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

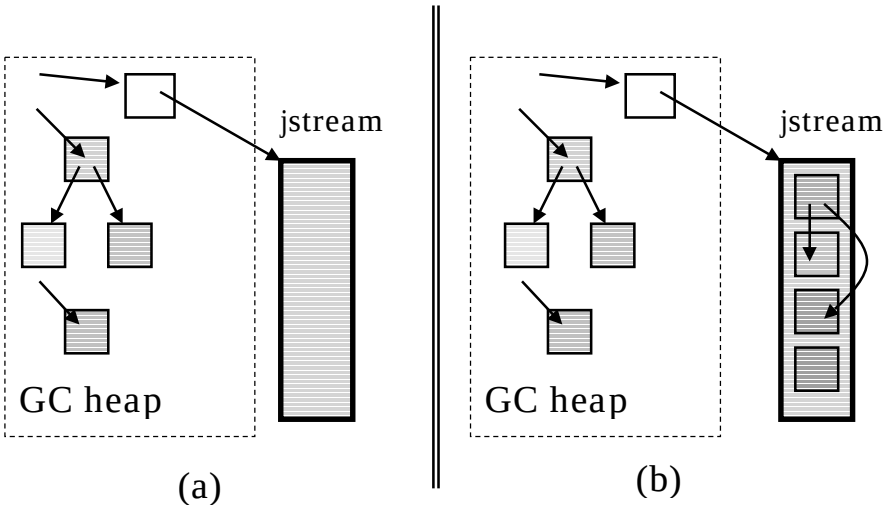

Figure 5.1 Serialization with jstreams. Objects in the GC heap (a) are copied into the jstream (b).

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).

