtual interfaces opened at a given time and a maximum of 1023 descriptors per send/receive queue. The virtual/physical translation table can hold over 229,000 entries. The maximum amount of pinned memory at any given time is over 930MBytes. The maximum transfer unit is 64Kbytes. The GNN-1000 does not support interrupt-driven message reception.

The cluster used consists of eight 450Mhz Pentium-II™ PCs with 128MBytes of RAM, 512KBytes second level cache (data and instruction) and running Windows2000 beta 3. A Giganet GNX-5000 (version A) switch connects all the nodes in a star-like formation. The network has a bi-directional bandwidth of 1.25 Gbps and interfaces with the nodes through the GNN-1000 adapter. Basic end-to-end round-trip latency is around 14μs (16μs without the switch) and the effective bandwidth is 85MBytes/s (100MBytes/s without the switch) for 4KByte messages.

2.1.3 Explicit Buffer Mapping: A Case for Buffer Re-Use

An essential advance made by modern network interfaces is *zero-copy* communication: network DMA engines read from and write into user buffers and descriptors without host processor intervention. Zero-copy requires that:

1. pages on which buffers and descriptors reside must be physically resident (e.g. pinned onto physical memory) during communication, and

2. the virtual to physical address mappings must be known to the network interface (pointers are specified as virtual addresses but the DMA engines must use physical address to access main memory).

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

The approaches pursued by several network-interface designs have generally fallen into two categories: implicit and explicit memory mapping. In implicit memory mapping, these two operations are performed automatically by the network-interface without application intervention. In systems such as StarT [ACR+96], FLASH [KOH+94], and Typhoon [RLW94], the network interface is attached to the memory bus and shares the *translation look-aside buffer* (TLB) with the host processor. The aggressive approach pursued in these systems suffers from poor cost-effectiveness since it requires special hardware support.

The Meiko CS-2 [HM93a] multi-computer incorporates a less aggressive design: the TLB is implemented on the adapter's on-board processor and coordinates with the host operating system (SunOS) running on SuperSparc processors. U-Net/MM [WBvE97] generalizes this approach for off-the-shelf operating systems and networks. The TLB is also integrated into the network interface and can be implemented entirely in the kernel, as in the DC21140/NT implementation, or partly in the network adapter, as in the PCA200/Linux implementation). Keeping the network-interface TLB consistent with that of the OS is no simple task. During a TLB miss, the network interface interrupts the host processor, which pins or pages-in a virtual page. Pinning pages in an interrupt context is non-standard and complicated. For example, the Linux implementation of U-Net/MM provides custom pinning/unpinning routines because the standard kernel ones cannot be used in the context of an interrupt handler. U-Net/MM also has to implement a page-in thread for pages that are temporarily swapped out to the disk. Another drawback is that the OS intervention during a page miss can be costly: around 50µs on 133Mhz Pentium with 33Mhz PCI bus if the page is present in the host TLB, and as high as 20ms if the page is not. This overhead is so critical that U-Net/MM can choose to drop the message if the page is not present. Furthermore, the user is unable to take advantage of application-specific optimization to "keep" the pages mapped in since it has no control over the paging behavior of the host machine.

The VI architecture adopts an explicit memory mapping approach that was first pursued by the Hamlyn [BJM+96] project and later by the Shrimp/VMMC [DBC+98] project. Applications are responsible for "registering" (`VipRegisterMemory`) and "de-registering" (`VipDeregisterMemory`) memory regions (in which user buffers and descriptors reside) with the VI architecture. The registration is initiated by the user and 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. De-registration undoes the above process: buffer and descriptor pages can be paged out and the virtual/physical mapping is dropped by the VI architecture.

By pushing the burden of pinning and unpinning buffers to applications, explicit buffer mapping greatly simplifies the design of network interfaces. Most importantly, buffer re-use based on application-specific information amortizes the costs of mapping (unmapping) memory onto (from) physical memory, which essentially eliminates the OS from the critical path. These costs can be high[6]: for example, Giganet's implementations of `VipRegisterMemory` and `VipDeregisterMemory` for Windows2000 beta3 have a combined cost of 20μs (i.e. over 10,000 machine cycles) on a 450Mhz Pentium-II. For comparison, the basic communication overheads are typically less than 1,000 machine cycles on the same platform.

A drawback of explicit buffer mapping is that system scalability is limited by the size of the translation table in the network interface, which in turn may depend on the host operating system.

Unfortunately, requiring applications to manage buffers in this manner is ill matched to the foundations of a Java.

