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/sec 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, parallel matrix multiplication, active 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, 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 80 Mbytes/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 for 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 Cluster

The network interface used throughout this dissertation is the commercially available GNN-1000 from Giganet [G98] 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 128MB 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 14us (16us without the switch) and the effective bandwidth is 82MB/s (100MB/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

3.1.1 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:

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

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 7-12) use a Java-level descriptor (`ViBATicket`) to hold a reference to the byte array being sent or received and other information such as the completion status, the transmission length, offset, and a 32-bit tag. Figure 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 15) because in virtually all cases the message is transmitted instantaneously—the extra completion check in an asynchronous send is more expensive than blocking in the native library. It also avoids accessing the ticket ring and enables two send variations. The first one

---

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

(*send-copy*) copies the data from the Java array to the buffer whereas the second (*send-pin*) pins the array on the fly, avoiding the copy[2].

The blocking receive call (line 18) polls the reception queue for a message, allocates a 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.

3.1.2 Implementation Status

Javia-I consists of 1960 lines of Java and 2800 lines of C++, and runs on Marmot and other publicly available JVMs such as JDK1.2 and Jview (build 3167). 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. Interrupt-driven message reception is not supported in Javia-I: the commercial network adapter used in the implementation does not currently support the notification API in the VI architecture.

3.1.3 Performance

The round-trip latency achieved between two cluster nodes (450Mhz Pentium-II boxes) is measured by a simple ping-pong benchmark sending an *N* byte message back and forth. The effective bandwidth is measured by transferring a total 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 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 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] The garbage collector must be disabled during the operation.

4 Javia-II

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

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

A buffer is abstracted by the `ViBuffer` class, as shown above. Allocating a `ViBuffer` with the static `alloc` method (line 3) causes Javia-II to allocate a buffer of the specified size *outside* the Java heap. The `register` method (line 7) pins the buffer to physical memory (so it can be used by the VI architecture), associates it with a VI, and obtains a descriptor to the buffer, which is represented by a `ViBufferTicket`. At that point, the buffer can be directly accessed by the DMA engine for communication. The buffer can be unregistered (line 8), which unpins it, and later re-registered with the same or a different VI. An application can access the buffer (e.g. perform read and write operations) as a Java primitive-typed array. For example, an invocation of `toByteArray` (line 11) returns a reference to a genuine Java byte array that is located in the buffer.

For transmission and reception of buffers, Javia-II provides only asynchronous primitives, as shown in lines 15-19. Javia-II differs from Javia-I in that the descriptors point directly to the Java-level buffers instead of native buffers (Figure 4). The application composes a message in the buffer (through array write operations) and enqueues the buffer for transmission using the `sendBufPost` method. `sendBufPost` is asynchronous and takes a `ViBufferTicket`, which is later used to signal completion. After the send completes, the application can compose a new message in the same buffer and enqueue it again for transmission. Reception is handled similarly—the application posts buffers for reception with `recvBufPost` and uses `recvBufWait` to retrieve received messages. For each message, it extracts the data through array read operations and can choose to post the buffer again.

An application can manifest its intention to re-use or de-allocate a buffer by invoking its `unRef` method (line 15). This call makes the buffer visible by the garbage collector, enabling it to track the array references into the buffer and to notify the application when such references no longer exist. At this point, an application can obtain new array references into the buffer, possibly of some other primitive type, or de-allocate the buffer.

Java's storage safety is preserved by ensuring that a `ViBuffer` will remain allocated as long as it is referenced as an array. For example, invocations of `free` result in a `ReferencedException` if an application holds one or more references into the buffer. Type safety is preserved by ensuring that an application will not obtain two differently typed array references into a single jbuf at any given time. For example, invocations of `toIntArray` will fail with a `TypedException` if `toByteArray` has been previously called on the same buffer. These properties are enforced through runtime checks. Accessing these buffers, however, require no runtime checks other than those already imposed by Java (e.g. array-bounds and null-pointer checks). A complete elaboration on the safety properties of `ViBuffers` and their integration into Marmot's copying collector can be found in [CvE99].

Javia-II adds about 100 lines of Java and 100 lines of C to Javia-I's implementation.

4.1.1 Performance

Table 1 and Figure 5 compare the round-trip latency obtained by Javia-II (*buffer*) with *raw* and two variants of Javia-I (*pin* and *copy*). The 4-byte round-trip latency of Javia-II is 20.5µs and the per-byte cost is 25ns, which is the same as that of *raw* because no data copying is performed in the critical path. The effective bandwidth achieved by Javia-II (Figure 6) is within 1 to 3% of that of *raw*, which is in the margin of error.

5 Applications

5.1 pMM: Parallel Matrix Multiplication

pMM represents a matrix as array of array of doubles and uses a parallel algorithm based on message passing and *block-gaxpy* operations [GvL89]. The core of the algorithm used in pMM is illustrated as follows, using Javia-I blocking receives:

```
int tau = myproc;
int stride = tau * r;
pvm.barrier(); /* global synchronization */
for (int k = 0; k < p; k++) {
  /* comm phase: send to right, recv from left using alloc receives */
  if (tau != myproc) {
  for (int j = 0; j < n/p; j++)
    rightVi.send(Aloc[j], 0, n, 0);
  for (int j = 0; j < n/p; j++) {
    do { Aloc[j] = leftVi.recvDoubleArray(0); } while (Aloc[j] == null);
  }
  /* computation phase */
  for (int j = 0; j < n/p; j++) {
    double[] c = Cloc[j];
    double[] b = Bloc[j];
    for (int i = 0; i < n; i++) {
      double sum = 0.0;
      for (int k = 0; k < n/p; k++) {
        double[] a = Aloc[k];
        sum += a[i] * b[stride+k];
      c[i] += sum;
    }
  }
  tau++;
```

```
  if (tau == p) tau = 0;
  stride = tau * r;
  pvm.barrier();
}
```

The computation kernel is a straightforward, triple-nested loop with three elementary optimizations: (i) one-dimensional indexing (columns are assigned to separate variables e.g. `c[i]` rather than `Cloc[j][k]`), (ii) scalar replacement (e.g the `sum` variable hoists the accesses to `c[i]` out of the innermost loop), and (iii) a 4-level loop unrolling (not shown above).