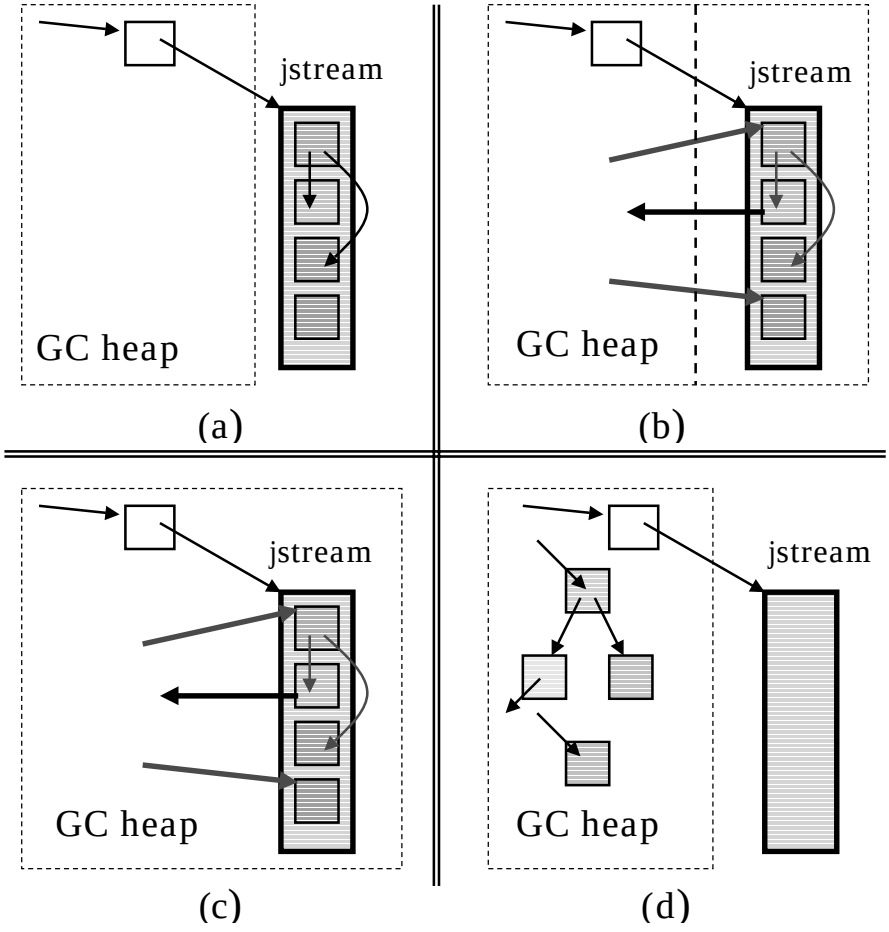

Figure 5.2 Object de-serialization with jstreams. Upon message arrival (a), the objects are de-serialized from the jstream (b): no restrictions are imposed on those objects. After the jstream is explicitly unrefed (c) and the callback invoked (d), it can be de-allocated or re-used.

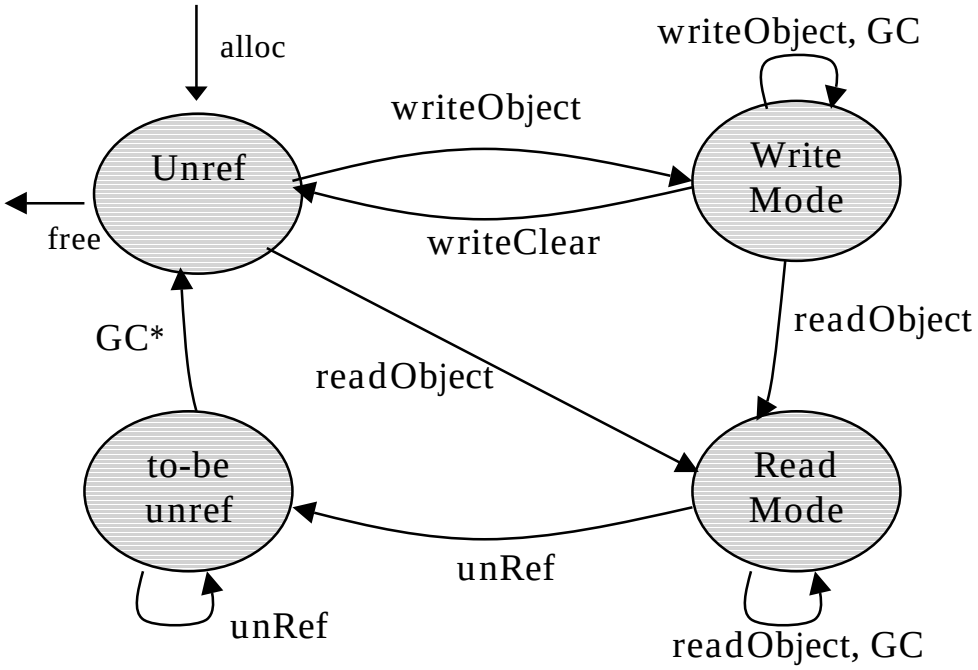

Figure 5.3 Jstreams state diagram for runtime safety checks. When the GC* transition takes place depends on whether the GC is copying or non-copying.