2.1.4 Java: A Safe Language

While user-level network interfaces are revolutionizing the networking architecture on PCs, building portable, robust, and high-performance cluster applications remains a daunting task. Programmers are increasingly relying on language support in order to make this task easier. Modern object-oriented programming languages such as Java [AG97] provide a high degree of port-

[6] At the time of this writing, the performance numbers of Berkeley's VIA implementation [BGC98] (for Myricom's Myrinet M2F [BCF+95] with LANai 4.x-based adapters on a 167Mhz SunUltra1 running Solaris 2.6) of VipRegisterMemory and VipDeregisterMemory were not available [Buo99].

ability, strong support for concurrent and distributed programming, and a safe programming environment:

- Portability is achieved by compiling the Java source program into an intermediate byte-code representation that can run on any Java Virtual Machine (JVM) platform.
- For multi-processor shared memory machines, Java offers standard multi-threading. For distributed machines, Java supports Remote Method Invocation (RMI), which is an object-oriented version of traditional remote procedure calls.
- By a safe programming environment we mean one that is *storage* and *type safe*.

Storage safety in Java, enforced by a garbage collector, guarantees that no storage will be prematurely disposed of whether at the explicit request of the programmer or implicitly by the virtual machine. This spares the programmer from having to track and de-allocate objects. However, the programmer has no control over object placement and little control over object de-allocation and lifetime. For example, consider the following Java code:

```
1  class Buffer {
2    byte[] data;
3    Buffer (int n) { data = new byte[n]; }
4  }
5  Buffer b = new Buffer(1024); /* allocation */
6  b = null; /* dropping the reference */
```

A `Buffer` is defined as a Java object with a pointer to a Java byte array. After allocation (line 5), the programmer knows that buffer and the byte array is in the garbage-collected heap, but cannot pinpoint their exact location because the garbage collector can move them around the heap[7]. By dropping the

[7] This is not the case if a non-copying garbage collector is used. However, Java programmers can make no assumptions about the garbage collection scheme of the underlying JVM.

reference to the buffer (line 6), the programmer only makes the buffer and the byte array *eligible* for garbage collection but does not actually free any storage.

Type safety ensures that references to Java objects cannot be forged, so that a program can access an object only as specified by the object's type and only if a reference to that object is explicitly obtained. Type safety is enforced by a combination of compile-time and runtime checks. For example, data stored in a `Buffer` can only be read as a byte array—accessing the data as any other type will require copying it. In order to access `data` as a double array, the following code allocates a new double array and copies the contents of `data` into it:

```
double[] data_copy = new double[1024/8];
for (int i=0,off=0;i<1024/8;i++,off+=8) {
  int upper = (((data[off]&0xff)<<24)+
               ((data[off+1]&0xff)<<16)+
               ((data[off+2]&0xff)<<8)+
                (data[off+3]&0xff));
  int lower = (((data[off+4]&0xff)<<24)+
               ((data[off+5]&0xff)<<16)+
               ((data[off+6]&0xff)<<8)+
                (data[off+7]&0xff));
  /* native call to transform a 64-bit long into a double */
  data_copy[i] = Double.toLongBits(((long)upper)<<32)+(lower&0xffffffffL))
}
```

In addition, the runtime representation of Java objects must include meta-data such as the method dispatch table and the object type. The latter is used by the Java system to perform runtime safety checks (such as array-bounds, array-stores, null pointer and down-casting checks) and to support the reflection API. The result is that object representations are sophisticated (Figure 2.2 shows a typical representation of a `Buffer`), implementation-dependent, and hidden from programmers, all of which make object serialization expensive. An evaluation of the serialization costs on several Java systems is presented in Chapter 4.

