SAFE AND EFFICIENT CLUSTER COMMUNICATION IN JAVA USING EXPLICIT MEMORY MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Chi-Chao Chang

January 2000

SAFE AND EFFICIENT CLUSTER COMMUNICATION IN JAVA USING EXPLICIT MEMORY MANAGEMENT

Chi-Chao Chang, Ph.D.
Cornell University 2000

This thesis presents a framework for using explicit memory management to improve the communication performance of Java™ cluster applications. The framework allows programmers to explicitly manage Java communication buffers, called *jbufs*, which are directly accessed by the DMA engines of high-performance network interfaces and by Java programs as primitive-typed arrays. The central idea is to remove the hard separation between Java's garbage-collected heap and the non-collected memory region in which DMA buffers must normally be allocated. The programmer controls when a jbuf is part of the garbage-collected heap so that the garbage collector can ensure it is safely re-used or de-allocated, and when it is not so it can be used for DMA transfers. Unlike other techniques, jbufs preserve Java's storage- and type-safety and do not depend on a particular garbage collection scheme.

The safety, efficiency, and programmability of jbufs are demonstrated throughout this thesis with implementations of an interface to the Virtual Interface Architecture, of an Active Messages communication layer, and of Java Remote Method Invocation (RMI). The impact on applications is also evalu-

ated using an implementation of cluster matrix multiplication as well as a publicly available RMI benchmark suite.

The thesis proposes *in-place object de-serialization*—de-serialization without allocation and copying of objects—to further enhance the performance of RMI on homogeneous clusters. This optimization takes advantage of the zero-copy capabilities of network devices to reduce the per-object de-serialization costs to a constant irrespective of object size, which is particularly beneficial for large objects such as arrays. In-place de-serialization is realized using *jstreams*, an extension of jbufs with object I/O streams. Jstreams use the explicit memory management offered by jbufs to incorporate de-serialized objects into the receiving Java virtual machine without compromising its integrity, without restricting the usage of those objects, and without making assumptions about the underlying garbage collection scheme. The performance impact of jstreams on Java RMI and the benchmark suite is evaluated.

Biographical Sketch

Chi-Chao was born on March 31, 1971, in Taipei, Taiwan, and grew up in Rio de Janeiro, Brazil, where he did basic schooling. In his childhood years he developed a keen interest in history, geography, and soccer but kept up with his math and science. He attended Colégio Marista São José, a catholic school, for 10 years and graduated in December of 1988. He entered the Federal University of Rio de Janeiro in the following year, but soon transferred to the University of Kansas, Lawrence, where he received a bachelor's degree in Electrical Engineering in May of 1993. On that beautiful campus, he met Marcia in the Fall of 1990.

He moved to Ithaca, New York, soon after graduation and entered the Ph.D. program in Computer Science at Cornell University. He was awarded a M.S. in Computer Science in August of 1996, and will receive his Ph.D in January of 2000. He is moving to sunny California, where he will join Yahoo! in September of 1999. Chi-Chao and Marcia will be wedded in the Summer of 2000.

To my parents.

Acknowledgements

I would like to thank my advisor, Thorsten von Eicken, for his constant support and guidance over the last five years; Keshav Pingali and my minor advisor in Finance, Peter Carr, for serving on my committee. Also thanks to Haitao Li, who agreed to serve as a proxy for Peter in my Ph.D. defense.

I also owe a debt of gratitude to my immediate colleagues: Grzegorz Czajkowski, Chris Hawblitzel, and Deyu Hu. Grzegorz was my officemate and principal collaborator during our early years in graduate school, when we both worked on a variety of parallel computing projects. All three provided invaluable help and were wonderful co-workers during the Safe Language Kernel endeavor. The work presented in this thesis originated from my previous experience with high-performance communication and our recent efforts in building extensible systems using safe languages such as Java.

I am indebted to the Phi Kappa Phi Honor Society and to CNPq/Brazil for providing me with graduate fellowships. I am also thankful to the Instituto Brasil-Estados Unidos (IBEU) for awarding the undergraduate scholarships that brought me to this country ten years ago. In particular, thanks to Rose Turano and Neide Monteiro of IBEU for their invaluable assistance during those years.

My thanks to all the friends I have had in Ithaca over the last six years. Special thanks to Vijay Menon and Ozan Hafizogullari for three years of memories in apartment K1; to Flávio Rezende and Henrique Araújo for their friendship and for teaching me the beautiful game of chess; to Roberto Malamut and Raul Casas who constantly remind me that rules are made to be broken; to my first year classmates who helped me learn Computer Science and to whom I now say, “The lights are off and the door is shut;” to many of my soccer team-mates from Cornell and from the local league; and to many of my tennis, bridge, and sheepshead partners.

And last, but not least, thanks to my family for their love and support, and to my beloved Marcia, for being my daily sunshine over the last nine and the next ninety years, and without whom none of this would matter.

Table of Contents

List of Tables

List of Figures

1 Introduction

Until recently, the performance of Java™ networking has not been a major concern. To begin with, Java programs run more slowly than comparable C or C++ programs, suggesting that the performance bottleneck of most applications may not yet be communication, but computation. Furthermore, distributed computing is largely based on Java Remote Method Invocation, which is designed first for flexibility and interoperability in heterogeneous environments and only second for performance. Because Java has been mainly used for applications running on wide-area networks (i.e. the Internet), the level of performance delivered by high-speed networks has not been particularly interesting.

The growing interest in using Java for high-performance cluster applications [JG98] has sparked the need for improving its communication performance on a cluster of workstations. These clusters are typically composed of homogeneous, off-the-shelf PCs equipped with high-performance network interfaces [Via97, Gig98] and connected by low-cost network fabrics with over 1Gbps of bandwidth. Recent research in just-in-time and static compilers, virtual machine implementations, and garbage collectors for Java have delivered promising results, reducing the performance gap between Java and C programs [ACL+98, ADM+98, BKM+98, FKR+99, MMG98].

Many attribute Java's success to its being a "better C++": not only is it object-oriented but it is also a *safe* language. By a safe language we mean one that is *storage* and *type* safe. Storage safety guarantees that no storage will be prematurely disposed of, whether at the explicit request of the programmer or implicitly by the runtime system [Ten81][1]. Storage safety spares the programmer from de-allocating objects explicitly and tracking object references: inaccessible objects are automatically searched for and de-allocated by a *garbage collector* whenever more storage is needed. Type safety ensures that references to Java objects cannot be forged, so that a program can access an object only as specified by the object's type and only if a reference to that object is explicitly obtained. Type safety is enforced by a combination of compile-time and runtime checks[2].

The goal of this thesis is to address two inefficiencies that arise when interfacing Java to the underlying networking hardware. First, storage safety in Java creates a hard separation between Java's heap, which is garbage-collected, and the native heap[3], which is not. In modern clusters of PCs, network interfaces make the raw performance of high-speed network fabrics—low message latency and high bandwidth—available to applications. The key advance is that data transfers between the network and application memory are performed by the DMA engines of network devices, offloading the host

---

[1]Another aspect of storage safety is to ensure that the contents of a location are never accessed before its initialization (Sections 12.4 and 12.5 [LY97]).

[2] Java virtual machines perform various runtime safety checks such as array bounds, array stores, null pointer, and down-casting checks.

[3] Native code (i.e. C, C++, or assembler code) is needed to interact with hardware devices directly or through native libraries.

processor. Memory pages in which data resides must be present in (or "pinned onto") the physical memory so they can be directly accessed by those DMA engines. This is ill matched to a garbage-collected system, where objects can be moved around without the application and the device's knowledge. Objects must be copied into memory regions in a native, non-collected heap. The result is that the performance benefits of direct DMA access are reduced substantially: 10% to 40% of the raw bandwidth can be lost.

Second, Java objects must be serialized and de-serialized across the network. Because type safety cannot be entirely enforced at compile-time, even the simplest objects such as arrays carry meta-data for runtime safety checks[4]. Meta-data complicates the in-memory layout of objects and calls for serialization of objects across the network. Serialization is an expensive operation since the entire object and its typing information must be copied onto the wire. De-serialization is equally expensive because types must be checked, new storage allocated, and data must be copied from the wire and into the newly allocated storage. Due to high serialization costs, the performance of popular communication paradigms such as RMI is over an order of magnitude worse in Java than in an unsafe language like C.

Recent efforts for improving the performance of Java communication have fallen into what will call "top-down" and "front-end" categories. The top-down approach consists of implementing a high-level Java communication abstraction (e.g. RMI) on top of a native communication library [BDV+98, MNV+99]. The front-end approach consists of providing "glue-code" to high-level native communication libraries (e.g. MPI), making them accessible from

---

[4] Incidentally, storage safety also generates meta-data, such as information for type-accurate garbage collection.

Java [GFH+98]. Both approaches in general yield reasonable performance, but generally lead to solutions that are specific to a particular abstraction, Java virtual machine implementation, and/or communication library.

This thesis pursues a "bottom-up" approach: Java is interfaced directly to the underlying network devices and higher level communications are built in Java on top of the low-level interface. By cutting through layers of abstractions, this thesis focuses on the inefficiencies in the data path that are *inherent* to the interaction between safe languages and hardware. Overheads in the control transfer path—context switching, scheduling, and software interrupts—are not fundamental to the language. Rather, they are associated to a particular communication model (e.g. RMI) and depend on the host operating system as well as the machine load.

1.1 Thesis Contribution

The main contribution is the thesis is a framework for using explicit memory management to improve the communication performance in Java. The framework is motivated by recent trends in network interface design, where applications, rather than devices, have full control over communication buffers and control structures. Because certain management operations (such as mapping user memory onto physical memory) can be very costly, the ability to *re-use* those data structures becomes paramount: programmers can amortize the high costs using application-specific information.

1.1.1 Jbufs: Safe and Explicit Management of Buffers

Jbufs are Java-level communication buffers that are directly accessed by the DMA engines of network interfaces and by Java programs as primitive-typed arrays. The central idea is to remove the hard separation between Java's garbage-collected heap and the non-collected memory region in which DMA buffers must normally be allocated. The programmer controls when a jbuf is part of the garbage-collected heap so that the garbage collector can ensure it is safely re-used or de-allocated, and when it is not, so it can be used for DMA transfers. Unlike other techniques, jbufs preserve Java's storage- and type-safety and do not depend on a particular garbage collection scheme.

The safety, efficiency, and programmability of jbufs are demonstrated throughout this thesis with implementations of an interface to the Virtual Interface architecture [Via97], of an Active Messages communication layer [MC95], and of Java Remote Method Invocation (RMI) [Rmi99]. The impact on applications is also evaluated using an implementation of cluster matrix multiplication as well as a publicly available RMI benchmark suite [NMB+99].

The impact of jbufs in basic, point-to-point communication performance is substantial: almost 100% of the raw performance of a commercial network interface is made available from Java. Results show that explicit memory management (i) outperforms conventional techniques in a cluster matrix multiplication application by at least 10%, and (ii) helps an active messages layer attain nearly 95% of the raw network capacity.

1.1.2 Jstreams: Optimizing Serialization for Cluster Applications

To further enhance the performance of RMI on homogeneous clusters, the thesis proposes *in-place object de-serialization*: de-serialization without allocation and copying of objects. This optimization takes advantage of the zero-copy capabilities of network devices to reduce the per-object de-serialization to a constant cost irrespective of object size, which is particularly beneficial for large objects such as arrays. In-place de-serialization is realized using *jstreams*, an extension of jbufs with object I/O streams. Jstreams use the explicit memory management offered by jbufs to incorporate de-serialized objects into the receiving Java virtual machine without compromising its integrity, without restricting the usage of those objects, and without making assumptions about the underlying garbage collection scheme.

The performance impact of jstreams on Java RMI and the benchmark suite is evaluated. Results show that jstreams improve the point-to-point performance of RMI by an order of magnitude, within a factor of two of the raw performance. This translates to an improvement of 2% to as much as 10% in the overall performance of several Java cluster applications.

1.1.3 Overview of Related Approaches

Several projects have recognized the importance of accessing non-collected memory regions from Java for enhanced performance. One common approach is to allocate specially annotated Java objects outside of the garbage-collected heap [Jd97]: these objects can be passed between Java and C programs by reference (without copy). Efficient implementations of this approach [WC99] achieve the same level of performance as that attained by jbufs when used for high-performance communication. However, these “external” or “pinned” objects require special annotation and are typically restricted in some way or another (e.g. they cannot contain pointers to other objects). Another approach is to integrate high-performance communication directly into custom Java virtual machines [MNV+99] with non-copying garbage collectors. Jbufs and jstreams impose no restrictions on the type or usage of Java objects, and can be implemented with both non-copying and copying garbage collectors.

The poor performance of Java object serialization and RMI is well known [Jg99] and is in part attributed to inefficient implementations in publicly available Java systems [NPH99]. The in-place object de-serialization presented in this thesis attempts to aggressively optimize this operation for the common case: homogeneous clusters of workstations equipped with zero-copy network devices.

Most proposals for using explicit memory management are largely aimed at improving the data locality and cache behavior of applications [Ros67, Han90, BZ93, Sto97, GA98]. Some projects also consider safe aspects of explicit memory management (e.g. safe regions [GA98] and mark-and-sweep zones [Sto97]) and rely on reference counting and non-copying garbage collectors. The framework proposed in this thesis considers explicit memory management in the presence of copying collectors as well.

1.2 Thesis Overview

Chapter 2 discusses the performance issues that arise when interfacing Java with the user-level network interfaces. First, it gives the necessary background on the Virtual Interface architecture, an industry standard of network interfaces, focusing on the features that are critical to high performance and the programming language requirements for applications to capitalize on these features. Second, it looks at conventional mechanisms for interfacing Java with native code. It contrasts two central issues—the hard separation between the garbage-collected and native heaps and the tension between efficiency and portability—and argues that the former is inherent to the language/hardware interaction. To motivate explicit buffer management, the chapter concludes with an evaluation a Java interface to the VI architecture (Javia-I) that respects the heap separation.

Chapter 3 introduces jbufs and provides an in-depth look at their safety properties and the interaction with the garbage collector. It justifies the need for explicit de-allocation and describes an implementation in the Marmot system with a copying collector. To demonstrate the use of jbufs as a framework for communication, the chapter presents implementations of Javia-II (an improvement over Javia-I), of a cluster matrix multiplication program, and of a messaging layer based on the Active Messages protocol. It concludes with a discussion on our experiences in using jbufs and on their limitations.

Chapter 4 makes a case for specializing object serialization for homogeneous cluster computing. It presents an evaluation of several implementations of the standard serialization protocol (Java Object Serialization) and analyses the impact of Marmot's static Java compiler on the costs of serialization. It studies the effect of these costs in the performance of a Java RMI system over Javia-I/II and of a benchmark suite consisting of six applications.

Chapter 5 introduces the in-place de-serialization technique and shows how it can be realized using jstreams. It describes additional safety constraints that must be met in order not to violate the integrity of the Java system on which de-serialization is taking place. It evaluates a prototype implementation of jstreams in Marmot and their impact in the performance of Java RMI and several applications. The chapter concludes with a discussion on our experiences in using jstreams.

Chapter 6 concludes the thesis with a summary and a discussion of some open-ended issues.

2 Interfacing Java to Network Interfaces

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

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

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

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

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

2.1 Background

2.1.1 The Virtual Interface Architecture

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

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

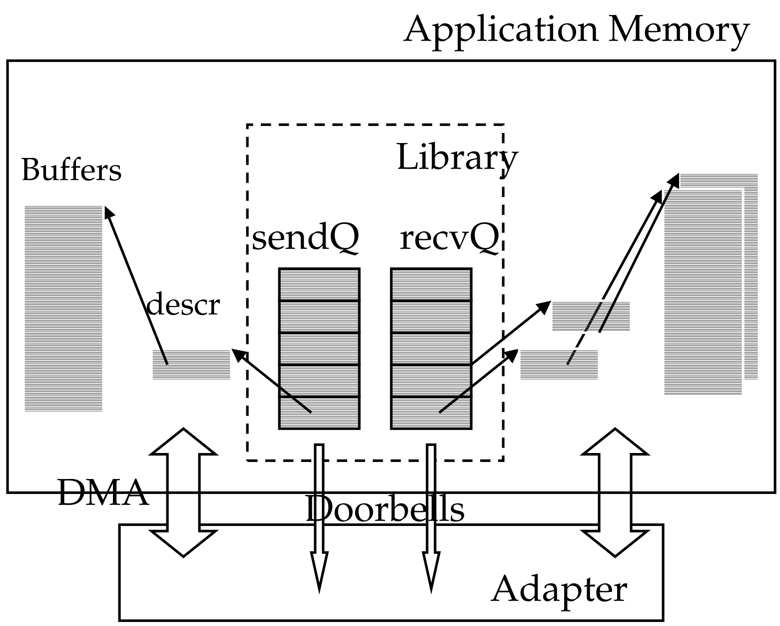

Figure 2.1 Virtual Interface data structures. Shaded structures must be pinned onto the physical memory so they can be accessed by DMA engines.

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

To send a message, an application composes the message in a buffer, builds a buffer descriptor, and adds it to the end of the send queue. The network interface fetches the descriptor, transmits the message using DMA, and sets a bit in the descriptor to signal completion. An application eventually checks the descriptors for completion (e.g. by polling) and dequeues them. Similarly, for reception, an application adds descriptors for free buffers to the end of the receive queue, and checks (polls) the descriptors for completion. The network interface fills these buffers as messages arrive and sets completion bits. Incoming packets that arrive at an empty receive queue are discarded. An application is permitted to poll at multiple receive queues at a time using VI completion queues. Apart from polling, the architecture also supports for interrupt-driven reception by posting notification handlers on completion queues.

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

2.1.2 Giganet cLAN™ GNN-1000 Cluster

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

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

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

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

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

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

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

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

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

The VI architecture adopts an explicit memory mapping approach that was first pursued by the Hamlyn [BJM+96] project and later by the Shrimp/VMMC [DBC+98] project. Applications are responsible for "registering" (`VipRegisterMemory`) and "de-registering" (`VipDeregisterMemory`) memory regions (in which user buffers and descriptors reside) with the VI architecture. The registration is initiated by the user and performed by the operating system, which pins the pages underlying the region and communicates the physical addresses to the network interface. The latter stores the translation in a table indexed by a region number. While all addresses in descriptors are virtual, the application is required to indicate the number of the region with each address (in effect all addresses are 64 bits consisting of a 32-bit region number and a 32-bit virtual address) so that the network interface can translate the addresses using its mapping table. De-registration undoes the above process: buffer and descriptor pages can be paged out and the virtual/physical mapping is dropped by the VI architecture.

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

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

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

2.1.4 Java: A Safe Language

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

---

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

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

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

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

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

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

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

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

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

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

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

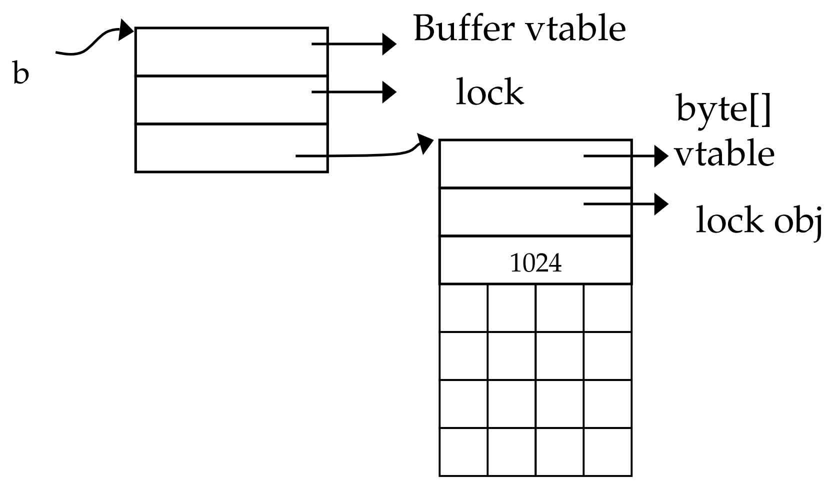

Figure 2.2 Typical in-memory representation of a Buffer object. Each Buffer object has two words of meta-data: one for the method dispatch table (from which the class object can be reached), and another for the monitor object. An array object also keeps the length of the array for runtime checks.

