3 Safe and Explicit Memory Management

Javia-I offers a straightforward front-end interface to the VI architecture within the bounds of the safety properties of Java: communication buffers and descriptors are managed entirely by a native library, which in turn interacts with the JVM through a Java/native interface. This approach is inefficient because it incurs overheads in the communication critical path that are attributed to copying data between the garbage-collected (GC) and the native heap as well as to pinning arrays on the fly. Javia-I results show that the hard separation between Java and native heaps yield a 10% to 40% hit in point-to-point performance for a range of message sizes.

This chapter addresses the shortcomings of Javia-I by first introducing the notion of *buffers*, or *jbufs*, to Java applications. The main motivation behind jbufs is to provide programmers with the same flexibility to manage (e.g. allocate, free, re-use) buffers in Java as they have in C. Besides explicit management, jbufs can be accessed efficiently from Java and, with the cooperation of the GC, can be re-used or freed without violating language safety. The key idea is to allow users to control whether a jbuf is part of the GC heap, *softening*

the hard separation that plagues Javia-I. Jbufs do not require changes to the Java source language or byte-code, and leverage most of the existing language infrastructure, including Java compilers, library support, and GC algorithms.

The chapter moves on to show that jbufs serve as a simple, powerful, and efficient framework for building communication software and applications in Java. Javia-II improves on Javia-I by defining communication buffers—jbufs extended with explicit pinning and unpinning capabilities—that are used directly by the VI architecture. Micro-benchmarks show that the raw performance achieved by the VI architecture becomes fully available to Java applications. These results are further corroborated by our experiences with pMM, a parallel matrix multiplication program, and Jam, an active messages communication layer, both of which are implemented on top of Javia-I/II and jbufs.

3.1 Jbufs

A jbuf is a region of memory that is abstracted by the `Jbuf` class:

```
1  public class Jbuf {
2
3    /* allocates a jbuf of size bytes */
4    public final static Jbuf alloc(int bytes);
5
6    /* attempts to free the jbuf */
7    public final void free() throws ReferencedException;
8
9    /* attempts to obtain a <p>[] reference to the jbuf, where */
10   /* p is a primitive type. Only byte[] and int[] are shown.*/
11   public final synchronized byte[] toByteArray() throws TypedException;
12   public final synchronized int[] toIntArray() throws TypedException;
13
14   /* claims that there are no references into the jbuf and */
15   /* waits for the GC to verify the claim. */
16   public final void unRef();
17
18   /* cb is invoked by GC after claim is verified */
19   public final void setCallBack(CallBack cb);
20
21   /* checks if a reference points into a jbuf */
22   public final boolean isJbuf(byte[] b);
23   public final boolean isJbuf(int[] i);
24   /* others omitted */
25 }
```

and provides users with three features:

1. *Lifetime control* through explicit allocation (`alloc`, line 4) and de-allocation (`free`, line 7).
2. *Efficient access* through direct references to Java primitive arrays (`toByteArray`, `toIntArray`, etc, lines 11-12).
3. *Location control* through interactions with the GC (`unRef` and `setCallBack`, lines 16 and 19).

In order to achieve lifetime control, jbufs differ from traditional Java objects in two ways. First, `alloc` allocates a jbuf *outside* of the Java heap and does *not* return a Java reference into that jbuf. Instead, it returns a wrapper object, which resides in the GC heap and contains a private C handle to the jbuf, as seen Figure 3.1(a). An application must explicitly obtain a genuine array reference into the allocated jbuf; it cannot access the jbuf through the wrapper object (Figure 3.1(b)). The second difference is that jbufs are not automatically freed—an application must invoke `free` on them explicitly.

Java's storage safety is preserved by ensuring that jbufs will remain allocated as long as they are referenced. For example, invocations of `free` result in a `ReferencedException` if an application holds one or more references into the jbuf. 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 jbuf.

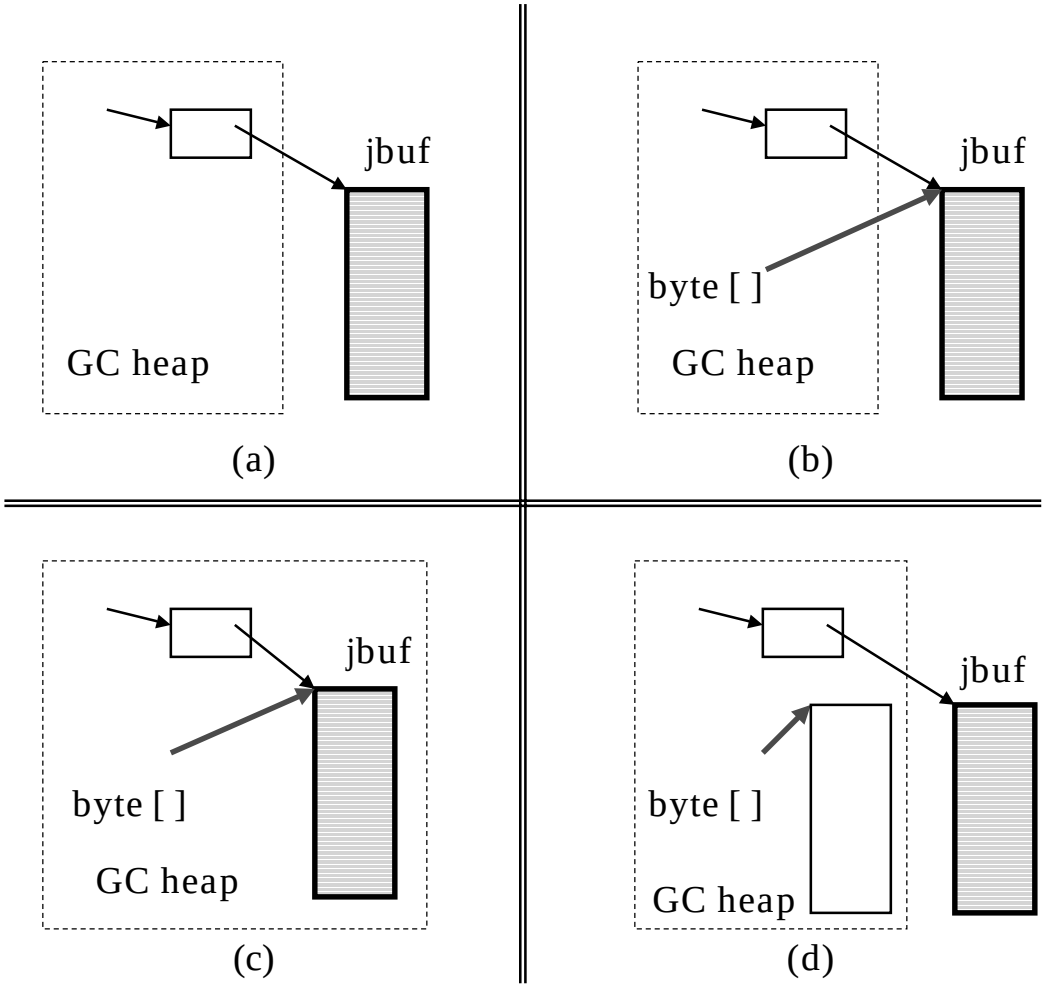

Figure 3.1 Typical lifetime of a jbuf in a copying GC. (a) After allocation, the jbuf resides outside of the GC heap. (b) The jbuf is accessed as a Java array reference. (c) The jbuf is added to the GC heap. (d) Upon callback invocation, the jbuf can be freed or re-used.

Location control enables safe de-allocation and re-use of jbufs by controlling whether or not a jbuf is part of the GC heap. The idea is to use the underlying GC to track references into the jbufs. The application indicates its willingness to free or re-use a jbuf by invoking its `unRef` method (line 9). It is thereby claiming that it no longer holds any references into the jbuf. The effect of the `unRef` call is that the jbuf becomes part of the GC heap. After at least one[14] GC occurrence, the collector verifies that there are no references into a jbuf and notifies the application through a callback (which is set by invoking `setCallback`, line 10). At this point, the jbuf is removed from the GC heap and can be safely re-used or freed. Figure 3.1(c-d) illustrates location control in the context of a copying collector. In essence, with location control the separation between GC and native heaps becomes *soft* (i.e. user-controlled).

A by-product of location control is that an array reference into a jbuf may become *stale* (e.g. one that no longer points to a jbuf as seen in Figure 3.1(d)). Programmers can check whether an array reference is stale by invoking the appropriate `isJbuf` method (lines 12-13).