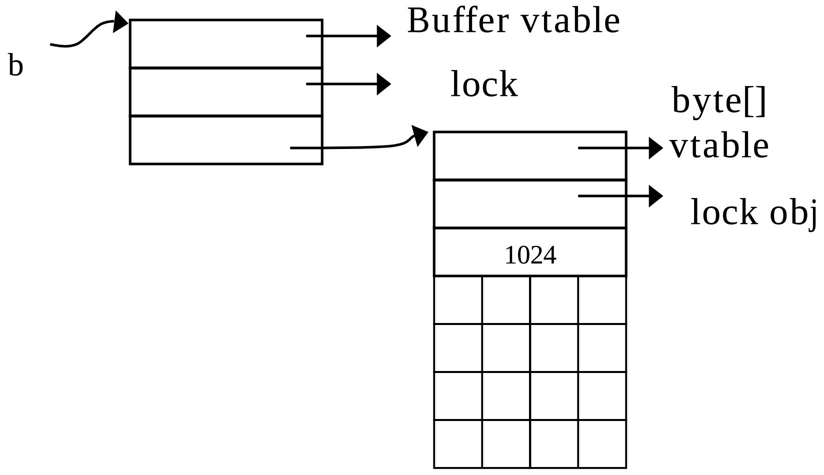
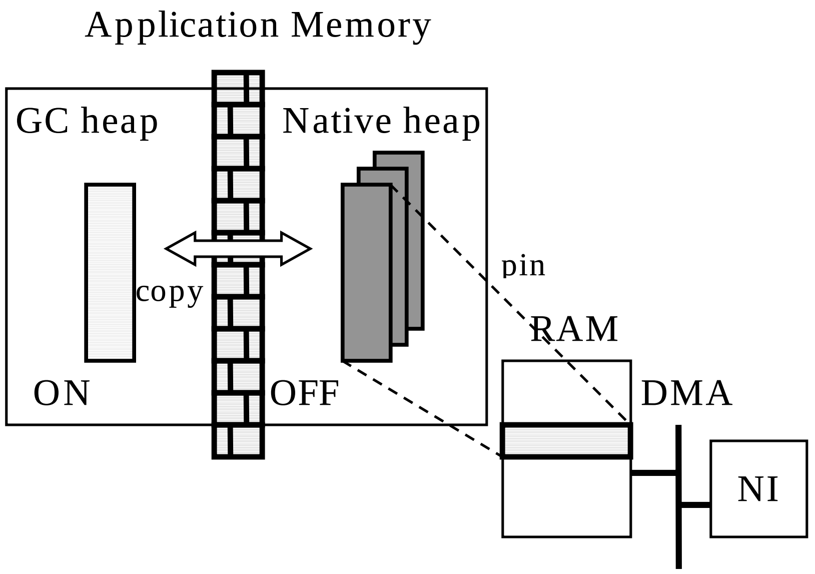


Figure 2.2 Typical in-memory representation of a Buffer object. Each Buffer object has two words of meta-data: one for the method dispatch table (from which the class object can be reached), and another for the monitor object. An array object also keeps the length of the array for runtime checks.

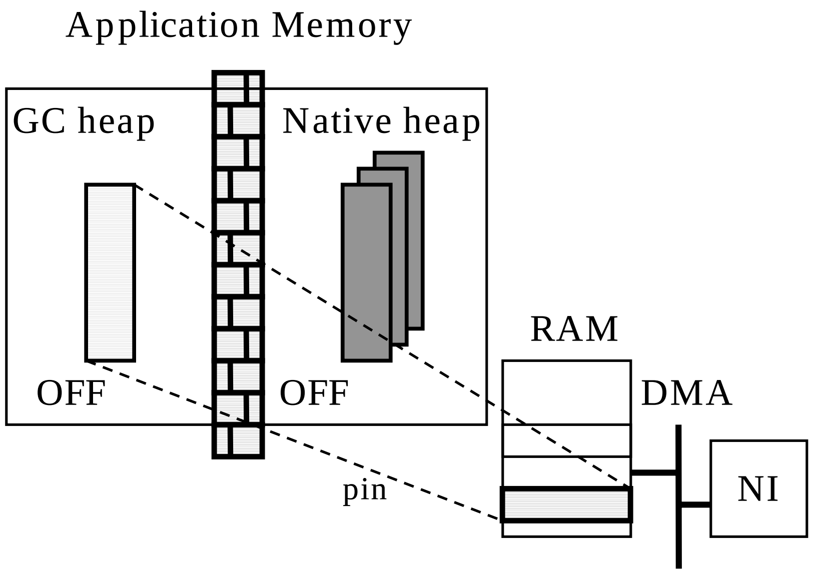
2.1.5 Separation between Garbage-Collected and Native Heaps

Because of the safety features in Java, programmers are forced to rely on native code (e.g. C) to access hardware resources and legacy libraries[8]. Figure 2.3 depicts the separation between Java's garbage-collected heap and the native, non-collected memory region in which DMA buffers must normally be allocated. Data has to be copied on demand between Java arrays and buffers that are pinned onto the physical memory so they can be directly accessed by the DMA engine (shown in diagram (a)). The garbage collector remains enabled except for the duration of the copy.

[8] The same applies to other safe languages such as ML [Hue96].



(a) Hard Separation: Copy-on-demand



(b) Optimization: Pin-on-demand

Figure 2.3 The hard separation between GC and native heaps. (a) Data has to be copied on demand from Java arrays into pinned native buffers so they can be accessed directly by the DMA engine. (b) GC can be disabled for a "short" time so Java arrays can be pinned and made visible to the DMA on demand. This optimization does not work well for receive operations because the GC has to be disabled indefinitely.

The pin-on-demand optimization (shown in diagram (b)) avoids the copying by pinning the Java array on the fly. To this end, the garbage collector must be disabled until DMA accesses are completed. This optimization, however, does not work well for receive operations: the garbage collector has to be disabled indefinitely, which is unacceptable.