2.1.5 Separation between Garbage-Collected and Native Heaps

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

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

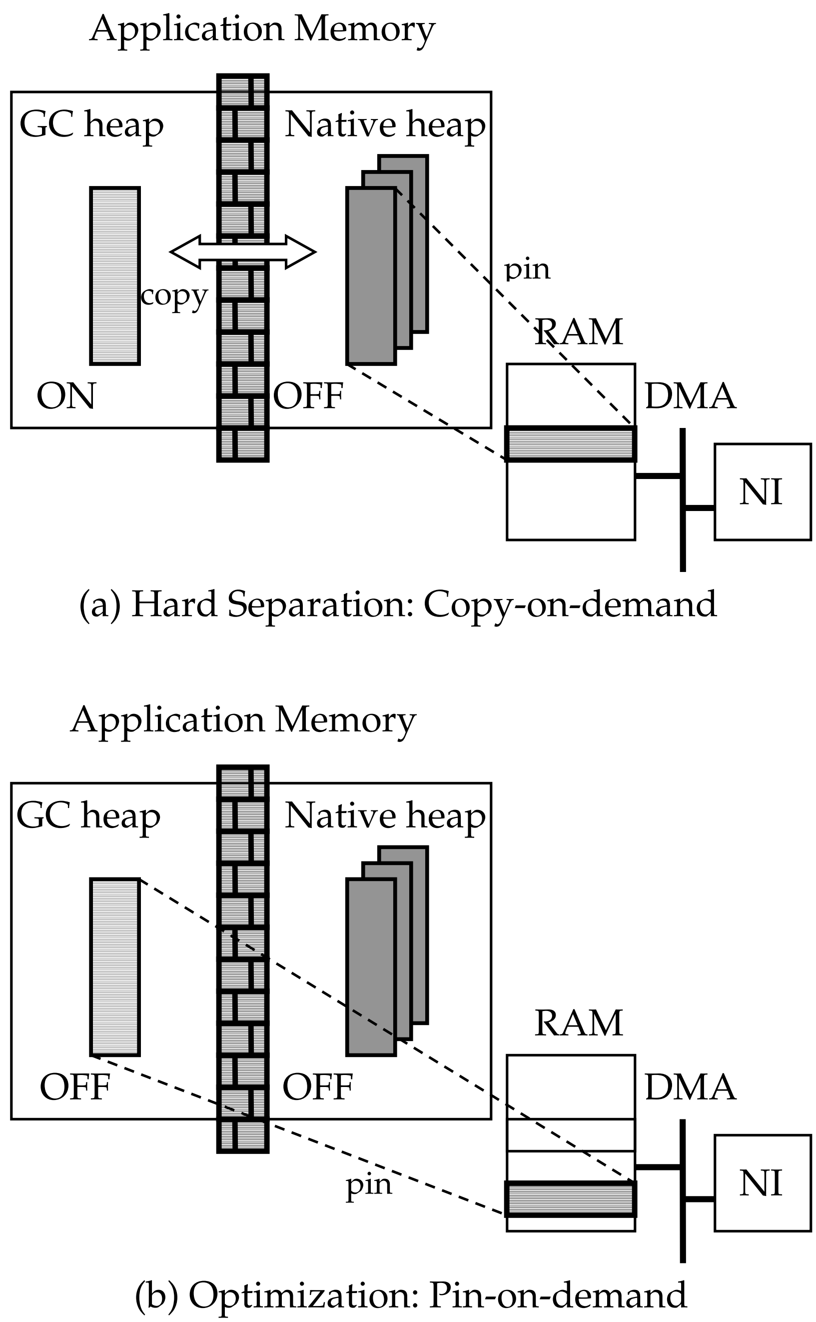

Figure 2.3 The hard separation between GC and native heaps. (a) Data has to be copied on demand from Java arrays into pinned native buffers so they can be accessed directly by the DMA engine. (b) GC can be disabled for a "short" time so Java arrays can be pinned and made visible to the DMA on demand.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

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

2.2 Java-Native Interfaces

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

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

---

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

2.2.1 The Marmot System

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

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

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

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

---

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

2.2.2 J/Direct

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

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

2.2.3 Java Native Interface (JNI)

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

---

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

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

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

2.2.4 Performance Comparison

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

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

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

Table 2.2 Cost of Java-to-C downcalls

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

Table 2.3 Cost of C-to-Java upcalls

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

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

Table 2.4 Cost of accessing Java fields from C

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

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

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

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

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

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

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

2.2.5 Summary

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

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

2.3 Javia-I : Interfacing Java to the VI Architecture

2.3.1 Basic Architecture

The general Javia-I architecture consists of a set of Java classes and a native library. The Java classes are used by applications and interface with a commercial VIA implementation through the native library. The core Javia-I classes are shown below:

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

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

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

  /* async recv */
  public void recvPost(ViBATicket t);
```

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

The class `Vi` represents a connection to a remote VI and borrows the connection set-up model from the JDK sockets API. When an instance of `Vi` is created a connection request is sent to the remote machine (specified by `ViAddress`) with a tag. A call to `ViServer.accept` (not shown) accepts the connection and returns a new `Vi` on the remote end. If there is no matching accept, the `Vi` constructor throws an exception.

Javia-I contains methods to send and receive Java byte arrays[12]. The asynchronous calls (lines 6-11) use a Java-level descriptor (`ViBATicket`, lines 20-24) to hold a reference to the byte array being sent or received and other information such as the completion status, the transmission length, offset, and a 32-bit tag. Figure 2.4 shows the Java and native data structures involved during asynchronous sends and receives. Buffers and descriptors are managed (pre-allocated and pre-pinned) in native code and a pair of send and receive ticket rings is maintained in Java and used to mirror the VI queues.

To post a Java byte array transmission, Javia-I gets a free ticket from the ring, copies the data from the byte array into a buffer and enqueues that on the VI send queue. `sendWait` polls the queue and updates the ring upon completion. To receive into a byte array, Javia-I obtains the ticket that corresponds to the head of the VI receive queue, and copies the data from the

---

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

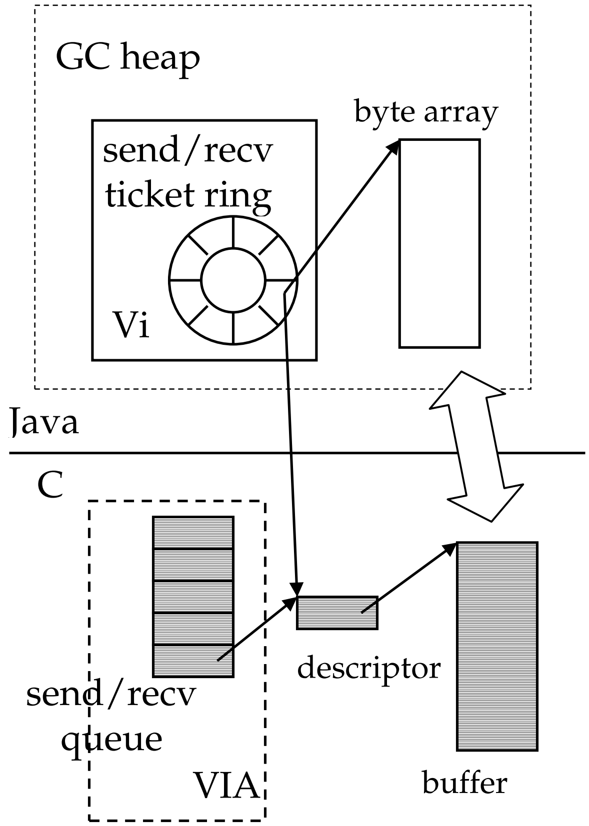

Figure 2.4 Javia-I per-endpoint data structures. Solid arrow indicates data copying.

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

Javia-I provides a blocking send call (line 14) because in virtually all cases the message is transmitted instantaneously—the extra completion check in an asynchronous send is more expensive than blocking in the native library. It also avoids accessing the ticket ring and enables two send variations. The first one (*send-copy*) copies the data from the Java array to the buffer whereas the second (*send-pin*) pins the array on the fly, avoiding the copy[13].

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

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

2.3.2 Example: Ping-Pong

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

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

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

2.3.3 Implementation Status

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

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

2.3.4 Performance

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

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

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

Pin's 4-byte latency includes the pinning and unpinning costs (around 20μs) and has a per-byte cost that is closest to that of raw (the difference is due to the fact that data is still being copied at the receive end). *Copy+alloc*'s 4-byte latency is only 1.5μs above that of *raw* because it bypasses the ticket ring on both send and receive ends. Its per-byte cost, however, is significantly higher than that of *copy* due to allocation and garbage collection overheads. The additional Java/native crossings take a toll in *JDK copy*: each downcall not only includes the overhead of a native method invocation in JNI, but also a series of calls to perform read/write operations to Java object fields. Although *JDK copy+alloc* is able to bypass the ring, the per-byte cost appears to be significantly higher, most likely due to garbage collections caused by excessive allocations during benchmark executions.

Pin's effective bandwidth is about 85% of that of *raw* for messages larger than 6Kbytes. Due to the high pinning costs, *copy* achieves an effective

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

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

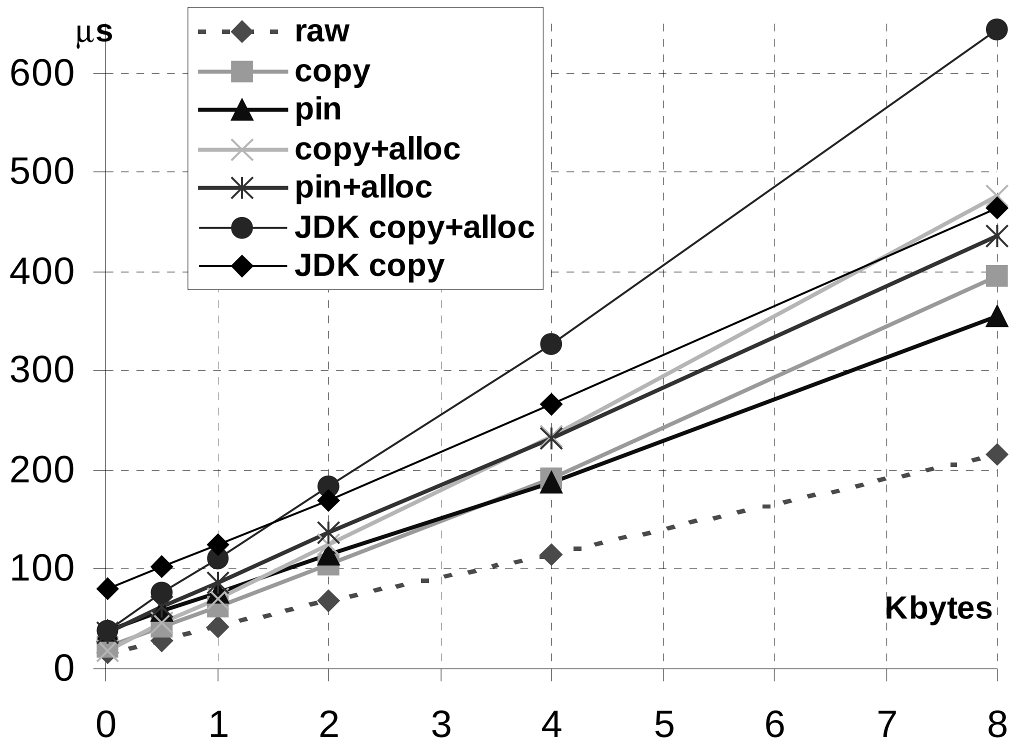

Figure 2.5 Javia-I round-trip latencies

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

2.4 Summary

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

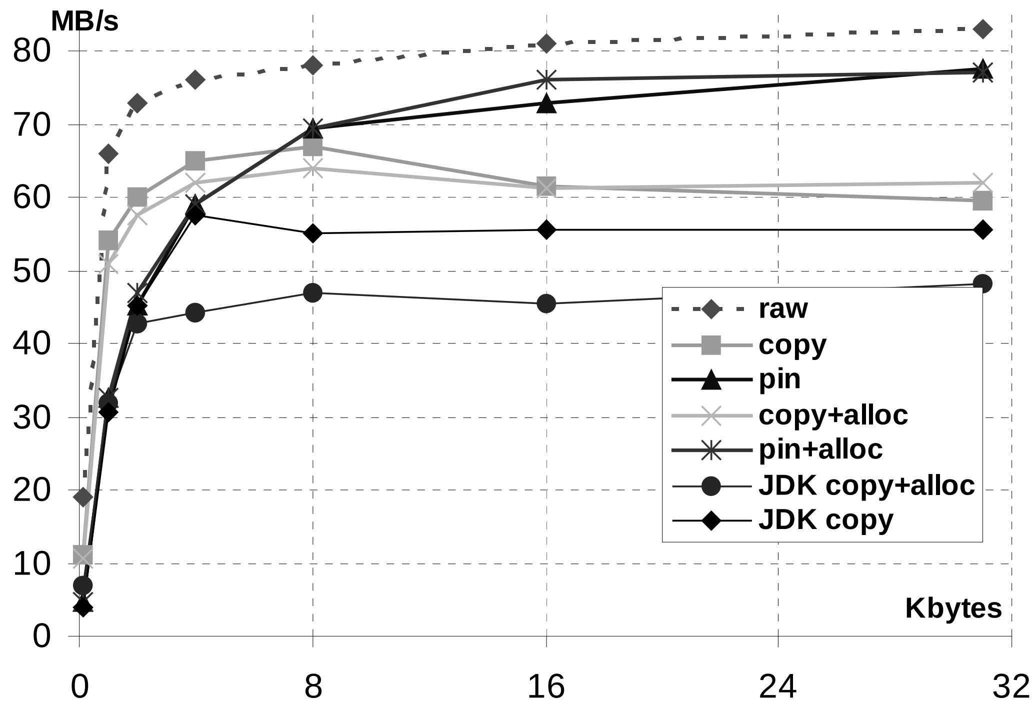

Figure 2.6 Javia-I effective bandwidth

Moreover, as mentioned before, pinning on the fly cannot be applied to the receiving end.

While this approach does not achieve the best performance with large messages, it is attractive for small messages and can be implemented on any off-the-shelf Java system that supports JNI. Even in the scenario where the native interface is efficient, as in Marmot, the hard separation between Java's garbage collected heap and native heap forces Javia-I to copy data or pin arrays on demand.

2.5 Related Work

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

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

A number of projects have adopted a "front-end" approach to developing communication software for Java applications: given a particular abstraction (e.g. sockets, RMI, MPI), they provide "glue-code" for interfacing with legacy libraries in native code. For example, implementations of the `java.net` package in most JVMs are typically layered on top of the sockets (TCP/IP) API. Central Data [Cd99] offers native implementations of the `portio` package for accessing serial and parallel ports from Java. [GFH+98] makes the MPI communication library available to Java applications by providing automatic tools for generating Java-native interface stubs. [BDV+98] deals with interoperability issues between Java RMI and HPC++, and [Fer98] presents a simple Java front-end to PVM. All these approaches respect the heap separation and do not address the performance penalty incurred during Java/native interactions.

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:

```
public class Jbuf {

  /* allocates a jbuf of size bytes */
  public final static Jbuf alloc(int bytes);

  /* attempts to free the jbuf */
  public final void free() throws ReferencedException;

  /* attempts to obtain a <p>[] reference to the jbuf, where */
  /* p is a primitive type. Only byte[] and int[] are shown.*/
  public final synchronized byte[] toByteArray() throws TypedException;
  public final synchronized int[] toIntArray() throws TypedException;

  /* claims that there are no references into the jbuf and */
  /* waits for the GC to verify the claim. */
  public final void unRef();

  /* cb is invoked by GC after claim is verified */
  public final void setCallBack(CallBack cb);

  /* checks if a reference points into a jbuf */
  public final boolean isJbuf(byte[] b);
  public final boolean isJbuf(int[] i);
  /* others omitted */
}
```

and provides users with three features:

1. *Lifetime control* through explicit allocation (`alloc`, line 4) and deallocation (`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.

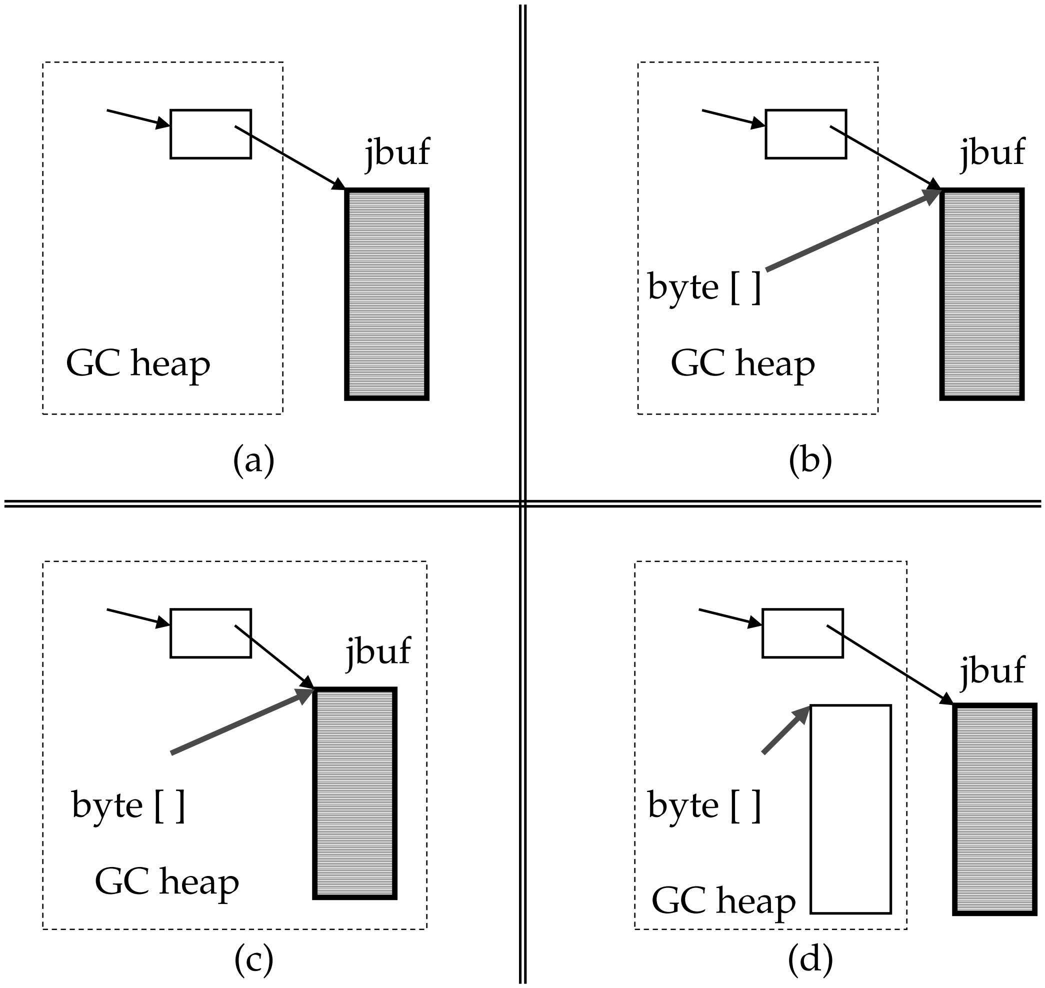

Figure 3.1 Typical lifetime of a jbuf in a copying GC. (a) After allocation, the jbuf resides outside of the GC heap. (b) The jbuf is accessed as a Java array reference. (c) The jbuf is added to the GC heap. (d) Upon callback invocation, the jbuf can be freed or re-used.

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.

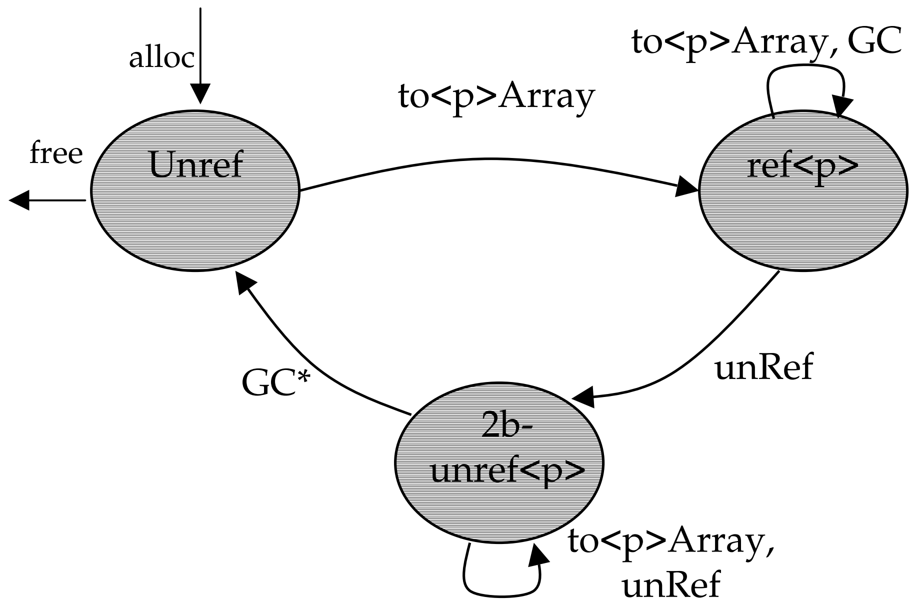

Figure 3.2 Jbufs state diagram for runtime safety checks. When the GC* transition takes place depends on whether the GC is copying or non-copying.

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

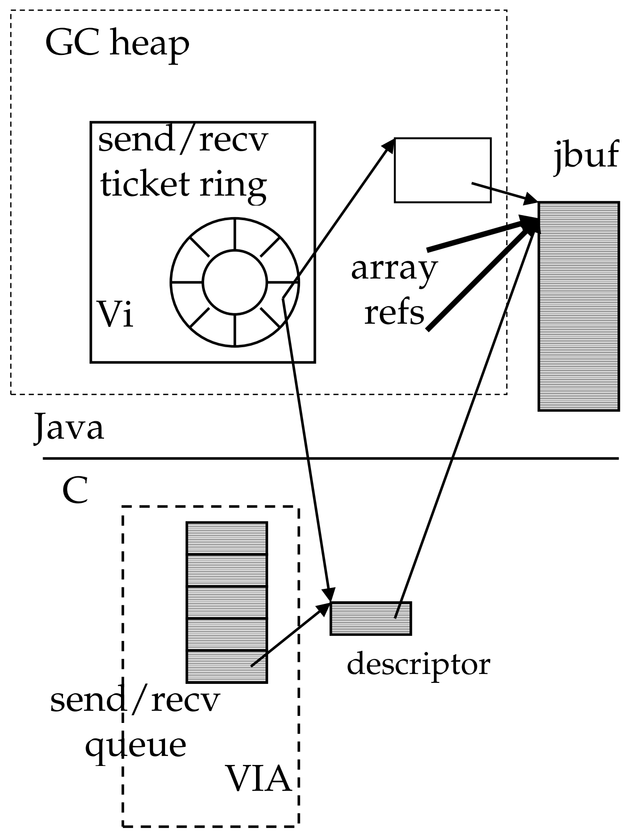

Figure 3.3 Javia-II per-endpoint data structures.

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 transferred 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:

```
ViBuffer buf = new ViBuffer(1024);
/* get send ticket */
ViBufferTicket sendT = buf.register(vi, attr);
/* get recv ticket */
ViBufferTicket recvT = buf.register(vi, attr);
byte[] b = vb.toByteArray();
/* initialize b… */
/* post recv ticket first */
vi.recvBufPost(recvT, 0);
if (ping) {
  /* send */
  vi.sendBufPost(sendT, 0, 1024);
  sendT = vi.sendBufWait(Vi.INFINITE);
  /* wait for reply */
  recvT = vi.recvBufWait(Vi.INFINITE);
  /* done */
} else { /* pong */
  vi.recvBufPost(recvT, 0);
  recvT = vi.recvBufWait(Vi.INFINITE);
  /* send reply*/
  vi.sendBufPost(sendT, recvT.off, recvT.bytesRecvd);
  sendT = vi.sendBufWait(Vi.INFINITE);
  /* done */
}
buf.deregister(sendT);
buf.deregister(recvT);
buf.unRef(new MyCallBack());
/* after callback invocation… */
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

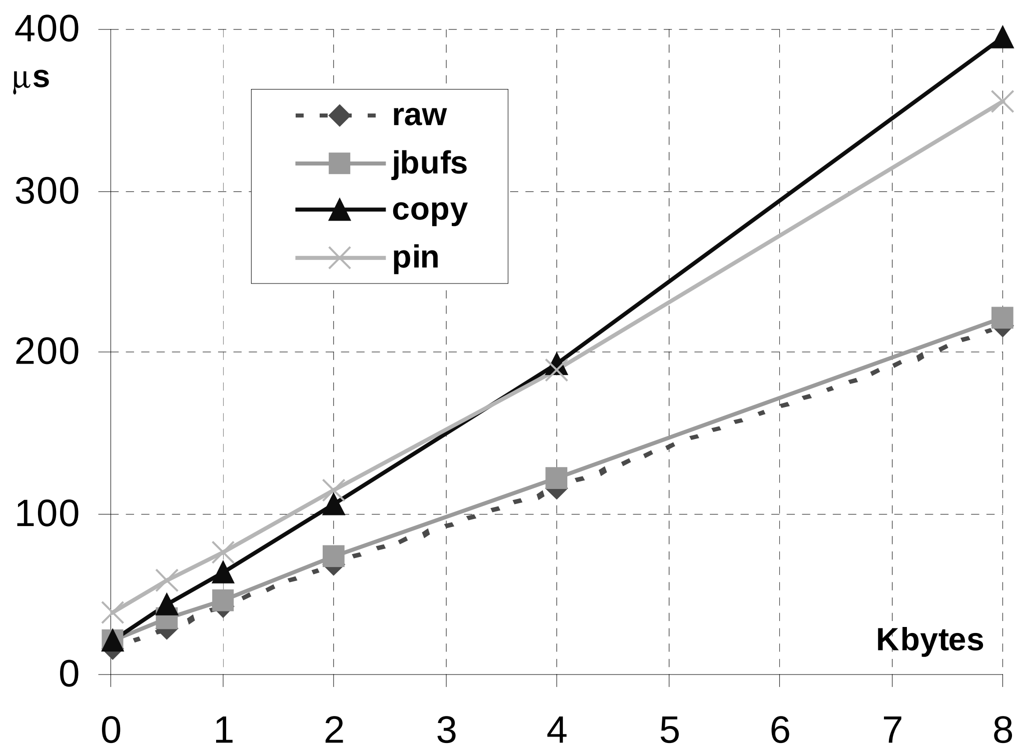

Figure 3.4 Javia-II round-trip latencies

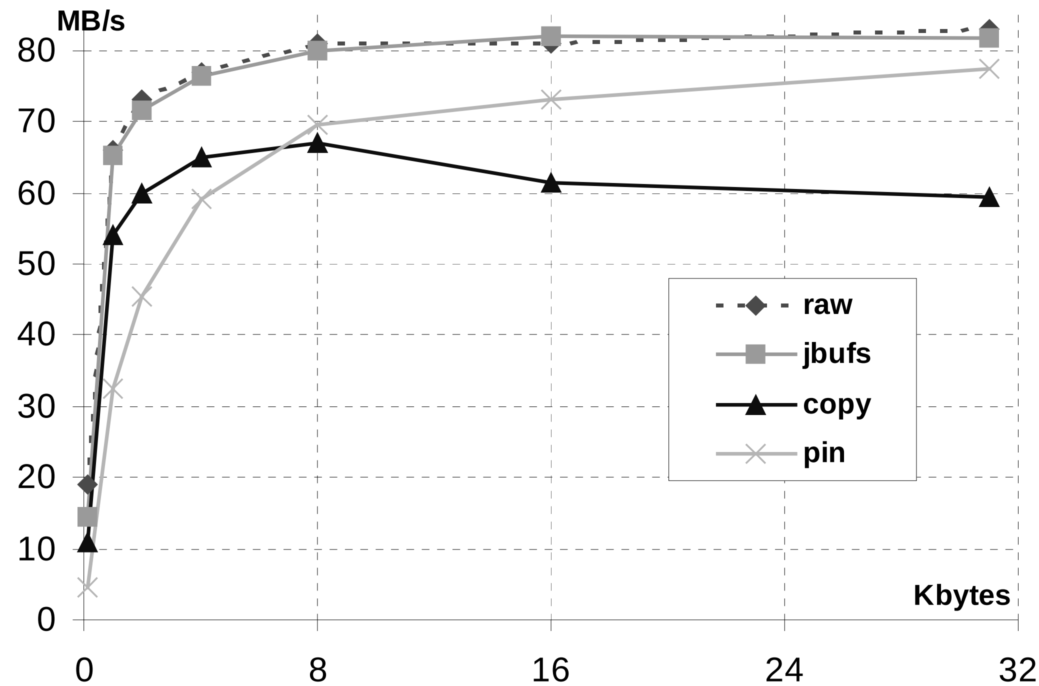

Figure 3.5 Javia-II effective

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.

```
/* Aloc is the local portion of A: array of n/p arrays of n doubles */
double[][] Aloc = new double[n/p][];
/* n/p jbufs, each being used as an array of n doubles */
ViBuffer[] bA = new ViBuffer[n/p];

/* bA's send and receive tickets */
ViBufferTicket[] sentT = new ViBufferTicket[n/p];
ViBufferTicket[] recvT = new ViBufferTicket[n/p];

/* initialize bA, tickets, and Aloc */
for (int j = 0; j < n/p; j++) {
  bA[j] = new ViBuffer(n*SIZE_OF_DOUBLE);    /* allocate jbufs */
  Aloc[j] = bA[j].toDoubleArray(n);          /* obtain double[] refs */
  sendT[j] = bA[j].register(rightVi, rattr); /* pin for sends */
  recvT[j] = bA[j].register(leftVi, lattr);  /* pin for recvs */
  for (int i = 0; i < n; i++) {
    /* Aloc initialization omitted */
  }
}
```

The core of the algorithm used in pMM is 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: iterate over columns A, B, and C*/
  for (int j = 0; j < n/p; j++) {
```

```
    double[] c = Cloc[j];
    double[] b = Bloc[j];
    /* iterate over rows */
    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).

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].

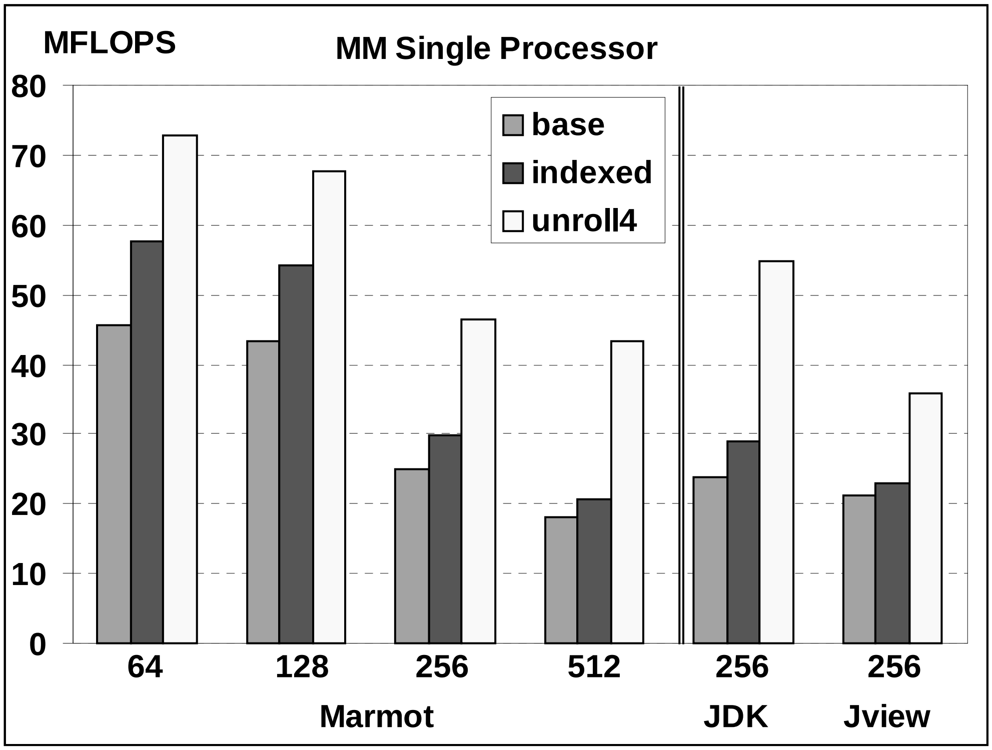

Figure 3.6 Performance of MM on a single 450Mhz Pentium-II

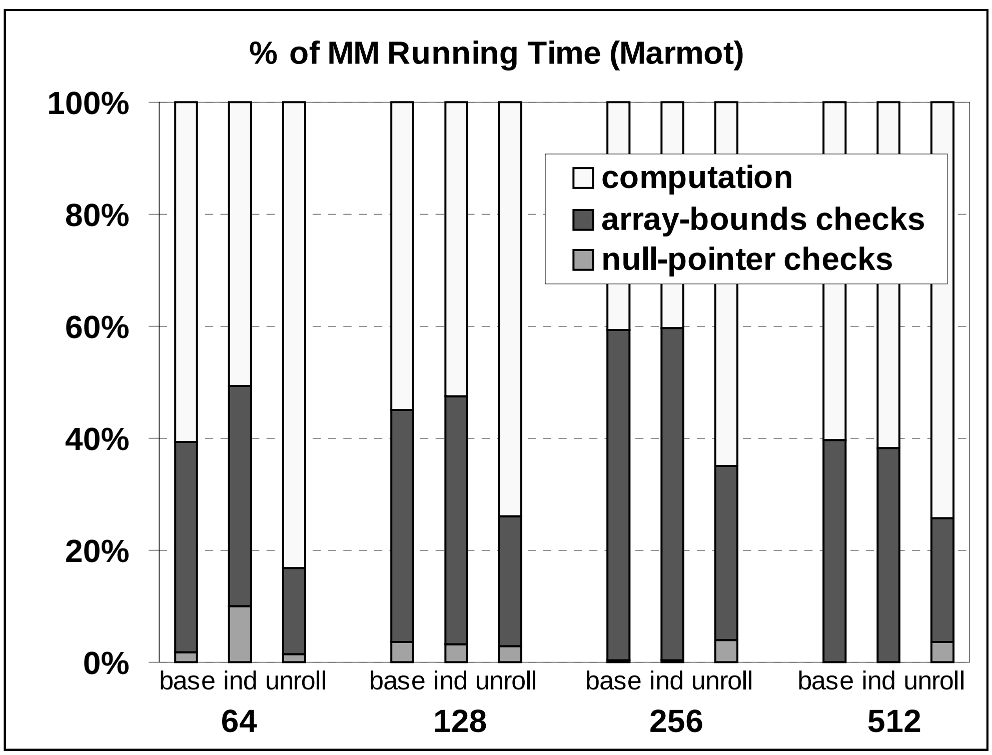

Figure 3.7 Impact of safety checks on MM

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).

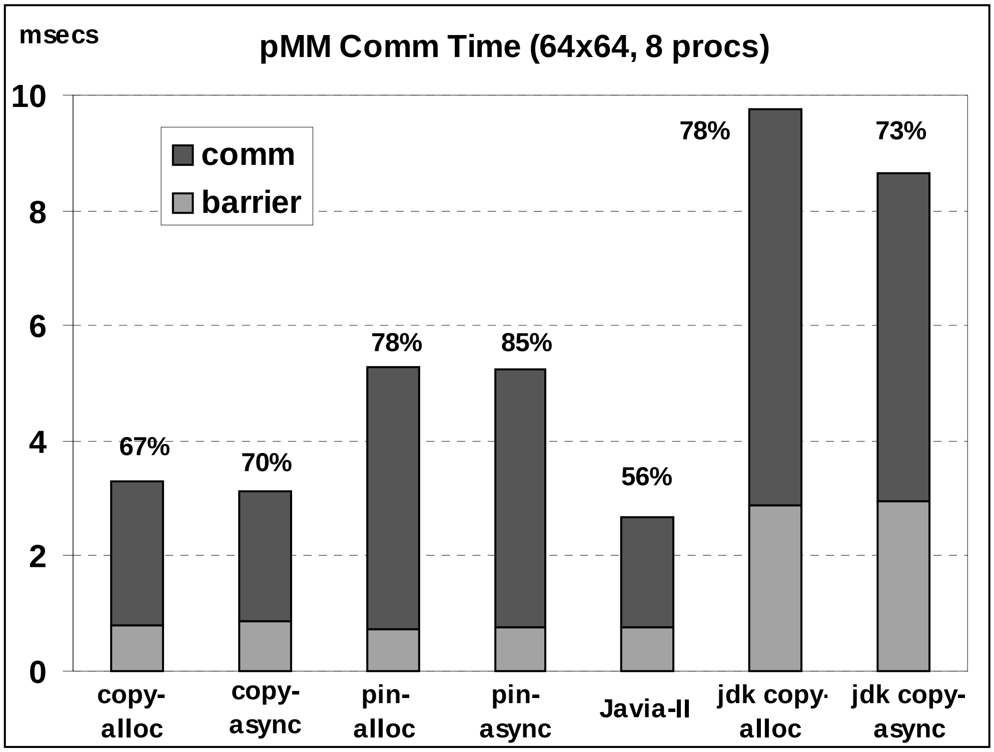

Figure 3.8 Communication time of pMM (64x64 mat., 8 processors)

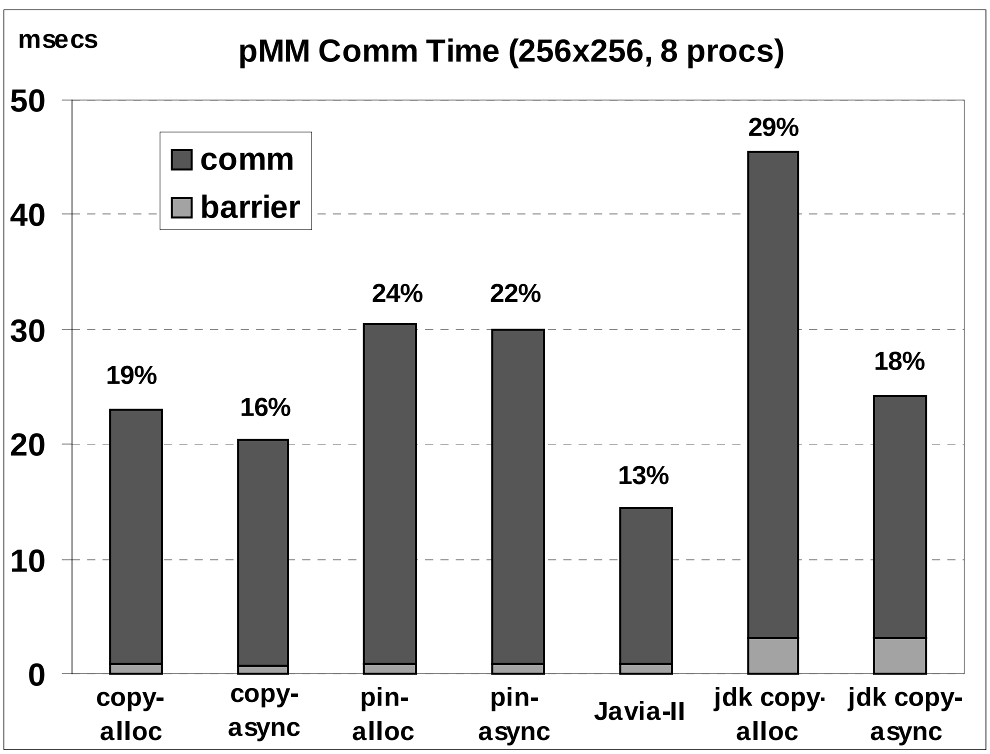

Figure 3.9 Communication time of pMM (256x256 mat., 8 processors)

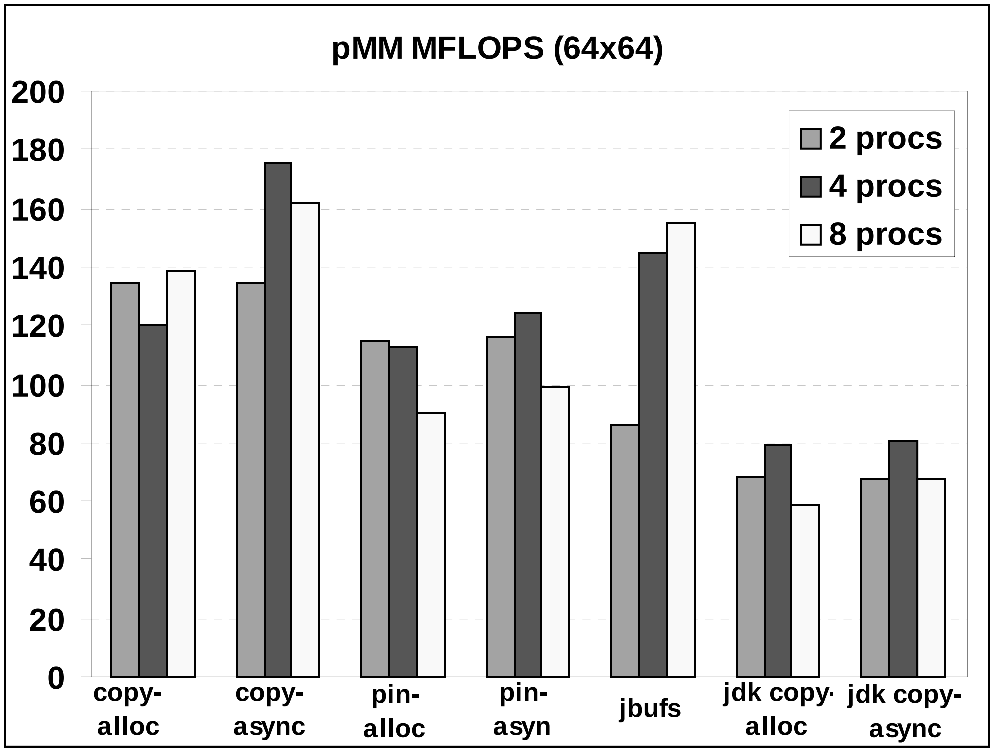

Figure 3.10 Overall performance of pMM (64x64 mat., 8 processors)

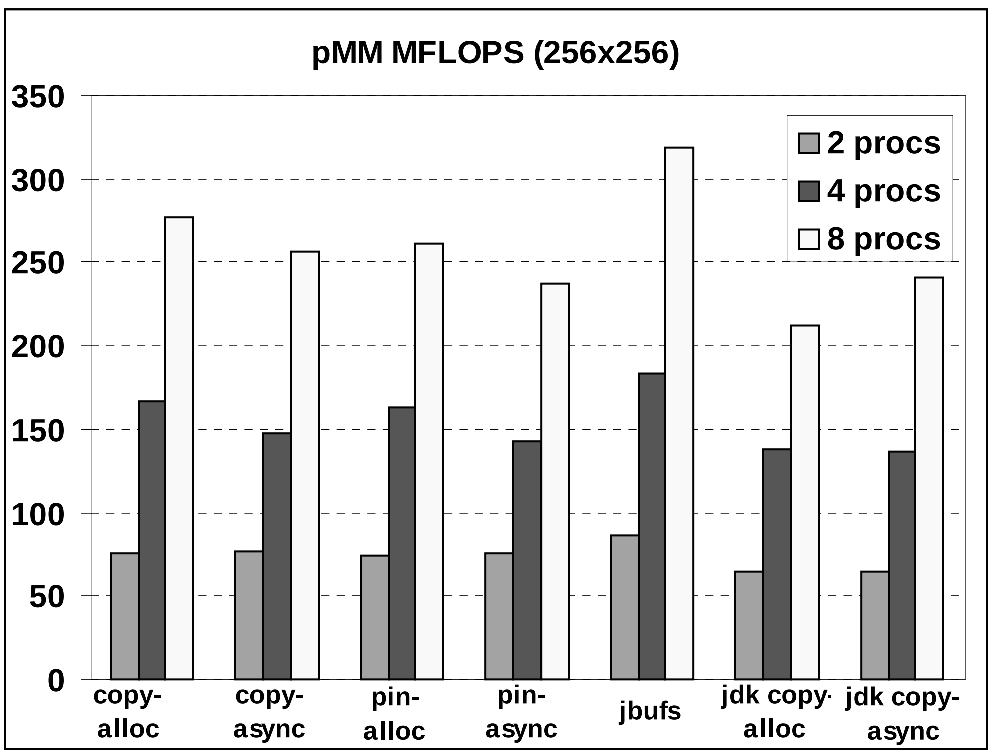

Figure 3.11 Overall performance of pMM (256x256 mat., 8 processors)

