Efficient Java RMI for Parallel Programming

JASON MAASSEN, ROB VAN NIEUWPOORT, RONALD VELDEMA,
HENRI BAL, THILO KIELMANN, CERIEL JACOBS, and RUTGER HOFMAN
Vrije Universiteit, Amsterdam

Java offers interesting opportunities for parallel computing. In particular, Java Remote Method Invocation (RMI) provides a flexible kind of remote procedure call (RPC) that supports polymorphism. Sun's RMI implementation achieves this kind of flexibility at the cost of a major runtime overhead. The goal of this article is to show that RMI can be implemented efficiently, while still supporting polymorphism and allowing interoperability with Java Virtual Machines (JVMs). We study a new approach for implementing RMI, using a compiler-based Java system called Manta. Manta uses a native (static) compiler instead of a just-in-time compiler. To implement RMI efficiently, Manta exploits compile-time type information for generating specialized serializers. Also, it uses an efficient RMI protocol and fast low-level communication protocols.

A difficult problem with this approach is how to support polymorphism and interoperability. One of the consequences of polymorphism is that an RMI implementation must be able to download remote classes into an application during runtime. Manta solves this problem by using a dynamic bytecode compiler, which is capable of compiling and linking bytecode into a running application. To allow interoperability with JVMs, Manta also implements the Sun RMI protocol (i.e., the standard RMI protocol), in addition to its own protocol.

We evaluate the performance of Manta using benchmarks and applications that run on a 32-node Myrinet cluster. The time for a null-RMI (without parameters or a return value) of Manta is 35 times lower than for the Sun JDK 1.2, and only slightly higher than for a C-based RPC protocol. This high performance is accomplished by pushing almost all of the runtime overhead of RMI to compile time. We study the performance differences between the Manta and the Sun RMI protocols in detail. The poor performance of the Sun RMI protocol is in part due to an inefficient implementation of the protocol. To allow a fair comparison, we compiled the applications and the Sun RMI protocol with the native Manta compiler. The results show that Manta's null-RMI latency is still eight times lower than for the compiled Sun RMI protocol and that Manta's efficient RMI protocol results in 1.8 to 3.4 times higher speedups for four out of six applications.

Authors' address: Division of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands.

1. INTRODUCTION

There is a growing interest in using Java for high-performance parallel applications. Java's clean and type-safe object-oriented programming model and its support for concurrency make it an attractive environment for writing reliable, large-scale parallel programs. For shared memory machines, Java offers a familiar multithreading paradigm. For distributed memory machines, such as clusters of workstations, Java provides Remote Method Invocation (RMI), which is an object-oriented version of Remote Procedure Call (RPC). The RMI model offers many advantages for distributed programming, including a seamless integration with Java's object model, heterogeneity, and flexibility [Waldo 1998].

Unfortunately, many existing Java implementations have inferior performance of both sequential code and communication primitives, which is a serious disadvantage for high-performance computing. Much effort is being invested in improving sequential code performance by replacing the original bytecode interpretation scheme with just-in-time compilers, native compilers, and specialized hardware [Burke et al. 1999; Krall and Grafl 1997; Muller et al. 1997; Proebsting et al. 1997]. The communication overhead of RMI implementations, however, remains a major weakness. RMI is designed for client/server programming in distributed (Web based) systems, where network latencies on the order of several milliseconds are typical. On more tightly coupled parallel machines, such latencies are unacceptable. On our Pentium Pro/Myrinet cluster, for example, Sun's JDK 1.2 implementation of RMI obtains a *null-RMI latency* (i.e., the roundtrip time of an RMI without parameters or a return value) of 1,316   s, compared to 31   s for a user-level Remote Procedure Call protocol in C.

Part of this large overhead is caused by inefficiencies in the JDK implementation of RMI, which is built on a hierarchy of stream classes that copy data and call virtual methods. Serialization of method arguments (i.e., converting them to arrays of bytes) is implemented by recursively inspecting object types until primitive types are reached, and then invoking the primitive serializers. All of this is performed at runtime for each remote invocation.

Besides inefficiencies in the JDK implementation of RMI, a second reason for the slowness of RMI is the difference between the RPC and RMI models. Java's RMI model is designed for flexibility and interoperability. Unlike RPC, it allows classes unknown at compile time to be exchanged between a client and a server and to be downloaded into a running program. In Java, an actual parameter object in an RMI can be of a subclass of the class of the method's formal parameter. In (polymorphic) object-oriented languages, the *dynamic* type of the parameter-object (the subclass) should be used by the method, not the static type of the formal parameter. When the subclass is not yet known to the receiver, it has to be fetched from a file or HTTP server and be downloaded into the receiver. This high level of flexibility is the key distinction between RMI and RPC [Waldo 1998]. RPC systems simply use the static type of the formal parameter (thereby type-converting the actual parameter), and thus lack support for polymorphism and break the object-oriented model.

The key problem is to obtain the efficiency of RPC *and* the flexibility of Java's RMI. This article discusses a compiler-based Java system, called *Manta*,[1] which was designed from scratch to efficiently implement RMI. Manta replaces Sun's runtime protocol processing as much as possible by compile-time analysis. Manta uses a native compiler to generate efficient sequential code and specialized serialization routines for serializable argument classes. Also, Manta sends type descriptors for argument classes only once per destination machine, instead of once for every RMI. In this way, almost all of the protocol overhead has been pushed to compile time, off the critical path. The problems with this approach are, however, how to interface with Java Virtual Machines (JVMs) and how to address dynamic class loading. Both are required to support interoperability and polymorphism. To interoperate with JVMs, Manta supports the Sun RMI and serialization protocol, in addition to its own protocol. Dynamic class loading is supported by compiling methods and generating serializers at runtime.

The general strategy of Manta is to make the frequent case fast. Since Manta is designed for parallel processing, we assume that the frequent case is communication between Manta processes, running, for example, on different nodes within a cluster. Manta supports the infrequent case (communication with JVMs) using a slower approach. Hence the Manta RMI system logically consists of two parts:

—A fast communication protocol that is used only between Manta processes. We call this protocol *Manta RMI*, to emphasize that it delivers the standard RMI programming model to the user; but it can only be used for communication between Manta processes.

—Additional software that makes the Manta RMI system as a whole compatible with standard RMI, so Manta processes can communicate with JVMs.

We refer to the combination of these two parts as the *Manta RMI system*. We use the term *Sun RMI* to refer to the standard RMI protocol as defined in the RMI specification [Sun Microsystems 1997]. Note that both Manta RMI and Sun RMI provide the same programming model, but their wire formats are incompatible.

The Manta RMI system thus combines high performance with the flexibility and interoperability of RMI. In a grid computing application [Foster and Kesselman 1998], for example, some clusters can run our Manta software and communicate internally using the Manta RMI protocol. Other machines may run JVMs, containing, for example, a graphical user interface program. Manta communicates with such machines using the Sun RMI protocol, allowing method invocations between Manta and JVMs. Manta implements almost all other functionality required by the RMI specification, including heterogeneity, multithreading, synchronized methods, and distributed garbage collection. Manta currently does not implement Java's security model, as the system is primarily intended for parallel cluster computing.

---

[1] A fast, flexible, black-and-white, tropical fish that can be found in the Indonesian archipelago.

The main contributions of this article are as follows.

—We show that RMI can be implemented efficiently and can obtain a performance close to that of RPC systems. The null-RMI latency of Manta RMI over Myrinet is 37 s, only 6 s slower than a C-based RPC protocol.

—We show that this high performance can be achieved while still supporting polymorphism and interoperability with JVMs by using dynamic bytecode compilation and multiple RMI protocols.

—We give a detailed performance comparison between the Manta and Sun RMI protocols, using benchmarks as well as a collection of six parallel applications. To allow a fair comparison, we compiled the applications and the Sun RMI protocol with the native Manta compiler. The results show that the Manta protocol results in 1.8 to 3.4 times higher speedups for four out of six applications.