2.2 Java-Native Interfaces

The Java language specification allows Java and C applications to interact with one another through a combination of language and runtime support[9]. Java programs can transfer control to native libraries by invoking Java methods that have been annotated with the `native` keyword. C programs not only can transfer control to Java programs but also obtain information about Java classes and objects at runtime via a *Java-native interface*.

Java-native interfaces have to cope with the separation between Java's and C's heap. How should Java objects be passed into C during a native method invocation? If they are passed by reference, how can the garbage collector track them? How should Java objects be accessed in C? The central theme behind the answers to these questions is the trade-off between efficiency and portability of Java applications that rely on native code. The following subsections look at three data points in the design space of these interfaces: two approaches, Marmot and J/Direct, emphasize performance whereas a third, JNI, is geared towards portability.

[9] Ideally, access to native code from Java should be prohibited: it defeats the purpose of a safe language. Once in native code, all the safety properties can be violated. The bottom-up approach pursued in this thesis seeks to minimize the amount of native code in Java communication software.

2.2.1 The Marmot System

Marmot [FKR+99] is a research Java system developed at Microsoft. It consists of a static, optimizing, byte-code to x86 compiler, and a runtime system with three different types of garbage collection schemes: a conservative mark-sweep collector, a semi-space and a two-generations copying collector[10].

Marmot's interaction with native code is very efficient. It translates Java classes and methods into their C++ counterparts and uses the same alignment and the "fast-call" calling convention as native x86 C++ compilers. C++ class declarations corresponding to Java classes that have native methods must be manually generated. All native methods are implemented in C++, and Java objects are passed by reference to native code, where they can be accessed as C++ structures.

Garbage collection is automatically disabled when any thread is running in native, but can be explicitly enabled by the native code. In case the native code must block, it can stash up to two (32-bit) Java references into the thread object so they can be tracked by the garbage collector. During Java-native crossings, Marmot marks the stack so the copying garbage collector knows where the native stack starts and ends.

Despite its efficiency, it is not surprising that native code written for Marmot is not compatible with other JVMs. JVM implementations differ in object layouts, calling conventions, and garbage collection schemes. For Java practitioners, the lack of native code portability severely compromises Java's future as a "write once, run everywhere" language.

[10] The generational collector is not available in the Marmot distribution used in this thesis.

2.2.2 J/Direct

J/Direct is a Java/native interface developed by Microsoft [Jd97] and is currently deployed in their latest JVM. The main motivation is to allow Java programs to interface directly with legacy C libraries, such as the Win32 API. J/Direct shields the garbage-collected heap (which is always enabled) from the native heap by automatically marshaling Java primitive-types and primitive-typed arrays into "equivalent" C data structures during a native method invocation. Arbitrary Java objects, however, are not allowed as arguments[11].

J/Direct requires special compiler support to generate the marshaling code. Native methods declarations are preceded with annotations in the form of a comment; these annotations are recognized by Microsoft's Java-to-byte-code compiler (*jvc*), which in turn propagates the annotations through unused byte-code attributes. The just-in-time compiler in *jview* generates the marshaling stubs based on the byte-code annotations. Since a program with J/Direct annotations is a valid Java program, it can be compiled and executed unmodified by a Java compiler or JVM from a different vender as long as the C library conforms to the native interface specification of that JVM.

2.2.3 Java Native Interface (JNI)

The definition of the JNI [Jni97] is a result of an effort by the Java community to standardize Java native interfacing. JNI has become widely accepted and has been deployed in several publicly available JVMs (e.g. JDK1.2 and Kaffe OpenVM [Kaf97]). JNI hides JVM implementation details from native code in four ways:

---

[11] Specially annotated, "shallow" (i.e. no pointer fields) Java objects can be passed as arguments (see Section 3.6.1).

1. By providing opaque references to Java objects, thereby hiding object implementation from native code;
2. By placing a function table between the JVM and native code and requiring all access to JVM data to occur through these functions;
3. By defining a set of native types to provide uniform mapping of Java types into platform-specific types; and
4. By providing flexibility to the JVM vendor as to how object memory is handled in cases where the user expects contiguous memory. JNI calls that return character string or scalar array data may lock that memory so that it is not moved by the memory management system during its use by native code

The JNI specification contains API calls for invoking Java methods, creating Java objects, accessing class and object variables, and catching and throwing exceptions in native code. In the presence of dynamic class loading, the API implementation must abide by the standard “class visibility” rules.

2.2.4 Performance Comparison

This section compares the performance of Marmot’s custom Java / native interface, J / Direct on Microsoft’s *jview* (build 3167), and two JNI implementations (Sun’s JDK 1.2 and *jview*). Table 2.1 summarizes the differences between Marmot, J / Direct, and JNI.