3.1.1 Example: A Typical Lifetime of a Jbuf

A typical use of jbufs is as follows:

```
1  Jbuf buf = Jbuf.alloc(1024);   /* allocate a jbuf of 1024 bytes */
2  Byte[] b = buf.toByteArray();  /* get a byte[] reference into buf */
3  for (int i=0; i<1024; i++) b[i] = (byte)i; /* initialize b */
4
5  /* use b: for example, send b across the network…*/
6
7  buf.unRef(new MyCallBack());   /* intends to free or re-use buf */
8  System.out.println(isJbuf(b));
9
10 /* callback has been invoked */
11
12 buf.free();
13 System.out.println(isJbuf(b));
```

[14] The required number of GC invocations depends on the GC scheme, as explained in the next section.

The print statement in line 8 outputs true because `b` is still a reference into a jbuf: the callback has not been invoked. If the underlying GC is a copying one, the statement in line 13 outputs false: `b` points to a regular byte array inside the GC heap. If the GC is a non-copying one, `b` in line 13 must be nil; or else line 12-13 will not have been reached because the callback will not be invoked.

3.1.2 Runtime Safety Checks

Safety is enforced using runtime checks and with the cooperation of the garbage collector. As shown in Figure 3.2, a jbuf can be in three states:

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

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

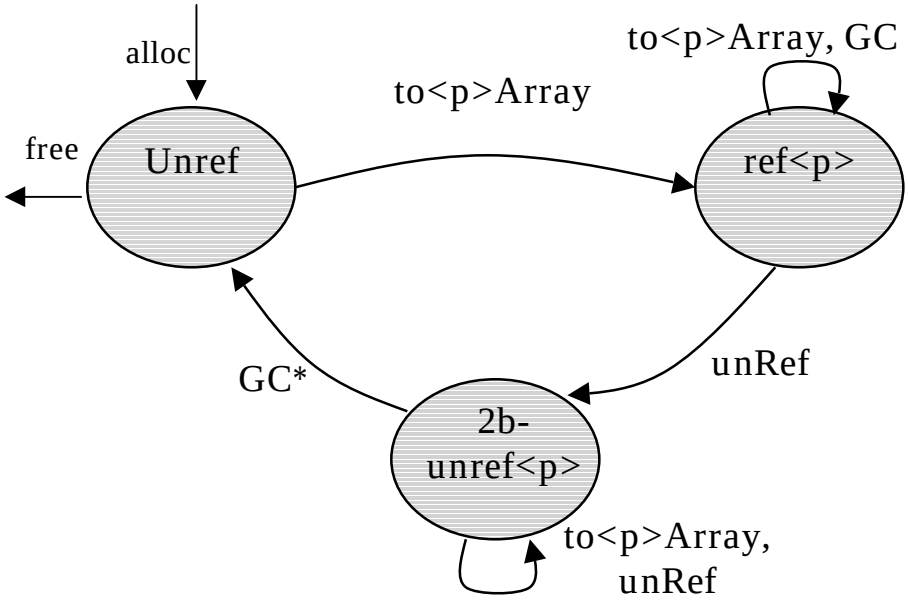

Figure 3.2 Jbufs state diagram for runtime safety checks. When the GC* transition takes place depends on whether the GC is copying or non-copying.

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

It is important to note that no additional runtime checks are needed to access jbufs apart from array-bounds and null-pointer checks imposed by Java. These runtime checks are only performed during jbuf management operations, which are typically not as performance critical as data access operations.

3.1.3 Explicit de-allocation

Two important issues regarding explicit de-allocation are (i) that it appears to violate Java's storage safety since an application may leak memory (accidentally or intentionally) by not freeing jbufs, and (ii) that it seems dispensable in the presence of object finalization (Section 12.12.7, [LY97]). Accidental memory leakage (e.g. an application has no array references into a jbuf but forgets to free it) is easily prevented by maintaining jbuf wrapper objects in an internal list and keeping track of the total jbuf-memory consumption. At some threshold value, the jbuf allocation routine traverses the list and frees all jbufs that are in the *unref* state. However, the Java language itself, let alone jbufs, cannot stop an application from leaking memory maliciously. For example, a user can consume unlimited memory by deliberately growing an unused linked-list.

Explicit de-allocation is indispensable. If the jbuf wrapper objects are kept in that internal list, then eliminating explicit de-allocation by having wrapper object finalizers[15] free jbufs will not work as the wrapper objects never become garbage. If wrapper objects need not be kept in that list, then they may become garbage. However, when the finalizer of the wrapper object is executed, the jbuf may well be in the *ref*<*p*> state in which it cannot be freed. Given that the finalizer is only executed once, jbufs in that state will never be freed.

3.1.4 Implementing Jbufs with a Semi-Space Copying Collector

Jbuf storage is allocated and de-allocated using Win32 `malloc` and `free` calls. The allocated memory region has a 4-word header to store array meta-data:

---

[15] It is not possible to overwrite the `finalize` methods of array objects.

one for the dispatch table, one for synchronization structure, one for the length, and another for padding purposes. The meta-data is (over)written during successful `to<p>Array` invocations. The state of the jbuf is stored in the wrapper object.

In order to support jbufs, the garbage collector must be able to change the scope of its collected heap dynamically. When a jbuf is `unRef`ed, the collector must add the jbuf's region of memory to the heap (`attachToHeap`), and remove it prior to invoking the callback (`detachFromHeap`).

We made minor modifications to Marmot's semi-space copying GC. The collector is based on Cheney's scanning algorithm [Wil92]: the collector copies the referenced object from the *from-space* to the *to-space*. In addition to the two semi-spaces, the augmented Marmot collector maintains a list of jbufs: `attachToHeap` simply adds a jbuf to that list whereas `detachFromHeap` removes it from the list. When following a reference, the from-space is always checked first so the GC performance of programs that do not use jbufs is not affected.

The current implementation of jbufs consists of 450 lines of Java and 390 lines of C. Fewer than 20 lines of code have been added/modified in Marmot's copying GC code (which is about 1000 lines of C, a third of Marmot's total GC code). Most of the C code for jbufs is for managing lists of jbuf segments.

3.1.5 Performance

The performance of jbufs is evaluated using three simple benchmarks on Marmot. The first one measures the overheads of `alloc` and `free`: *M* jbufs of 4 bytes each (excluding meta-data) are allocated and freed in separate loops,

repeated over *N* iterations. The second measures the cost of invoking `toByteArray`, `isJbuf`, and `unRef`. One loop invokes `toByteArray` on each of the *M* jbufs, another that invokes `isJbuf` on each byte array reference, and followed by a third loop that invokes `unRef` on each jbuf. The third synthetic benchmark measures the performance impact of jbufs on Marmot's copying collector: the cost of collecting a heap[16] with *M unreferenced*, 4-byte jbufs is subtracted from the cost of collecting the same heap with the *M* jbufs *referenced.*

Table 3.1 Jbufs overheads in Marmot

	cost (us)
alloc	2.72
free	2.24
toByteArray	0.50
isJbuf	0.30
unRef	2.54
gc overhead (p/jbuf)	0.55

Table 3.1 shows the micro-benchmark results for *M=1000* and *N=100*. The cost of `alloc` is about 2μs higher than that of allocating a byte array of the same size (which is 0.7μs). The overheads in `unRef` include accessing two critical sections (one to update the state of the jbuf, another to update the GC region list) compared to one in `toByteArray`. The per-jbuf overhead in GC includes tracking the reference into the jbuf, copying a 4-byte array, and invoking the callback method. Overall, the overall copying GC performance in Marmot is fairly unaffected.

[16] The heap *includes* the jbufs wrapper objects. The size of the heap is immaterial: the difference between the two heaps is essentially the jbuf segments.

3.1.6 Implications on Other Garbage Collection Schemes

Similar modifications made to Marmot's semi-space copying collector are also applicable the conservative mark-sweep collector as well as the two-generations copying collector [Tar99].

The conservative mark-sweep collector divides a large, contiguous heap space into blocks and keeps a list of free blocks. The blocks are maintained by several large bit-maps, one of which is used by the mark phase. The collector segregates objects based on size. It does not rely on per-object pointer information except for checking if an array is an array of objects. During the mark phase, the collector checks if a pointer points to a jbuf only after it has determined that it does not point to the original heap. As jbufs do not contain pointers, they need not be further scanned by the mark phase. After the sweep phase, the list of jbufs is traversed: unmarked ones have their callbacks invoked and are detached from the list[17].