The remainder of the article is structured as follows. Design and implementation of the Manta system are discussed in Section 2. In Section 3, we give a detailed analysis of the communication performance of our system. In Section 4, we discuss the performance of several parallel applications. In Section 5, we look at related work. Section 6 presents conclusions.

2. DESIGN AND IMPLEMENTATION OF MANTA

This section will discuss the design and implementation of the Manta RMI system, which includes the Manta RMI protocol and the software extensions that make Manta compatible with Sun RMI.

2.1 Manta Structure

Since Manta is designed for high-performance parallel computing, it uses a native compiler rather than a JIT. The most important advantage of a native compiler is that it can perform more time consuming optimizations, and therefore (potentially) generate better code.

The Manta system is illustrated in Figure 1. The box in the middle describes the structure of a Manta process, which contains the executable code for the application and (de)serialization routines, both of which are generated by Manta's native compiler. Manta processes can communicate with each other through the Manta RMI protocol, which has its own wire format. A Manta process can communicate with any JVM (the box on the right) through the Sun RMI protocol, using the standard RMI format (i.e., the format defined in Sun's RMI specification).

A Manta-to-Manta RMI is performed with the Manta protocol, which is described in detail in the next section. Manta-to-Manta communication is the common case for high-performance parallel programming, for which our system is optimized. Manta's serialization and deserialization protocols support heterogeneity (RMIs between machines with different byte-orderings or alignment properties).

A Manta-to-JVM RMI is performed with a slower protocol that is compatible with the RMI specification and the standard RMI wire format. Manta uses
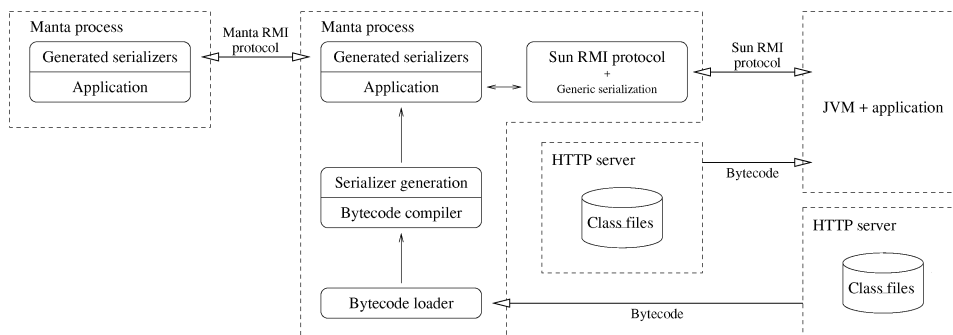
Fig. 1.   Manta/JVM interoperability.

generic routines to (de)serialize the objects to or from the standard format. These routines use reflection, similar to Sun's implementation. The routines are written in C, as is all of Manta's runtime system, and execute more efficiently than Sun's implementation, which is partly written in Java.

To support polymorphism for RMIs between Manta and JVMs, a Manta application must be able to handle bytecode from other processes. When a Manta application requests bytecode from a remote process, Manta will invoke its bytecode compiler to generate the metaclasses, the (de)serialization routines, and the object code for the methods as if they were generated by the Manta source code compiler. Dynamic bytecode compilation is described in more detail in Section 2.4. The dynamically generated object code is linked into the application with the operating system's dynamic linking interface. If a remote process requests bytecode from a Manta application, the JVM bytecode loader retrieves the bytecode for the requested class in the usual way through a shared filesystem or through an HTTP daemon. Sun's `javac` compiler is used to generate the bytecode at compile time.

The structure of the Manta system is more complicated than that of a JVM. Much of the complexity of implementing Manta efficiently is due to the need to interface a system based on a native-code compiler with a bytecode-based system. The fast communication path in our system, however, is straightforward: the Manta protocol just calls the compiler-generated serialization routines and uses a simple scheme to communicate with other Manta processes. This fast communication path is described below.

2.2 Serialization and Communication

RMI systems can be split into three major components: low-level communication, the RMI protocol (stream management and method dispatch), and serialization. Below, we discuss how the Manta protocol implements each component.

Low-level communication.   RMI implementations are typically built on top of TCP/IP, which was not designed for parallel processing. Manta uses the Panda communication library [Bal et al. 1998], which has efficient implementations on a variety of networks. Panda uses a scatter/gather interface to minimize the number of memory copies, resulting in high throughput.

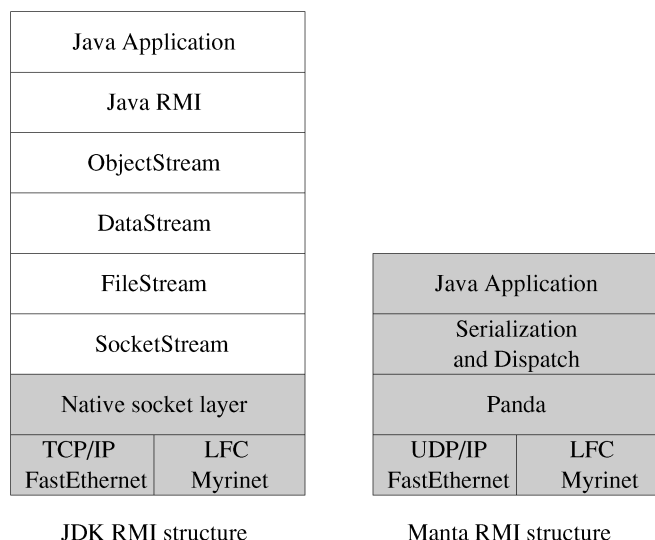| JDK RMI structure	Manta RMI structure
Java Application	
Java RMI	
ObjectStream	
DataStream	
FileStream	Java Application
SocketStream	Serialization and Dispatch
Native socket layer	Panda
TCP/IP FastEthernet — LFC Myrinet	UDP/IP FastEthernet — LFC Myrinet

Fig. 2.   Structure of Sun and Manta RMI protocols; shaded layers run compiled code.

On Myrinet, Panda uses the LFC communication system [Bhoedjang et al. 2000], which provides reliable communication. LFC is a network interface protocol for Myrinet that is both efficient and provides the right functionality for parallel programming systems. LFC itself is implemented partly by embedded software that runs on the Myrinet Network Interface processor and partly by a library that runs on the host. To avoid the overhead of operating system calls, the Myrinet Network Interface is mapped into user space, so LFC and Panda run entirely in user space. The current LFC implementation does not offer protection, so the Myrinet network can be used by a single process only. On Fast Ethernet, Panda is implemented on top of UDP, using a 2-way sliding window protocol to obtain reliable communication. The Ethernet network interface is managed by the kernel (in a protected way), but the Panda RPC protocol runs in user space.

The Panda RPC interface is based on an *upcall* model: conceptually, a new thread of control is created when a message arrives which will execute a handler for the message. The interface was designed to avoid thread switches in simple cases. Unlike active message handlers [von Eicken et al. 1992], upcall handlers in Panda are allowed to block to enter a critical section, but a handler is not allowed to wait for another message to arrive. This restriction allows the implementation to handle all messages using a single thread, so handlers that execute without blocking do not need any context switches.

The RMI protocol.   The runtime system for the Manta RMI protocol is written in C. It was designed to minimize serialization and dispatch overhead such as copying, buffer management, fragmentation, thread switching, and indirect method calls. Figure 2 gives an overview of the layers in the Manta RMI protocol and compares it with the layering of the Sun RMI system.

The shaded layers denote statically compiled code, while the white layers are mainly JIT-compiled Java (although they contain some native calls). Manta avoids the stream layers of Sun RMI. Instead, RMI parameters are serialized directly into an LFC buffer. Moreover, in the JDK, these stream layers are written in Java, and therefore their overhead depends on the quality of the Java implementation. In Manta, all layers are either implemented as compiled C code or compiler-generated native code. Also, the native code generated by the Manta compiler calls RMI serializers directly, instead of using the slow Java Native Interface. Heterogeneity between little-end and big-end machines is handled by sending data in the native byte order of the sender, and having the receiver do the conversion, if necessary.