Table 2.1 Marmot, J / Direct, and JNI’s GC-related features

	GC during native code (default)	*GC Enable and Disable*	*Data Copy*	*Pin a Java object in C*
Marmot	Off	Yes	Manual	Yes
J/Direct	On	No	Automatic	No
JNI	On	Yes	Automatic	Yes

Table 2.2 Cost of Java-to-C downcalls

Java/Native call	*Marmot*		*J/D (jview3167)*		*JNI (jdk1.2)*		*JNI (jview3167)*	
(in us)	*virtual*	*static*	*virtual*	*static*	*virtual*	*static*	*virtual*	*static*
null	0.252	0.250	N/A	4.065	0.118	0.118	3.993	4.171
one int	0.254	0.252	N/A	4.336	0.124	0.126	4.132	4.364
two ints	0.260	0.254	N/A	4.386	0.214	0.407	4.246	4.520
three ints	0.260	0.258	N/A	4.476	0.214	0.443	4.282	4.648
one object	0.258	0.256	N/A	5.187	0.132	0.132	4.730	5.211

Table 2.3 Cost of C-to-Java upcalls

Native/Java call	*Marmot*		*J/D (jview3167)*		*JNI (jdk1.2)*		*JNI (jview3167)*	
(in us)	*virtual*	*static*	*virtual*	*static*	*virtual*	*static*	*virtual*	*static*
null	0.276	0.272	N/A	N/A	2.507	2.577	14.042	13.172
one int	0.280	0.280	N/A	N/A	2.898	2.667	13.802	13.483
two ints	0.284	0.274	N/A	N/A	2.662	2.477	14.359	14.257

Table 2.2 and 2.3 show the basic costs of transferring control from Java to C (*downcalls*) and from C to Java (*upcalls*) respectively. The cost of a downcall in J/Direct and JNI on *jview* is surprisingly high. JNI on JDK1.2 is roughly 50% faster than Marmot—Marmot spends extra cycles checking call stack alignment and marking the Java stack for GC purposes. On the other hand, upcalls in JNI over JDK1.2 are about 10x slower than in Marmot because of function calls to obtain the class of the object (`GetObjectClass`), to obtain the method identifier (`GetMethodID`), and to perform the call (`CallMethod`).

Table 2.4 Cost of accessing Java fields from C

Object Access (in us)	*Marmot*	*J/D (jview3167)*	*JNI (jdk1.2)*	*JNI (jview3167)*
read int field	0.012	N/A	1.215	2.335
write int field	0.018	N/A	1.272	2.463
read obj field	0.018	N/A	1.724	2.473

Table 2.4 shows that accessing fields of an object is expensive in JNI because of function calls to obtain the field identifier (`GetFieldID`) and to access the field *per se* (`GetIntField`, `SetIntField`, etc). In Marmot, these

costs are reduced by nearly 100-fold: roughly 5 to 8 machine cycles on a 450Mhz Pentium-II processor.

Table 2.5 Cost of crossing the GC / Native separation by copy and pin-on-demand

(in us)	*Marmot*	*J/D (jview3167)*	*JNI (jdk1.2)*	*JNI (jview3167)*
pin/unpin	0.000	N/A	0.619	broken
copy 10 bytes	3.649	3.024	3.883	6.829
copy 100 bytes	4.068	3.589	4.072	7.300
copy 1000 bytes	5.742	9.297	5.858	13.269

Table 2.5 shows the costs of crossing the heap separating using copy and pin-on-demand. JNI lets native code obtain a direct pointer to the array elements as long as pinned arrays are supported by the JVM. This pointer is only valid in the critical section delimited by explicit calls to `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical`. Since garbage collection is disabled in the critical section, the understanding is that one should not run "for too long" or block the thread while running in the critical section. In JDK1.2, it costs about 0.6μs to enter and exit the critical section. (At the time of this writing, implementation of these two JNI calls on *jview* is broken.) Since garbage collection in Marmot is automatically disabled in native code, arrays are automatically pinned. J/Direct does not support pinned Java arrays, although it allows programs to access C arrays from Java.

If pinned arrays are not supported, then native code can only access a copy of the Java array. The copying costs are roughly the same in both Marmot and JNI, with JDK1.2 outperforming *jview* by nearly a factor of two.

2.2.5 Summary

Java is not suitable for writing programs that interact directly with the underlying hardware. The primary reason is the strong typing and garbage collection, which gives programmers no control over objects' lifetime, location, and layout. Java programs can, however, call native libraries (which in turn interface to the devices) through well-defined Java / native interfaces. Tailoring the implementation of these interfaces to a particular JVM leads to good performance, as seen in Marmot, but sacrifices the portability of the native code.

The copying overheads incurred by the hard separation between Java and native heaps is fundamental to the language—Java programs are garbage collected and the scheme used is shielded from programmers—and are therefore orthogonal to portability. The following section studies the impact of this separation on the performance of Java communication over the VI architecture.