The generational collector is a simple two-generation collector with an allocation area and an older generation. It implements write-barriers based on a sequential-store-buffer technique to track pointers from the older generation into the allocation area. Jbufs added to the list are part of the allocation area[18] and thus have to be checked when following a pointer into that area.

3.1.7 Proposed JNI support

An extension to the JNI can enable more portable implementations of jbufs without revealing two JVM-specific information: the meta-data layout of ar-

---

[17] The Boehm-Demers-Weiser [BW88, Boe93] conservative collector uses lazy sweeping for better performance: instead of sweeping the whole heap after each collection, the collector incrementally sweeps the heap on demand until the sweep is complete. Lazy sweeping can be implemented with jbufs without much effort.

[18] In fact, jbufs should always be part of the youngest generation regardless of the number of generations.

rays and the GC scheme. The proposed extension consists of three functions as follows, where `<Type>` is a placeholder for a primitive type:

`jint get<Type>ArrayMetaDataSize(JNIEnv *env);`

This function returns the storage size (in bytes) for the array meta-data. If the array meta-data and body (in this order) are not contiguous in memory, the function returns zero.

`j<Type>Array Alloc<Type>Array(JNIEnv *env, int array_size, char *seg, int seg_size, void *body);`

This function allocates a primitive-typed array of size `array_size` in a memory segment `seg` supplied by the user. The function fails if `seg_size` is smaller than the size of array (in bytes) plus the meta-data storage size. The function returns both a pointer to the body of the array (`body`) and a reference to the array itself. If `body` is null, then the array can only be accessed through JNI only (the implementation is being very conservative here, but it is still ok). If not, then body must be a C pointer into the memory segment.

`void AttachHeap(char *seg, void (*callback )(char *));`

This function attaches the memory segment `seg` to the GC heap along with a callback function. It only succeeds after a successful invocation of `Alloc<type>Array` associated with `seg` and prior to the invocation of a callback associated with the same. Attaching an already attached segment results in a nop.

3.2 Javia-II

3.2.1 Basic Architecture

Javia-II defines the `ViBuffer` class, which extends a jbuf with methods for pinning (and unpinning) its memory region onto the physical memory so that the VI architecture can DMA directly into and out of jbufs.

```
1  /* communication buffer */
2  public class ViBuffer extends Jbuf {
3    /* pinning and unpinning */
4    public ViBufferTicket register(Vi vi);
5    public void deregister(ViBufferTicket t);
6  }
7
8  /* ticket is returned by register and used by deregister */
9  public class ViBufferTicket {
10   /* no public constructor */
11   ViBuffer buf; private int bytesRecvd, off, tag;
12   /* public methods to access fields ommited */
13 }
14
15 public class Vi {
16   /* async send */
17   public void sendBufPost(ViBufferTicket t);
18   public void sendBufWait(int millisecs);
19   /* async recv */
20   public void recvBufPost(ViBufferTicket t);
21   public void recvBufWait(int millisecs);
22 }
```

The `register` method (line 4) pins the buffer to physical memory, associates it with a VI, and obtains a descriptor to the memory region, which is represented by a `ViBufferTicket` (lines 9-13). At that point, the buffer can be directly accessed by the VI architecture for communication. A jbuf can be de-registered (line 5), which unpins it, and later re-registered with the same or a different VI. If `register` is invoked multiple times on the same jbuf, the jbuf is pinned only once; *all* tickets have to be de-registered before the jbuf is unpinned.

For transmission and reception of buffers, Javia-II provides only asynchronous primitives, as shown in lines 17-21. Javia-II differs from Javia-I in that the VI descriptors point directly to the Java-level buffers instead of native

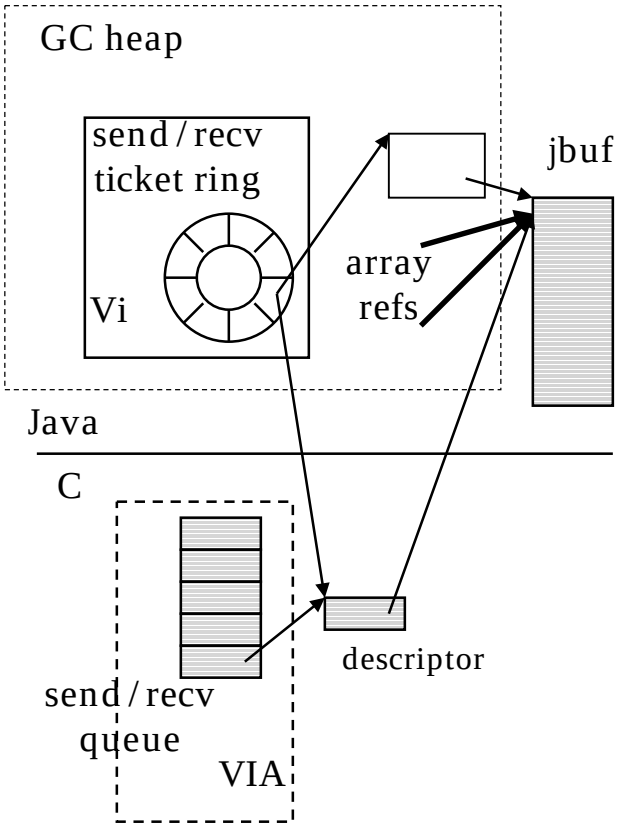

Figure 3.3 Javia-II per-endpoint data structures.

buffers (Figure 3.3). 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.

Javia-II provides two levels of type safety. In the first level, no type checking is performed during message reception: for instance, data trans-

ferred out of a jbuf referenced as a `double` array can be deposited into a jbuf that is typed as an `int` array. In the second level, type checking is performed by tagging[19] the message with the source type before transmission and matching that tag with the destination type during reception.

By using jbufs, Javia-II itself remains very simple: it adds about 100 lines of Java and 100 lines of C to the Javia-I implementation.

3.2.2 Example: Ping-Pong

The following is a simplified ping-pong program using Javia-II and jbufs:

```
1  ViBuffer buf = new ViBuffer(1024);
2  /* get send ticket */
3  ViBufferTicket sendT = buf.register(vi, attr);
4  /* get recv ticket */
5  ViBufferTicket recvT = buf.register(vi, attr);
6  byte[] b = vb.toByteArray();
7  /* initialize b… */
8  /* post recv ticket first */
9  vi.recvBufPost(recvT, 0);
10 if (ping) {
11   /* send */
12   vi.sendBufPost(sendT, 0, 1024);
13   sendT = vi.sendBufWait(Vi.INFINITE);
14   /* wait for reply */
15   recvT = vi.recvBufWait(Vi.INFINITE);
16   /* done */
17 } else { /* pong */
18   vi.recvBufPost(recvT, 0);
19   recvT = vi.recvBufWait(Vi.INFINITE);
20   /* send reply*/
21   vi.sendBufPost(sendT, recvT.off, recvT.bytesRecvd);
22   sendT = vi.sendBufWait(Vi.INFINITE);
23   /* done */
24 }
25 buf.deregister(sendT);
26 buf.deregister(recvT);
27 buf.unRef(new MyCallBack());
28 /* after callback invocation… */
29 buf.free();
```

3.2.3 Performance

Table 3.2 and Figure 3.4 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-

[19] Using 32-bit message tags supported by the VI architecture.

Table 3.2 Javia-II 4-byte round-trip latencies and per-byte overhead

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

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 3.5) is between 1% to 3% of that of *raw*, which is within the margin of error.