Another optimization in the Manta RMI protocol is avoiding thread switching overhead at the receiving node. In the general case, an invocation is serviced at the receiving node by a newly allocated thread, which runs concurrently with the application threads. With this approach, however, the allocation of the new thread and the context switch to this thread will be on the critical path of the RMI. To reduce the allocation overhead, the Manta runtime system maintains a pool of preallocated threads, so the thread can be taken from this pool instead of being allocated. In addition, Manta avoids the context-switching overhead for simple cases. The Manta compiler determines whether a remote method may block. If the compiler can guarantee that a given method will never block, the receiver executes the method without doing a context switch to a separate thread. In this case, the current application thread will service the request and then continue. The compiler currently makes a conservative estimation, and only guarantees the nonblocking property for methods that do not call other methods and do not create objects (since that might invoke the garbage collector, which may cause the method to block). This analysis has to be conservative, since a deadlock situation might occur if an application thread services a method that blocks.

The Manta RMI protocol cooperates with the garbage collector to keep track of references across machine boundaries. Manta uses a local garbage collector based on a mark-and-sweep algorithm. Each machine runs this local collector, using a dedicated thread that is activated by the runtime system or the user. The distributed garbage collector is implemented on top of the local collectors, using a reference-counting mechanism for remote objects (distributed cycles remain undetected). If a Manta process communicates with a JVM, it uses the distributed garbage collection algorithm of the Sun RMI implementation, which is based on leasing.

The serialization protocol.   The serialization of method arguments is an important source of overhead in existing RMI implementations. Serialization takes a Java object and converts (serializes) it into an array of bytes, making a deep copy that includes the referenced subobjects. The Sun serialization protocol is written in Java and uses reflection to determine the type of each object during runtime. The Sun RMI implementation uses the serialization protocol for converting data that are sent over the network. The process of serializing all arguments of a method is called *marshalling*.

With the Manta protocol, all serialization code is generated by the compiler, avoiding most of the overhead of reflection. Serialization code for most classes is generated at compile time. Only serialization code for classes which are not locally available is generated at runtime, by the bytecode compiler. The overhead of this runtime code generation is incurred only once—the first time the new class is used as an argument to some method invocation. For subsequent uses, the efficient serializer code is then available for reuse.

The Manta compiler also generates the marshalling code for methods. The compiler generates method-specific marshall and unmarshall functions, which (among others) call the generated routines to serialize or deserialize all arguments of the method. For every method in the method table, two pointers are maintained to dispatch to the right marshaller or unmarshaller, depending on the dynamic type of the given object. A similar optimization is used for serialization: every object has two pointers in its method table to the serializer and deserializer for that object. When a particular object is to be serialized, the method pointer is extracted from the method table of the object's dynamic type and the serializer is invoked. On deserialization, the same procedure is applied.

Manta's serialization protocol performs optimizations for simple objects. An array whose elements are of a primitive type is serialized by doing a direct memory copy into the LFC buffer, so the array need not be traversed, as is done by the JDK. In order to detect duplicate objects, the marshalling code uses a table containing objects that have already been serialized. If the method does not contain any parameters that are objects, however, the table is not built up, which again makes simple methods faster.

Another optimization concerns the type descriptors for the parameters of an RMI call. When a serialized object is sent over the network, a descriptor of its type must also be sent. The Sun RMI protocol sends a complete type descriptor for every class used in the remote method, including the name and package of the class, a version number, and a description of the fields in this class. All this information is sent for every RMI call; information about a class is only reused within a single RMI call. With the Manta RMI protocol, each machine sends the type descriptor only once to any other machine. The first time a type is sent to a certain machine, a type descriptor is sent and the type is given a new type-id that is specific to the receiver. When more objects of this type are sent to the same destination machine, the type-id is reused. When the destination machine receives a type descriptor, it checks if it already knows this type. If not, it loads it from the local disk or an HTTP server. Next, it inserts the type-id and a pointer to the metaclass in a table, for future references. This scheme thus ensures that type information is sent only once to each remote node.

2.3 Generated Marshalling Code

Figures 3, 4, and 5 illustrate the generated marshalling code. Consider the `RemoteExample` class in Figure 3. The `square()` method can be called from another machine, so the compiler generates marshalling and unmarshalling code for it.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class RemoteExample extends UnicastRemoteObject
  implements RemoteExampleInterface {
    int value;
    String name;

    synchronized int square(int i, String s1, String s2) throws RemoteException {
        value = i;
        name = s1 + s2;
        System.out.println("i = " + i);
        return i*i;
    }
}
```

Fig. 3.   A simple remote class.

```
marshall__square(class__RemoteExample *this, int i, class__String *s1, class__String *s2) {
    MarshallStruct *m = allocMarshallStruct();
    ObjectTable = createObjectTable();

    writeHeader(m->outBuffer, this, OPCODE_CALL, CREATE_THREAD);
    writeInt(m->outBuffer, i);
    writeObject(m->outBuffer, s1, ObjectTable);
    writeObject(m->outBuffer, s2, ObjectTable);

    // Request message is created, now write it to the network.
    flushMessage(m->outBuffer);

    fillMessage(m->inBuffer);  // Receive reply.
    opcode = readInt(m->inBuffer);
    if (opcode == OPCODE_EXCEPTION) {
        class__Exception *exception = readObject(m->inBuffer, ObjectTable);
        freeMarshallStruct(m);
        THROW_EXCEPTION(exception);
    } else {
        result = readInt(m->inBuffer);
        freeMarshallStruct(m);
        RETURN(result);
    }
}
```

Fig. 4.   The generated marshaller (pseudocode) for the square method.

```
unmarshall__square(class__RemoteExample *this, MarshallStruct *m) {
    ObjectTable = createObjectTable();

    int i = readInt(m->inBuffer);
    class__String *s1 = readObject(m->inBuffer, ObjectTable);
    class__String *s2 = readObject(m->inBuffer, ObjectTable);

    result = CALL_JAVA_FUNCTION(square, this, i, s1, s2, &exception);
    if (exception) {
        writeInt(m->outBuffer, OPCODE_EXCEPTION);
        writeObject(m->outBuffer, exception, ObjectTable);
    } else {
        writeInt(m->outBuffer, OPCODE_RESULT_CALL);
        writeInt(m->outBuffer, result);
    }