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

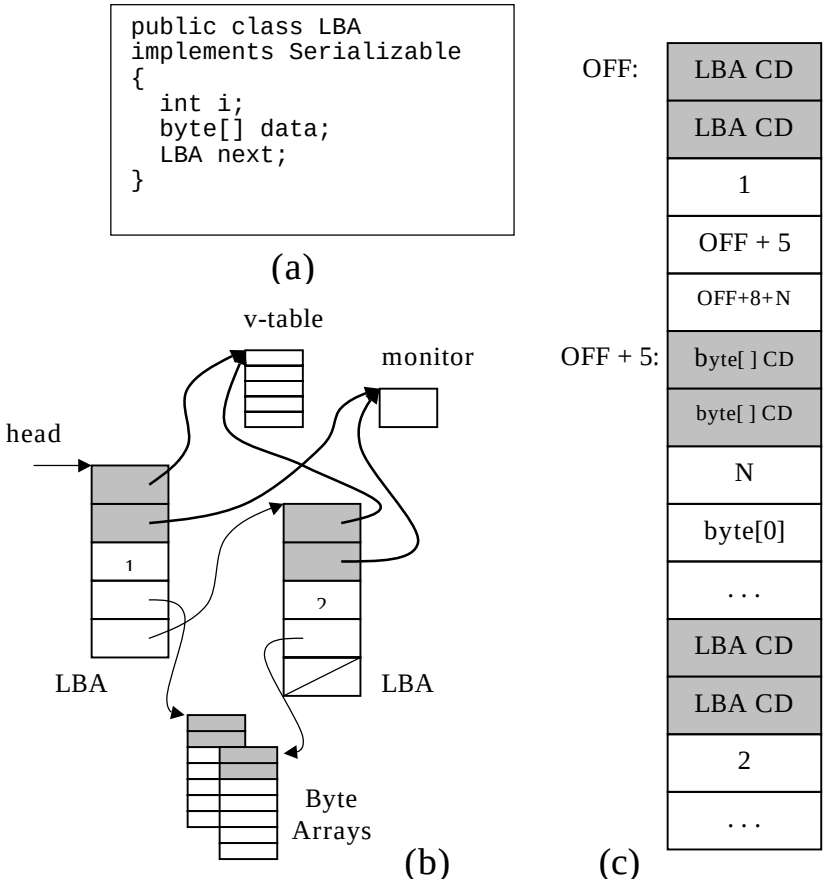

Figure 5.4 Jstreams wire protocol in Marmot. (a) Class definition of a linked-list of byte arrays. (b) In-memory layout of a two-element list in Marmot. (c) The wire representation of the list on Marmot.

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.

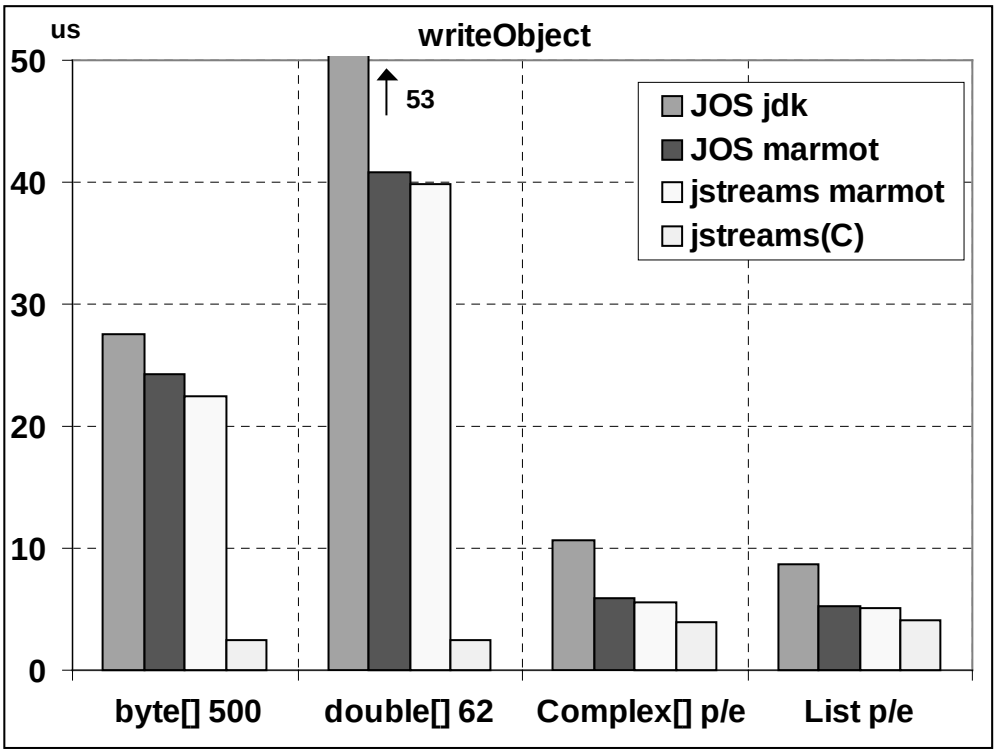

Figure 5.5 Serialization overheads of jstreams in Marmot.