3.3 pMM: Parallel Matrix Multiplication in Java

pMM consists of a single program image (same set of Java class files, or same executable in the case of Marmot) running on each node in the cluster. The

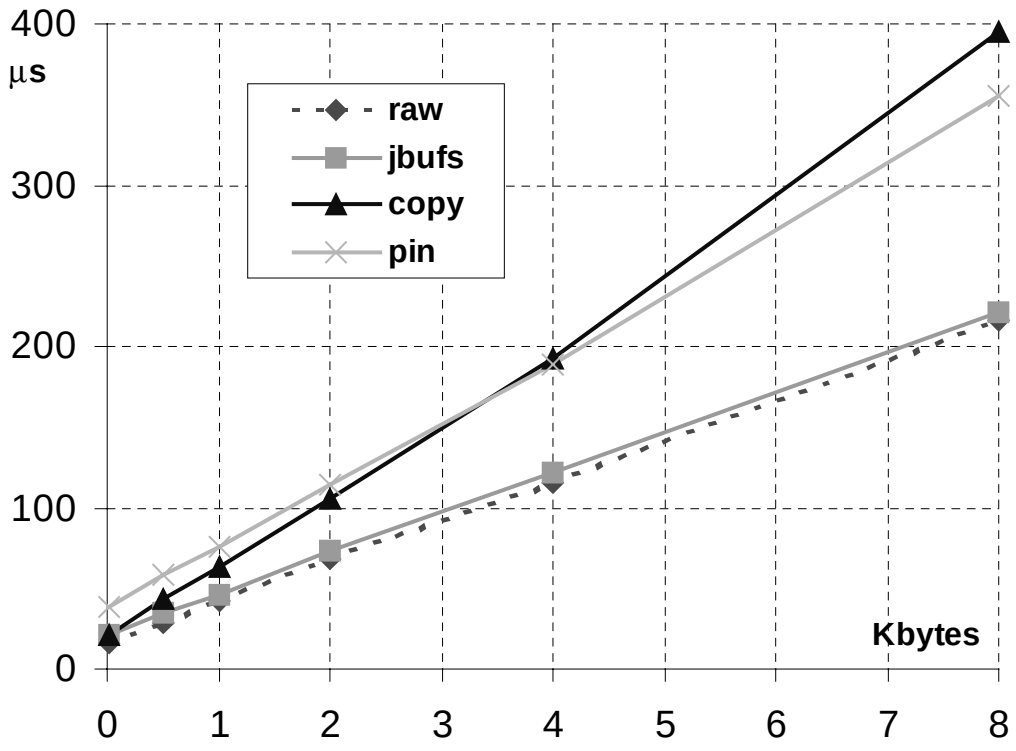

Figure 3.4 Javia-II round-trip latencies

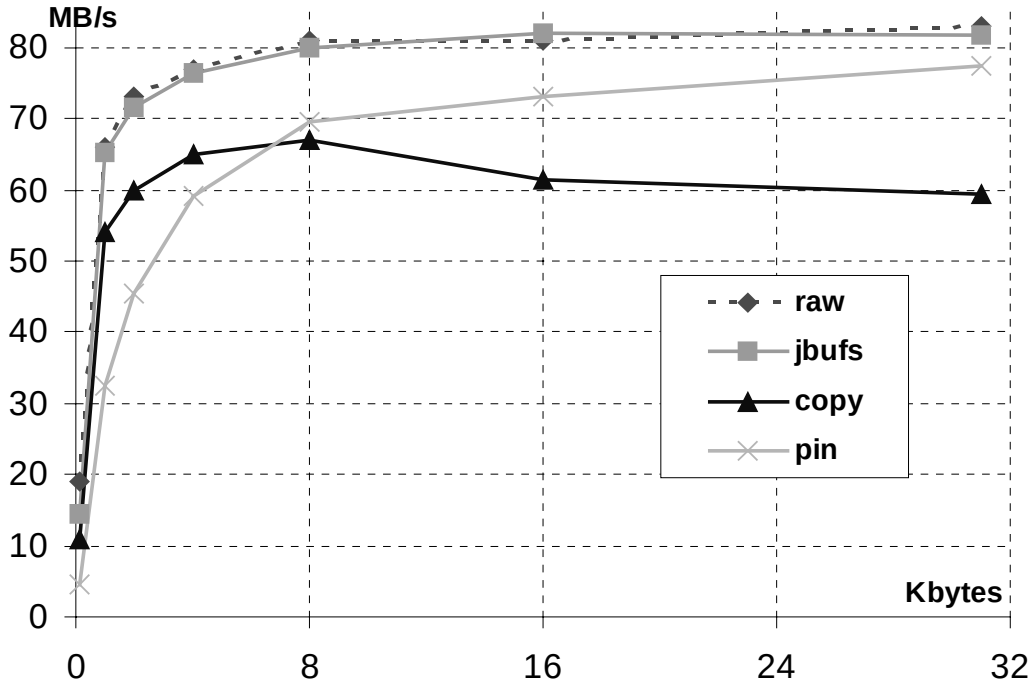

Figure 3.5 Javia-II effective

program image is spawned manually on every node and runs on top of a simple parallel virtual machine called *Pack*. During initialization, Pack is responsible for setting up a complete connection graph between the cluster nodes and providing global synchronization primitives based on barriers.

pMM represents a matrix as an array of arrays of doubles and uses a parallel algorithm based on message passing and *block-gaxpy* operations [GvL89]. The algorithm starts with the input matrices A and B and the output matrix C distributed across all processors in a block-column fashion so each processor owns a "local" portion of each matrix. To perform a block gaxpy procedure, each processor needs its local portion of B but the entire matrix A. To this end, the algorithm circulates portions of A around the ring of processors in a "merry-go-round" fashion. At every iteration (out of *p*, where *p* is the

total number of processors), the communication phase consists of having each processor send its local portion of A to its right neighbor and update it with the new data received from its left neighbor. The computation phase consists of updating its local portion of C with the result of multiplying the local portions of A and B.

The first implementation of pMM uses Javia-I to send and receive arrays of doubles whereas the second implementation uses jbufs that are accessed as arrays of doubles. The jbufs are pinned throughout the program and array references never become stale. The following code shows how jbufs are set up for communication.

```
1  /* Aloc is the local portion of A: array of n/p arrays of n doubles */
2  double[][] Aloc = new double[n/p][];
3  /* n/p jbufs, each being used as an array of n doubles */
4  ViBuffer[] bA = new ViBuffer[n/p];
5
6  /* bA's send and receive tickets */
7  ViBufferTicket[] sentT = new ViBufferTicket[n/p];
8  ViBufferTicket[] recvT = new ViBufferTicket[n/p];
9
10 /* initialize bA, tickets, and Aloc */
11 for (int j = 0; j < n/p; j++) {
12   bA[j] = new ViBuffer(n*SIZE_OF_DOUBLE);    /* allocate jbufs */
13   Aloc[j] = bA[j].toDoubleArray(n);          /* obtain double[] refs */
14   sendT[j] = bA[j].register(rightVi, rattr); /* pin for sends */
15   recvT[j] = bA[j].register(leftVi, lattr);  /* pin for recvs */
16   for (int i = 0; i < n; i++) {
17     /* Aloc initialization omitted */
18   }
19 }
```

The core of the algorithm used in pMM is as follows, using Javia-I blocking receives:

```
1  int tau = myproc;
2  int stride = tau * r;
3  pvm.barrier(); /* global synchronization */
4  for (int k = 0; k < p; k++) {
5    /* comm phase: send to right, recv from left using alloc receives */
6    if (tau != myproc) {
7    for (int j = 0; j < n/p; j++)
8      rightVi.send(Aloc[j], 0, n, 0);
9    for (int j = 0; j < n/p; j++) {
10     do { Aloc[j] = leftVi.recvDoubleArray(0); } while (Aloc[j] == null);
11   }
12   /* computation phase: iterate over columns A, B, and C*/
13   for (int j = 0; j < n/p; j++) {
```

```
14      double[] c = Cloc[j];
15      double[] b = Bloc[j];
16      /* iterate over rows */
17      for (int i = 0; i < n; i++) {
18        double sum = 0.0;
19        for (int k = 0; k < n/p; k++) {
20          double[] a = Aloc[k];
21          sum += a[i] * b[stride+k];
22        }
23        c[i] += sum;
24      }
25    }
26    tau++;
27    if (tau == p) tau = 0;
28    stride = tau * r;
29    pvm.barrier();
30  }
```

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).

3.3.1 Single Processor Performance

The performance of Java matrix multiplication on a single processor is still far from that achieved by the best numerical kernels written in Fortran or C. Representing a matrix as an array of arrays hinders the effectiveness of traditional block-oriented algorithms (e.g. level-2/3 BLAS found in LAPACK [ABB+92]) which rely on the contiguity of blocks for improved cache behavior. Another major impediment is the generation of precise exception handlers for array-bounds and null-pointer checks in Java. High-order compiler transformations, such as blocking, often restructure loops and move code around. Because of Java's strict sequential semantics imposed by exceptions, these transformations are not legal in Java [MMG98].