    // Reply message is created, now write it to the network.
    flushMessage(m->outBuffer);
}
```

Fig. 5.   The generated unmarshaller (pseudocode) for the square method.

The generated marshaller for the `square()` method is shown in Figure 4 in pseudocode. Because `square()` has `Strings` as parameters (which are objects in Java), a table is built to detect duplicates. A special `create_thread` flag is set in the header data structure because `square` potentially blocks: it contains a method call that may block (e.g., in a `wait()`) and it creates objects, which may trigger garbage collection and thus may also block. The `writeObject` calls serialize the string objects to the buffer. `flushMessage` does the actual writing out to the network buffer. The function `fillMessage` initiates reading the reply.

Pseudocode for the generated unmarshaller is shown in Figure 5. The header is already unpacked when this unmarshaller is called. Because the `create_thread` flag in the header was set, this unmarshaller will run in a separate thread obtained from a thread pool. The marshaller itself does not know about this. Note that the `this` parameter is already unpacked and is a valid reference for the machine on which the unmarshaller will run.

2.4 Dynamic Bytecode Compilation

To support polymorphism, a Manta program must be able to handle classes that are exported by a JVM, but that have not been statically compiled into the Manta program. To accomplish this, the Manta RMI system contains a bytecode compiler to translate classes to object code at runtime. We describe this bytecode compiler below. Manta uses the standard dynamic linker to link the object code into the running application.

As with the JDK, the compiler reads the bytecode from a file or an HTTP server. Next, it generates a Manta metaclass with dummy function entries in its method table. Since the new class may reference or even subclass other unknown classes, the bytecode compiler is invoked recursively for all referenced unknown classes. Subsequently, the instruction stream for each bytecode method is compiled into a C function. For each method, the used stack space on the Virtual Machine stack is determined at compile time, and a local stack area is declared in the C function. Operations on local variables are compiled in a straightforward way. Virtual function calls and field references can be resolved from the running application, including the newly generated metaclasses. Jumps and exception blocks are implemented with labels, gotos, and nonlocal gotos (`setjmp/longjmp`). The resulting C file is compiled with the system C compiler, and linked into the running application with the system dynamic linker (called `dlopen()` in many Unix implementations). The dummy entries in the created metaclass method tables are resolved into function pointers in the dynamically loaded library.

One of the optimizations we implemented had a large impact on the speed of the generated code: keeping the method stack in registers. The trivial implementation of the method stack would be to maintain an array of N 32-bit words, where N is the size of the used stack area of the current method. Since bytecode verification requires that all stack offsets can be computed statically, it is, however, possible to replace the array with a series of N register variables, so the calls to increment or decrement the stack pointer are avoided and the
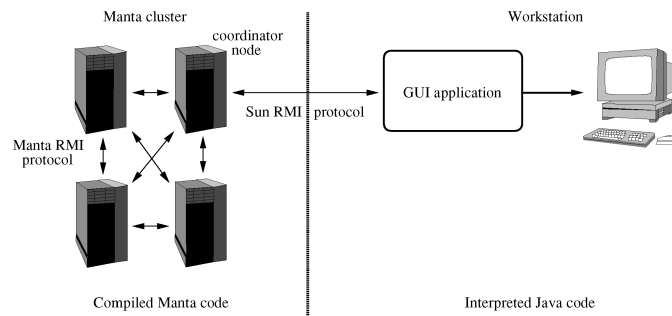
Fig. 6.　Example of Manta's interoperability.

C compiler can keep stack references in registers. A problem is that in the JVM, 64-bit variables are spread over two contiguous stack locations. We solve this by maintaining two parallel stacks, one for 32-bit and one for 64-bit words. Almost all bytecode instructions are typed, so they need to operate only on the relevant stack. Some infrequently used instructions (the dup2 family) copy either two 32-bit words or one 64-bit word, and therefore operate on both stacks. The memory waste of a duplicate stack is moderate, since the C compiler will remove any unreferenced local variables. With this optimization, the application speed of compiled bytecode is generally within 30% of compiled Manta code.

2.5 Example Application

Manta's RMI interoperability and dynamic class loading are useful to interoperate with software that runs on a JVM and uses the Sun RMI protocol. For example, consider a parallel program that generates output that must be visualized. The parallel program is compiled with Manta and uses the Manta RMI protocol. The software for the visualization system to be used, however, may run on the Sun JDK and use the Sun RMI protocol. To illustrate this type of interoperability, we implemented a simple example, using a graphical version of one of our parallel applications (successive overrelaxation; see Section 4).

The computation is performed by a parallel program that is compiled with Manta and runs on a cluster computer (see Figure 6). The output is visualized on the display of a workstation, using a graphical user interface (GUI) application written in Java. The parallel application repeatedly performs one iteration of the SOR algorithm and collects its data (a 2-dimensional array) at one node of the cluster, called the *coordinator*. The coordinator passes the array via the Sun RMI protocol to a remote *viewer* object, which is part of the GUI application on the workstation. The viewer object creates a window and displays the data. Since the GUI application is running on a Sun JDK, communication between the coordinator and the GUI uses the Sun RMI protocol. The Manta nodes internally use the Manta RMI protocol.

The RMI from the coordinator to the GUI is implemented using a remote *viewer* interface, for which an RMI stub is needed. This stub is not present in the compiled Manta executable, so the coordinator node dynamically

retrieves the bytecode for the stub from the code base of the GUI, compiles it using Manta's dynamic bytecode compiler, and then links it into the application.

3. COMMUNICATION PERFORMANCE

In this section, the communication performance of Manta RMI is compared against several implementations of Sun RMI. Experiments are run on a homogeneous cluster of Pentium Pro processors, each containing a 200 MHz Pentium Pro and 128 MByte of main memory. All boards are connected by two different networks: 1.2 Gbit/sec Myrinet [Boden et al. 1995] and Fast Ethernet (100 Mbit/s Ethernet). The system runs RedHat Linux 6.2 (kernel version 2.2.16). Both Manta and Sun RMI run over Myrinet and Fast Ethernet.

For the comparison, we used three systems that use Sun RMI: the Sun (Blackdown) JDK 1.2 with JIT, the IBM JDK 1.1.8 also with JIT, and a Sun RMI system (based on the JDK 1.1) compiled with our native Manta compiler. For all three systems, we built an interface to run over Myrinet. We describe these systems in detail in Section 3.1, including important optimizations. Since the Sun system compiled with Manta turned out to be the fastest of the three RMI systems, we use this system in the following sections to represent Sun's RMI protocol. Next, we discuss the latency (Section 3.2) and throughput (Section 3.3) obtained by Manta RMI and Sun RMI. Finally, we analyze the impact of several optimizations in the protocols (Section 3.4).

3.1 Implementation of Sun RMI over Myrinet

A difference between Manta and the Sun JDK is that Manta uses a native compiler, whereas the JDK uses a JIT. The sequential speed of the code generated by the Manta compiler is much better than that of the Sun JDK/JIT system, and comparable to the IBM JDK/JIT. The overhead of the Java Native Interface and differences in sequential code speed obscures the comparison between the Manta and Sun RMI protocols. To allow a fair comparison, we built a third system, which uses Sun RMI, but with the Java RMI code compiled by the native Manta compiler. This system is called *Sun compiled*. Below, we discuss the latter system and the optimizations we implemented for the Sun RMI protocol on Myrinet.

The Sun compiled system. We built a Sun RMI system by compiling the Sun RMI code with Manta. Sun's native code had to be replaced by new Manta native C code. This native code is used for object serialization and interfacing to the network layer. The new object serialization code is similar to the Sun implementation, using reflection to convert objects to bytes. To improve performance, we reimplemented a larger part of the serialization code in C. In the new native code, we can exploit knowledge about the memory layout of objects: we directly use the class information and data fields present in the object, instead of calling the reflection mechanism in Java as the Sun JDK does. Also, in the Sun JDK, expensive Java Native Interface calls are required to convert scalar