2.3 Javia-I : Interfacing Java to the VI Architecture

2.3.1 Basic Architecture

The general Javia-I 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-I classes are shown below:

```
public class Vi { /* connection to a remote VI */

  public Vi(ViAddress mach, ViAttributes attr) { … }

  /* async send */
  public void sendPost(ViBATicket t);
  public ViBATicket sendWait(int millisecs);

  /* async recv */
  public void recvPost(ViBATicket t);
11    public ViBATicket recvWait(int millisecs);
12
13    /* sync send */
14    public void send(byte[] b,int len,int off,int tag);
15
16    /* sync recv */
17    public ViBATicket recv(int millisecs);
18  }
19
20  public class ViBATicket {
21    private byte[] data; private int len, off, tag;
22    private boolean status;
23    /* public methods to access fields ommited */
24  }
```

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.

Javia-I contains methods to send and receive Java byte arrays[12]. The asynchronous calls (lines 6-11) use a Java-level descriptor (`ViBATicket`, lines 20-24) 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 2.4 shows the Java and native data structures involved during asynchronous sends and receives. Buffers and descriptors are managed (pre-allocated and pre-pinned) in native code and a pair of send and receive ticket rings is maintained in Java and used to mirror the VI queues.

To post a Java byte array transmission, 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. `sendWait` polls the queue and updates the ring upon completion. 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

[12] The complete Javia-I interface provides send and receive calls for all primitive-typed arrays.

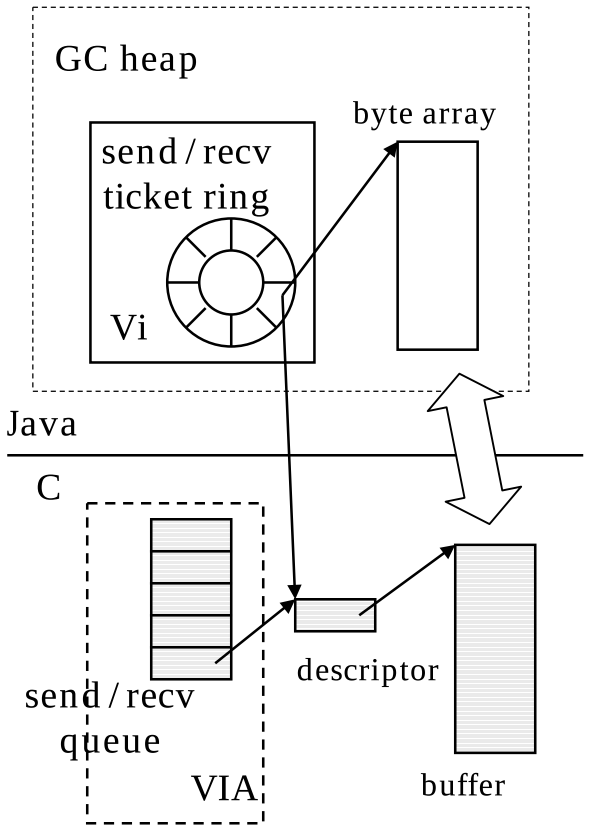


Figure 2.4 Javia-I per-endpoint data structures. Solid arrow indicates data copying.

buffer into the byte array. This requires two *additional* Java / native crossings: upon message arrival, an upcall is made in order to dequeue the ticket from the ring, followed by a downcall to perform the actual copying. Synchronized accesses to the ticket rings and data copying are the main overheads in the send / receive critical path.

Javia-I provides a blocking send call (line 14) 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 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[13].

The blocking receive call (line 17) polls the reception queue for a message, allocates a ticket and byte array of the right size on-the-fly, copies data into it, and returns a ticket. Blocking receive not only eliminates the need for a ticket ring, it also fits more naturally into the Java coding style. However, it requires an allocation for every message received, which may cause garbage collection to be triggered more frequently.

Pinning the byte array for reception is unacceptable because it would require the garbage collector to be disabled indefinitely.

2.3.2 Example: Ping-Pong

The following is a segment of a simplified ping-pong program using Javia-I with asynchronous receives:

```
byte[] b = new byte[1024];
/* initialize b… */
/* create and post receive ticket */
ViBATicket t = new ViBATicket(b, 0);
vi.recvPost(t);
if (ping) {
  vi.send(b, 0, 1024);
  t = vi.recvWait(Vi.INFINITE);
  b = t.getByteArray();
  /* read b… */
  /* done */
} else { /* pong */
  t = vi.recvWait(Vi.INFINITE);
  b = t.getByteArray();
  /* read b… */
  /* send reply */
  vi.send(b, 0, b.length);
  /* done */
}
```

[13] The garbage collector must be disabled during the operation.