The single processor performance of the multiplication kernel compiled using Marmot is about 70Mflops for 64x64 matrices, and 45Mflops for 256x256 matrices (compared to JDK1.2's 55Mflops and Jview's 37Mflops for same size matrices) on a 450Mhz Pentium-II. In contrast, the performance of DGEMM found in Intel's Math Kernel library [Int99] is over 400Mflops for 64x64 matrices on the same machine. In Marmot, the cost of array-bounds checks account for 40 to 60% of the total execution time, whereas null-pointer checks account for less than 5%.

5.1.1 Cluster Performance

Figures 7 and 8 compare the absolute time spent in communication (in milliseconds) with input matrices of size 64x64 and 256x256, respectively, on 8 processors. Total communication time is obtained by commenting out the computation phase of pMM. The cost of barrier synchronization is factored out of the total communication time by skipping both communication and computation phases. *Javia-II*'s communication time is consistently smaller than the rest: with 256x256 matrices, where message payload is 2048 bytes, *Javia-II* spent 25% less time than *copy-async* in communication, as predicted by micro-benchmarks. For an input size of 64x64, the fraction of communication time in the total execution time is around 73% (median) for *jdk-copy-async*, with a high of near 85% for *pin-async* and a low of 56% for *Javia-II*. For 256x256, the median percentage is around 20%, with a low of 13% for *Javia-II*.

Figure 10 shows that the overall performance of pMM using 256x256 matrices correlates well with the communication performance seen in Figure 8. A peak performance of 320Mflops is attained by *Javia-II*, followed by 275Mflops attained by *copy-alloc* on 8 processors. *Javia-II* consistently outperforms the other versions on 2 and 4 processors as well (not shown). However, this "nice" correlation is not the case for 64x64 matrices, as shown in Figure 9: a peak performance of 175Mflops goes to *copy-async* on 4 processors. In fact, the overall performance of *Javia-II* is inferior to those of Javia-I on 2 processors (not shown). The erratic results seen are most likely due to obscure cache effects. This is a clear indication that, at this point, faster communication in Java does not necessarily lead to better overall performance: data locality is essential for numerically intensive applications.

Another interesting data point is that allocating an array on every message reception can actually improve locality. For example, although *copy/pin-alloc* spends about 15% more time than *copy/pin-async* in communication on 8 processors (Figure 8), *cop/piny-alloc*'s Mflops is as much as 10% higher than that of *copypin-async* (Figure 10).

5.2 Jam: Active Messages for Java

Jam is an implementation of AM-II, the second-generation active messages layer targeted for cluster computing [AC95].In Jam, endpoints are connected across the network by a pair of virtual interface connections: one for small messages and another for large messages. Each entry in the endpoint's translation table corresponds to one such pair. Endpoints need to be registered with the local name server in order for them to be visible to remote endpoints. The name server uses a simple naming convention: *<remote machine, endpoint name>*. A *map* call initiates the setup of a connection: the name of the remote endpoint is sent to the remote machine; the connection request is accepted only if the remote endpoint is registered. Jam provides reliable, ordered delivery of messages. While the interconnections between virtual interfaces and the back-end switch are highly reliable, a flow control mechanism (similar to the one in [CCH+96]) is still needed to avoid message losses due to receive queue overflows or send/receive mismatches.

A key design issue in Jam is how to provide an adequate bulk transfer interface to Java programmers. AM-II instead lets the sender specify an integer offset into a "virtual segment" supplied by the receiver: senders no longer have to deal with remote virtual addresses. This means that receivers in Java would have to operate on arrays using offsets (assuming no extra copying of data), which would be inconvenient at best.

Jam exploits two bulk transfer designs. The first design, which is based on Javia-I, does not require the receiver to supply a virtual segment—byte arrays are allocated upon message arrival and are passed directly to the handlers. Despite the allocation and copying overheads, this design works with any GC scheme and fits more naturally into the Java coding style.

The second design, which is based on Javia-II and calls for a copying collector, requires the receiver to supply a list of `ViBuffers` to an endpoint. The endpoint manages this list as a pool of receive buffers for bulk transfers and associates it to a separate virtual interface connection. Upon bulk data arrival, the dispatcher obtains a Java array reference from the receiving `ViBuffer` and passes that reference directly to the handler. The receiving `ViBuffer` is `unRef`ed after the handler's execution. When the pool is (about to be) empty, the dispatcher reclaims the buffers in the pool by triggering a garbage collection. Jam *knows* whether the underlying GC is a copying one after the first attempt to reclaim the buffers: if they are still in the referenced state, Jam dynamically switches back to the first design.

This design avoids copying data in the communication critical path and defers copying to GC time only if it is indeed necessary. For example, consider two types of active message handlers:

```
class First extends AM_Handler {
  private byte first;
  void handler(Token t, byte[] data, . . .) {
    first = data[0];
  }
}
class Enqueue extends AM_Handler {
  private Queue q;
  void handler(Token t, byte[] data, . . . ) {
    q.enq(data);
  }
}
```

The handler named `First` looks at the first element of `data` but does not keep a reference to it whereas the handler named `Enqueue` save the reference to data for later processing. A copying garbage collector will only have to copy `data` in the latter case.

5.2.1 Implementation Status

Jam consists of 1000 lines of Java code. A Jam endpoint is an abstract Java class that can be sub-classed for a particular transport layer. Jam currently has endpoint implementations for Javia-I and Javia-II. Jam implements all of AM-II short request (`AM_RequestM`) and reply (`AM_ReplyM`) calls, one bulk transfer call (`AM_RequestIM`), message polling (`AM_Poll`) and most of bundle and endpoint management functions. Unimplemented functionality includes asynchronous bulk transfers (`AM_RequestXferAsynchM`), moving endpoints across different bundles (`AM_MoveEndpoint`) and the message error model.