types like long and double to bytes before writing them to the output stream. In the new code, these values can be written directly into the stream buffer, converting them on the fly.

Interfacing to Myrinet.   To run the Sun JDK, IBM JDK, and the *Sun compiled* system over Myrinet, we use a socket interface on top of Panda/Myrinet. The socket interface is called FastSockets, and is a reimplementation of Berkeley FastSockets [Rodrigues et al. 1997]. Its main virtues are zero-copy streams, a one-to-one mapping of socket messages and Panda messages, and a performance quite close to Panda's. The Sun API currently does not allow replacement of its sockets with an alternative implementation like FastSockets, so a marginal change to the API was necessary. It was sufficient to declare the constructor of class `java.net.InetAddress` as public. (This API problem has been registered in Sun's bug database.)

Performance optimizations for Sun RMI.   Below, we describe several performance problems we addressed with the FastSockets layer, to allow a fairer comparison with Manta. In general, we tried to eliminate the most important sources of overhead for Sun RMI, as far as these optimizations could be done in the FastSockets layer. In particular, we optimized the interaction with the thread package and with the garbage collector.

The first problem is that many socket calls (e.g., `send`, `receive`, `accept`, `connect`) are blocking, in the sense that they suspend the calling *process* until the system call is completely serviced. Java applications are multithreaded, and the semantics of blocking calls must be that the caller *thread* is suspended, and any other runnable thread in the process is scheduled: otherwise, deadlock may occur. Virtual machine implementations handle blocking system calls in their thread package: all sockets are set to nonblocking mode, and blocking calls are intercepted, so threads that would block are suspended. The thread scheduler polls any sockets on which a call has been posted (e.g., with `select`), and wakes up a suspended thread if some action can be performed on its socket. This functionality to block a thread and schedule another one is not exported by virtual machine implementations. Therefore, we must emulate it in our interface layer.

All sockets are set to nonblocking mode by our interface layer. A naive emulation of blocking would have a thread invoke `Thread.yield()` until its socket call has completed. However, for performance reasons it is important that the blocking mechanism does not involve unnecessary thread switches, even though multiple threads can usually be scheduled (e.g., a thread in the server that listens to the registry). For the JDKs, the penalty is hundreds of microseconds per RMI, since the kernel is involved in each thread switch. So our implementation uses condition variables for condition synchronization. For the JDKs, these were implemented in Java. Manta's thread package exports condition variables to native functions. A thread that would block enters the wait state instead. It is signaled when a poller thread notices that the socket is ready. The role of poller thread is taken by one of the blocked threads: a separate poller thread always involves thread switches. Usually, no thread switches are incurred on the critical path of the RMI latency test.

Table I. Null-RMI Latency and Throughput on Myrinet and Fast Ethernet

System	Version	Network	Latency (μs)	Throughput (Mbyte/s)
Sun JIT (Blackdown)	1.2	Myrinet	1316	3.8
IBM JIT	1.1.8		550	7.9
Sun compiled	1.1.4		301	15
Manta RMI			37	54
Panda RPC			31	60
Sun JIT (Blackdown)	1.2	Fast Ethernet	1500	3.1
IBM JIT	1.1.8		720	6.0
Sun compiled	1.1.4		515	7.2
Manta RMI			207	10.5
Panda RPC			173	11.1

The second optimization we performed concerns garbage collection. The problem is that RMIs create many objects that become garbage at the end of the invocation. The Manta compiler provides support for the server side: it determines whether the objects that are an argument or result of the invocation may *escape* (i.e., whether a reference to the object is retained outside the RMI invocation). If not, Manta allows such objects to be immediately returned to the heap. For the *Sun compiled* system, we modified Sun's skeleton compiler rmic, so the generated skeletons use this compiler information. This substantially improved latency and throughput for RMIs that use objects as arguments or return value (see Section 3.4). There is no mechanism in the JDK to explicitly free objects, so we could not apply this technique for the Sun or IBM JDK. The garbage collector is responsible for the low throughput achieved even by the fast IBM JDK system.

Performance.   Table I shows the null-RMI latency and throughput of various RMI implementations on Myrinet and Fast Ethernet. On Myrinet, Manta RMI obtains a null-RMI latency of 37 μs, while the Sun JDK 1.2 (with just-in-time compilation enabled) obtains a latency of 1316 μs, which is 35 times higher. *Sun compiled* obtains a null-RMI latency of 301 μs, which is still eight times slower than Manta RMI. In comparison, the IBM JIT 1.1.8 obtains a latency of 550 μs (measured on RedHat Linux 6.1). The *Sun compiled* system uses more efficient locking than the Sun and IBM JITs (using Manta's user space thread package) and a much faster native interface. It also reduces the garbage collection overhead for objects passed to RMI calls. Finally, its sequential code speed is much better than that of the Sun JDK JIT, and is comparable to the IBM JIT. The table also gives the performance of a conventional Remote Procedure Call protocol (of the Panda library [Bal et al. 1998]), which is implemented in C. As can be seen, the performance of the Manta protocol comes close to that of Panda RPC.

The throughput obtained by Manta RMI (for sending a large array of integers) is also much better than that of the Sun JDK: 54 MByte/s versus 3.8 MByte/s (over Myrinet). The throughput of *Sun compiled* is 15 MByte/s, 3.6 times less than for Manta RMI, but better than that for the Sun JDK. The table also gives performance results on Fast Ethernet. Here, the relative differences are smaller because the communication costs are higher.

Table II. Breakdown of Manta and *Sun Compiled* RMI on Pentium Pro and Myrinet
(times are in s)

	Manta				*Sun Compiled*			
	empty	1 object	2 objects	3 objects	empty	1 object	2 objects	3 objects
Serialization	0	6	10	13	0	195	210	225
RMI overhead	5	10	10	10	180	182	182	184
Communication	32	34	34	35	121	122	124	125
Method call	0	1	1	1	0	1	1	1
Total	37	51	55	59	301	500	517	535

As the *Sun compiled* system is by far the most efficient implementation of the Sun RMI protocol, we use this system in the following sections to represent Sun's protocol.

3.2 Latency

We first present a breakdown of the time that Manta and *Sun compiled* spend in remote method invocations. We use a benchmark that has zero to three empty objects (i.e. objects with no data fields) as parameters, while having no return value. The benchmarks are written in such a way that they do not trigger garbage collection. The results are shown in Table II. The measurements were done by inserting timing calls, using the Pentium Pro performance counters, which have a granularity of 5 nanoseconds. The serialization overhead includes the costs to serialize the arguments at the client side and deserialize them at the server side. The RMI overhead includes the time to initiate the RMI call at the client, handle the upcall at the server, and process the reply (at the client), but excludes the time for (de)serialization and method invocation. The communication overhead is the time from initiating the I/O transfer until receiving the reply, minus the time spent at the server side. For Manta, the measurements do not include the costs for sending type descriptors (as these are sent only once).

The simplest case is an empty method without any parameters, the null-RMI. On Myrinet, a null-RMI takes about 37 s with Manta. Only 6 s are added to the roundtrip latency of the Panda RPC, which is 31 s. The large difference between passing zero or one object parameters can be explained as follows. First, the runtime system has to build a table used to detect possible cycles and duplicates in the objects. Second, RMIs containing object parameters are serviced by a dedicated thread from a thread pool, since such RMIs may block by triggering garbage collection. The thread-switching overhead in that case is about 5 s. Finally, the creation of the parameter object also increases the latency.

For the *Sun compiled* system, a null-RMI over Myrinet takes 301 s, which is 8 times slower than Manta, even with all the optimizations we applied. Manta RMI obtains major performance improvements in all layers: compiler-generated serializers win by a factor 17 or more; the RMI overhead is 18 times lower; and the communication protocols are 4 times faster.