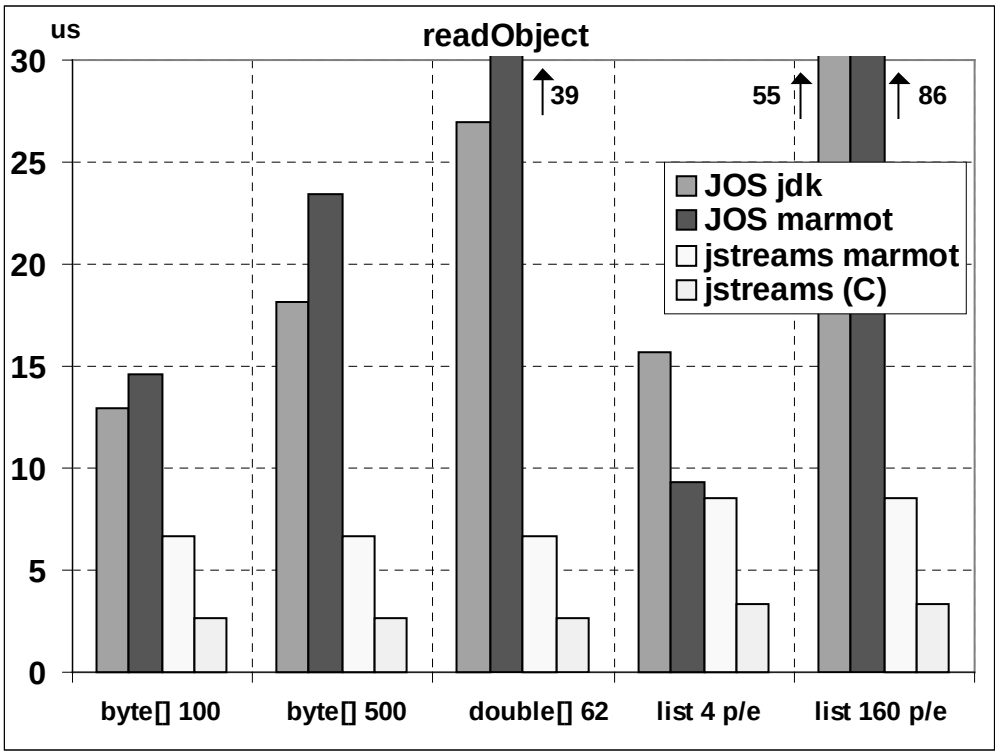

Figure 5.6 De-serialization overheads of jstreams in Marmot.

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 metadata (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 metadata (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:

```
1  public class Jstream extends Jbuf {
2
3    /* readObject checks if the descriptor of the class (or of     */
4    /* any of its super-class) is the same as that of the argument */
5    /* class. */
6    public Object readObject(Class arg) throws TypedException,
7      UnrefException, ClassMismatchException;
8  }
```