5.2.2 Performance

A simple ping-pong benchmark using `AM_Request0` and `AM_Reply0` shows a 0-byte round-trip latency of 31us, about 11 us higher than that of Javia-II (Figure 11). This value increases by about 0.5us for every four additional words. For large messages, Jam round-trip latency is within 25us of Javia-II and has the same per-byte cost. The main overheads in Jam's critical path are: (i) an extra pair of send/recv overheads (due to two separate VI connections: one for small messages, another for bulk transfers); (ii) synchronized access to bundle and endpoint structures; (iii) handler and translation table lookup, and (iv) protocol processing (header parsing and flow control).

Jam achieves an effective bandwidth of 75Mbytes/s, within 5% of Javia-II, as seen in Figure 12.

6 Related Work

One design alternative for Javia would have been to use the *JDirect* technology developed by Microsoft [JD97] and deployed in its latest JVM. JDirect enables a Java programmer to define pinned (non-collectable) objects using source-level, "comment-like" annotations that propagate via byte-code to the JIT compiler. Although a pinned object can be passed between Java and C without copying, in the current implementation a pinned object mirrors an actual Java object. The JIT compiler looks at the byte-code "annotations" and re-directs read/write operations to the pinned object. This re-direction incurs on level of indirection (i.e. looking up the actual location of the pinned object), which is prohibitively expensive. Javia-II allows the VI architecture to access pinned `ViBuffer` directly and lets applications read/write from/into these buffers through regular Java array references. These accesses are only subjected to safety checks (e.g. array-bounds checks) already imposed by the Java language.

The Jaguar project [WC99] essentially overcomes the level of indirection that plagued JDirect: extensions to the JIT compiler generate code that directly accesses a pinned object's fields. The code generation is triggered by object typing information (i.e. external objects) rather than source-level annotations as in JDirect. An implementation of the Berkeley/Linux VI architecture using Jaguar achieves the same level of performance as Javia: within 1% of the raw hardware. In spite of the high performance, extending the JIT compiler raises a security concern: whether or not the generated code actually preserves the type-safety properties of the byte-code. For example, Jaguar would have to generate explicit array-bounds check when accessing an external array. This is not a concern in Javia since it accesses buffers through genuine Java references. Another difference between Jaguar and Javia is that Jaguar trades trusted protection for the ability to access hardware control resources, such as network and file descriptors, in a fine-grain manner. Javia-II uses special buffers to handle inefficiencies in the data transfer path only while maintaining the safety properties of Java.

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

Recently, several projects have focused on making the performance of Java RMI suitable for parallel computing on clusters of workstations. Manta [MNV+99] is a "Java-like" language and implements Java RMI efficiently over a Panda, a custom communication system. Manta relies on compiler-support for generating marshaling and unmarshaling code in C, thereby avoiding type checking at runtime. It communicates using both JDK's serialization protocol (for compatibility) as well as a custom protocol (for performance). Manta is able to avoid array copying in the critical path by relying on a non-copying collector and scatter/gather primitives in Panda. The authors report a RMI latency of 35μs and a throughput of 51.3 Mbytes/s on a PII-200/Myrinet cluster, which is within 15% of the throughput achieved by Panda.

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

7 Concluding Remarks

The research presented here pursues the simple goal of exposing user-level network interfaces to Java applications. By first exploiting native buffers, Javia-I motivates the need for explicit management of buffers and Javia-II. The performance achieved with Javia-II is encouraging: the overhead of the interface is small compared to the round-trip latency and the peak bandwidth is essentially the same as that achieved using a C program. This is a clear indication that it is worth removing the copies from the critical path.

Javia is intended as a building block for the construction of parallel Java applications as well as communication libraries *entirely* in Java. Javia has been used for implementing a parallel matrix multiplication application (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 lower communication times if there is sufficient data locality. A point-to-point evaluation of Jam shows that high-level communication protocols can be implemented efficiently in Java.

8 Acknowledgements

This research is funded by DARPA ITO contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations. Chi-Chao Chang is supported in part by a doctoral fellowship (200812/94-7) from CNPq/Brazil.

9 References

[AC95] A. Mainwaring and D. Culler. Active Messages Application Programming Interface and Communication Subsystem Organization. *Draft technical report*, 1995.

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

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

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

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

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

[CvE99] C-C. Chang and T. von Eicken. Jbufs: Enabling Safe and Efficient Networking and I/O in Java. *Cornell CS TR*, September 1999.

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

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

[G98] Giganet, Inc. http://www.giga-net.com.

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

[Int99] Intel Corporation. http://developer.intel.com/vtune/perflibst/mkl/mklperf.htm

[JD97] Microsoft Corporation. http://www.microsoft.com/java/resource/jdirect.htm.

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

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

[NPH99] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM SIGPLAN Java Grande Conference*, San Francisco, CA, June 1999.

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

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

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

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

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

[WC99] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. Submitted for publication, August 1999.

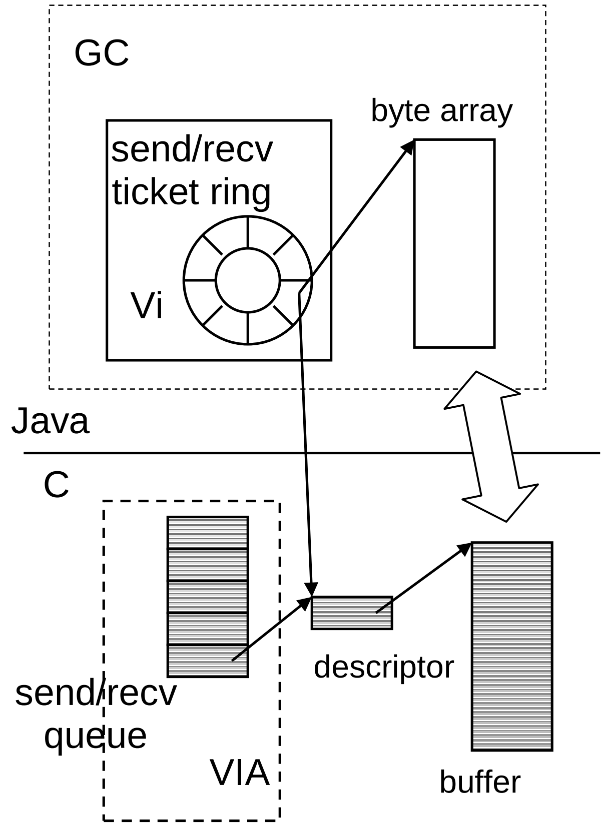

Figure 1. Javia-I per-endpoint architecture. Solid arrow indicates data copying.