Next, we study the latency for Manta RMI and the *Sun compiled* protocol for various combinations of input parameters and return values. We use similar

Table III. RMI Latency on Myrinet for Manta, *Sun Compiled*, and KaRMI for Different Parameters and Return Values (in   s). (The KaRMI latencies were measured on a 500 MHz Digital Alpha with a slow JIT.)

Benchmark	Manta	*Sun-compiled*	KaRMI
void (void)	37	301	117
void (int, int)	39	309	194
int32	77	1500	328
tree-1	66	763	279
tree-15	264	1610	1338
float[50]	88	522	483
float[5000]	1430	4215	8664

benchmarks as described in Nester et al. [1999] for the Karlsruhe RMI (KaRMI) system. The results are shown in Table III. For comparison, we also include the latency over Myrinet obtained by KaRMI, taken from Nester et al. [1999]. These measurements were done on a 500 MHz Digital Alpha and obtained with the JDK 1.1.6 (which used a low-quality JIT). KaRMI (like Manta RMI) is not compatible with Sun RMI because it uses its own, more compact, serialization format rather than the format specified by the RMI standard. The first two benchmarks use RMIs with an empty return value; the remaining benchmarks return the value passed as input parameter. The *int32* benchmark sends and receives an object containing 32 integers. The tree benchmarks send and receive balanced trees with 1 and 15 nodes, each containing four integers. The last two benchmarks transfer arrays of floating point numbers. As can be seen from this table, Manta RMI obtains much lower latencies on all benchmarks than the *Sun compiled* system. Manta RMI also obtains better performance than KaRMI.

3.3 Throughput

Next, we study the RMI throughput of Manta and *Sun compiled*. We use a benchmark that measures the throughput for a remote method invocation with various types of arguments and no return value. (As RMIs are synchronous, however, the sender does wait for the remote method to return.) The benchmark performs 10,000 RMIs, with about 100,000 bytes of arguments each. The reply message is empty. The results are shown in Table IV.

Manta achieves a throughput of 54 MByte/s for arrays of integers, compared to 60 MByte/s for the underlying Panda RPC protocol (see Table I). In comparison, the throughput of *Sun compiled* is only 15 MByte/s.

The throughput for *Sun compiled* for arrays of integers is substantially higher than for the Sun JIT (15 MByte/s versus 3.8 MByte/s, see Table I), due to our optimizations described in Section 3.1. Still, the throughput for *Sun compiled* is much lower than for Manta RMI. The Sun serialization protocol internally buffers messages and sends large messages in chunks of 1 KByte, which decreases throughput. Even more important, Sun RMI (and *Sun compiled*) performs unnecessary byte-swapping. The sender and the receiver use the same format for integers, but this format differs from the standard RMI format. *Sun compiled* then uses serialization to convert the data to the standard format. Manta RMI, on the other hand, always sends the data in the format of the sender, and lets the receiver do byte-swapping only when necessary. The

Table IV. RMI Throughput (in MByte/s) on Myrinet of Manta
and *Sun Compiled* for Different Parameters

	Manta	*Sun-compiled*
Array of bytes	54	37
Array of integers	54	15
Array of floats	54	15
Array of doubles	54	15
Binary tree	2.4	0.6

throughput obtained by the *Sun compiled* system for an array of bytes, for which no byte swapping is needed, is 37 MByte/s (see Table IV). This throughput is high because all I/O layers in the Sun system have short-cuts for long messages. When writing a large buffer, each I/O layer passes the buffer directly to the layer below it, without copying. Similarly, when a large read request is done, it is passed on to the bottom layer, and the result is passed back up, without copying.

For KaRMI, only one throughput test is available [Philippsen et al. 2000]. The throughput for an array of 200 KByte over Myrinet is 23 MByte/s, which is less than half the throughput of Manta RMI. This low throughput is attributed to the overhead of thread scheduling and the interaction between Java threads and system threads [Philippsen et al. 2000].

The binary tree throughput benchmark is based on the KaRMI latency benchmark described in Nester et al. [1999], but using input parameters and no return values. The benchmark sends balanced trees with 1,000 nodes, each containing four integers. The reported throughput is that for the user "payload" (i.e., the four integers), although more information is sent over the network to rebuild the tree structure. The throughput for this benchmark is very low in comparison with the throughput achieved for arrays. The overhead can be attributed to the small size of the nodes and the dynamic nature of this data type, which makes especially (de)serialization expensive: the tree is written to and read from the network buffer a tree node at a time, and for *Sun compiled* even a byte at a time; therefore the overhead of network access is incurred much more often than for arrays.

3.4 Impact of Specific Performance Optimizations

Below, we analyze the impact of specific optimizations in more detail.

Type descriptors.   As explained in Section 2.2, the Sun protocol always sends a complete type descriptor for each class used in the RMI. Manta RMI sends this type information only once for each class; it uses a type-id in subsequent calls. The amount of data that Manta RMI sends for object and array parameters thus depends on whether a parameter of the same class has been transmitted before. Table V shows the amount of data sent for both cases, and for both Manta RMI and *Sun compiled* RMI. For each case, the table gives the number of bytes for RMIs with no arguments, with a 100 element array of integer argument, and with a single object containing an integer and a double. It also shows the times on a 200 MHz Pentium Pro to write the type descriptor at the sending side and read it at the receiving side.

Table V. Amount of Data Sent by Manta RMI and Sun RMI; Runtime Overhead of Type Descriptors

	Manta RMI			Sun compiled RMI		
	empty	int [100]	1 object	empty	int [100]	1 object
Bytes (using type descriptor)	44	484	96	63	487	102
Bytes (using type-id)	44	452	64	—	—	—
Writing type descriptor (s)	—	11	12	—	25	27
Reading type descriptor (s)	—	15	17	—	55	73

As can be seen, the type-descriptor optimization saves 32 bytes for each array or object parameter. The runtime costs saved by the optimization for reading and writing the type descriptors is 26 s for arrays and 29 s for objects. Moreover, a type descriptor includes the name of its class. We used a single-letter class name (and no package) in the benchmark, so the optimization wins even more for classes with longer names.

The Sun RMI protocol sends only moderately more data than the Manta protocol, yet it spends a considerable amount of time in processing and communicating the data. The Sun protocol spends 80 s handling the type descriptors for arrays and 100 s for objects. It pays this price at every invocation, whereas the Manta protocol incurs the overhead only once.

Using a scatter/gather interface. As explained in Section 2.2, the Panda library on which Manta is built, uses a scatter/gather interface to minimize the number of memory copies needed. This optimization increases the throughput for Manta RMI. To assess the impact of this optimization we also measured the throughput obtained when the sender makes an extra memory copy. In this case, the maximum throughput decreases from 54 to 44 MByte/s, since memory copies are expensive on a Pentium Pro [Brown and Seltzer 1997]. This experiment clearly shows the importance of the scatter/gather interface. Unfortunately, dereferencing the scatter/gather vector involves extra processing, so the null-RMI latency of the current Manta RMI system is slightly higher than that for an earlier Panda version without the scatter/gather interface (34 versus 37 s) [Maassen et al. 1999].

Reducing byte swapping. Another optimization that increases the throughput is to avoid byte swapping between identical machines. As described in Section 3.3, with Sun RMI the sender always converts the arguments of an RMI call to the wire format defined by the RMI specification; the receiver converts the format back to what it requires. In Manta, on the other hand, the data are transmitted in the native byte order of the sender, and the receiver only does the conversion if necessary. So if the sender and receiver have the same format, but this format is different from the standard RMI format, Sun RMI will do two byte-swap conversions while Manta will not do any byte swapping.