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

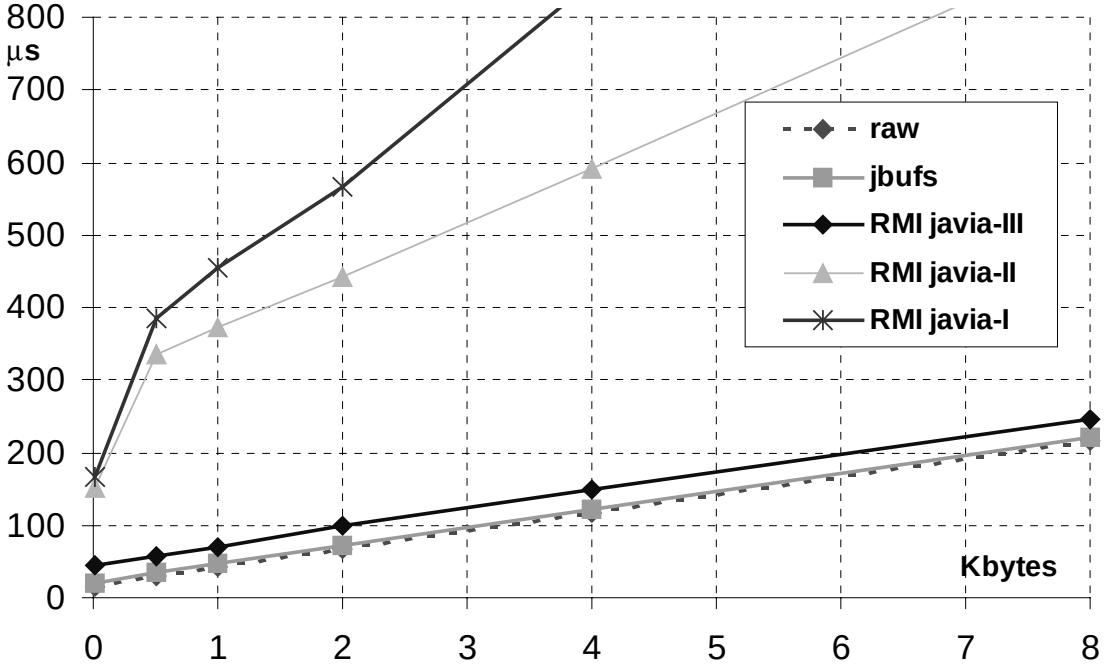

Figure 5.7 RMI round-trip latencies using jstreams.

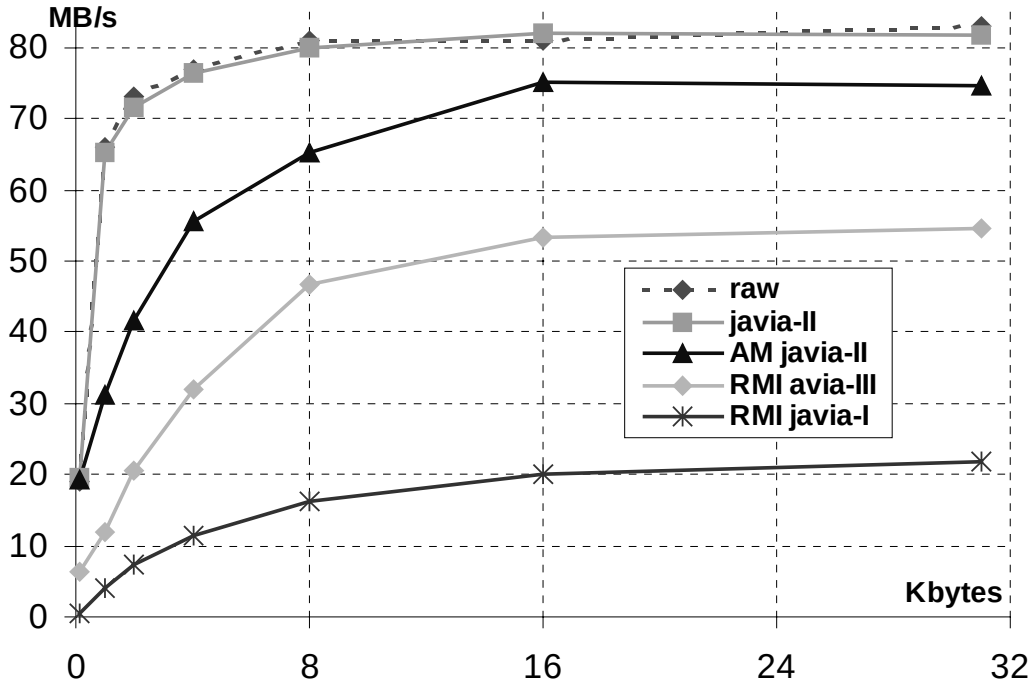

Figure 5.8 RMI effective bandwidth using jstreams.

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].