	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. Javia-I's 4-byte round-trip latencies and per-byte additional overhead.

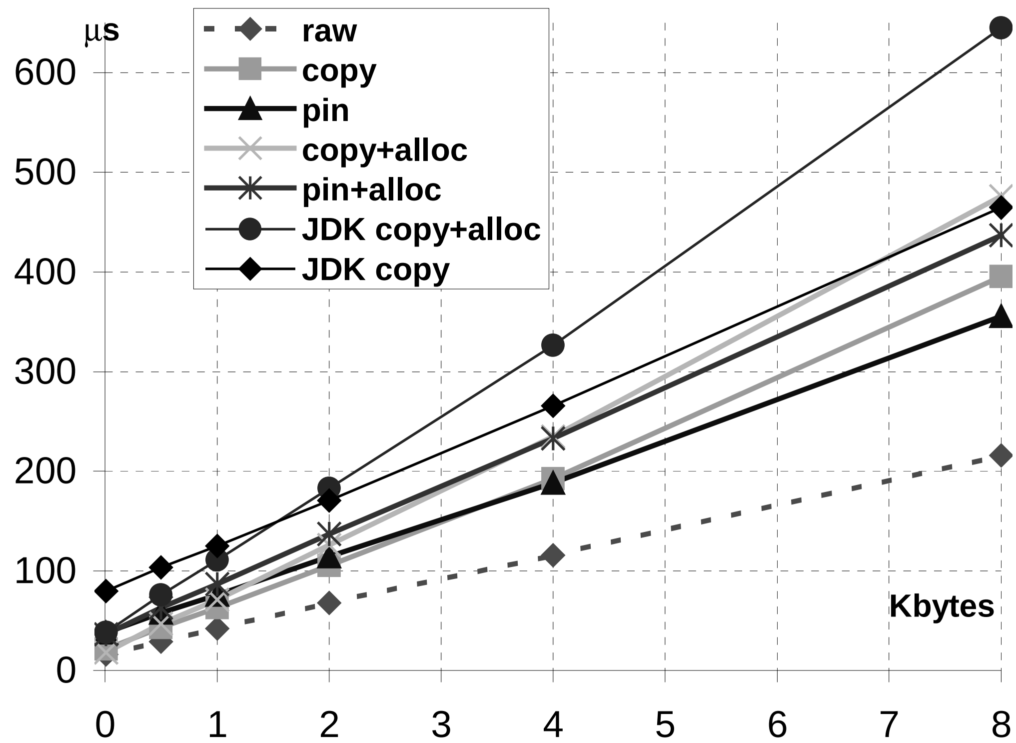

Figure 2. Javia-I's round-trip latencies.

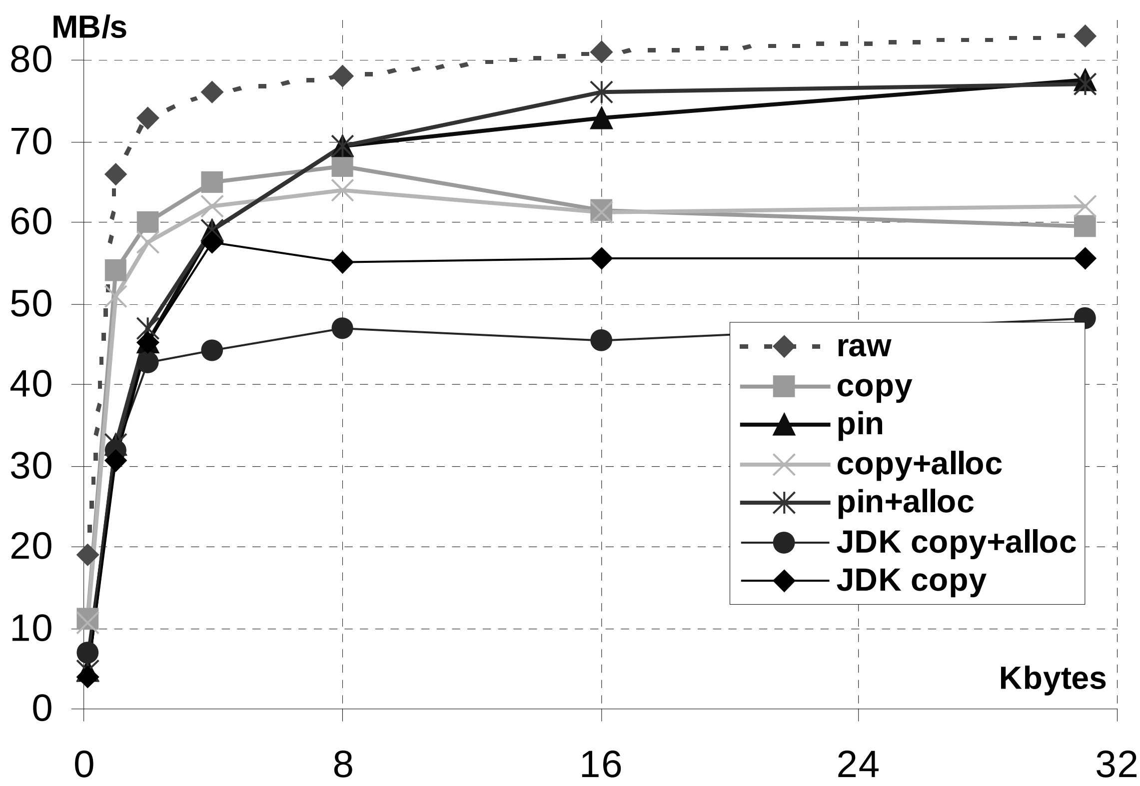

Figure 3. Javia-I's effective bandwidth.