We measured the impact of this optimization by adding byte swapping code to the sender side of Manta RMI. (This code is not present in the normal Manta system, since the sender never does byte swapping with Manta.) If byte swapping is performed by the sender and receiver (as Sun RMI does), the throughput of Manta RMI for arrays of integers or floats decreases by

almost a factor two. The maximum throughput obtained with byte swapping enabled is decreased from 54 to 30 MByte/s. This experiment clearly shows that unnecessary byte swapping adds a large overhead, which is partly due to the extra memory copies needed.

Escape analysis. As described in Section 3.1, we implemented a simple form of escape analysis. With this analysis, objects that are argument or result of an RMI but that do not escape from the method will be immediately returned to the heap. Without this optimization, such objects would be subject to garbage collection, which reduces the RMI throughput. Without escape analysis the throughput for Manta is reduced from 54 to 30 MByte/s. For *Sun compiled*, the throughput for byte arrays is reduced from 37 to 25 MByte/s, but the other throughput numbers are hardly affected (since these cases also suffer from other forms of overhead, in particular byte swapping).

4. APPLICATION PERFORMANCE

The low-level benchmarks show that Manta obtains a substantially better latency and throughput than the Sun RMI protocol. For parallel programming, however, a more relevant metric is the efficiency obtained with applications. To determine the impact of the RMI protocol on application performance, we have written six parallel applications with different granularities. We briefly describe the applications and the input sizes used below, and then we discuss their performance using Manta and *Sun compiled*. Each application program typically first creates the remote objects needed for interprocess communication and exchanges the references to these objects among the machines. Therefore, the overhead of distributed garbage collection and reference counting only occurs during initialization, and has hardly any impact on application performance.

SOR. Red/black SOR (successive overrelaxation) is an iterative method for solving discretized Laplace equations on a grid. The program distributes the grid rowwise among the processors. Each processor exchanges one row of the matrix with its neighbors at the beginning of each iteration. We used a 578×578 grid as input.

ASP. The ASP (all-pairs shortest paths) program computes the shortest path between any two nodes of a given 1,280-node graph. It uses a distance table that is distributed rowwise among the processors. At the beginning of each iteration, one processor needs to send a row of the matrix to all other processors. Since Java lacks broadcasting, we expressed this communication pattern using a spanning tree. Each processor forwards the message along a binary spanning tree to two other processors, using RMIs and threads.

Radix is a histogram-based parallel sort program from the SPLASH-2 suite [Woo et al. 1995]. We have rewritten the program in Java, using RMI. The program repeatedly performs a local sort phase (without communication) followed by a histogram merge phase. The merge phase uses combining-tree communication to transfer histogram information. After this merge phase the program

moves some of the keys between processors, which also requires RMIs. The radix program performs a large number of RMIs. We used an array with 3,000,000 numbers as input.

FFT is a 1-D fast Fourier transform program based on the SPLASH-2 code, which we rewrote in Java. The matrix is partitioned rowwise among the different processors. The FFT communication pattern is a personalized all-to-all exchange, implemented using an RMI between every pair of machines. We used a matrix with 2^{20} elements.

Water is another SPLASH application that we rewrote in Java. This N-body simulation is parallelized by distributing the bodies (molecules) among the processors. Communication is primarily required to compute interactions with bodies assigned to remote machines. Our Java program uses message combining to obtain high performance: each processor receives all bodies it needs from another machine using a single RMI. After each operation, updates are also sent using one RMI per destination machine. Since Water is an $O(N^2)$ algorithm and we optimized communication, the relative communication overhead is low. We used 1,728 bodies for the measurements.

Barnes-Hut is an $O(N \log N)$ N-body simulation. We wrote a Java program based on the code by Blackston and Suel [1997]. This code is optimized for distributed memory architectures. Instead of finding out at runtime which bodies are needed to compute an interaction, as in the SPLASH-2 version of Barnes-Hut, this code precomputes where bodies are needed, and sends them in one collective communication phase before the actual computation starts. In this way, no stalls occur in the computation phase [Blackston and Suel 1997]. We used a problem with 30,000 bodies.

Figures 7 to 12 show the speedups for these six applications obtained by Manta and *Sun compiled*. For both systems, the programs are compiled statically using the Manta compiler. The speedups for each system are computed relative to the parallel Manta program on a single CPU. The sequential execution times of Manta and *Sun compiled* are very similar, as the applications are compiled with the Manta compiler for both systems (for some applications Manta is slightly faster due to caching effects).

Table VI gives performance data of the six applications, including the total number of messages sent (summed over all CPUs) and the amount of data transferred, using 16 or 32 CPUs. These numbers were measured at the Panda layer, so they include header data. Also, a Manta RMI generates two Panda messages, a request and a reply.

Figures 7 to 12 show that Manta's higher communication performance results in substantially better application speedups. *Sun compiled* performs well for only two applications, Water and ASP. Water has by far the lowest communication overhead of the six applications. On 32 CPUs, it sends 2,708 (44984/16.61) messages and 1.20 (20.05/16.61) MBytes per second (for *Sun compiled*). ASP communicates more than Water, but it performs relatively few RMIs per second. With the other four applications, Manta obtains much better maximal speedups, ranging from a factor 1.8 (for Barnes-Hut and Radix) to 3.4 (for SOR).
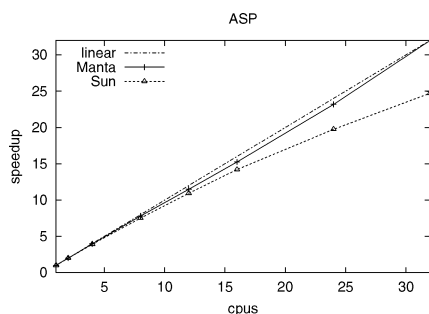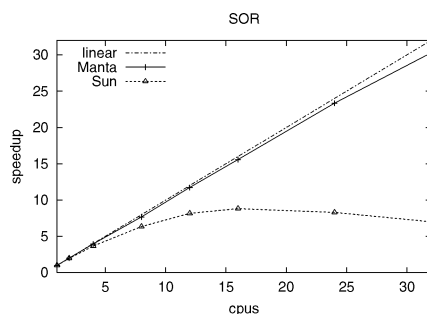
Fig. 7.   Speedup of ASP.
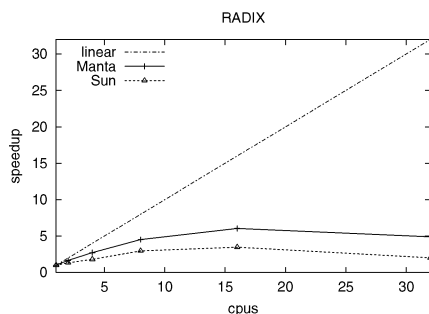

Fig. 8.   Speedup of SOR.
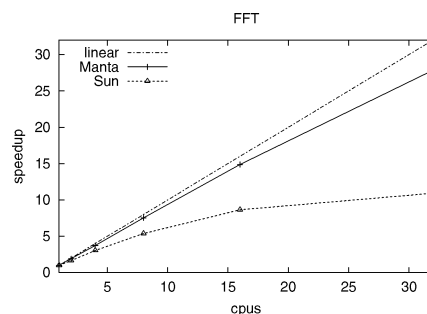

Fig. 9.   Speedup of Radix.
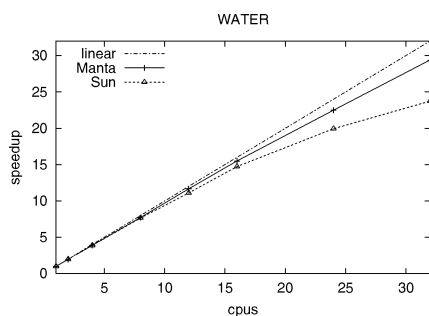

Fig. 10.   Speedup of FFT.
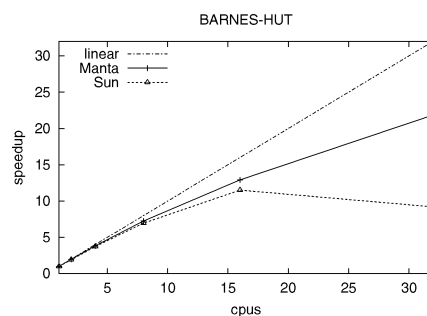

Fig. 11.   Speedup of Water.

Fig. 12.   Speedup of Barnes-Hut.

Manta obtains high efficiencies for all applications except Radix sort. Radix sends the largest number and volume of messages per second of all six applications; on 32 CPUs, it sends almost 27,000 (39418/1.46) messages and 85 (124.68/1.46) MBytes per second, summed over all CPUs.

Table VI shows that the Sun RMI protocol sends far more messages for all applications than Manta, because the Sun serialization protocol buffers messages and transfers large messages in chunks of 1 KBytes (see Section 3.3). The volume of the data transferred by the Manta protocol is somewhat lower than

Table VI. Performance Data for Manta and *Sun Compiled* on 16 and 32 CPUs

Program	System	Time (s.)	16 CPUs #messages	Data (MByte)	Time (s.)	32 CPUs #messages	Data (MByte)
ASP	Manta	25.64	38445	95.30	12.22	79453	196.96
	Sun compiled	27.56	154248	100.23	15.83	319870	207.17
SOR	Manta	10.96	44585	80.38	5.64	92139	166.11
	Sun compiled	19.38	134765	84.62	24.39	285409	175.11
Radix	Manta	1.19	9738	62.46	1.46	39418	124.68
	Sun compiled	2.06	78674	64.87	3.56	183954	130.35
FFT	Manta	3.52	8344	152.06	1.88	33080	157.14
	Sun compiled	6.07	173962	157.37	4.80	204949	163.45
Water	Manta	25.46	6023	8.44	13.41	23319	17.30
	Sun compiled	26.78	16088	9.59	16.61	44984	20.05
Barnes-Hut	Manta	14.90	18595	23.78	8.81	107170	52.26
	Sun compiled	16.74	45439	25.20	20.91	171748	57.58

that for the Sun protocol, since Manta does not send type descriptors for each class on every call and Manta sends fewer messages and thus fewer headers.

5. RELATED WORK

We discuss related work in three areas: optimizations to RMI, fast communication systems, and parallel programming in Java.