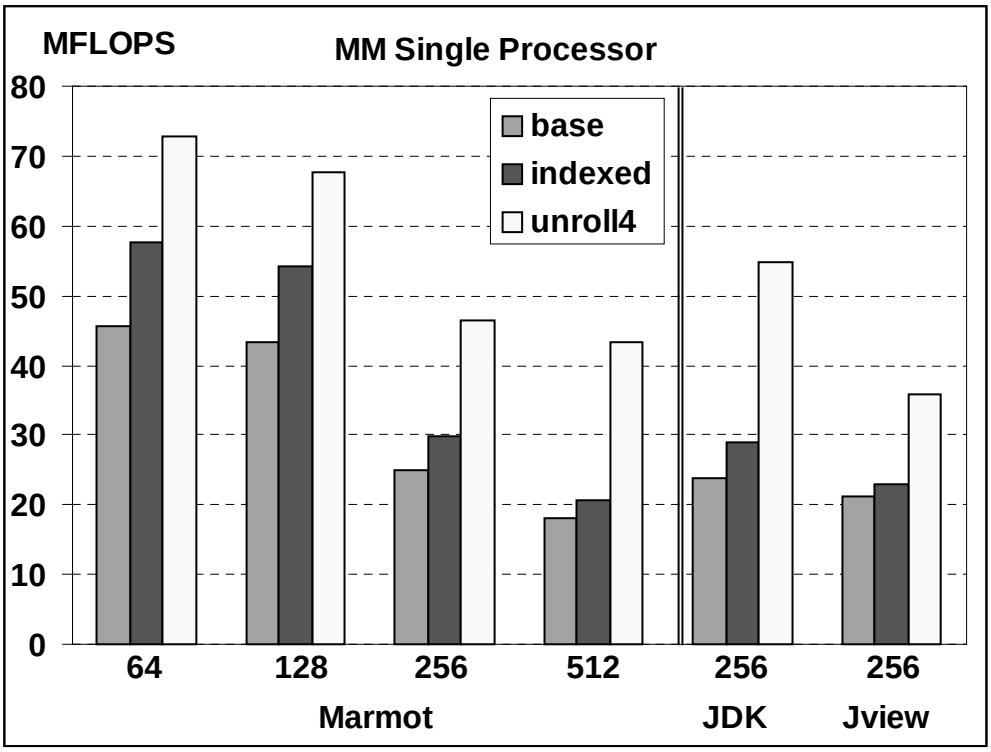

Figure 3.6 Performance of MM on a single 450Mhz Pentium-II

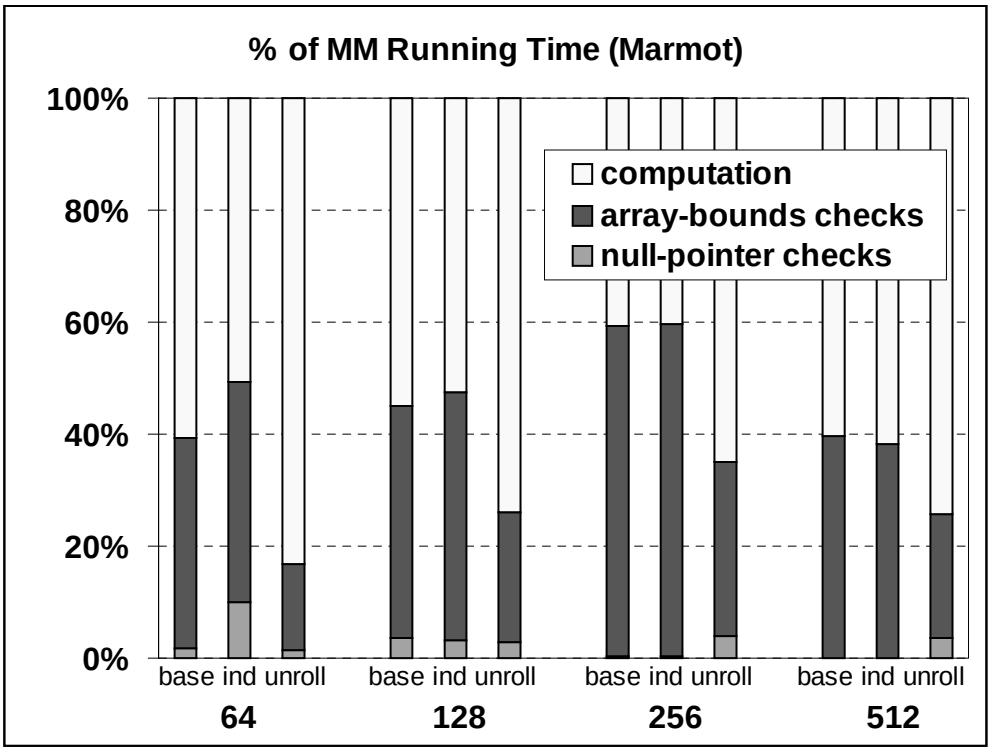

Figure 3.7 Impact of safety checks on MM

Figure 3.6 compares the performance of MM with different optimizations on a single cluster node. *indexed* implements 1-D indexing and scalar replacement, and *unroll4* performs 4-level loop unrolling on top of *indexed*. On a single cluster node, Marmot's *unroll4* achieves a peak performance of over 70Mflops for 64x64 matrices. As the data size increases, the performance drops significantly mostly due to poor cache behavior. For 256x256 matrices, Marmot achieves about 45Mflops, compared to 55Mflops and 37Mflops attained by JDK and Jview respectively. In comparison, the performance of DGEMM (i.e. matrix multiply) found in Intel's Math Kernel library [Int99] is over 400Mflops for 64x64 matrices. (All the numbers are for a 450Mhz Pentium-II).

Marmot allows us to selectively turn off particular safety checks, namely array-bounds and null-pointer checks, to determine their cost. Since none of these checks actually fail during the execution of MM for any matrix size, eliminating these checks does not affect the overall execution. 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% (median of 3%), as seen in Figure 3.7.

3.3.2 Cluster Performance

Figures 3.8 and 3.9 compare the absolute time pMM spends in communication (in milliseconds) using different configurations of Javia-I and using Javia-II (labeled *jbufs*). The input matrices used are 64x64 and 256x256 doubles and the benchmark is run on eight processors. Total communication time is obtained by commenting out the computation phase of pMM. The cost of barrier synchronization is measured by skipping both communication and computation phases. *Jbufs*' communication time is consistently smaller than the rest: with

256x256 matrices, where message payload is 2048 bytes, *jbufs* spent 25% less time than *copy-async* in communication, as predicted by micro-benchmarks. Figures 3.8 and 3.9 also show the percentage of the total execution time attributed to communication (on top of each bar). For an input size of 64x64, this percentage is around 73% (median) for *jdk-copy-async*, with a high of near 85% for *pin-async* and a low of 56% for *jbufs*. For 256x256, the median percentage is around 20%, with a low of 13% for jbufs.

Figure 3.11 shows that the overall performance of pMM using 256x256 matrices correlates well with the communication performance seen in Figure 3.9. A peak performance of 320Mflops is attained by *jbufs*, followed by 275Mflops attained by *copy-alloc* on eight processors. Jbufs consistently outperform the other versions on two and four processors as well. However, this "nice" correlation is not the case for 64x64 matrices, as shown in Figure 3.10: a peak performance of 175Mflops goes to *copy-async* on four processors. In fact, the overall performance of *jbufs* is inferior to those with Javia-I on two processors. These results are most likely due to cache effects. This is a clear indication that, at this point, faster communication in Java does not necessarily lead to better overall performance of parallel, numerically-intensive applications (in particular, those with level-3 BLAS operations).

Another interesting data point is that allocating an array on every message reception can actually improve locality. For example, although *copy-alloc* spends about 15% more time than *copy-async* in communication on eight processors (Figure 3.9), *copy-alloc*'s Mflops is 10% higher than that of *copy-async* (Figure 3.11).