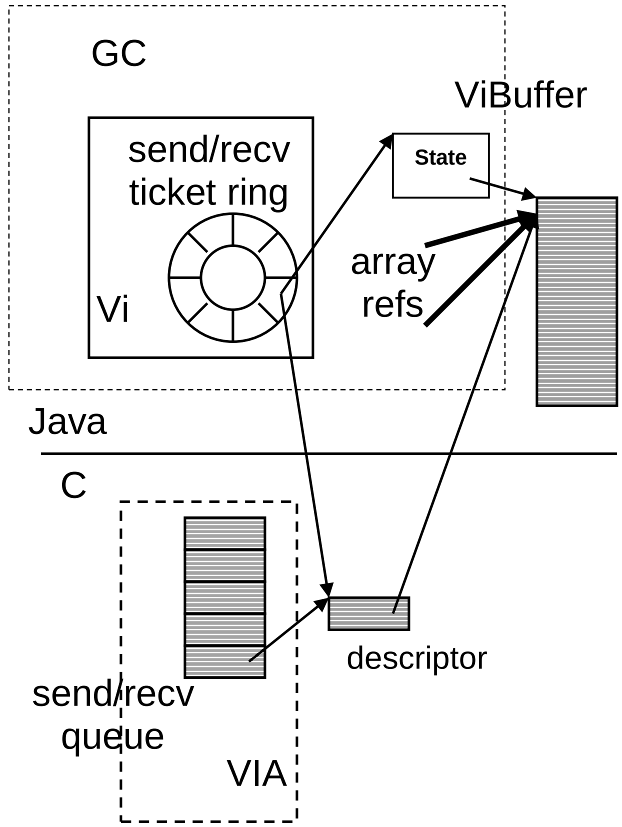

Figure 4. Javia-II per-endpoint architecture.

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

Table 2. Javia-II's 4-byte round-trip latencies and per-byte additional overhead.

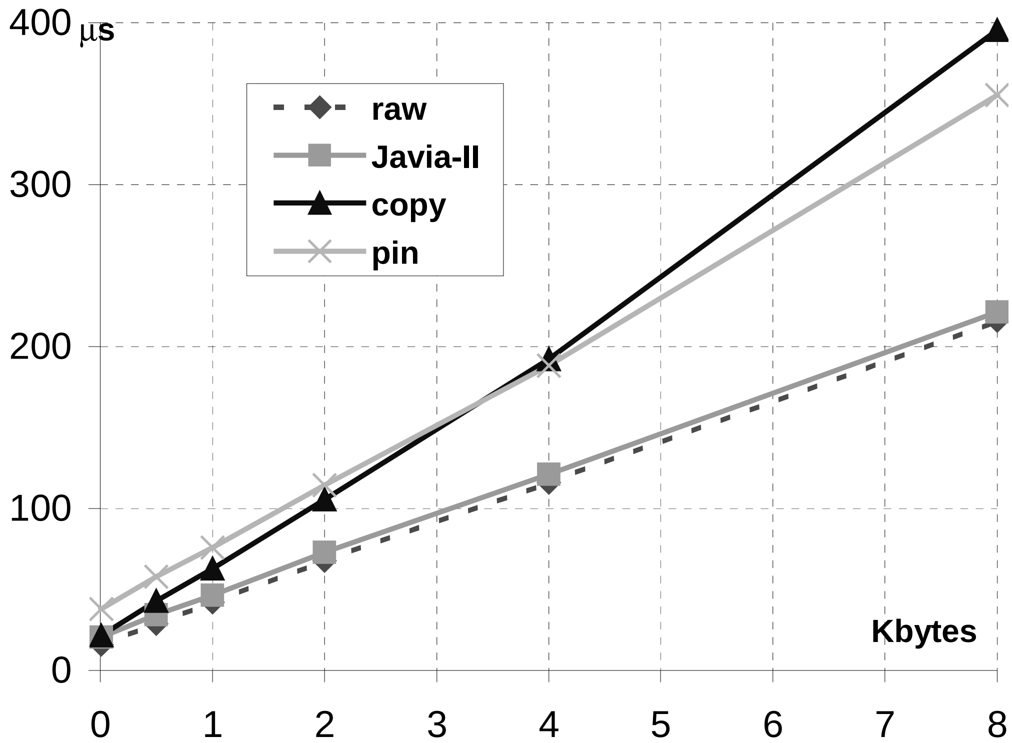

Figure 5. Javia-II's round-trip latencies.