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:

```
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.

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).

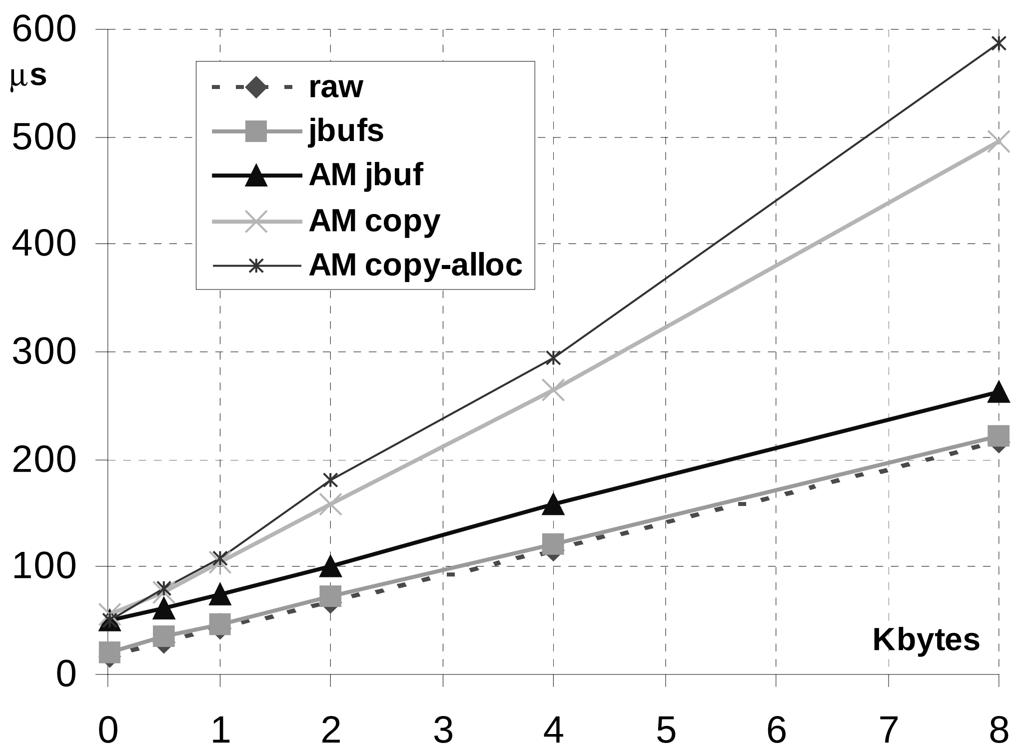

Figure 3.12 Jam round-trip latencies

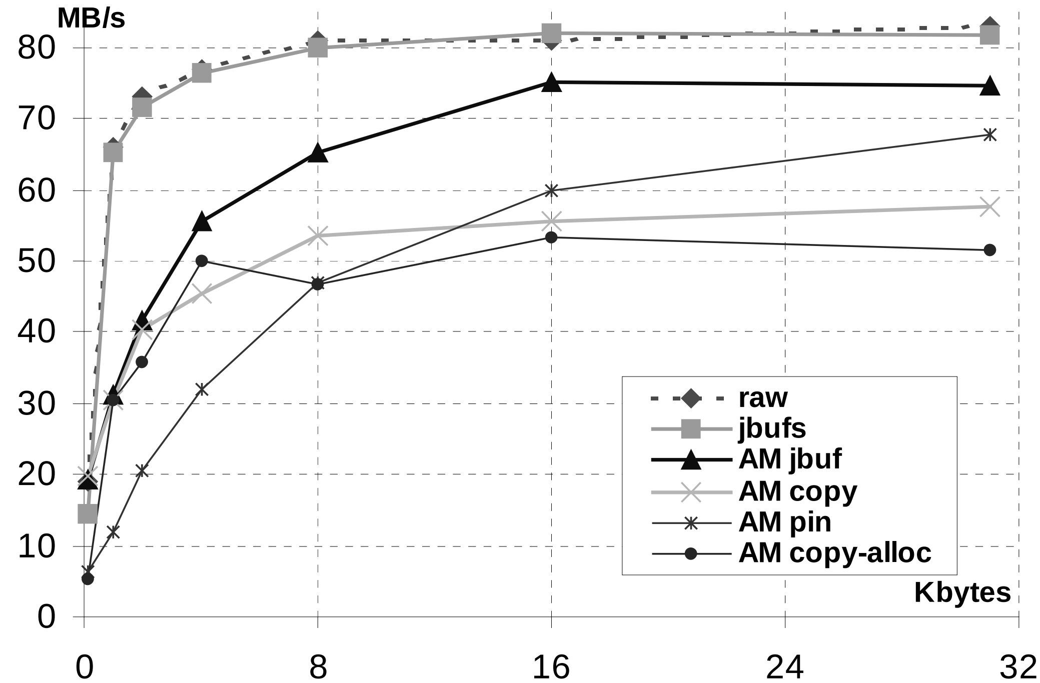

Figure 3.13 Jam effective bandwidth

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 re-use 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 applications 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.

4 Object Serialization: A Case for Specialization

The ability to send and receive primitive-type arrays efficiently is essential for high-performance communication in Java. Jbufs enable the VI architecture to transfer the contents of arrays in a zero-copy fashion by exploiting two facts: primitive-type arrays are shallow objects (i.e. contain no pointers to other Java objects) and their elements are typically contiguous in memory[20]. Array-based cluster applications written in Java can take advantage of jbufs to improve their communication performance, as demonstrated in the previous chapter.

During a remote method invocation (RMI) in Java, however, arbitrary linked object data structures are frequently passed by copy and must be transmitted over the wire. Since the objects forming the data structure are not guaranteed to be contiguous in memory, they need to be serialized onto the wire on the sending side and de-serialized from the wire on the receiving side. Standard serialization protocols are designed first for flexibility, portability, and interoperability of the RMI layer and only second for performance.

---

[20] Virtually all JVM implementations lay out array elements contiguously in memory for efficiency purposes, although it is not guaranteed by the JVM specification.

This chapter argues that the costs of object serialization are prohibitively high for cluster applications. It evaluates several implementations of the JDK serialization protocol using micro-benchmarks and an RMI implementation over Javia-I/II. Although efficient array transfer improves the performance of RMI (because serialization of Java objects ultimately yields byte arrays) the overheads of serialization are still an order of magnitude higher than the basic send and receive overheads in Javia-II. The impact of serialization on point-to-point RMI performance is substantial: the zero-copy benefits achieved by jbufs become negligible. For some applications in an RMI benchmark suite, the cost of serialization is estimated to account for up to 15% of their total execution time.

4.1 Object Serialization

As seen in Figure 4.1, serializing an object consists of converting its in-memory representation into a stream of bytes. This conversion makes a deep copy of the object: all transitively reachable objects are also serialized. The resulting

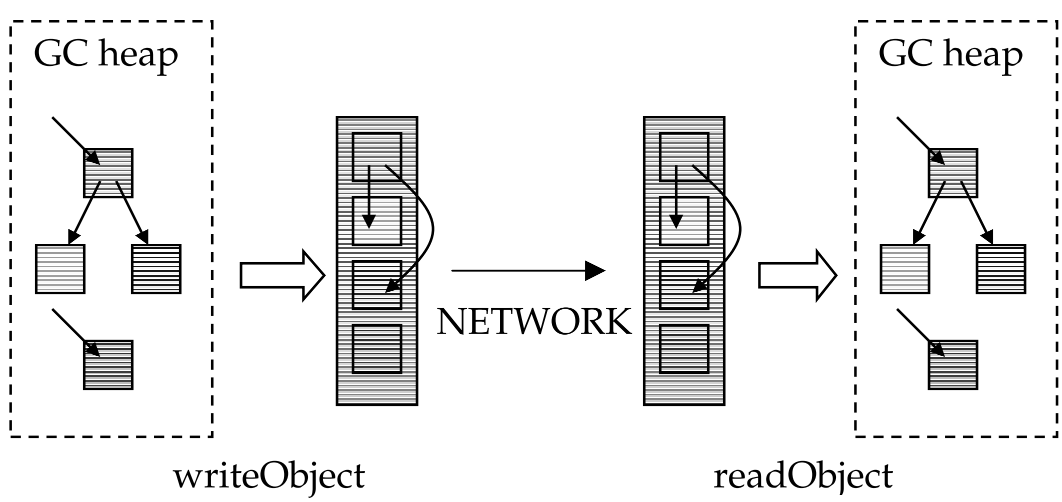

Figure 4.1 Object Serialization and De-serialization.

stream, typically stored in a Java byte array, is sent over the network. At the receiving end, the objects are retrieved (de-serialized) from the stream: for each de-serialized object, data is copied from the stream into its newly allocated storage.

Most publicly available JVMs implement the Java Object Serialization (JOS) protocol [Jos99] that is designed for flexibility and extensibility. JOS introduces object I/O streams (`ObjectInputStream`/`ObjectOutputStream`) with methods to write "serializable" Java objects into the stream (`writeObject`) and to read them from the stream (`readObject`). The protocol serializes the description of an object's class along with the object itself. If the class is available in the JVM during de-serialization, both the wire and the local versions are compared using "class compatibility" rules. If the class is not available or is incompatible, JOS provides a mechanism to annotate serialized classes (via the `annotateClass` method) so users can send along the original byte-code or an URL from where it can be fetched. Users can also define the external format of an object by overriding `read/writeObject` methods in object I/O stream classes with protocol-specific ones, or by providing object-specific implementations of `read/writeExternal` methods.

4.1.1 Performance

This section shows that the performance of JOS is inadequate for cluster computing using results from micro-benchmarks. Figures 4.2 and 4.3 show the performance of three implementations—Marmot, JDK1.2, and Jview3167 on a 450Mhz Pentium-II—of `writeObject` and `readObject` methods respectively. The types of objects used in the experiment are `byte` and `double` arrays with comparable sizes (around 100 and 500 bytes), an array of `Complex` numbers (each element with a real and a imaginary field of type `double`), and a linked list (each element with a `int` value and a "next" pointer). The costs for the latter two are reported on a per-element basis. The numbers reported are averages with standard deviation of less than 5% (maximum is 4.7% in *byte[] 500*).

The results shown in Figures 4.2 and 4.3 lead to the following observations:

1. Serialization overheads are in tens of microseconds: an order of magnitude higher than basic send and receive overheads in Javia I/II (around 3μs). Reading a byte array of 500 elements costs around 20μs; in comparison, a `mempcy` of 500 bytes costs around 0.8μs;
2. Serialization of arrays with 16, 32, and 64-bit primitive-type elements

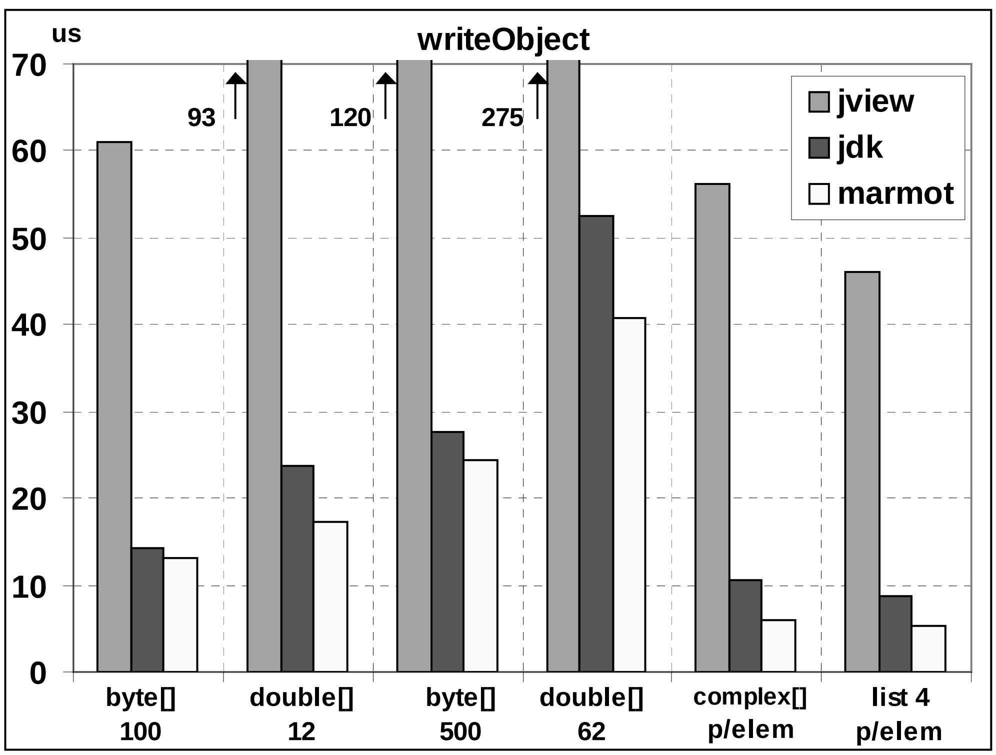

Figure 4.2 Comparing the cost of serialization in three implementations of Java Object Serialization.

into byte arrays takes a significant performance hit due to Java's type safety. Serializing a `double` array of 62 elements is nearly 50% more expensive than a byte array of 500 elements;

3. Costs grow as a function of object size both in `writeObject`, due to the deep-copy, and in `readObject`, due to storage allocation and data copying. It costs about 9µs to read one linked-list element with an `int` field out of the stream, and around 86µs to read one with 40 `int` fields.

It seems unlikely that better compilation technology will improve the performance of serialization in a substantial way. Tables 4.1 and 4.2 show the percentage change in the cost of `writeObject` and `readObject` on Marmot in the absence of method inlining, synchronization, and safety checks. Changes in cost are more significant for the array of complex numbers and the linked

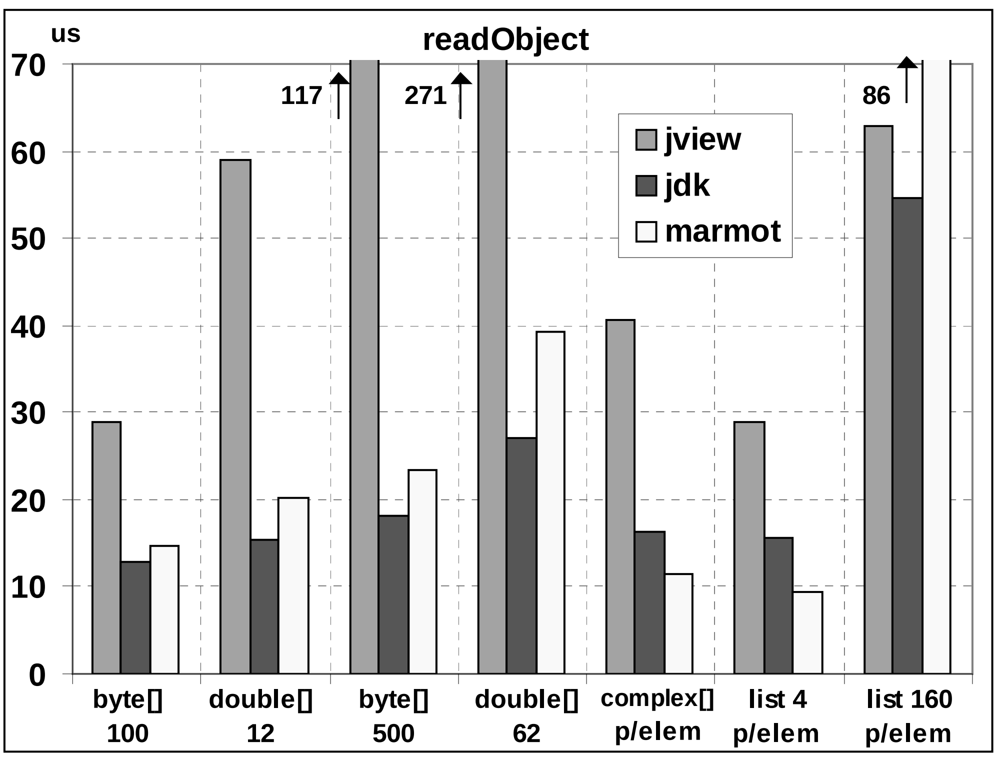

Figure 4.3 Comparing the cost of de-serialization in three implementations of Java Object Serialization.

list. Method inlining in Marmot already reduces the costs by 60%. Even if Marmot were able to successfully eliminate *all* safety checks and *all* synchronization, performance would improve by another 30% at best.

Table 4.1 Impact of Marmot's optimizations in serialization.

Cost Difference (writeObject)	no method inlining	no locks	no array-bounds checks	no null pointer checks	no casts checks	no array-store checks
byte[] 500	24.2%	-4.5%	-0.4%	0.0%	-2.5%	-1.3%
double[] 100	14.1%	-3.2%	-1.5%	0.0%	-3.3%	-0.5%
complex[] p/elem	56.7%	-12.9%	-0.7%	0.0%	-0.6%	-7.3%
list p/elem	61.7%	-12.9%	-0.7%	0.0%	0.0%	-6.7%

Table 4.2 Impact of Marmot's optimizations in de-serialization

Cost Difference (readObject)	no method inlining	no locks	no array-bounds checks	no null pointer checks	no casts checks	no array-store checks
byte[] 500	48.7%	-4.2%	0.0%	0.0%	-0.9%	0.0%
double[] 100	23.2%	-2.5%	0.0%	0.0%	-1.5%	0.0%
complex[] p/elem	80.5%	-12.6%	0.0%	0.0%	-6.1%	0.0%
list p/elem	77.7%	-20.7%	0.0%	-2.9%	-11.2%	-0.7%

4.2 Impact of Serialization on RMI

The high serialization costs reported in the previous section affect the performance of Java RMI significantly. This section starts with an overview of Java RMI and briefly describes an implementation over Javia-I/II. Readers familiar with RMI can jump to the section on micro-benchmark performance (Section 4.2.3).

4.2.1 Overview of RMI

RMI enables the creation of distributed Java applications in which methods of remote Java objects can be invoked from other JVMs, possibly on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in a name service provided by RMI, or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI implementations in publicly available JVMs are based on the Java RMI specification [Rmi99].

RMI relies on JOS to serialize and de-serialize remote objects (which are passed by reference) and regular objects (which are passed by value). RMI takes advantage of JOS' extensibility and class serialization protocol to support "polymorphic"[21] method invocations: an actual parameter object can be a subclass of the remote method's formal parameter class. This means that the receiver may not know the actual subclass of the argument, and may have to fetch it from the wire or from a remote location. This flexibility makes RMI applications potentially more tolerant to service upgrades and different versions of class files, and is the key distinction between RMI and traditional remote procedure call systems.

4.2.2 An Implementation over Javia-I/II

This section describes a straightforward RMI implementation over Javia-I/II based on the RMI specification.

Remote objects (that extend the `RemoteObject` class and implement the `Remote` interface) are bound (i.e. exported) to a simple RMI `Registry`. The registry creates corresponding stub and skeleton (i.e. server side stub) objects for the remote object, spawns a transport-dependent server thread that waits for incoming connections, and updates its service database. When a client binds to an exported remote object, the registry ships the stub to the client,

---

[21] This term is in quotes because there is no true polymorphism in Java [OW97].

which is instantiated in the client's JVM. During an RMI, the stub creates a transport-dependent `RemoteCall` object that connects to the server thread and initializes communication structures. The server thread spawns a new thread to service calls from that stub upon accepting the connection. The remote call object is cached by the stub for subsequent invocations to the same remote object in order to avoid creating a new connection for every RMI.

The implementation uses JOS for serialization and de-serialization of arguments and relies on a RMI protocol that is similar to the one described in the specification. It also uses a simple distributed GC scheme based on reference counting [BEN+94].

The system consists of about 4000 lines of Java and currently supports three transport layers: TCP/IP sockets, Javia-I and Javia-II. A remote call object using Javia-I connects to the server thread through a virtual interface that can be configured in four different send/receive combinations (Section 2.3.1). In the case of Javia-II, a connection is composed of two virtual interfaces: one for RMI headers (up to 40 bytes) and another for the payload. Jbufs posted on the header VI are accessed as `int` arrays; those posted on the payload VI are accessed as `byte` arrays by the object I/O streams.

Because of RMI's blocking semantics, the number of jbufs posted on each VI (which is a service parameter) essentially indicates the maximum number of concurrent calls (e.g. client threads) the remote object can handle for each connection. A remote call object tracks the number of outstanding RMIs to ensure that that number is not exceeded. When waiting for an incoming message (either a call or a reply), the thread polls for a while (around twice the round trip latency) before blocking.

4.2.3 Performance

The round-trip latency of an RMI between two cluster nodes is measured by a simple ping-pong benchmark that sends back and forth a byte array of size *N* as argument using a single RMI connection. The effective bandwidth is measured by sending 10MBytes of data one-way, using various byte array sizes as fast as possible. RMI implementations over several configurations of Javia-I and over Javia-II (labeled as *RMI jbufs*) are compared on Marmot and JDK1.2. Both experiments exclude context switch costs since unloaded machines are used and reception takes place primarily by polling (due to the above optimization).

Figure 4.4 shows the round-trip latencies. Although jbufs yield some improvement, the RMI performance is far from that of Javia-II. Table 4.3 shows the round-trip latencies of an RMI with an integer argument and includes the number for RMI over sockets as well. A significant fraction of the 150µs achieved by *RMI jbufs* goes to setting up object I/O streams for argument passing. For instance, the round-trip latency of a null RMI drops to slightly less than 30µs, which is about the same as that achieved by Jam.

Figure 4.5 shows effective bandwidth achieved by RMI. A peak bandwidth of about 22MBytes/s is attained by *RMI jbufs*, which is about 25% of total capacity. The bandwidth curve reaches the peak at a much slower rate than Javia-I and Javia-II because RMIs are not pipelined (due to their blocking semantics) during the experiment[22].

---

[22] Bandwidth experiments involving multiple client connections have not been carried out because the cluster nodes have a single processor.

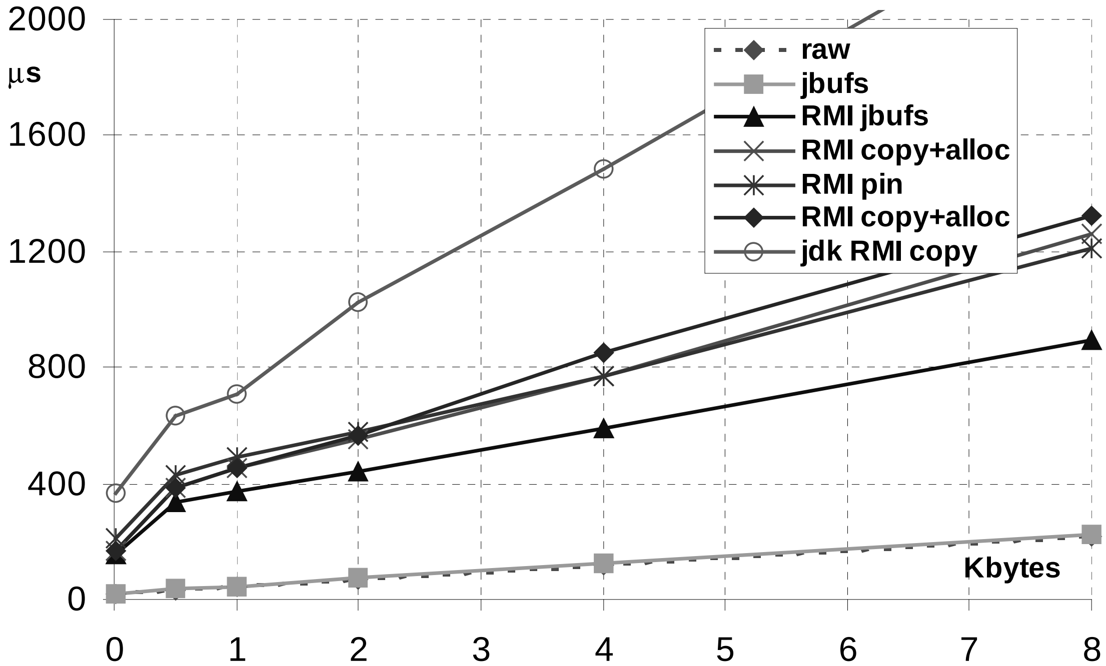

Figure 4.4 RMI round-trip latencies.

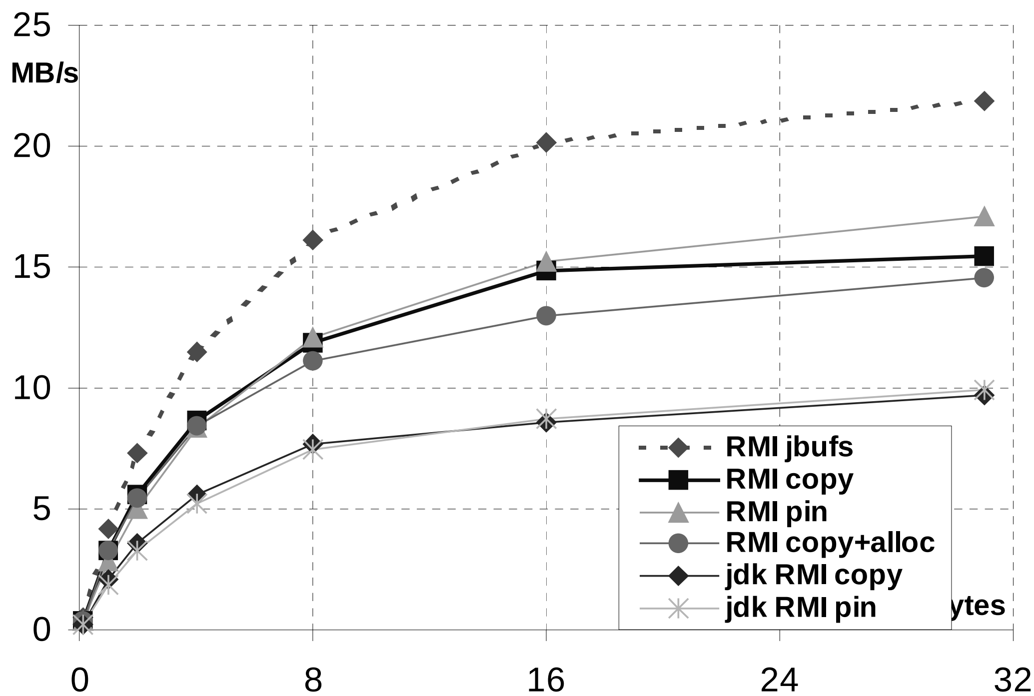

Figure 4.5 RMI effective bandwidth.

Table 4.3 RMI 4-byte round-trip latencies.

RMI	*4-byte (us)*
jbufs	150.4
copy+alloc	161.9
copy	164.5
pin	211.8
jdk copy	271.0
sockets	482.3
jdk sockets	520.1

4.3 Impact of Serialization on Applications

Poor RMI performance can have a significant effect on overall application performance. This section reports the impact of object serialization on a benchmark suite consisting of six RMI-based applications. Table 4.4 provides a summary of the applications used. A brief description of each application is presented in the following subsection.

4.3.1 RMI Benchmark Suite

Two applications, Traveling Salesman Problem (*TSP*) and Iterative Deepening A* (*IDA*), fall into the traditional producer-consumer model [NMB+99]. TSP computes the shortest path to visit all cities exactly once from a starting city by using a "branch-and-bound" algorithm. The algorithm prunes search subspaces by ignoring partial routes that are longer than the current shortest path. Because the amount of computation for a search sub-space is not known a-priori, the implementation adopts the producer-consumer model for (potentially) better load balancing. Workers running on cluster nodes repeatedly fetch jobs from a centralized job queue using RMIs. During execution, each worker keeps a local copy of the current best solution—if a worker finds a

Table 4.4 Summary of RMI benchmark suite.

Application	*Origin*	*Description*	*Model*	*Input*
TSP	Manta	shortest path to visit all other cities exactly once	Work Queue	17 cities
IDA	Manta	solving a 15-tile puzzle using repeated DFS	Work Stealing	depth of 58 moves
SOR	Manta	iterative method for Laplace equations	Sync Master Slave	1600-1600 grid, 100 iterations
EM3D arrays	Split-C Suite	simulation of EM wave propagation using RMI of `double[]`	Sync Master Slave	100K edges/proc, 100 iterations
FFT complex	Split-C Suite	1-D Fast Fourier Transform using `Complex[]`	Sync Master Slave	1 million points
FFT arrays	Split-C Suite	1-D Fast Fourier Transform using `double[]`	Sync Master Slave	1 million points
MM	Javia	MM using RMIs	Sync Master Slave	256x256 matrices

shorter solution, it updates the values of all other workers through RMI. The computation terminates when there are no jobs left in the job queue. The size of the input set used is 17 cities.