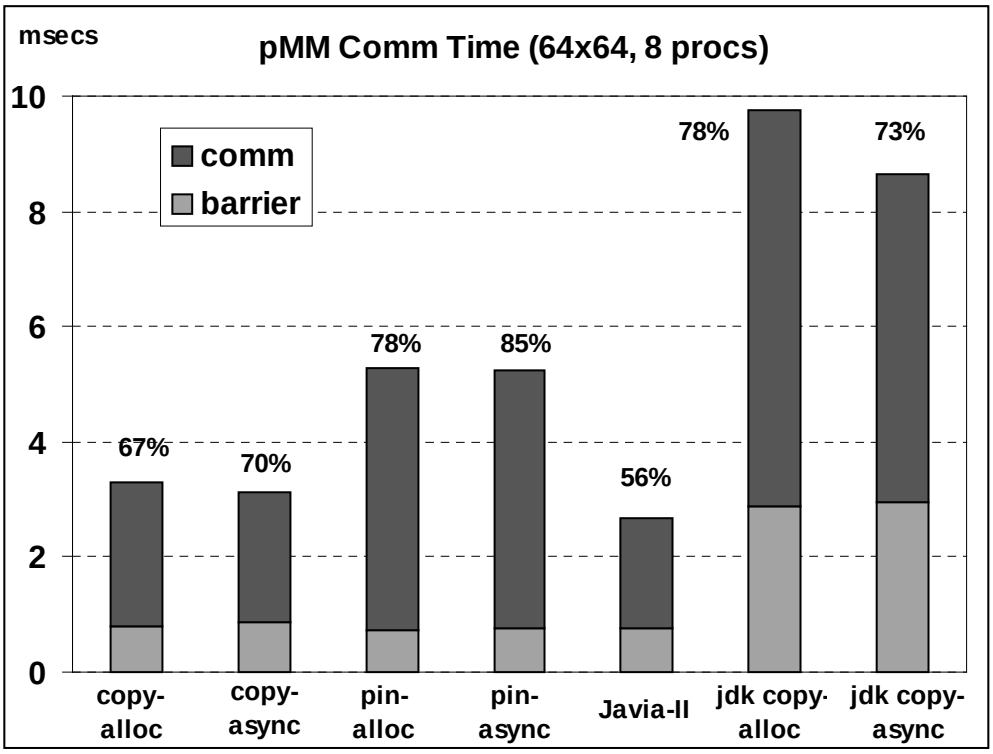

Figure 3.8 Communication time of pMM (64x64 mat., 8 processors)

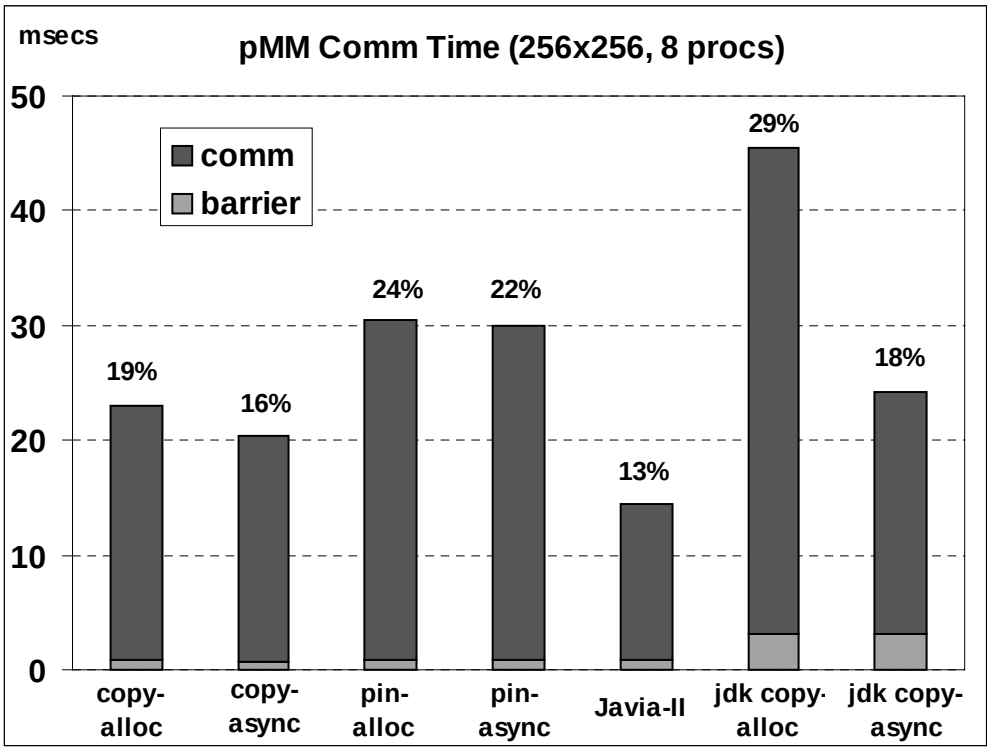

Figure 3.9 Communication time of pMM (256x256 mat., 8 processors)

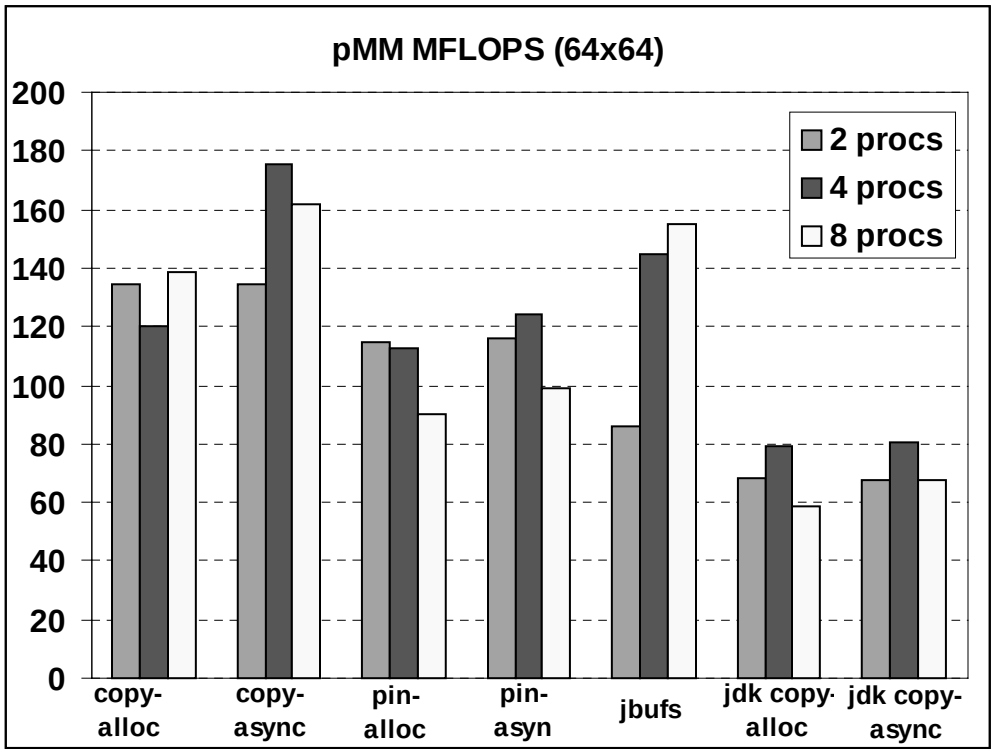

Figure 3.10 Overall performance of pMM (64x64 mat., 8 processors)

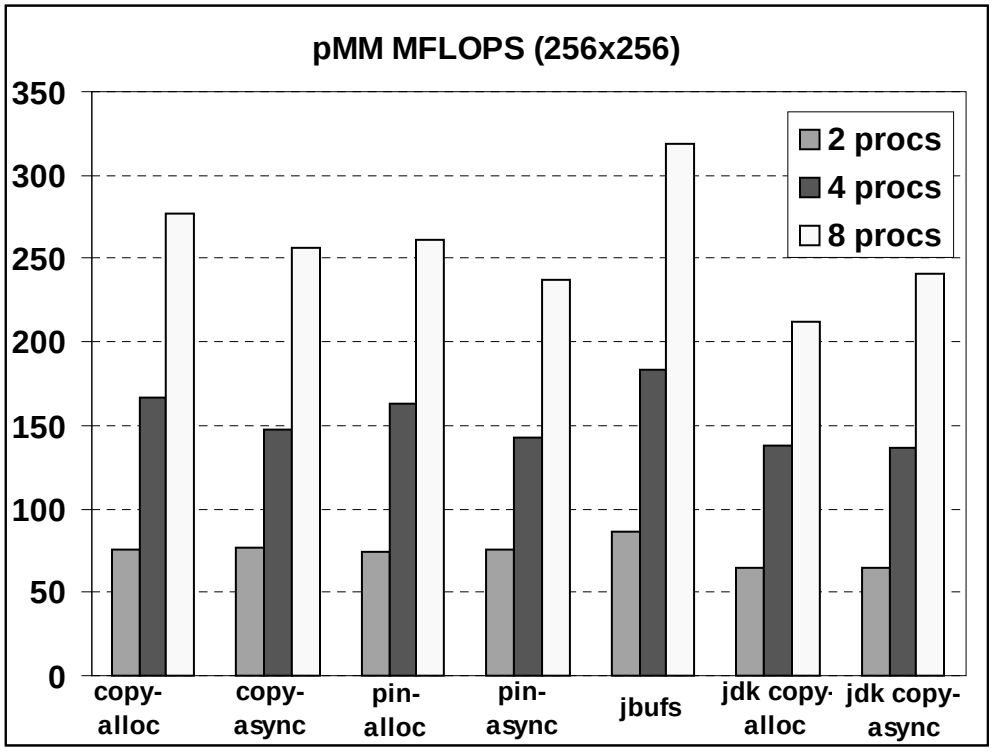

Figure 3.11 Overall performance of pMM (256x256 mat., 8 processors)

