Javia: A Java Interface to the Virtual Interface Architecture

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

{chichao,tve}@cs.cornell.edu

Abstract

The Virtual Interface (VI) architecture has become the industry standard for user-level network interfaces. This paper presents the implementation and evaluation of Javia, a Java interface to the VI architecture. Javia explores two points in the design space. The first approach manages buffers in C and requires data copies between the Java heap and native buffers. The second approach relies on a Java-level buffer abstraction that eliminates the copies in the first approach. Javia achieves an effective bandwidth of 80Mbytes/s for 8Kbyte messages, which is within 1% of those achieved by C programs. Performance evaluations of parallel matrix multiplication and of the Active Messages communication protocol show that Javia can serve as an efficient building block for Java cluster applications.

Keywords Java, Virtual Interface Architecture, user-level network interfaces, high-performance communication, memory management.

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 (VI) architecture [Via97]. At this point, commercial hardware that implements the VI architecture is available for Windows NT/2000. Several studies demonstrate the architecture’s potential for high performance [BGC98] as well as for supporting higher level communication abstractions and clustered applications [SPS98, SSP99].

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+99, MGG98] have delivered promising results, gradually reducing the performance gap between Java and C programs. Thus, providing access to the VI architecture 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 present a two-level Java interface to the VI architecture called Javia. The first level of Javia (Javia-I) manages the buffers used by the VI architecture 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.

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 the VI architecture and to transmit/receive Java arrays directly. Micro-benchmarks show that programs using Javia-II can achieve bandwidths of over 80Mbytes/s for large messages (> 8Kbytes), which are within 1% (error range) of those achieved by C programs.

Javia is intended as a building block for the construction of parallel Java applications as well as higher level communication libraries *entirely* in Java. Javia has been used for an implementation of parallel matrix multiplication (pMM) as well as an active messages library (Jam). Performance results of pMM on an 8-node PC cluster show that the overall application performance can improve with faster communication times. A point-to-point performance evaluation of Jam shows that high-level communication libraries can be implemented efficiently in Java.

Section 2 provides background on the VI architecture and on the experimental setup used in the paper. Sections 3 and 4 describe the Javia-I and Javia-II architectures, respectively. Section 5 presents the design and evaluation of pMM and Jam over Javia. Section 6 relates Javia to other efforts in improving Java's communication performance and section 7 concludes.

2 Background

2.1 Virtual Interface Architecture

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. 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 implementation 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 interrupt-driven reception by posting notification handlers on completion queues.

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 the VI architecture, 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 the VI architecture. 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 Experimental Setup

2.2.1 Giganet cLAN™ Cluster

The network interface used throughout this paper is the commercially available GNN-1000 from Giganet [Gig98] for Windows 2000 beta 3. The network adapter is accompanied by the following software: (i) a custom firmware that implements the packet multiplexing and address translation in hardware, (ii) a device driver that implements VI setup and tear-down, page pinning and unpinning routines that coordinate with the TLB in the adapter, among other things, (iii) and a user-level library (Win32 dll) that implements the VI Architecture API.

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 Pro 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 1.25Gbytes/s 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 82Mbytes/s (100Mbytes/s without the switch) for 4Kbyte messages.

2.2.2 Marmot

Marmot [FKR+99] is a Java system developed at Microsoft Research. It consists of a static, optimizing, byte-code to x86 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. Marmot does not rely on any external compiler or back-end and currently runs on P-II based PCs with Windows NT/2000. Java programs compiled by Marmot run roughly 1.5x to 5x faster than using Microsoft's JVM (build 3168). Because it relies solely on a static compiler, Marmot does not support dynamic loading of classes.

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: `java.lang`, `java.util`, `java.io`, and `java.awt`. Support for object serialization (in the `java.io` package) and reflection (`java.lang.reflect`) have also been added. Marmot is configured to use a semi-space copying collector based on the Cheney scanning algorithm. All objects are allocated in the garbage-collected heap.

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. A call of a null Java-to-native method costs about 0.3µs on a 450Mhz Pentium-II.

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.

3 Javia-I

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 VI architecture implementation through the native library. The core Javia-I classes are shown below:

```
1   public class Vi { /* connection to a remote VI */
2
3     public Vi(ViAddress mach, ViAttributes attr) { … }
4
5     /* async send */
6     public void sendPost(ViBATicket t);
7     public ViBATicket sendWait(int millisecs);
8
9     /* async recv */
10    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[1]. The asynchronous calls (lines 6-11) 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 1 shows the Java and native data structures involved during asynchronous send and receive. 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 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

---

[1] The complete Javia-I interface provides send and receive calls for all primitive-typed arrays (`double`, `float`, etc)

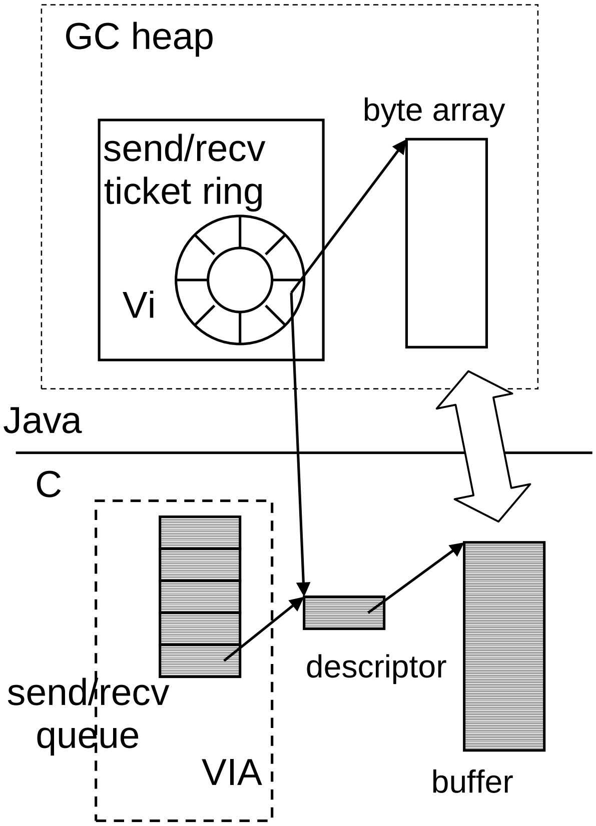

Figure 1. Javia-I per-endpoint architecture. Solid arrow indicates data copying.

(*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 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.

3.2 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] The garbage collector must be disabled during the operation.

	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

Table 1.

3.3 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 and 3 show the round-trip and the bandwidth plots respectively, and Table 1 shows the 4-byte latencies and the per-byte costs. Numbers have been taken on both Marmot and Sun's JVM running JDK1.2/JNI (only *copy* and *copy+alloc* are reported here). Sun's JVM numbers are annotated with the *jdk* label.