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.