3.4 Jam: Active Messages for Java

Active messages [vECS+92] are a portable instruction set for communication. Its primitives map efficiently onto lower-level network hardware and compose well into higher-level protocols and applications. The central idea of active messages is to incorporate incoming data quickly into the ongoing computation by invoking a handler upon a message arrival. Initially developed for the CM-5 and later ported onto many other multi-computers [SS95, CCvE96, KSS+96], the original specification (called Generic Active Messages [CKL+94], or Gam, version 1.0+) was inadequate for mainstream cluster computing. It relied heavily on the single-program-multiple-data (SPMD) execution model supported by most parallel computers. For example, active message users tag request messages with a destination processor id and refer to remote handlers and memory locations by their virtual addresses.

The second version of active messages, AM-II [MC95], is more general and better suited for cluster computing. It uses a flexible naming scheme that is not bound to a particular execution model (e.g. SPMD or gang-scheduling), network configuration, and name service implementation. It adopts a connection-oriented protection model that enables multiple applications to access the network devices simultaneously and in a protected fashion. It also introduces a descriptive error model that goes beyond the rudimentary “all-or-nothing” fault model and provides synchronization support for thread-safe multiprogramming.

The main data structure in AM-II is an *endpoint*. An endpoint is like a user or kernel port with a tag and a global name associated with it. An endpoint contains send and receive message pools for small messages, a handler table that maps integers to (local) function pointers, a translation table that

maps integers to remote endpoints, and a virtual memory segment for bulk transfers. Endpoints are aggregated in bundles in order to avoid deadlock scenarios [MC95]: incoming messages for endpoints within a bundle are serviced atomically.

The following subsection provides a brief description of Jam, an implementation of AM-II over Javia-I/II.

3.4.1 Basic Architecture

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. Sequence numbers are used to keep track of packet losses and a sliding window is used for flow control; unacknowledged messages are saved by the sending endpoint for retransmissions. When a message with the wrong sequence number is received, it is dropped and a negative acknowledgement is returned to the sender, forcing a retransmission of the missing as well as subsequent messages. Acknowledgements are piggybacked

onto requests and replies whenever possible; otherwise explicit acknowledgements are issued when one quarter of the window remains unacknowledged.

3.4.2 Bulk Transfers: Re-Using Jbufs

A key design issue in Jam is how to provide an adequate bulk transfer interface to Java programmers. In the Gam specification, the sender specifies a virtual address into which data should be transferred. 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 specification is well suited for C but is ill matched to Java. Integer offsets would have to be offsets into Java arrays; assuming no extra copying of data, having to operate on arrays using offsets 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. While this design incurs allocation and copying overheads, it works with any GC scheme and fits 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 jbufs to an endpoint. The endpoint manages this list as a pool of receive buffers for bulk transfers and associates it with a separate virtual interface connection. Upon bulk data arrival, the dispatcher obtains a Java array reference from the receiving jbuf and passes that reference directly to the handler. The receiving jbuf is `unRef`ed after the handler's execution. When the pool is (about to be) empty, the dispatcher reclaims jbufs 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 jbufs: if the jbufs 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:

```
1  class First extends AM_Handler {
2    private byte first;
3    void handler(Token t, byte[] data, . . .) {
4      first = data[0];
5    }
6  }
7  class Enqueue extends AM_Handler {
8    private Queue q;
9    void handler(Token t, byte[] data, . . . ) {
10     q.enq(data);
11   }
12 }
```

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.

3.4.3 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.

3.4.4 Performance

A simple ping-pong benchmark using `AM_Request0` and `AM_Reply0` shows a 0-byte round-trip latency of 31µs, about 11µs higher than that of Javia-II (Figure 3.12). This value increases by about 0.5µs for every four additional words. For large messages, Jam round-trip latency is within 25µs of Javia-II and has the same per-byte cost. Additional overheads include:

- an extra pair of send/receive overheads (due to two separate VI connections: one for small messages, another for bulk transfers);
- synchronized access to bundle and endpoint structures;
- handler and translation table lookup, and
- protocol processing (header parsing and flow control).