IDA solves the 15-tile puzzle using repeated depth-first searches. The program uses a decentralized job queue model with work stealing. Each job corresponds to a state in the search space, and each cluster node maintains a local job queue. When a node fetches a job from its job queue, it first checks whether the job can be pruned. If not, it expands the job by computing the successor states (e.g. making all possible next "moves") and enqueues the new jobs. If the local job queue becomes empty, a node tries to "steal" jobs from other nodes. The initial state of the puzzle is obtained by making 58 moves from the final state.

The remaining applications fall into the "structured" category: processing nodes have distinct computation and communication phases and are globally synchronized using barriers (*Barrier*). Upon reaching barrier point, program execution is blocked until all nodes reach a corresponding barrier point. To reduce the network traffic, each node communicate only with a parent node (if it is not the root) and (up to) two children nodes through RMIs.

Red-black Successive Over-relaxation (*SOR*) [NMB+99] is an iterative method for solving discrete Laplace equations: it performs multiple passes over a rectangular grid, updating each grid point using a stencil operation (a function of its four neighbors). The grid is distributed across all nodes in a row-wise fashion so each node receives several contiguous rows of the grid. Due to the stencil operation and the row-wise distribution, at each iteration every node (except for the first and last) needs to exchange its boundaries rows with its left and right neighbors using RMIs before updating the points. Each iteration has two exchange phases and two computation phases. The input used is a 1600x1600 grid of `double` values.

EM3D is a parallel application that simulates electromagnetic wave propagation [CDG+93]. The main data structure is a distributed graph. Half of its nodes represent values of an electric field (E) at selected points in space, and the other corresponds to values of the magnetic field (H). The graph is bipartite: no two nodes of the same type (e.g. E or H) are adjacent. Each of the processors has the same number of nodes, and each node has the same number of neighbors. Computation consists of a sequence of identical steps: each processor updates values of its local H- and E-nodes as a weighed sum of their neighbors. A naïve version of EM3D performs an RMI to fetch the value from a remote node each time the value is needed. An optimized version uses a simple pre-fetching scheme: a ghost-node is introduced for each a remote node that is shared by many local nodes. During the pre-fetching phase, each ghost node fetches the data from its corresponding remote node, eliminating redundant RMIs. There are no remote accesses during the computation phase.

The version of EM3D used here aggregates ghost nodes on a per-processor basis and issues a single RMI per processor. It uses a `double` array as argument and explicitly copies data between the graph and the array itself. The benchmark uses a synthetic graph of 40,000 nodes distributed across 8 processors where each node has degree 20 for a total of 800,000 edges. The fraction of edges that cross processor boundaries is varied from 0% to 50% in order to change the computation to communication ratio.

Fast Fourier Transform (*FFT*) [CcvE99] computes the n-input butterfly algorithm for the discrete one-dimensional FFT problem using P processors. The algorithm is divided into three phases: (i) $log(n) - log(P)$ local FFT computation steps using a cyclic layout where the first row of the butterfly is assigned to processor 1, the second to processor 2, and so on; (ii) a data re-mapping phase towards a blocked layout where the n/P rows are placed on the first processor, the next n/P rows on the second processor, and so on; and (iii) $log(P)$ local FFT computation steps using the blocked layout. In the first and third phases, each processor is responsible for transforming n/P elements. Each processor allocates a single n/P-element vector to represent its portion of the butterfly. Communication occurs only in the data re-mapping phase where each processor uses RMIs to send a n/P^2-element chunk of data to each remote processor[23]. The communication is staggered to avoid hot spots at the destination. Two versions—one using an array of `Complex` (with two `double` fields) and another using two `double` arrays—are run with an input of one million points.

The last application is a version of pMM (Section 3.3) where the communication using Javia-I/II is replaced with RMIs. pMM is run using 256x256 matrices on 8 processors.

4.3.2 Performance

RMI performance has little impact, if any, on the two irregular applications. Figure 4.6 shows the speedups of TSP and IDA. In TSP, the load is fairly balanced though coarse-grained; in IDA, idle workers ping other workers in a tight loop trying to steal work, congesting the network with RMIs.

The benefits of a fast transport layer are more pronounced in applications with distinct communication and computation phases. SOR (Figure 4.7) using RMI over Javia-II attains a speedup of 6.3 on 8 processors compared to about 1.8 if RMI over sockets are used. The per-edge cost of EM3D (Figure 4.8) grows as the percentage of remote edge grows, as expected. The array-based versions of FFT using RMI over Javia-I/II are able to achieve a peak 7Mflops (Figure 4.9). In comparison, the best FFT performance by a C program reported on a similar machine (a single 300Mhz Pentium-II) is about 70Mflops [Fft99].

---

[23] Because the FFT transfer size far exceeds the maximum transfer unit (MTU) of the RMI implementation (32Kbytes), data has to be further segmented to fit in MTU-sized chunks.

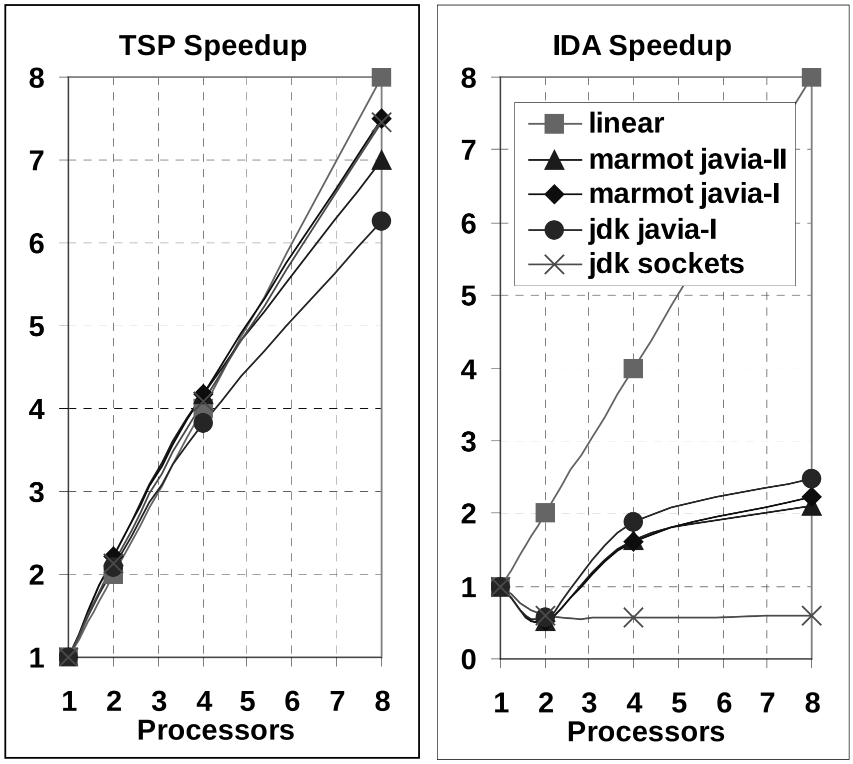

Figure 4.6 Speedups of TSP and IDA.

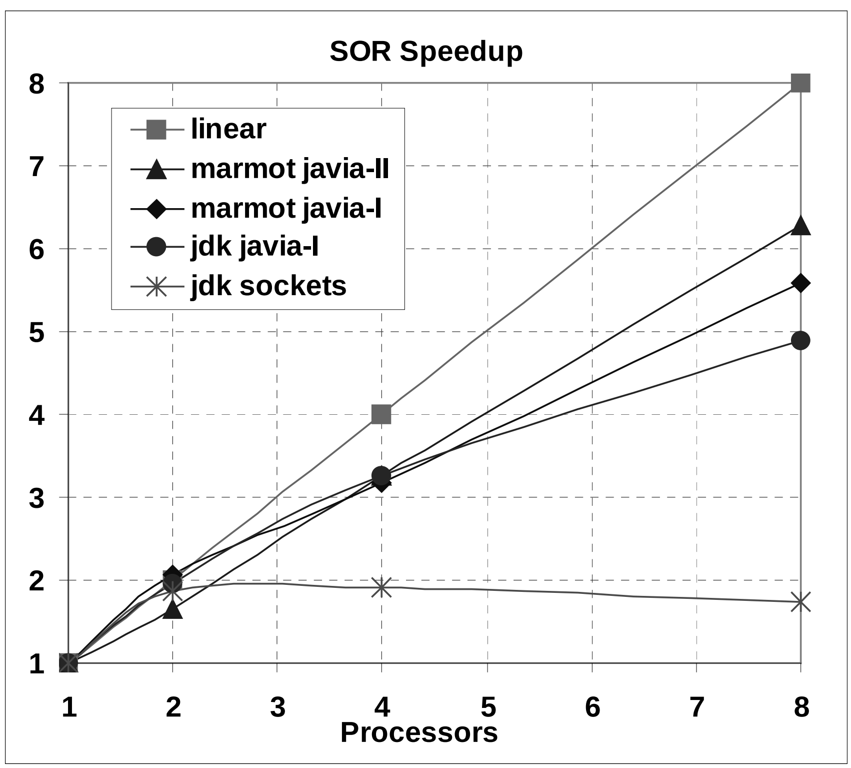

Figure 4.7 Speedup of SOR.

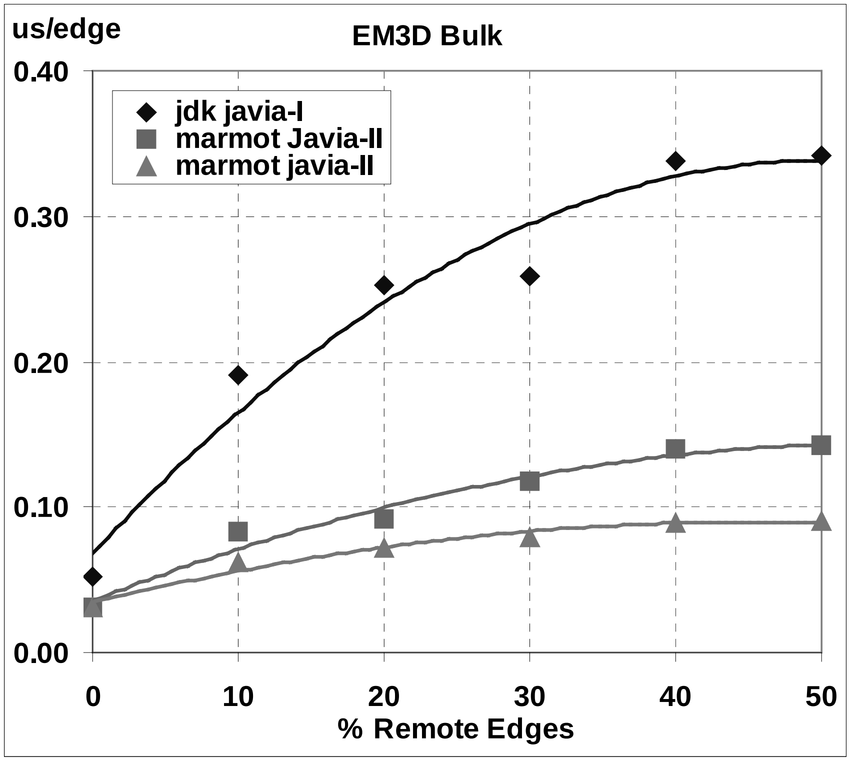

Figure 4.8 Performance of EM3D on 8 processors.

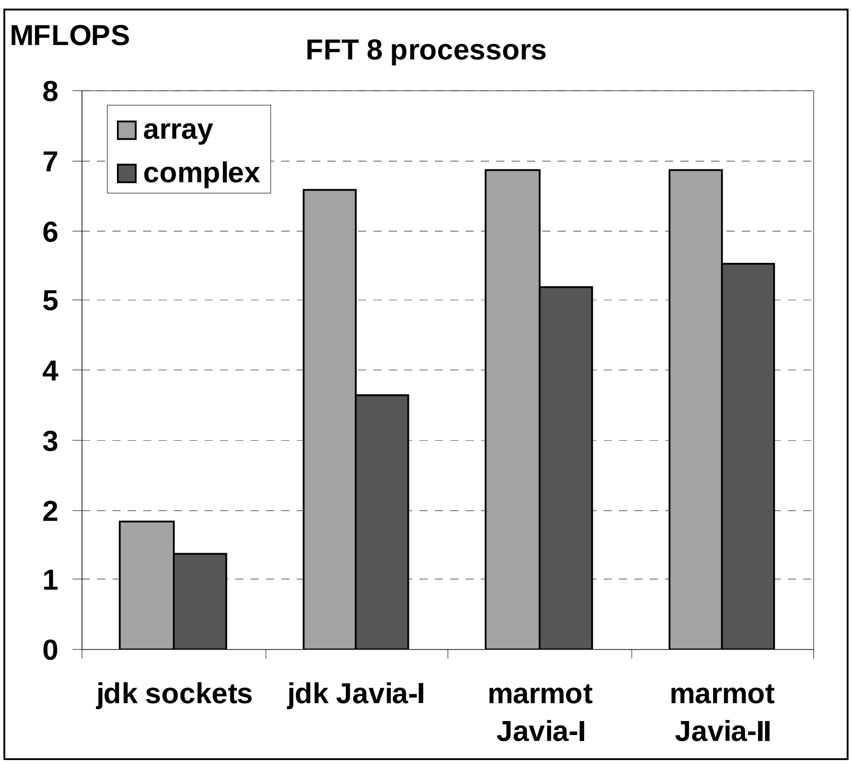

Figure 4.9 Performance of FFT on 8 processors.

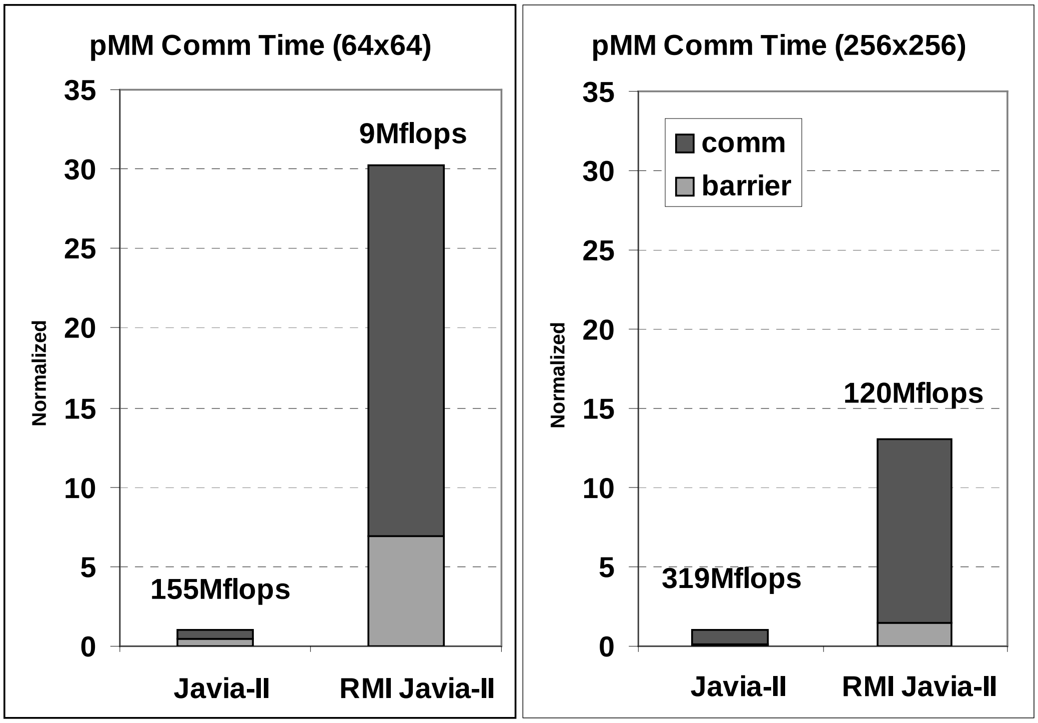

Figure 4.6 Performance of pMM over RMI on 8 processors.

Table 4.5 Communication Profile of Structured RMI Applications

Application	*#incoming RMIs*	*#outgoing RMIs*	*data type*	*data size (elements)*
SOR	400	400	`double[]`	1600
EM3D arrays	400	400	`double[]`	2300~2350
FFT complex	113	113	`Complex[]`	1024
FFT arrays	113	113	`double[]`	2 x 1024
pMM	22400	22400	`double[]`	256

Table 4.6 Estimated Impact of Serialization on Application Performance

Application	*comm meas. (secs)*	*total meas. (secs)*	*serial est. (secs)*	*serial est. (% comm)*	*serial est. (% total)*
SOR	4.59	19.78	0.54	11.76%	2.73%
EM3D arrays	2.20	4.60	0.24	10.90%	5.22%
FFT complex	18.30	19.03	2.61	14.28%	13.73%
FFT arrays	14.82	15.36	0.21	1.42%	1.37%
pMM	190.58	280.00	14.56	7.64%	5.20%

pMM using RMI over Javia-II (Figure 4.10) achieves a peak performance of 120Mflops, which is less than 40% of that achieved by pMM over Javia-II. The high communication time is partly attributed to context switches between the main and the remote object threads[24].

4.3.3 Estimated Impact of Serialization

To evaluate the effect of high serialization costs more precisely, we estimate the fraction of the communication time and of the total execution time in which the processor spends in serialization alone. The methodology relies heavily on the application's structured communication pattern and exploits two facts: (i) the cluster nodes have a single processor, and (ii) each application invokes a single remote method during the communication phase. For each processor, the total number of *incoming* RMIs during the communication phase is multiplied by the cost of *de-serializing* the arguments (both type and size are considered); the total number of *outgoing* RMIs[25] is multiplied by the total cost of *serializing* the arguments. The sum of the two resulting quantities is an estimate of the time spent in object serialization. Table 4.5 summarizes the communication profile of the structured applications in RMI benchmark suite.

Table 4.6 shows that the estimated serialization and de-serialization costs can account for as much as 15% of an application's execution time.

---

[24] Pipelining RMIs with multiple sender threads to hide network latency improved the communication time by less than 10%.

[25] The total number of incoming and outgoing RMIs reported in Table 4.5 have been validated by runtime RMI profiling.

4.4 Summary

The performance of object serialization is currently inadequate for cluster computing. Java's type safety causes array serialization to be over an order of magnitude higher than basic communication overheads as well as memory-to-memory transfer latencies. Better compiler technology will unlikely yield substantial improvements in serialization. Because of data copying, serialization costs grow as a function of object size. This essentially nullifies the "zero-copy" benefits offered by modern network interfaces.

The Java I/O model dictates that objects be serialized and de-serialized via cascading I/O streams, which leads to inefficient data access and buffering [NPH99]. Setting up these streams is also very costly: for example, experiments with RMI over Javia-II indicate that the round-trip latency of a null, optimized RMI is 5x faster than that of an RMI with one integer argument. Overall, serialization costs are estimated to account for 3% to 15% of total execution time of communication intensive applications.

4.5 Related Work

4.5.1 Java Serialization and RMI

KaRMI [NPH99] presents a ground-up implementation of object serialization and RMI entirely in Java. Unlike Manta (see below), 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 over 2MBytes/s (compared to a raw throughput of 50MBytes/s). The low bandwidth is attributed to 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 the serialization improvements over JDK1.4 vanish quickly as transfer size increases.

Several other projects [JCS+99, CFK+99] have shown that the performance of serialization is poor in the context of messaging layers such as MPI.

Breg et al. [BDV+98] recognizes the poor performance of Java RMI but advocates a "top-down" solution: it designs a subset of RMI that can be layered on top of the HPC++ runtime system. Krishnaswamy et al. [KWB+98] improves the performance of RMI over UDP with clever caching.

4.5.2 High Performance Java Dialects

Manta [MNV+99] is a "Java-like" language and implements Java RMI efficiently over 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 garbage collector and scatter/gather primitives in Panda. The authors report a RMI latency of 35µs and a throughput of 51.3MBytes/s on a PII-200/Myrinet cluster, which is within 15% of the throughput achieved by Panda.

Titanium [YSP+98] is a Java dialect for parallel computing that is inspired by Split-C [CDG+93], a parallel extension to C with split-phase operations. Titanium is designed first for high performance on large-scale multiprocessors and clusters, and only second to safety, portability, and support for building complex data structures. Titanium supports contiguous multi-dimensional arrays that map efficiently onto bulk transfers in Active Messages.

4.5.3 Compiler-Support for Serialization

More generally, previous work has demonstrated that optimizing stub compilers are required to reduce the serialization overheads that plague many distributed systems for heterogeneous environments. Schmidt *et al.* [SHA95, GS97] studied the performance of `rpcgen` and two commercial CORBA implementations. They reported that traditional stub compilers produced inferior code compared to hand-written stubs. The Flick IDL Compiler [EFF+97] uses custom intermediate representations and traditional compiler optimizations to produce stub code that is superior to most stub compilers. Object serialization in Java is inherently more expensive because object types (i.e. classes) have to be serialized as well.

5 Optimizing Object Serialization

Object serialization, with the cooperation of the RMI system, can be aggressively specialized for homogeneous clusters. This chapter presents a specialization technique called *in-place de-serialization*—de-serialization without allocation and copying of objects—that leverages the zero-copy capabilities of the VI architecture during object de-serialization. The technique makes de-serialization costs independent of object size, which can be beneficial especially when dealing with large objects such as arrays. The challenge is to incorporate incoming objects (in message buffers) into the receiving JVM without compromising the JVM's integrity, without placing any restrictions on subsequent uses of those objects, and without making assumptions about the garbage collector.

In-place de-serialization is realized using *jstreams*, an extension of jbufs with methods to read and write objects from and into communication buffers. The in-memory layout of objects is preserved during serialization so storage allocation and data copying are not needed during de-serialization. By controlling whether a jstream is part of the GC heap, de-serialized objects can be cleanly and safely incorporated into the JVM. Jstreams do not require changes to the Java source language or byte-code and is not dependent on a particular JVM implementation or GC scheme.

The performance of a prototype implementation in Marmot is compared to standard object serialization. Results show that de-serialization costs are comparable to Javia's receive overheads, especially when the implementation is in native code. Jstreams have been incorporated into the RMI system over Javia-II (presented in the previous chapter) with minor modifications to the RMI stub compiler in order to support "polymorphic" remote methods. By eliminating data copying during de-serialization, Jstreams improve the point-to-point performance of RMI substantially, bringing it to within a factor two of that achieved by Javia-II. For many structured applications in the RMI benchmark suite, this improvement reduces overall execution time by 3-10%.

5.1 In-Place Object De-serialization

A JVM-specific protocol preserves the in-memory layout of objects on the wire during serialization. Upon message arrival, serialized objects are integrated into the receiving JVM without having to copy the data from the receive buffers into a newly allocated storage. The design requirements are:

1. The integrity of the JVM on which de-serialization takes place must be preserved. De-serialization should not compromise the JVM and should not corrupt its storage integrity. De-serialized objects should be genuine Java objects as if they had been allocated by the JVM itself.
2. De-serialized objects should be *arbitrary* Java objects. They should not require special annotation other than that required by standard serialization (i.e. objects must implement the `Serializable` interface.)

3. The implementation of de-serialization should be independent of a particular GC scheme.

Because in-place de-serialization requires a JVM-specific protocol, jstreams do not support user-specific extensions to the wire protocol.

The security of the wire itself is not an issue in homogeneous clusters: to make messages truly secure on the wire, one has to resort to cryptographic techniques that are antithetical to high performance.

5.2 Jstreams

A jstream extends a jbuf as follows:

```
1  public class Jstream extends Jbuf {
2
3    /* serializes an object into the stream */
4    public final void writeObject(Object o) throws TypedException,
5      ReadModeException, EndOfStreamException;
6
7    /* clears the stream */
8    public final void writeClear() throws TypedException, ReadModeException;
9
10   /* de-serializes an object, makes stream visible by GC */
11   public final Object readObject() throws TypedException, UnrefException;
12
13   /* unRef and setCallBack methods not shown */
14
15   /* checks if an object resides in a jstream */
16   public final boolean isJstream(Object o);
17
18 }
```

Jstreams replace efficient access through arrays with an object I/O streaming interface. They inherit jbufs' lifetime location control. As seen in Figure 5.1, `writeObject` (line 4) serializes a Java object onto a jstream. If a jstream is full, an `EndOfStreamException` is thrown. `writeClear` (line 8) resets the stream.

`readObject` (line 11) de-serializes an object from a jstream—subsequent `writeObject` and `writeClear` invocations on the same jstream fail with a `ReadModeException` so de-serialized objects are not clobbered. A `readObject` will succeed even without previous `writeObject` invocations

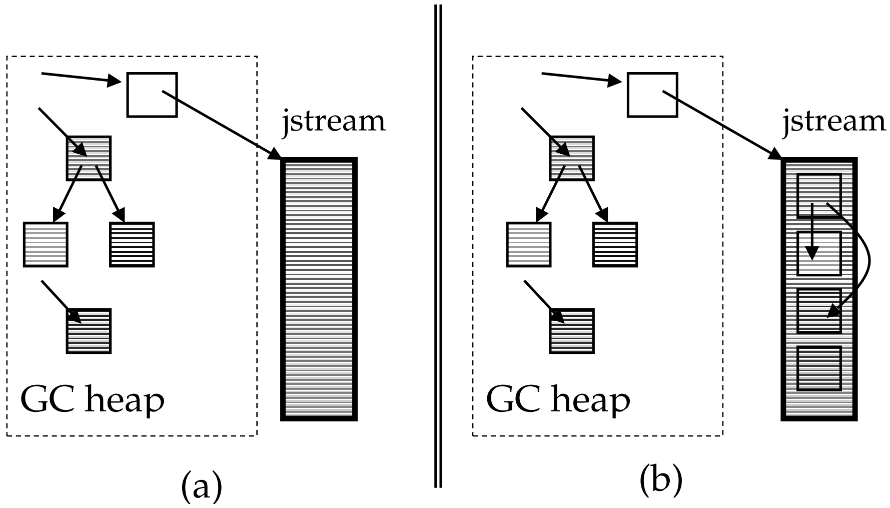

Figure 5.1 Serialization with jstreams. Objects in the GC heap (a) are copied into the jstream (b).

on the same jstream: the data may have been written by a network or I/O device. If no object is left in the stream, an `EndOfStreamException` is thrown.

A jstream becomes part of the GC heap after the first `readObject` invocation (as seen in Figure 5.2(b)) so the GC can track references coming out of the de-serialized objects. To free or re-use a jstream, an application has to first explicitly invoke `unRef` (after which `readObject` will fail with an `UnrefException`), and then wait until the GC invokes the corresponding callback method, as seen in Figure 5.2(c).

As with jbufs, references to de-serialized objects can be *stale* (e.g. the object has been moved out the jstream). Programmers can check whether an object reference is stale by invoking the `isJstream` method line 16).

A `TypedException` is thrown during any of the write and read calls if the jstream is currently being referenced as a jbuf.

Jstreams also support serialization and de-serialization of primitive types with `write/readByte`, `write/readInt`, etc (not shown).

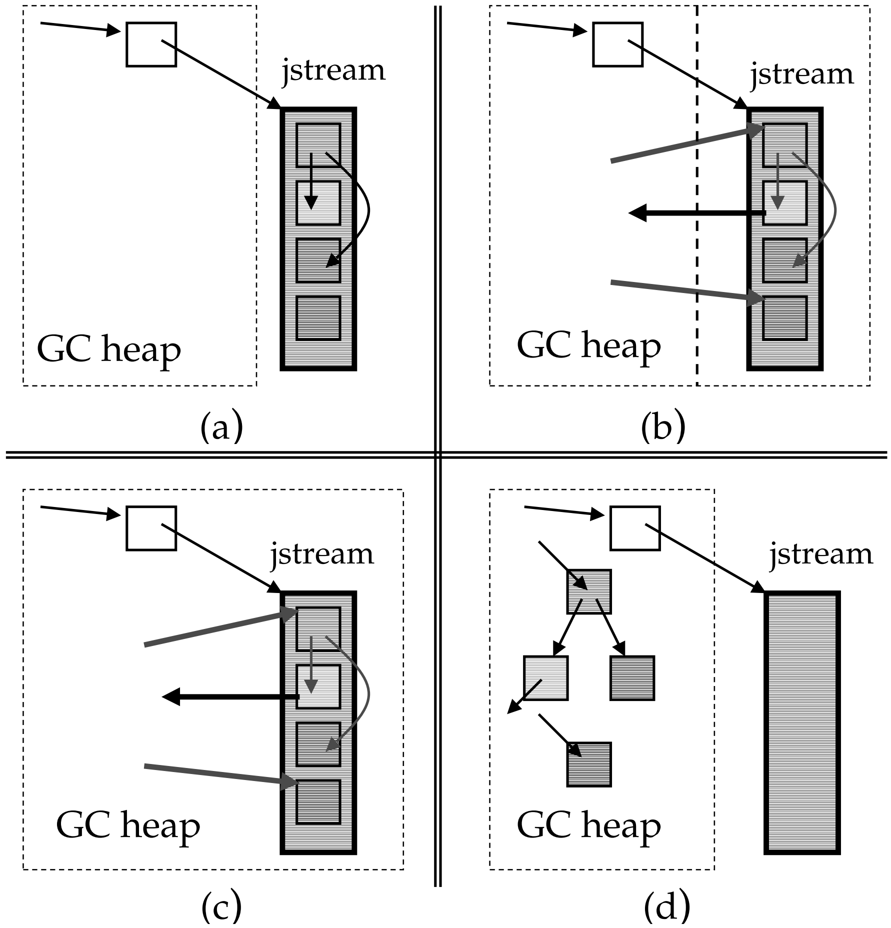

Figure 5.2 Object de-serialization with jstreams. Upon message arrival (a), the objects are de-serialized from the jstream (b): no restrictions are imposed on those objects. After the jstream is explicitly unrefed (c) and the callback invoked (d), it can be de-allocated or re-used.

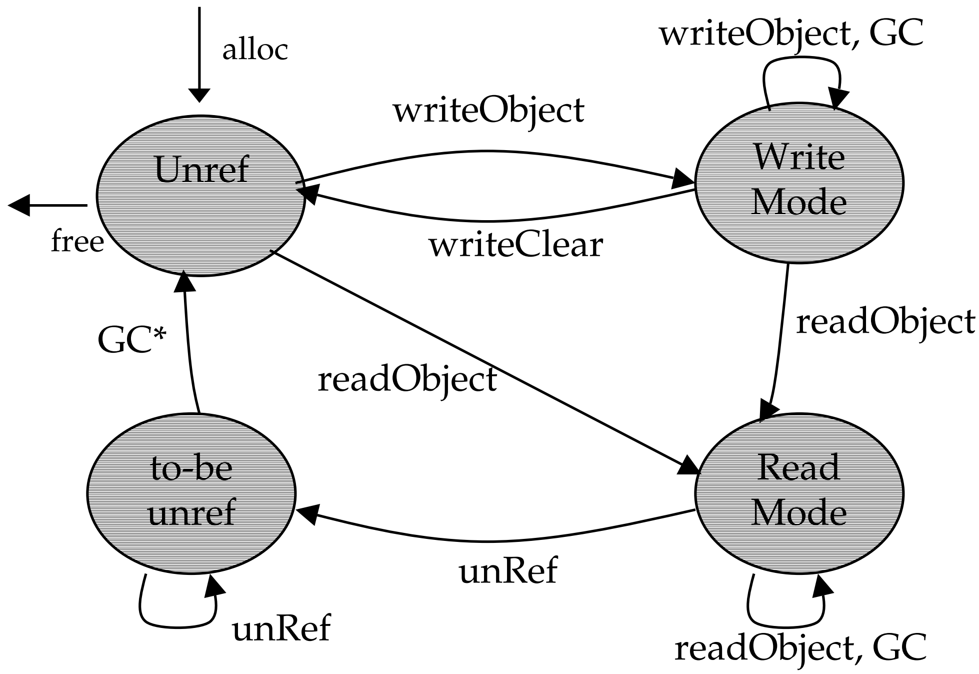

Figure 5.3 Jstreams state diagram for runtime safety checks. When the GC* transition takes place depends on whether the GC is copying or non-copying.

5.2.1 Runtime Safety Checks

Safety is enforced through runtime checks and with the cooperation of the garbage collector. Essentially, a jstream has two modes of operation: a write and a read mode. The state diagram of a jbuf is augmented with three states (shown in Figure 5.3):

1. Write mode (*write*). The jstream contains at least one serialized object and only permits `writeObject` and `writeClear` invocations.
2. Read mode (*read*). There is at least one reference to a de-serialized object in the jstream. Note that, unlike the *ref*<*p*> state of a jbuf, this state is not parameterized by a primitive-type. Only `readObject` calls are permitted in this state.

3. To-be-unreferenced (*2b-unref*). The application has claimed that the jstream has no object references and is awaiting on the GC to verify that claim. Again, note that this state is not parameterized.

A jstream starts at *unref* and goes into *write* mode after a successful invocation of `writeObject`. When a `writeClear` is invoked, the jstream returns to the *unref* state. When a `readObject` is invoked, it goes into the *read* mode (from either *write* or *unref* states). It makes a transition from *read* to the *2b-unref* state after an `unRef` invocation, and returns to the *unref* state after the callback. Note that neither *read* nor *2b-unref* states are parameterized by a primitive-type.

Jstreams also require the cooperation of the network interface so they are not clobbered by the DMA engine. As stated in Section 5.2.7, receive posts are only allowed if the jstream is in the *unref* state.

5.2.2 Serialization

Serialization is implemented by `writeObject` and is based on a JVM-specific wire protocol. `writeObject` traverses the object and all transitively reachable ones and copies them into the stream. The in-memory layout of objects is preserved on the wire[26]. Class objects are not serialized onto the wire; instead, a 64-bit class descriptor[27] is placed in the object's meta-data fields[28]. Jstreams

---

[26] To this end, serialization of object references is delayed until the serialization of the current object is completed.

[27] The descriptor is a checksum with the property that, with very high probability, two classes have the same descriptor only if they are structurally equivalent. A descriptor for class is obtained by invoking the static method `GetSerialVersionUID` in the `ObjectStreamClass` class provided by JOS[27] (in `java.io` package).

[28] Pointers to the virtual method dispatch table (vtable) and to the monitor object are mandatory meta-data fields in an object. Dispatch table is required to support virtual methods in Java. The monitor structure is needed to support per-object synchronization (Section 17.17.3, [LY97]). If these fields are not adjacent to each other on a particular JVM, the type-descriptor can be truncated into 32 bits.

require a two-way mapping between the meta-data and the corresponding 64-bit descriptor in order to expedite serialization and to look up the corresponding meta-data during de-serialization.

Pointers to objects are *swizzled* [Mos92]: they are replaced with offsets to a base address. Offsets of serialized objects are temporarily recorded so they are not re-serialized if the data structure is circular.

As an example, consider the Java class definition for an element of a linked list of byte arrays (`LBA`) as seen in Figure 5.4(a). It contains an integer field `i`, a reference to a byte array `data`, and a pointer to the `next` element in

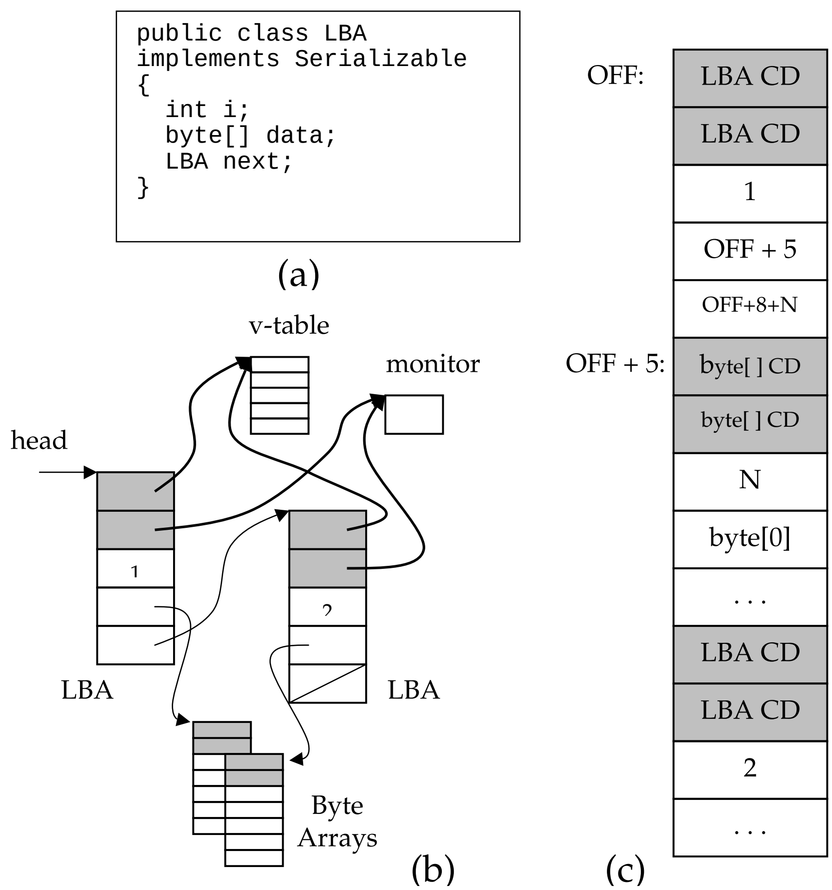

Figure 5.4 Jstreams wire protocol in Marmot. (a) Class definition of a linked-list of byte arrays. (b) In-memory layout of a two-element list in Marmot. (c) The wire representation of the list on Marmot.

the list. Figure 5.4(b) shows the in-memory layout of the list in the Marmot system, and Figure 5.4(c) shows the contents of a jstream after the list is serialized.

5.2.3 De-Serialization

De-serialization is implemented by `readObject`, which traverses the jstream in a depth-first order. For each object, it reads the 64-bit class descriptor from the wire, overwrites the descriptor with the corresponding meta-data, and performs the same procedure recursively on objects referenced by the pointer fields. "Offsets" are first bounds-checked (so as not to exceed the size of the jstream) and then *unswizzled*: actual pointers are obtained by adding the jstream's base address to an offset.

De-serialization must protect a jstream from corrupted or malicious data from the wire: for example, a portable pointer can point to a location that overlaps a previously de-serialized object. To this end, `readObject` tracks "black-out" regions in the jstream—regions that contain de-serialized objects—and rejects any "offset" pointing to one of those regions.

5.2.4 Implementing Jstreams in Marmot

Jstream extends the implementation of the Marmot jbufs two implementations of `writeObject` and `readObject`, one written in C and one in Java. The C implementation contains about 600 lines of code. Three implementation details are worth mentioning:

1. The mapping between vtable pointers and 64-bit type descriptors is constructed during static initialization of class reflection tables (used to support reflection).

2. When traversing the fields of an object, the Java implementation of `writeObject` uses reflection[29] to obtain the object layout information and to tell the reference fields from the non-reference ones; `readObject` needs reflection for the latter reason only. The C implementation relies on the same reflection information for `writeObject`, but uses a concise 32-bit pointer-tracking information that is stored in each class object (this information is also used by the copying garbage collector for the same purpose).
3. After the first invocation of `readObject`, a jstream is incorporated into the copying GC heap as a "pinned segment" in Marmot. It is only promoted to a "collectable" segment after it has been explicitly `unRef`ed. As a pinned segment, the contents of the jstream are visible but not copied during the Cheney scan. The collector handles Java objects residing in a pinned jstream as if they have been stack-allocated [SG98].

5.2.5 Performance

Figures 5.5 and 5.6 show the performance of the Java and C versions of `writeObject` and `readObject` in Marmot respectively. Although the Java version is just as expensive as JOS' in Marmot, the C version is substantially faster primarily because it uses `memcpy`. The de-serialization costs in C are about 2.6µs for arrays and 3.3µs for list elements and are constant with respect to object sizes. Incorporating a jstream into the copying collector as a pinned segment incurs an additional 3.6µs at the first invocation of `readObject`.

---

[29] Reflection in Marmot is augmented to provide object layout information as well: each object field has an offset (to the vtable) and padding information associated with it. The Java Reflection API [Jav99] does not provide object layout information.

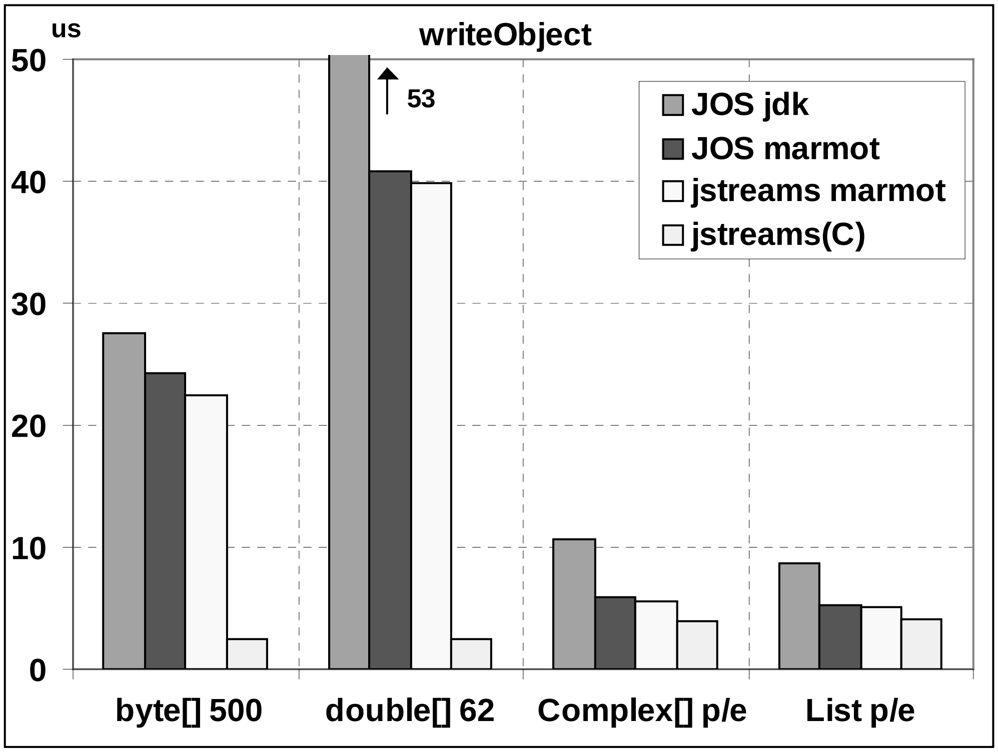

Figure 5.5 Serialization overheads of jstreams in Marmot.

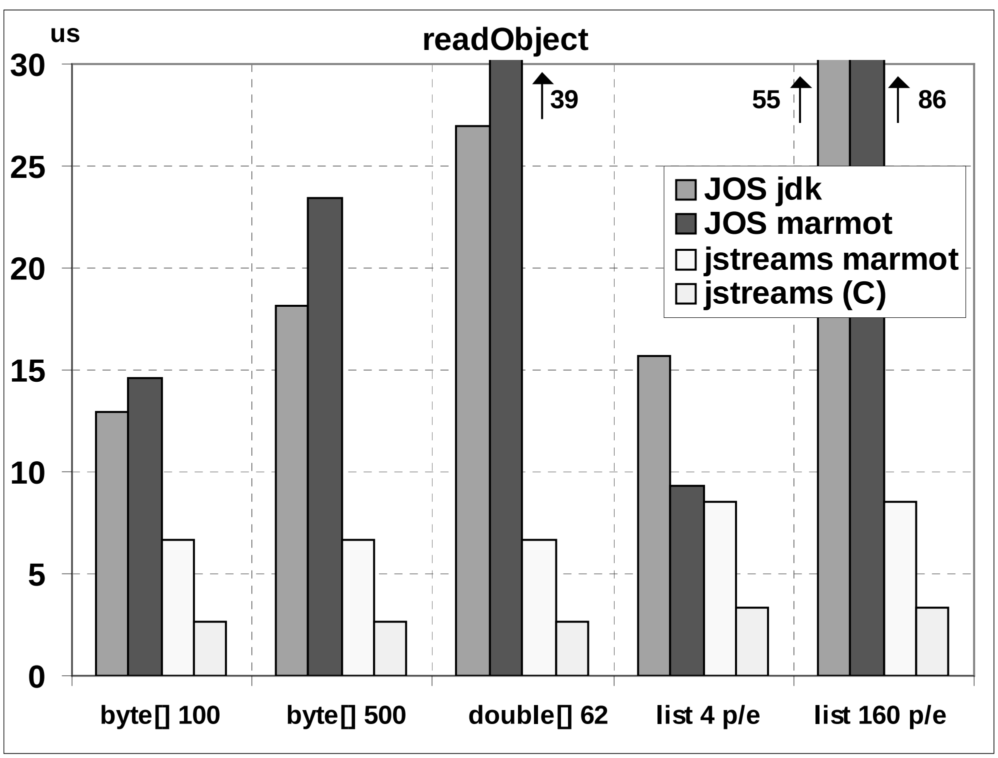

Figure 5.6 De-serialization overheads of jstreams in Marmot.

5.2.6 Enhancements to Javia-II

Jstreams requires the definition of a communication stream, or `ViStream`. A `ViStream` is analogous to a `ViBuffer` (Section 3.2.1) in that it extends a jstream with methods to pin (and unpin) its memory region onto physical memory so it can be directly access by the DMA engine.

A pair of asynchronous send (`sendStreamPost/sendStreamWait`) and receive calls (`recvStreamPost/recvStreamWait`) are added to Javia-II's interface. Receive posts are only allowed if the jstream is in the *unref* state. No architectural changes are made to Javia-II.

5.2.7 Proposed JNI Support

An extension to the JNI can enable more portable implementations of jstreams without revealing two JVM-specific information: the in-memory layout of objects (including pointer-tracking information) and the GC scheme. The proposed extension to JNI consists of five functions as follows:

`void *createMapping(JNIEnv *env);`

> This function returns a two-way table that maps JVM-specific meta-data (e.g. vtables) to 64-bit class descriptors for all classes; null if an error occurs.

`void *createMappingForClass(JNIEnv *env, jobject class);`

> This function returns a two-way table that maps JVM-specific meta-data (e.g. vtables) to 64-bit class descriptors for the specified class, its super-class, and all transitively reachable classes; null if an error occurs. This function is used to support "polymorphic" RMIs (Section 5.3.1).

`void writeObjectNative(JNIEnv *env, void *mapping, jobject obj, char *seg, int seg_size);`

This function makes a deep copy of `obj` based on a JVM-specific protocol that maintains the object layout in the wire. It translates `obj`'s class-related meta-data to 64-bit class descriptors based on `mapping`.

`jobject readObjectNative(JNIEnv *env, void *mapping, char *seg, int seg_size, boolean *isCopy);`

This function returns null if the de-serialization process fails. Class descriptors are translated to meta-data based on `mapping`. If `isCopy` is true, the returned reference has been copied into the GC heap, so `seg` can be re-used (this is conservative: no zero-copy). If `isCopy` is false, then the returned reference points to the object in `seg`, `seg` is automatically added to the garbage-collected heap, and the user can access the object through JNI access methods.

`jobject readObjectNativeCheck(JNIEnv *env, void *mapping, jobject class, char *seg, int seg_size, boolean *isCopy);`

Same as above except that the class of the returned object must match `class`. This function is used to support "polymorphic" RMIs (Section 5.3.1).

5.3 Impact on RMI and Applications

This section evaluates the effect of jstreams on the point-to-point performance of RMI as well as on the RMI benchmark suite. The section starts with a brief description of the modifications to the RMI implementation presented in Section 4.2.2 and of the zero-copy optimization for arrays.

5.3.1 "Polymorphic" RMI over Javia-I/II

In order to support polymorphic RMIs, the following method is added to the `Jstream` class:

```
public class Jstream extends Jbuf {

  /* readObject checks if the descriptor of the class (or of     */
  /* any of its super-class) is the same as that of the argument */
  /* class. */
  public Object readObject(Class arg) throws TypedException,
    UnrefException, ClassMismatchException;
}
```

The method `readObject` checks whether the class descriptor of the de-serialized object's class or any of its super-classes[30] match that of the argument class and throws a `ClassMismatchException` if no match is found. This requires a simple modification to RMI stub compilers: each `readObject` invocation must take the class of the formal parameter as argument.

A remote call object over Javia-II augmented with jstreams has been incorporated into the RMI implementation. A connection is also composed of two virtual interfaces, except that the one for RMI payload is posted with jstreams. As in Jam (Section 3.4.2), the number of jstreams posted on each remote call object is a service parameter. Each incoming RMI consumes a jstream, which in turn is reclaimed on demand by the remote call object.

Specialization is achieved by making the type of the RMI transport a service parameter as well. A high-speed transport is used as long as it supported on both the server and the client sides; otherwise, a slower type of transport (e.g. sockets or Javia-I) is used.

[30] The list of super-class descriptors for each class is computed using reflection during static initialization.

5.3.2 Zero-Copy Array Serialization

Serialization of a primitive-typed array residing in a pinned jbuf is optimized: `writeObject` writes the jbuf's base address and transfer length instead of copying the elements of the array into the stream; `readObject` essentially performs a `to<p>Array`.

Although a jstream is a jbuf itself, the jbuf in which the array resides and the jstream into which the array is serialized can *not* be the same—there is no transition from the *ref*<*p*> state into the *write* state. Therefore, the Java application has to explicitly manipulate arrays in jbufs. Jstreams are used by automatically generated RMI stubs and are thus hidden from applications.

5.3.3 RMI Performance

Figure 5.7 shows the round trip latencies of RMI over Javia-II using jstreams (with zero-copy array serialization, which is about 25μs above that achieved

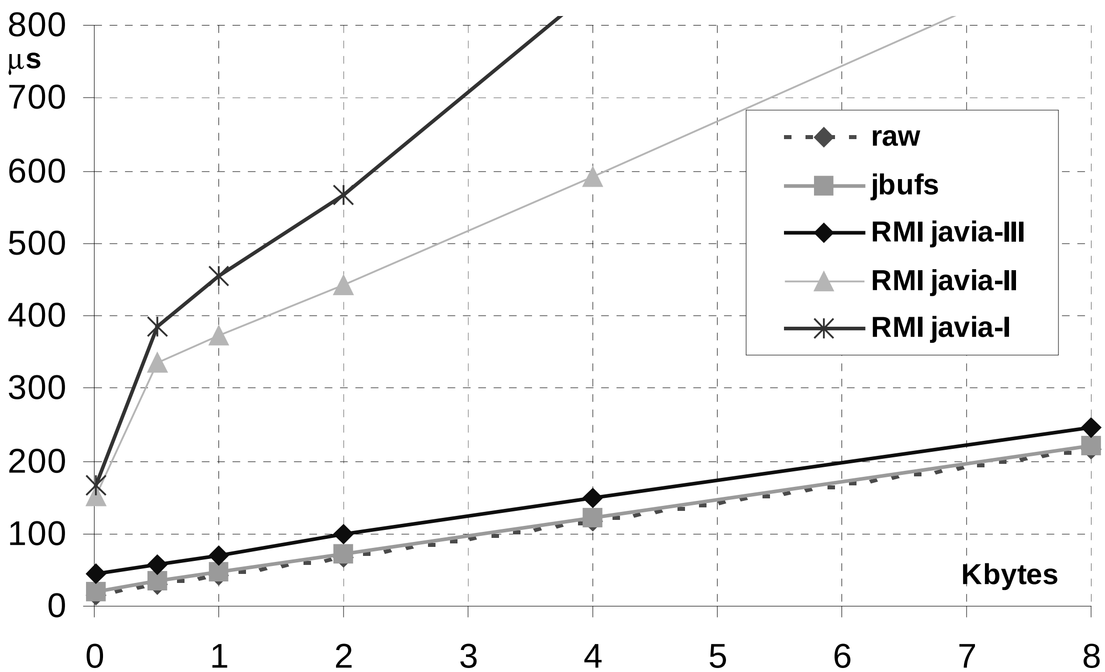

Figure 5.7 RMI round-trip latencies using jstreams.

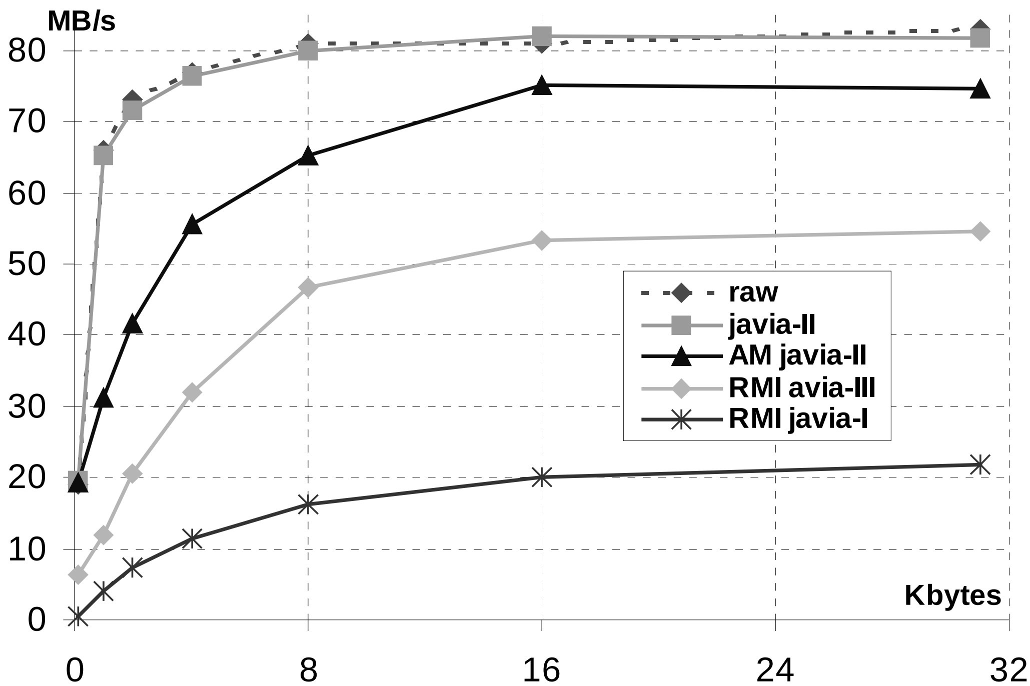

Figure 5.8 RMI effective bandwidth using jstreams.

by a ping-pong program using C. This cost is fixed with respect to the transfer size[31]. Figure 5.8 shows the peak effective bandwidth: about 52MBytes/s compared to about 20MBytes/s attained by RMI over jbufs using JOS. The bandwidth curve reaches the peak at a slower rate than Jam and Javia-II because RMIs are not pipelined (in accordance with the blocking semantics).

5.3.4 Impact on Applications

The same set of benchmark runs as described in Section 4.3.1 are repeated using RMI over Javia-II with jstreams. The runs are taken with zero-copy array serialization enabled. The number of jstreams for each remote object has been chosen so that the fraction of the total time spent in GC is minimized. For SOR, EM3D, and FFT, a pool of 100 jstreams is sufficient to bring that fraction

---

[31] Up to MTU (32Kbytes), beyond which the payload needs to be fragmented (which incurs additional overheads).

to about 1.5%. For pMM, a pool of 1000 jstreams (each of size 2Kbytes) brings the fraction of GC time to about 15%[32].

Table 5.1 Measured Impact of Jstreams on Application Performance. All columns are measured times except for the last two, which are estimated % improvements from Section 4.3.3

Application	*JOS comm (secs)*	*JOS total (secs)*	*jstreams comm (secs)*	*jstreams total (secs)*	*% improv. comm*	*% improv. total*	*% improv. comm (est.)*	*% improv. total (est.)*
SOR	4.59	19.78	3.99	19.08	13.20%	3.52%	11.76%	2.73%
EM3D arrays	2.20	4.60	1.99	4.37	9.50%	4.85%	10.90%	5.22%
FFT complex	18.30	19.03	16.16	17.26	11.70%	9.30%	14.28%	13.73%
FFT arrays	14.82	15.36	14.29	14.83	3.57%	3.40%	1.42%	1.37%
pMM	190.58	280.00	170.91	307.80	10.32%	-9.93%	7.64%	5.20%

Table 5.1 shows the effect of jstreams on the performance of the RMI benchmark suite. Compared with the benchmark results obtained using RMI and JOS over jbufs, the measured improvements in total execution time for all applications but pMM—from 2% to 10%—are comparable to the estimated values presented in Section 4.3.3. Although the communication time in pMM improves by about 10% (about 2x higher than the estimate), the overall execution time actually takes nearly a 10% hit due to poor cache behavior.

5.4 Summary

Jstreams enable zero-copy de-serialization of arbitrary objects by leveraging the location control provided by jbufs. Serialization of simple objects such as primitive-typed arrays can be optimized for zero-copy as well. These optimizations bring the performance of RMI to within a factor two of the raw hardware performance in a homogeneous cluster environment. This translates to a

[32] There are 26 GC occurrences (about 1.2ms per GC); the total amount of time spent in GC (~32ms) is less than 15% of a total execution time of 280ms.

2% to 10% improvement in overall execution time for most applications in RMI benchmark suite.

The jstreams architecture assumes that optimized implementations of `readObject` and `writeObject` should be integrated into the JVM. First, the wire protocol is JVM-specific so as to preserve the layout of objects. Second, implementations in native code are not only faster, but can also utilize internal JVM support for optimizations. For example, the implementation in Marmot takes advantage of the 32-bit pointer-tracking vector used for forwarding pointers during a copying collection. This information would otherwise have to be obtained through some form of reflection, which would certainly be more expensive.

Jstreams do not currently allow for extensibility of the wire protocol, as does JOS. Extensibility through class annotations and user-defined external wire formats is useful for supporting many higher-level communication abstractions. For example, the RMI implementation relies on custom definitions of `readExternal` and `writeExternal` methods to serialize and de-serialize remote objects across the network. Rather than passing remote objects, Jstreams are only used for invoking remote methods that transfer large amounts of data—these methods are typically invoked very frequently and therefore are worth optimizing.

Jstreams in fact only provide limited type checking: they are incapable of preserving *type invariance* [Ten81]. For example, consider a class named `SortedList`, a linked list whose elements are sorted in some fashion. Jstreams check whether the class (or any super-class) of an incoming list matches the expected formal parameter; however, they do not check whether the elements of the list are actually sorted. Checking for type invariance can be accomplished through user-defined extensions. For example, users can define a method called `CheckSorted` in `SortedList` that is invoked by `readObject` before it returns a de-serialized object.

5.5 Related Work

5.5.1 RPC Specialization

Ever since the conception of the Remote Procedure Call in 1984 [BN84], research in RPC systems over the last decade has focused largely on specializing RPC for different types of platforms.

Initial research on RPC focused on inter-machine calls on conventional workstations connected by a regular network (e.g. Ethernet). Key features were reliability, security, and the ability to handle a variety of argument types and to support for multiple transport layers over local or wide-area networks. The overhead introduced by these requirements is well understood and thoroughly reported by Schroeder and Burrows in the context of the DEC Firefly OS [SB90]. In the late 80's, mainstream research was on tuning the RPC implementation for best performance across the network. In the early 90's, focus shifted from cross-machine RPC to cross-domain, or local RPC. Bershad *et. al.* [BAL+90] argued that in micro-kernel operating systems RPC calls occur predominantly between different protection domains (i.e. processes) within the same machine. Lightweight RPC (LRPC) [BAL+90] was motivated by this observation and specializes RPC for the local case by reducing the role of the kernel without compromising safety on uni-processor machines. User-level RPC (URPC) [BAL+92] generalized this idea for shared-memory multiprocessors.

Just as it became necessary to optimize RPC for the local case in conventional operating systems, RPC should be specialized for high performance within a parallel machine. A great deal of research projects in the parallel computing such as MRPC [CCvE99], Concert [KC95], and Orca [BKT92] aimed at improving the performance of RPC on multi-computers. The main theme was to demonstrate that RPC can be efficiently layered over standard message passing libraries while reducing the overheads of method dispatch, multi-threading and synchronization. MRPC specializes RPC for multiple-program-multiple-data parallel programming on multi-computers, replacing heavy-weight, general-purpose RPC runtime systems such as Nexus [FKT96] and Legion [GW96]. Concert depended on special compiler support for performance, while MRPC and Orca only relied on compilers for stub generation. Because multi-computers offers parallel programs dedicated access to the network fabric, security in general was not an issue.

5.5.2 Optimizing Data Representation

Researchers have long pointed out that inadequate data representations and presentation layers exacerbate the cost of serialization. Clark and Tennenhouse [CT90] identified data representation conversion to be the bottleneck for most communication protocols. They advocate the importance of optimizing the presentation layer of a protocol stack. Hoshcka and Huitema [HH94] have attempted to improve protocol processing of self-describing presentation layers (i.e. ASN.1) by combining compiled stub code with interpretation. Despite these efforts, such presentation layers are rarely used by RPC systems due to the high decoding cost. The Universal Stub Compiler [OPM94] provides the user with the flexibility to specify the representation of data types. It uses this information to reduce the amount of copying during marshaling.

In-place de-serialization uses the in-memory representation of data as their wire representation, making it possible to eliminate copying altogether.

5.5.3 Zero-Copy RPC

Many high-performance RPC systems designed and implemented for commodity workstations have attempted to reduce the amount of data copying to the extent possible. In the Firefly RPC system [SB90], data representation is negotiated at bind time and copies are avoided by direct DMA transfers from a marshaled buffer and by receive buffers statically-mapped into all address spaces. Amoeba's [TvRS+91] RPC is built on top of message passing primitives and does not enforce any specific data representation. In the Peregrine system [JZ91], arguments are passed on client stub's stack. The stub traps into the kernel so that the DMA can transfer data directly out of the stack into the server's stub stack. Peregrine also supports multi-packet RPC efficiently. The authors reported a round trip RPC overhead of 309μs on diskless Sun-3/60 connected by 10MBits/s Ethernet. About 50% of this overhead were due to kernel traps, context switches, and receive interrupt handling.

The RPC overheads of the above systems are dominated by kernel's involvement in the critical pat. SHRIMP Fast RPC project [BF96] optimizes RPC for commodity workstations equipped with user-level network interfaces. Fast RPC achieves a round-trip latency of about 9.5μs, 1μs above the hardware minimum (between two 60Mhz Pentium PCs running Linux), and uses a custom format for data streams. This is closely related to the JVM-specific wire format required by jstreams. It is unclear whether Fast RPC is able to support linked C structures; jstreams not only handles linked object structures but also incorporates typing information.

A more recent effort by the Shah et al. [SPM99] shows implementations of legacy RPC systems on top of a Giganet GNN-1000 adapter. A user-level implementation of RPC achieves a 4-byte round-trip latency of about 110μs on a server system with four 400Mhz Pentium-II Xeon™ processors, which is over 5x times higher than GNN-1000 raw latency.

The idea of in-place de-serialization was first adopted by J-RPC, a zero-copy RPC system for Java [CvE98]. Object de-serialization and the interaction with the GC are hardwired in J-RPC, making it difficult to generalize the idea for other communication models.

5.5.4 Persistent Object Systems

The idea of *unswizzling* pointers before they are incorporated into the object heap has been exploited in many systems, most notably in persistent object systems [Kae86, Mos92, WK92, HM93b]. The problem is that persistent stores may grow so large that they contain more objects than can be addressed directly by the available hardware. Persistent store pointers have thus to be converted into virtual memory addresses when objects are read from persistent storage much like having to convert "offsets" into pointers in the receiving JVM. Unlike unswizzling on discovery (doing the conversion in a lazy fashion, at use) [WK92], jstreams perform unswizzling all at once [Mos92].

6 Conclusions

The networking performance in state-of-the-art Java systems is not commensurate with that of high-performance network interfaces for cluster computing. This thesis argues that the fundamental bottlenecks in the data-transfer path are the (i) separation between Java's garbage-collected heap and the native, non-collected heap that is directly accessible to network interfaces (Chapter 2), and (ii) the high costs of object serialization (Chapter 4). The first bottleneck is inherent to the interaction between a storage-safe language and the underlying networking hardware; the second is inherent to the language's type-safety. Although this thesis studies these bottlenecks in the context of Java, we believe they are applicable to other safe languages.

The approach proposed in this thesis—explicit buffer management—is motivated by state-of-the-art user-level network interfaces. The thesis is that, in order to take advantage of zero-copy capabilities of network devices, programmers should be able to perform buffer management tasks in Java just as they can in C, and most importantly, without breaking the storage and type safety in Java. To this end, the main contributions of *jbufs* (Chapter 3) are in (i) *recognizing* the role of the garbage collector in explicit memory management, namely the ability to verify whether a buffer can be re-used or de-allocated safely, in (ii) *exposing* this role to programmers through a simple interface (`unRef` and a callback), and in (iii) *identifying* the essential support needed from a garbage collector that is independent of the collection scheme, namely the ability to change the scope of the collected heap dynamically.