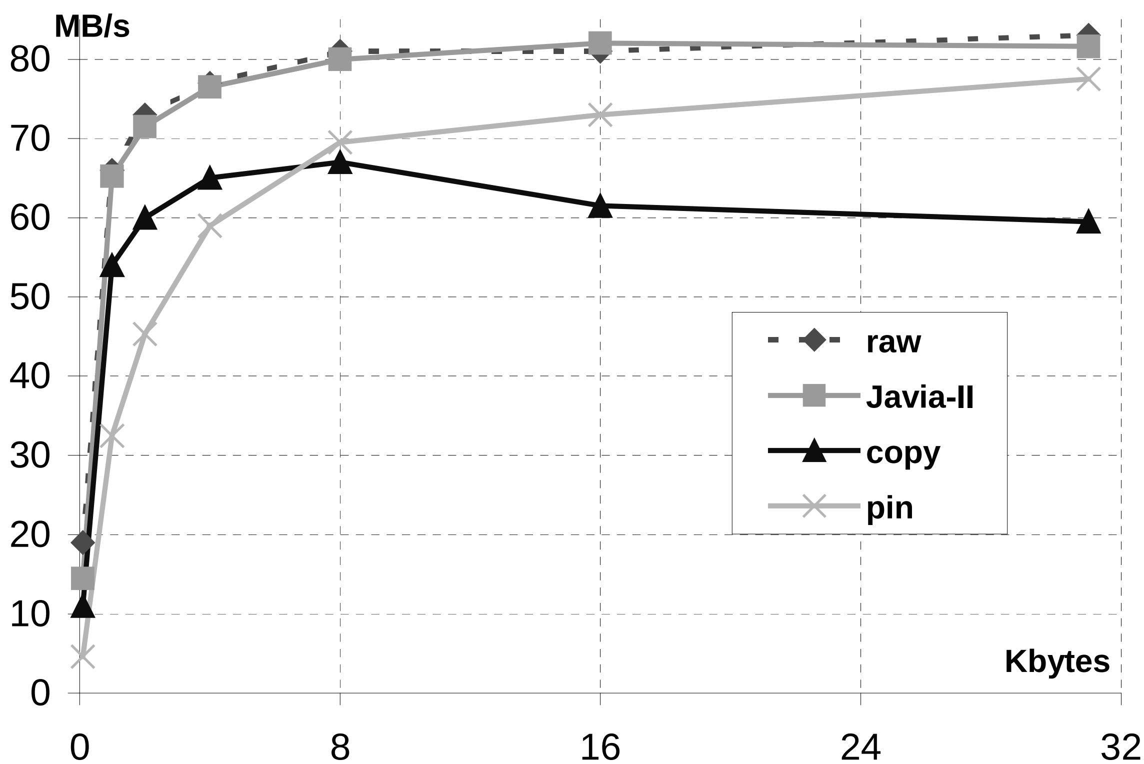

Figure 6. Javia-II's effective bandwidth.

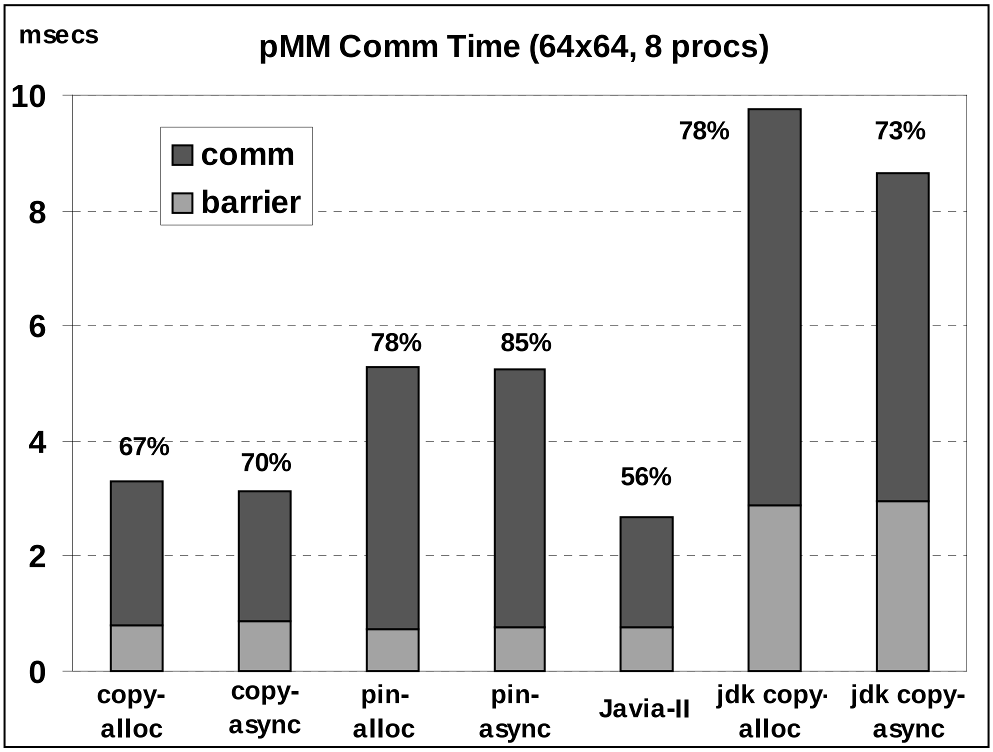

Figure 7. Comparison of pMM's communication time using Javia-I and Javia-II for 64x64 matrices on 8 cluster nodes. The percentage of communication in the total execution time is indicated on top of each bar.