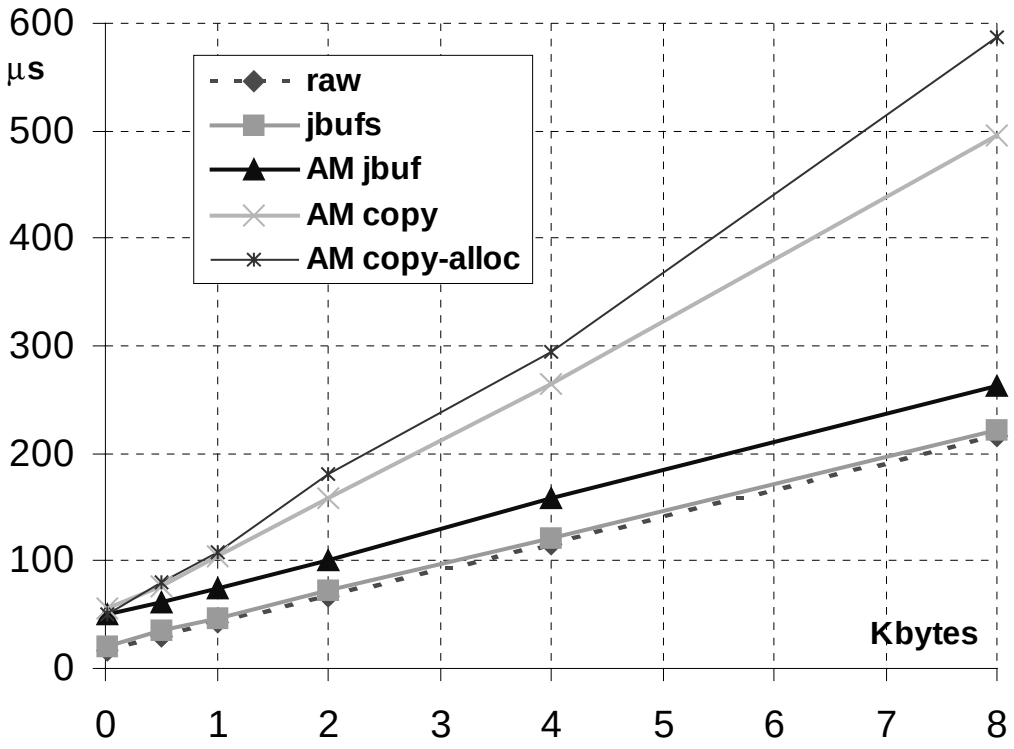

Figure 3.12 Jam round-trip latencies

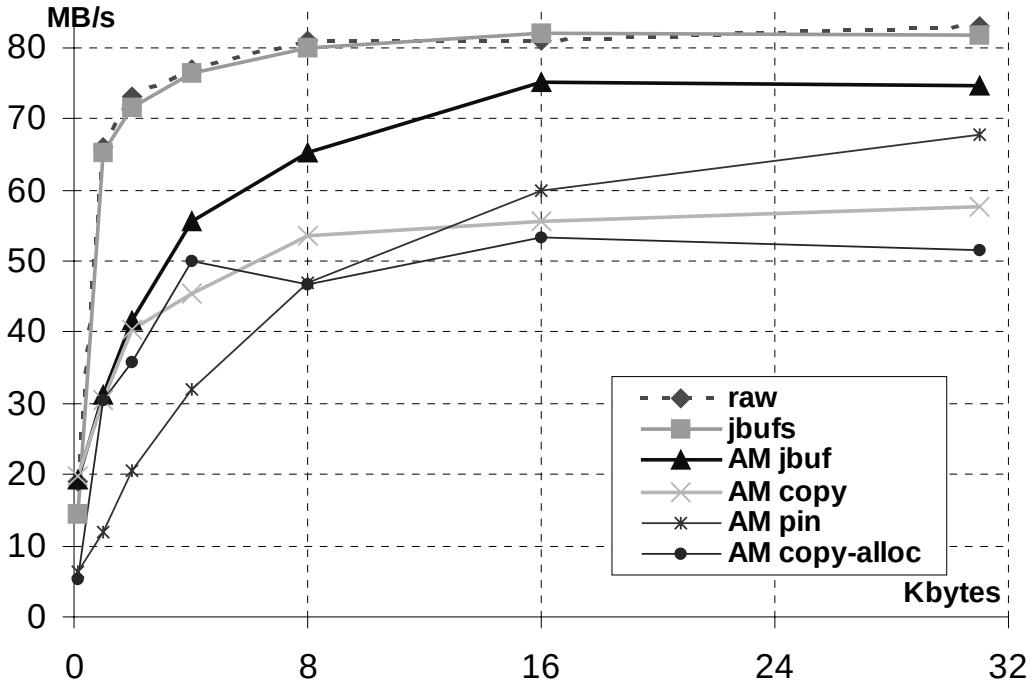

Figure 3.13 Jam effective bandwidth

Jam achieves an effective bandwidth of 75MBytes/s, within 5% of Javia-II, as seen in Figure 3.13.

3.5 Summary

Jbufs are Java-level buffers that can be managed explicitly by applications without breaking the language. By controlling whether jbufs are subject to garbage collection, the separation between GC and native heap becomes dynamic. Javia-II exploits this fact to make nearly 100% of the raw network performance available to Java.

The benefits of accessing jbufs via genuine array references should be clear: it eliminates indirect access via method invocations, promotes code reuse of large numerical kernels, and leverages optimization infrastructure for eliminating array-related safety checks. The latter benefit is currently difficult

to substantiate experimentally because Java compilation technology is still too immature despite dramatic progress over the last two years. For example, Marmot's rudimentary schemes to eliminate array-bound checks fail to remove any of the checks encountered in a simple level 3 BLAS loop used by pMM.

Unlike in pMM, where jbufs are allocated at program initialization and de-allocated at its termination, the ability to manage jbufs explicitly has helped in the design and implementation of Jam tremendously. In situations where a copying collector is used, Jam is able to defer data copying to GC time. An important concern is that a messaging layer may have to trigger a garbage collection whenever it needs to reclaim jbufs for re-use, which could be counter-productive. Instead of a "one-size-fits-all" solution, Jam lets users decide how frequently jbuf reclamation occurs by having them supply a list of receive jbufs to an endpoint. Users can utilize application-specific information to fine-tune performance.

Although one can explicitly manage jbufs, our experience indicates that jbufs are still not as flexible as C buffers. For example, during protocol processing in Jam, it would have been convenient to access different parts of a jbuf with different array types (e.g. accessing the first 10 words of the jbuf as `int` arrays, and pass the remaining to the handler as a `byte` array). Currently, Jam uses two message pools, one for small messages (i.e. the entire message can be treated as a protocol "header"), and another for bulk payload, so it can assign two different jbufs to each. This leads to extra cost to active messages round-rip latency.

Another concern is that location control may produce stale references into jbufs. Our experience so far indicates that stale references are not an issue.

Jbuf management has not been stressed in pMM—the main benefit of jbufs there is the ability to transfer arrays with zero-copy. Jam has a rather centralized control over jbuf management since it is a communication layer. It remains to be seen whether stale references will cause much headaches to Java programmers.

3.6 Related Work

3.6.1 Pinned Java Objects

Two closely-related projects have recognized the need to access the native (i.e. pinned) heap from Java: Microsoft's J/Direct technology and Berkeley's Jaguar project.

1.1.1.1 Microsoft J/Direct

In addition to the features introduced in Section 2.2.2, J/Direct allows Java applications to define pinned, non-collectable objects using source-level annotations (i.e. `/** @dll.struct */`). Programmers must manually supply the C data type that corresponds to the annotated Java object and must allocate them in C. After allocation, these objects can passed between Java and C by reference (as an `int` handle). To use them from Java, J/Direct provides functions converts a handle into a genuine Java object (`dllLib.ptrToStruct`).

The implementation of `dllLib.ptrToStruct` allocates a "mirror" Java object—the JIT compiler re-directs read and write operations on the mirror object to the pinned object. This re-direction incurs a level of indirection (i.e. looking up the pinned object), which is prohibitively expensive (about 10x higher than a regular Java array access). Jbufs allows the VI architecture to access pinned communication buffers (e.g. in the referenced state) and lets appli-

cations read/write from/into these buffers through array references. These accesses are only subjected to the safety checks already imposed by the Java language.

1.1.1.2 Jaguar

The Jaguar project [WC99] essentially overcomes the level of indirection that plagues J/Direct: 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. Unlike J/Direct, these external objects are allocated from Java. 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-bound checks when accessing an external array. This is not a concern with jbufs because accesses go through genuine array references.

Another difference between the Jaguar and the Jbufs approaches 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. Instead, Javia-I/II focuses only on large data transfers—the rationale is that control structures are often "small" enough so the data to be written can be passed as native method arguments. For example, Javia-I/II passes control information as byte or word arguments to native methods and uses tem to update VI descriptors. This would avoid fetching that data from native code—though inexpensive in Marmot, it is rather costly in JDK.

1.1.1.3 Other approaches based on custom JVMs

Many JVMs support in one way or another pinning of objects, mostly for performance reasons. For example, Microsoft's *jview* allocates large arrays (> 10Kbytes) in a pinned heap so they are not moved by its generational copying collector. Systems like Javia can be integrated into JVMs with non-copying GCs (such as KaffeVM [Kaf97]) with undue effort and are likely to achieve good performance. The problem is that Java applications that interact with network devices directly are (and should be) oblivious to the underlying GC scheme. Jbufs incorporates user-managed buffers safely into any garbage-collected environment: all that is required from the GC is the ability to dynamically change the scope of the GC heap.

It is not possible to attain safe and explicit memory management without adequate GC or finalization support. Array "factories" that produce arrays outside of the GC heap would leak memory unless there is finalization of array objects. Explicit de-allocation of such arrays would violate memory safety unless references are tracked appropriately.

The real motivation for explicit management in jbufs is that it provides a clean framework for optimization de-serialization of Java objects. Neither J/Direct nor Jaguar can provide zero-copy de-serialization without introducing certain "restrictions" to the de-serialized objects. Jbufs allows incoming, *arbitrary* Java objects to be integrated into a JVM without violating its integrity. This is the subject of Chapter 5.

3.6.2 Safe Memory Management

The central motivation for developing jbufs, namely zero-copy data transfers, differs from that of most explicit allocation and de-allocation proposals, which is to improve data locality. Ross [Ros67] presents a storage package that lets

applications allocate objects in zones. Each zone has a different allocation policy and de-allocation is on a per-object basis. Vo [Vo96] introduces a similar library named *Vmalloc*: objects are allocated in regions, each with a different allocation policy. Some regions allow per-object de-allocation, while others de-allocates them all at once (by freeing a region). None of the above approaches attempts to provide safety along with explicit memory management. Surveys on explicit memory management and garbage collectors can be found in [WJN+95] and [Wil92] respectively.

Gay and Aiken [GA98] propose explicit memory management with *safe regions*. Objects are allocated in regions, and de-allocating a region frees all objects within that region. De-allocation is made safe by keeping a reference count for each region. They rely on compiler support to generate code that performs reference counting; jbufs, on the other hand, requires no compiler assistance and relies on the underlying GC. Stoutamire [Sto97] defines regions of memory (or *zones*) that are mapped efficiently onto hardware abstractions such as a page or even a cache line. Zones are first-class objects in the *Sather* programming language in order to enable explicit programming for locality. Memory reclamation is on a per-object basis using a non-copying (mark-and-sweep) GC. The authors of safe regions and zones do not consider scenarios in which copying GC techniques might be employed. Jbufs attain explicit memory management in the presence of non-copying as well as copying GC schemes.