2.3.3 Implementation Status

Javia-I consists of 1960 lines of Java and 2800 lines of C++. The C++ code performs native buffer and descriptor management and provides wrapper calls to Giganet's implementation of the VI library. A significant fraction of that code is attributed to JNI support.

Most of the VI architecture is implemented, including query functions and completion queues. Unimplemented functionality includes interrupt-driven message reception: the commercial network adapter used in the implementation does not currently support the notification API in the VI architecture. This is not a prime concern in this thesis: software interrupts are typically expensive (one order of magnitude higher than send / receive overheads) and depend heavily on the machine load and on the host operating system.

2.3.4 Performance

The round-trip latency achieved between two cluster nodes (450Mhz Pentium-II boxes) is measured by a simple ping-pong benchmark that sends a byte array of size *N* back and forth. The effective bandwidth is measured by transferring 15MBytes of data using various packet sizes as fast as possible from one node to another. A simple window-based, pipelined flow control scheme [CCH+96] is used. Both benchmarks compare four different `Vi` configurations,

1. Send-copy with non-blocking receive (*copy*),
2. Send-copy with blocking receive (*copy+alloc*),
3. Send-pin with non-blocking receive (*pin*), and
4. Send-pin with blocking receive (*pin+alloc*),

with a corresponding C version that uses Giganet's VI library directly (*raw*). Figures 2.5 and 2.6 show the round-trip and the bandwidth plots respectively, and Table 2.6 shows the 4-byte latencies and the per-byte costs. Numbers have been taken on both Marmot and JDK1.2 / JNI (only *copy* and *copy+alloc* are reported here). JDK numbers are annotated with the *jdk* label.

Pin's 4-byte latency includes the pinning and unpinning costs (around 20μs) and has a per-byte cost that is closest to that of raw (the difference is due to the fact that 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 allocation and garbage collection overheads. The additional Java / native crossings take a toll in *JDK copy*: each downcall not only includes the overhead of a native method invocation in JNI, but also a series of calls to perform read / write operations to Java object fields. Although *JDK copy+alloc* is able to bypass the ring, the per-byte cost appears to be significantly higher, most likely due to garbage collections caused by excessive allocations during benchmark executions.

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

Table 2.6 Javia-I 4-byte round-trip latencies and per-byte overhead

	4-byte(us)	*per-byte(ns)*
raw	16.5	25
pin	38.0	38
copy	21.5	42
JDK copy	74.5	48
copy+alloc	18.0	55
JDK copy+alloc	38.8	76

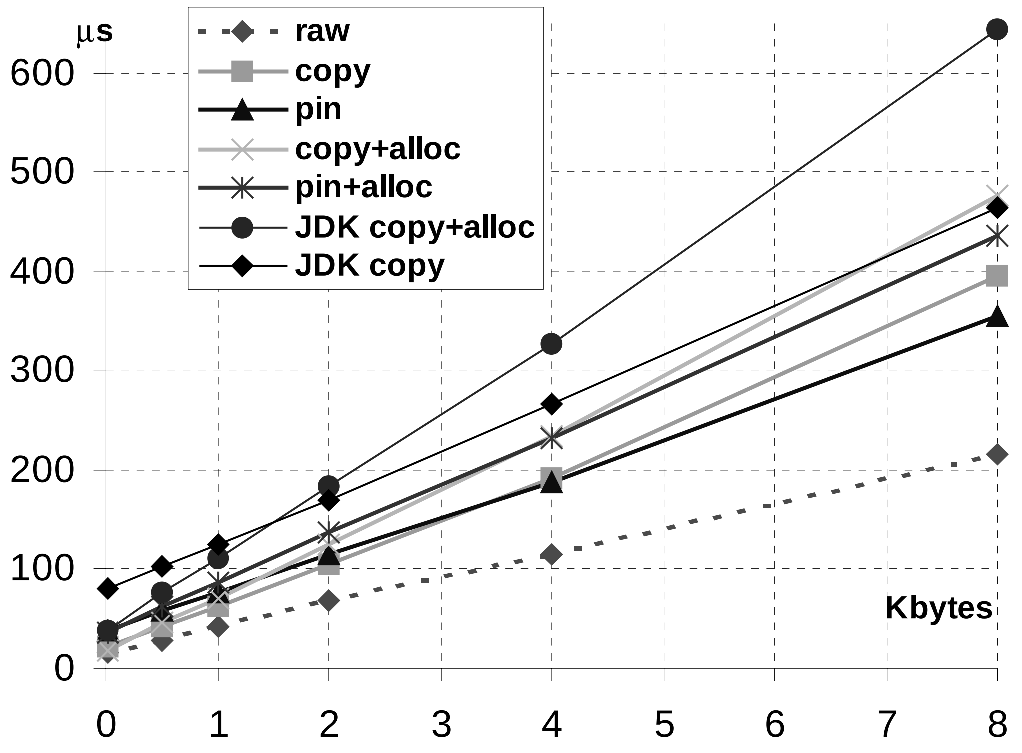


Figure 2.5 Javia-I round-trip latencies

bandwidth (within 70-75% of *raw*) that is higher than that of *pin* for messages smaller than 6Kbytes. *JDK copy* peaks at around 65% of raw.