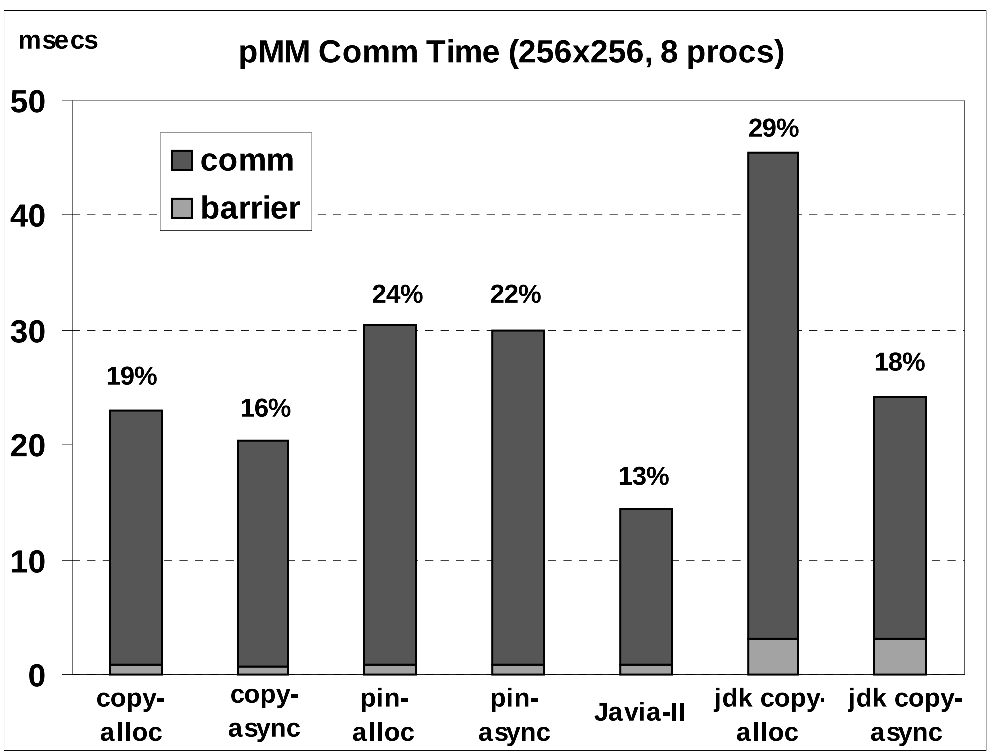

Figure 8. Comparison of pMM's communication time using Javia-I and Javia-II for 256x256 matrices on 8 cluster nodes. The percentage of communication in the total execution time is indicated on top of each bar.

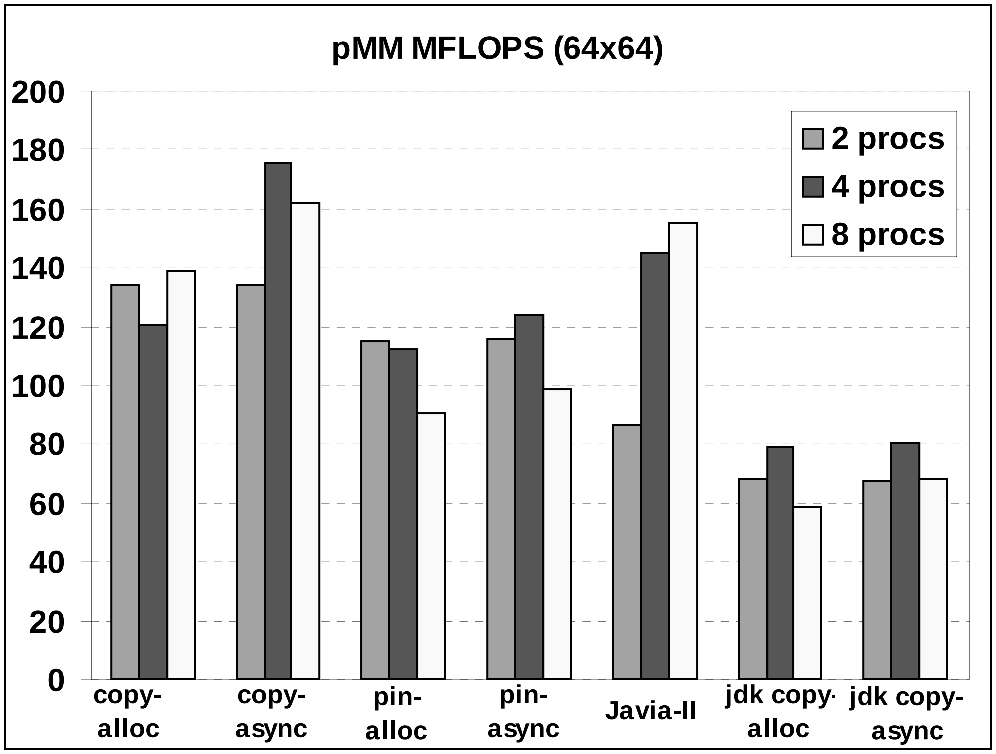

Figure 9. Comparison of pMM's overall performance using Javia-I and Javia-II for 64x64 matrices on 2, 4, and 8 cluster nodes.

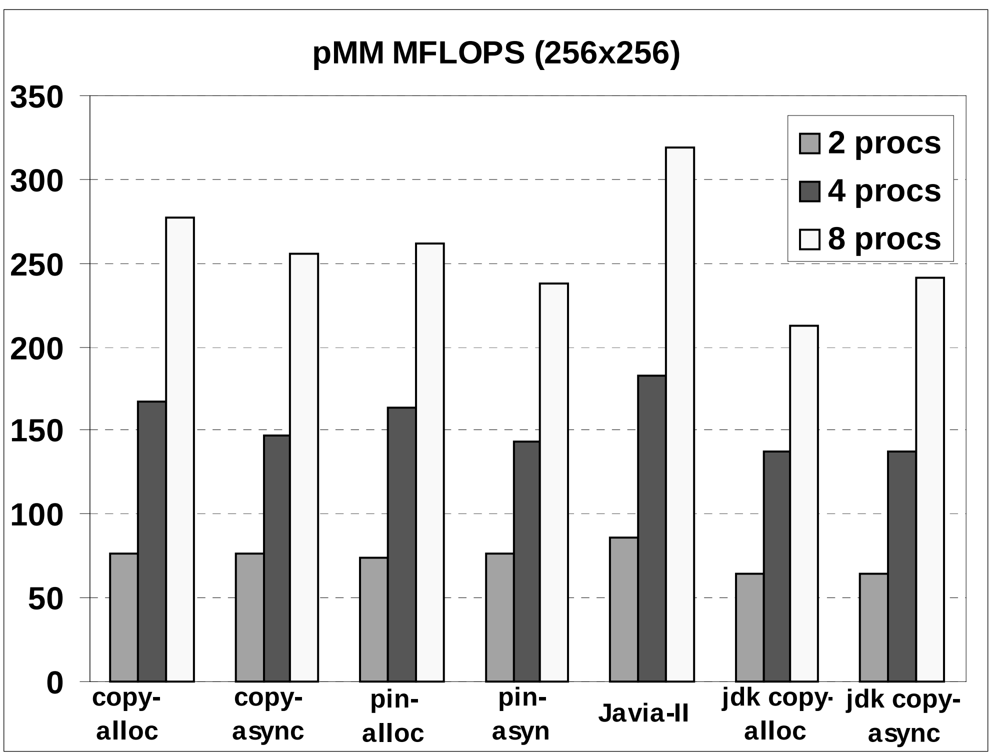

Figure 10. Comparison of pMM's overall performance using Javia-I and Javia-II for 256x256 matrices on 2, 4, and 8 cluster nodes.