Jbufs offer two key benefits for Java applications that directly interact with network interfaces: efficient access to a non-collected region of memory as primitive-typed arrays and the ability to re-use that region. Our experiences with cluster matrix multiplication (Section 3.3) suggest that efficient access is convenient, reduces communication times, but currently has limited impact on overall performance, which is dominated by poor cache locality and by runtime safety checks. Our experiences with an implementation of Active Messages (Section 3.4) indicate that the buffer re-use is useful: for example, communication system designers can implement their own buffer management schemes or delegate them to applications. However, our experiences with Java RMI (Section 4.2) using standard object serialization reveal that efficient access and buffer re-use are essentially immaterial: overheads are dominated by high serialization costs.

Jbufs stand out from related approaches in that they can be extended to support in-place object de-serialization in a clean, safe, and efficient manner (Chapter 5). The resulting abstraction, *jstreams*, is able to cut the cost of object de-serialization to a constant irrespective to object size on homogeneous clusters. This translates to an order of magnitude improvement in point-to-point RMI performance and improvements (of up to 10%) to a set of benchmarked RMI-based applications. The re-use of jstreams allows applications to tune the RMI system for performance; measuring the effectiveness of this tuning, however, is beyond the scope of this thesis.

For applications that exchange RMIs intensively, such as cluster matrix multiplication, two variables come in play when tuning the size of the buffer pool: data locality and garbage collection. A pool with a large number of jstreams decreases the frequency of garbage collections, but harms the data locality of the application. Our experiences with cluster matrix multiplication reveal that tuning the buffer pool size is hard and that the overall application performance with jstreams can be actually worse (by at least 10%) than with standard object serialization. This is in part attributed to the semi-space copying collector being used, which yields high collection costs (15% of total execution time). A generational collector will likely reduce these costs in a substantial way.

Jstreams employ a simple optimization during serialization: array objects that reside in jbufs are transferred in a zero-copy fashion. While it works well for all the RMI applications used in this thesis, achieving zero-copy serialization of *arbitrary* objects in a *clean* fashion is still an open problem. The fundamental difficulty is that objects can be scattered all over the heap[33]; even with user cooperation (e.g. having the user allocate objects into a single jbuf[34] so they remain "adjacent" with one another), it is still difficult to control the location of all Java objects[35].

The ideas presented in this thesis are applicable to other kinds of high-performance Java applications that interact with I/O devices, such as file systems and persistent object systems. For example, a file could be memory-

---

[33] Solutions based on DMA scatter-gather operations are vulnerable in that they do not scale well (due to resource limitations of the underlying network interface) and that scatter-gather operations are expensive to set up [MNV+99].

[34] A jbuf can be easily extended with an object allocation interface.

[35] For example, character arrays of Java strings are typically "interned" in some internal table maintained by the JVM.

mapped into a jbuf and accessed directly by Java file I/O streams. Also, a heap of persistent objects could be serialized into stable storage, and later in-place de-serialized and incorported into the JVM. It is clear that the base performance of these systems will improve [Wel99], though substantial improvements in overall application performance remains to be seen.

Bibliography

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Cruz, A. Greebaum, S. Hammarling, A. McKeneey, S. Ostrouchov, and D. Sorensen. LAPACK User's Guide. *SIAM*, Philadelphia, 1992.

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

[ACR+96] B. Ang, D. Chiou, L. Rudolph, and Arvind. Message Passing Support on Star-T Voyager. *CSG-Memo-387*, MIT Laboratory for Computer Science, July 1996.

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

[AG97] K. Arnold and J. Gosling. The Java Programming Language. *Addilson-Wesley*, 1997.

[BAL+90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. In *ACM Transactions on Computer Systems (TOCS),* 8(1):37-55, February 1990.

[BAL+92] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. In *ACM Transactions on Computer Systems (TOCS),* 9(2):175-198, May 1991.

[BCF95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro, vol.15, no.1*, February 1995, pp29-36.

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

[BEN+94] A. Birrell, D. Evers, G. Nelson, S. Owicki, and T. Wobber. Distributed Garbage Collection for Network Objects. *DEC Systems Research Center Technical Report*, 1994.

[BF96] A. Bilas and E. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *Tech. Rep. TR-512-96*, Princeton University, 1996.

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

[BJM+96] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996, pp245-260.

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

[BKT92] H. Bal, M. F. Kaashoek, and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. In *IEEE Trans. on Software Engineering*, 18(3):190-205, March 1992.

[BN84] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 4(1):39-59, February 1984.

[Boe93] H-J. Boehm. Space-efficient conservative garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993, pp197-206.

[Buo99] Philip Buonadonna. Personal communication, September 1999.

[BW88] H-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807-820, 1988.

[BZ93] D. Barrett and B. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, Albuquerque, NM, June 1993, pp187-196.

[CCH+96] C-C. Chang, G. Czajkowski, C. Hawblitz, and T. von Eicken. Low Latency Communication on the IBM RISC System/6000 SP. In *Proceedings of ACM/IEEE Supercomputing*, Pittsburgh, PA, November 1996.

[CCvE96] C-C. Chang, G. Czajkowski, and T. von Eicken. Design and Implementation of Active Messages on the IBM SP-2. *Cornell CS Technical Report 96-1572*, February 1996.

[CCvE99] C-C. Chang, G. Czajkowski, and T. von Eicken. MRPC: A High Performance RPC System for MPMD Parallel Computing. *Software: Practice and Experience (29)1:43-66*, 1999.

[Cd99] Central Data. http://www.cd.com.

[CDG+93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of ACM/IEEE Supercomputing*, Portland, OR, November 1993.

[CFK+99] B. Carpenter, G. Fox, S-H. Ko, and S. Lim. Object Serialization for Marshaling Data in a Java Interface to MPI. In *ACM SIGPLAN Java Grande Conference*, San Francisco, CA, June 1999.

[CKL+94] D. Culler, K. Keeton, L. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright, Generic Active Messages Specification, white paper, November 1994.

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

[CT90] Clark, D., and Tennenhouse, D. Architectural Considerations for a New Generation of Protocols. In *Proc. of the SIGCOMM*, 1990, pp.200-208.

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

[CvE99a] C-C. Chang and T. von Eicken. Interfacing Java to the Virtual Interface Architecture. In *Proceedings of ACM SIGPLAN Java Grande Conference*, San Francisco, CA, June 1999, pp51-57.

[CvE99b] C-C. Chang and T. von Eicken. Javia: A Java Interface to the Virtual Interface Architecture. Concurrency: Practice and Experience, to appear.

[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 1998.

[EFF+97] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *Proc. of ACM Conference on Programming Language Design and Implementation (PLDI),* Las Vegas, NV, 1997.

[Fer98] A. Ferrari. JPVM: Network Parallel Computing in Java. In *ACM 1998 Workshop for Java for High-Performance Network Computing*, February 1998.

[Fft99] Fastest Fourier Transform in the West. http://www.fftw.org

[FKR+99] R. Fitzgerald, T. Knoblock, E. Ruf, B. Steensgard, and D. Tarditi. Marmot: An Optimizing Compiler for Java. *Microsoft Research Technical Report 99-33*, June 1999.

[FKT96] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach for Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing (JPDC)*, 37, 1996.

[GA98] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

[GFH+98] V. Getov, 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.

[Gig98] Giganet, Inc. http://www.giganet.com

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

[GvL89] G. Golub and C. van Loan. Matrix Computations. *The John Hopkins University Press*, 2nd Edition, 1989.

[GW96] A. Grimshaw, and W. Wulf. Legion -- A View From 50,000 Feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing,* Los Alamitos, California, August 1996.

[Han90] D. Hanson. Fast allocation and de-allocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5-12, January 1990.

[HH94] Hoshcka, P., and Huitema, C. Automatic Generation of Optimized Code for Marshaling Routines. In *Proc. of Int'l Working Conference on Upper Layer Protocols, Architectures, and Applications*, Barcelona, Spain, 1994.

[HM93a] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In Proceedings of Hot Interconnects, August 1993.

[HM93b] A. Hosking and J. E. Moss. Protection Traps and Alternatives for Memory Management of an Object-Oriented Language. In *Proceedings of ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[Hue96] L. Huelsbergen. A Portable C Interface for Standard ML of New Jersey. *AT&T Bell Laboratories Technical Memorandum*, January 1996.

[Int99] Intel® Math Kernel Library Performance Specification http://developer.intel.com/vtune/perflibst/mkl/mklperf.htm

[Jav99] Javasoft. The Source for Java Technology. http://www.javasoft.com

[JCS+99] G. Judd, M. Clement, and Q. Snell. Design Issues for Efficient Implementation of MPI in Java. In *ACM SIGPLAN Java Grande Conference*, San Francisco, CA, June 1999.

[Jd97] J/Direct: Microsoft Technologies for Java. http://www.microsoft.com/java.

[Jg98] The Java Grande Consortium. http://www.javagrande.org

[Jg99] Interim Java Grande Forum Report JGF-TR-4. http://www.javagrande.org/reports.htm

[Jos99] Javasoft. Java Object Serialization Specification. http://www.javasoft.com

[Jni97] Javasoft. Java Native Interface Specification. http://www.javasoft.com.

[JZ91] D. Johnson and W. Zwaenepoel. The Peregrine high performance RPC system. *Tech. Rep. COMPTR91-152*, Dept. of Computer Science, Rice University, 1991.

[Kae86] T. Kaehler, Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *Proceedings of the Conference on OOSPLA*, Portland, OR, September 1986.

[Kaf97] Transvirtual Technologies. The Kaffe OpenVM. http://www.transvirtual.com

[KC95] V. Karamcheti, and A. Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of ACM/IEEE Supercomputing*, Portland, OR, Nov 1993.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Henessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, April 1994.

[KSS+96] Krishnamurthy, K. Schauser, C. Scheiman, R. Wang, D. Culler, and K. Yelick. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996.

[KWB+98] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient Implementations of Java

RMI. In 4^{th} *USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, NM, 1998.

[LY97] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. *Addilson-Wesley*, 2^{nd} Edition, September 1996.

[MC95] A. Mainwaring and D. Culler. Active Messages Application Programming Interface and Communication Subsystem Organization. Draft technical report, 1995.

[Mic98] Microsoft. Notice Regarding Java Lawsuit Rulling http://msdn.microsoft.com/visualj/statement.asp

[Mic99] Microsoft Technologies for Java. http://www.microsoft.com/java

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

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

[Mos92] J. E. Moss. Working with Persistent Objects: to Swizzle or not to Unswizzle. *IEEE Transactions on Software Engineering*, 18(8):657-673, August 1992.

[NMB+99] R. Nieuwpoort, J. Maassen, H. Bal, T. Kielmann, R. Veldema. Wide-Area Parallel Computing in Java. In *ACM SIGPLAN Java Grande Conference*, San Francisco, CA, June 1999, pp8-14.

[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, pp152-159.

[OPM94] O'Malley, S. Proebsting, T., and Montz, A. USC: A Universal Stub Compiler. In *Proceedings. of the SIGCOMM*, London, UK, 1994, pp.295-306.

[OW97] M. Odersky and P. Wadler. Pizza into Java: translating theory into practice. In *Proceedings of the* 24^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997, pp.146-159.

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

[RLW94] S. Reinhardt, J. Larus, and D. Wood. Typhoon and Tempest: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, April 1994.

[Rmi99] Java Remote Method Invocation. http://java.sun.com.

[Ros67] D. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481-492, August 1967.

[SB90] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *ACM Transactions on Computer Systems (TOCS)*, 8(1):1-17, February 1990.

[SG98] B. Steengard and D. Gay. Stack Allocating Objects in Java. Microsoft Research Technical Report, October 1998.

[SHA95] D. Schmidt, T. Harrison, and E. Al-Shaer. Object-Oriented Components for High-Speed Network Programming. In *Proc. of the First USENIX Conference on Object-Oriented Technologies and Systems*, Monterrey, CA, June 1995.

[SPM99] H. Shah, C. Pu, R. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC)*, 1999.

[SS95] K. Schauser and C. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS),* Santa Barbara, CA, April 1995.

[Sto97] D. Stoutamire. Portable, Modular Expression of Locality. *PhD thesis*, University of California, Berkeley, 1997.

[Tar99] D. Tarditi. Personal communication. August 1999.

[Ten81] R. Tennent. Principles of Programming Languages. *Prentice Hall*, 1981.

[TvRS+91] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. vanRossum. Experiences with the

Amoeba distributed operating system. Comm of the ACM, 33(12):46-63, Dec 1990.

[vEBB+95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. *U-Net: A User-level Network Interface for Parallel and Distributed Computing.* In Proceedings of the 15th Annual Symposium on Operating System Principles, Copper Mountain Resort, Colorado, Dec. 1995, pp40-53.

[vECS+92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium in Computer Architecture (ISCA)*, Gold Coast, Australia, May 1992.

[Via97] The Virtual Interface Architecture. http://www.via.org

[Vo96] K-P. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software: Practice and Experience*, 26(3):357-374, March 1996.

[WBvE97] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Stanford, CA, August 1997.

[WC99] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, to appear.

[Wel99] M. Welsh. Tigris: A Cluster-Based I/O System. Submitted for publication, June 1999.

[Wil92] P. Wilson. Uniprocessor Garbage Collection. In *Proceedings of Int'l Workshop on Memory Management*, St. Malo, France, September 1992. LNCS 637, Springer-Verlag.

[WJN+95] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of Int'l Workshop on Memory Management*, Kinross, Scotland, September 1995. LNCS 986, Springer-Verlag.

[WK92] P. Wilson and S. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proceedings of 1992 Int'l Workshop on Object Orientation in Operating Systems (WOOOS)*, Paris, France, September 1992, pp.364-377.

[YSP+98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-Performance Java Dialect. In *ACM 1998 Workshop for Java for High-Performance Network Computing*, February 1998.