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.