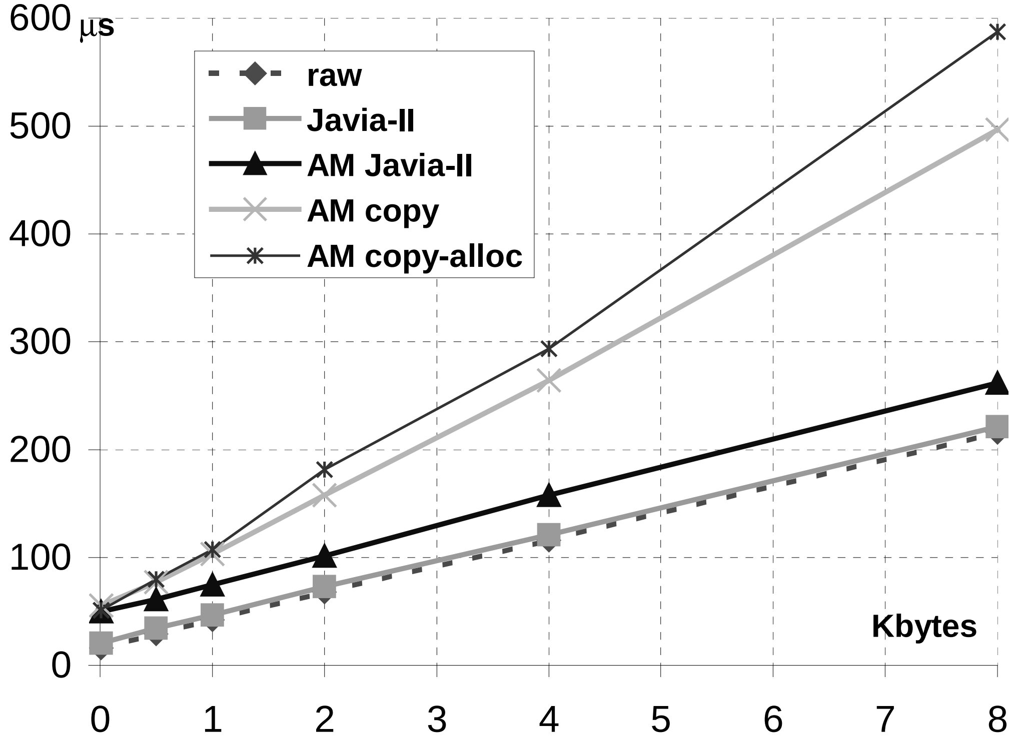

Figure 11. Jam round-trip latencies using Javia-I (*AM copy*, *AM copy-alloc*) and Javia-II, and compared with raw and Javia-II.

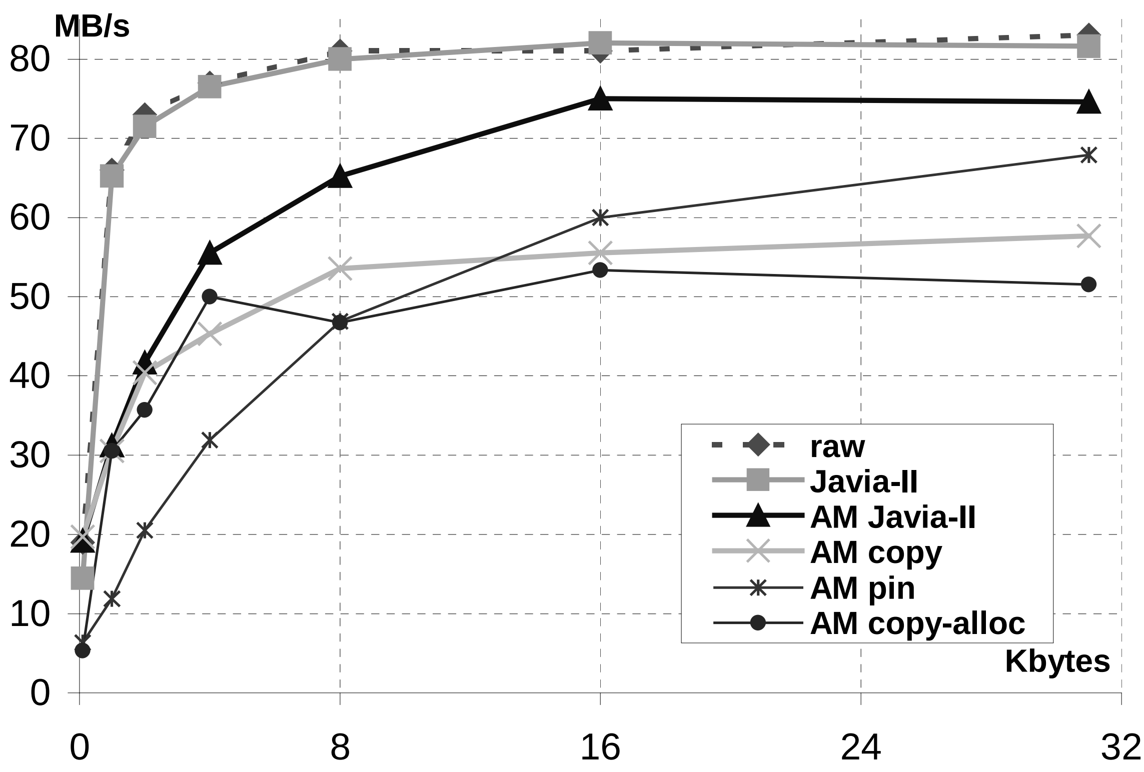

Figure 12. Jam's effective bandwidth using Javia-I (*AM copy*, *pin*, *copy-alloc*) and Javia-II, and compared with raw and Javia-II.