2.4 Summary

Javia-I provides a simple interface to the VI architecture. It respects the heap separation by hiding all the VI architecture data structures in native code and copying data between buffers and Java arrays. By exploiting the blocking semantics of send, the *pin* variant replaces the copy costs on the sending side with those of pinning and unpinning an array. While C applications can amortize the high (one-time) cost of pinning by re-using buffers, Java programmers cannot because of the lack of explicit control over object location and lifetime.

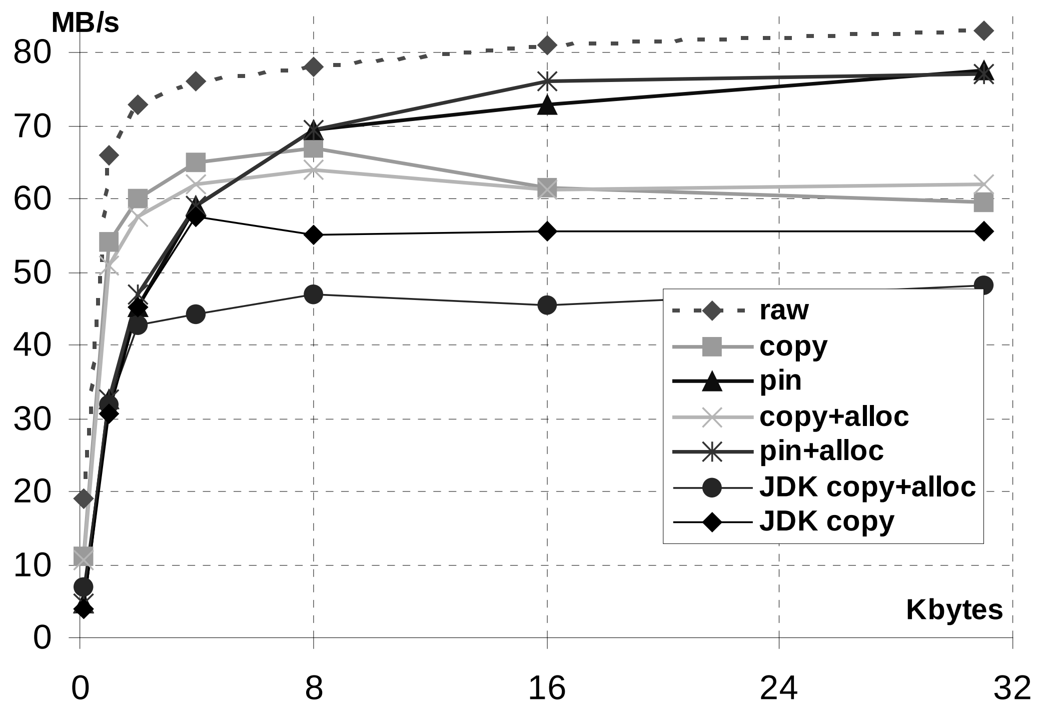


Figure 2.6 Javia-I effective bandwidth

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 JNI. Even in the scenario where the native interface is efficient, as in Marmot, the hard separation between Java's garbage collected heap and native heap forces Javia-I to copy data or pin arrays on demand.

2.5 Related Work

The inefficiencies that arise during Java-native interfacing are well known. Microsoft [Mic99] provides custom native interfaces: the Raw Native Interface for enhanced performance, and J/Direct for convenience. The measured performance of J/Direct is far from impressive; as discussed in the next chapter, Jaguar [WC99] improves on J/Direct by providing more flexibility and better performance. Javasoft [Jav99] has continuously improved its JNI implementation and has shown that JNI can be implemented efficiently.

The separation between garbage-collected and native heaps is applicable to other safe languages as well. Huelsbergen [Hue96] presents a portable C interface for Standard ML/NL. An ML program uses user-supplied data types to register a C function with the interface and to build specifications of corresponding C data structures. The interface runtime system performs automatic marshaling of data: it allocates storage for C function arguments and copies data into the allocated storage during a native function call. In order to cope with different data representations and garbage-collection schemes, the interface does not consider pinning ML data structures.

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, implementations of the `java.net` package in most JVMs are typically layered on top of the sockets (TCP/IP) API. Central Data [Cd99] offers native implementations of the `portio` package for accessing serial and parallel ports from Java. [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 [Fer98] presents a simple Java front-end to PVM. All these approaches respect the heap separation and do not address the performance penalty incurred during Java/native interactions.