Optimizations for RMI.   RMI performance is studied in several other papers. KaRMI is a new RMI and serialization package (drop-in replacement) designed to improve RMI performance [Nester et al. 1999; Philippsen et al. 2000]. The Manta performance is better than that of KaRMI (see Table III in Section 3.2). The main reasons are that Manta uses static compilation and a completely native runtime system (implemented in C). Also, Manta exploits features of the underlying communication layer (the scatter/gather interface). KaRMI uses a low-quality JIT (JDK 1.1.6 in the cited publications) and a runtime system written mostly in Java (which thus suffers from the poor JIT performance). KaRMI is designed to be portable and therefore avoids using native code (e.g., KaRMI throughput could have been improved in certain cases, but the designers deliberately chose not to because it would require native code). Manta, on the other hand, was developed from scratch to obtain high communication performance. Both KaRMI and Manta RMI use a wire format that is different from the standard RMI format.

Krishnaswamy et al. [1998] improve RMI performance somewhat by using caching and UDP instead of TCP. Their RMI implementation, however, still has high latencies (e.g., they report null-RMI latencies above a millisecond on Fast Ethernet). Also, the implementation requires some modifications and extensions of the interfaces of the RMI framework. Javanaise [Hagimont and Louvegnies 1998] and VJava [Lipkind et al. 1999] are other Java systems that implement object caching. Javanaise proposes a new model of distributed shared objects (as an alternative to RMI). Breg et al. [1998] study RMI performance and interoperability. Hirano et al. [1998] provide performance figures of RMI and RMI-like systems on Fast Ethernet.

Fast communication systems.   Much research has been done since the 1980s on improving the performance of remote procedure call protocols [Hutchinson et al. 1989; Johnson and Zwaenepoel 1991; van Renesse et al. 1989; Schroeder and Burrows 1990; Thekkath and Levy 1993]. Several important ideas resulted from this research, including the use of compiler-generated (un)marshalling routines, avoiding thread-switching and layering overhead, and the need for efficient low-level communication mechanisms. Many of these ideas are used in today's communication protocols, including RMI implementations.

Except for support polymorphism, Manta's compiler-generated serialization is similar to Orca's serialization [Bal et al. 1997]. The optimization for nonblocking methods is similar to the single-threaded upcall model [Langendoen et al. 1997]. Small, nonblocking procedures are run in the interrupt handler to avoid expensive thread switches. Optimistic active messages is a related technique based on rollback at runtime [Wallach et al. 1995].

Instead of kernel-level TCP/IP, Manta uses Panda on top of LFC, a highly efficient user-level communication substrate. Lessons learned from the implementation of other languages for cluster computing were found to be useful. These implementations are built around user level communication primitives, such as active messages [von Eicken et al. 1992]. Examples are Concert [Karamcheti and Chien 1993], CRL [Johnson et al. 1995], Orca [Bal et al. 1998], Split-C [Culler et al. 1993], and Jade [Rinard et al. 1993]. Other projects on fast communication in extensible systems are SPIN [Bershad et al. 1995], Exo-kernel [Kaashoek et al. 1997], and Scout [Mosberger and Peterson 1996]. Several projects are currently also studying protected user-level network access from Java, often using VIA [Chang and von Eicken 1998, 1999; Welsh and Culler 2000]. However, these systems do not yet support remote method invocation.

Parallel programming in java.   Many other projects for parallel programming in Java exist.[2] Titanium [Yelick et al. 1998] is a Java-based language for high-performance parallel scientific computing. It extends Java with features like immutable classes, fast multidimensional array access, and an explicitly parallel SPMD model of communication. The Titanium compiler translates Titanium into C. It is built on the Split-C/Active Messages back-end.

The JavaParty system [Philippsen and Zenger 1997] is designed to ease parallel cluster programming in Java. In particular, its goal is to run multithreaded programs with as little change as possible on a workstation cluster. It allows the methods of a class to be invoked remotely by adding a `remote` keyword to the class declaration, removes the need for elaborate exception catching of remote method invocations, and, most importantly, allows objects and threads to be created remotely. Manta optionally allows a similar programming model, but it also supports the standard RMI programming model. JavaParty was originally implemented on top of Sun RMI, and thus suffered from the same performance problem as Sun RMI. The current implementation of JavaParty uses KaRMI.

---

[2]See for example the JavaGrande Web page at *http://www.javagrande.org/*.

Spar/Java is a data and task-parallel programming language for semiautomatic parallel programming [van Reeuwijk et al. 1997]. The Do! project tries to ease parallel programming in Java using parallel and distributed frameworks [Launay and Pazat 1998]. Ajents is a Java system that supports object migration [Izatt et al. 1999]. Several Java-based programming systems also exist for developing wide-area metacomputing applications [Alexandrov et al. 1997; Baldeschwieler et al. 1996; Christiansen et al. 1997].

An alternative for parallel programming in Java is to use MPI instead of RMI. Several MPI bindings for Java already exist [Carpenter et al. 1999; Getov et al. 1998; Judd et al. 1999]. This approach has the advantage that many programmers are familiar with MPI and that MPI supports a richer set of communication styles than RMI, in particular collective communication. However, the MPI message-passing communication style is difficult to integrate cleanly with Java's object-oriented model. MPI assumes an SPMD programming model that is quite different from Java's multithreading model. Also, current MPI implementations for Java suffer from the same performance problem as most RMI implementations: the high overhead of the Java Native Interface. For example, for the Java-MPI system described in Getov [1999], the latency for calling MPI from Java is 119 s higher than calling MPI from C (346 versus 227 s, measured on an SP2).

IceT [Gray and Sunderam 1997] also uses message passing instead of RMI. It enables users to share Java Virtual Machines across a network. A user can upload a class to another virtual machine using a PVM-like interface. By explicitly calling *send* and *receive* statements, work can be distributed among multiple JVMs.

Another alternative to RMI is to use a Distributed Shared Memory (DSM) system. Several DSM systems for Java exist, providing a shared memory programming model instead of RMI, while still executing on a distributed memory system. Java/DSM [Yu and Cox 1997] implements a JVM on top of the TreadMarks DSM [Keleher et al. 1994]. No explicit communication is necessary, all communication is handled by the underlying DSM. DOSA [Hu et al. 1999] is a DSM system for Java based on TreadMarks that allows more efficient fine-grained sharing. Hyperion [Antoniu et al. 2000; Macbeth et al. 1998] tries to execute multithreaded shared-memory Java programs on a distributed-memory machine. It caches objects in a local working memory, which is allowed by the Java memory model. The cached objects are flushed back to their original locations (main memory) at the entry and exit of a `synchronized` statement. Jackal is an all-software fine-grained DSM for Java based on the Manta compiler [Veldema et al. 2001a; Veldema et al. 2001b]. cJVM is another Java system that tries to hide distribution and provides a single system image [Aridor et al. 1999].

6. CONCLUSION

In this article we investigated how to implement Java's remote method invocation efficiently, with the goal of using this flexible communication mechanism for parallel programming. Reducing the overhead of RMI is more challenging

than for other communication primitives, such as remote procedure call, because RMI implementations must support interoperability and polymorphism. Our approach to this problem is to make the frequent case fast. We have designed a new RMI protocol that supports highly efficient communication between machines that implement our protocol. Communication with Java virtual machines (running the Sun RMI protocol) is also possible but slower. As an example, all machines in a parallel system can communicate efficiently using our protocol, but they can still communicate and interoperate with machines running other Java implementations (e.g., a visualization system). We have implemented the new RMI protocol (called Manta RMI) in a compiler-based Java system, called Manta, which was designed from scratch for high-performance parallel computing. Manta uses a native Java compiler; but to support polymorphism for RMIs with other Java implementations, it is also capable of dynamically compiling and linking bytecode.

The efficiency of Manta RMI is due to three factors: the use of compile time type information to generate specialized serializers; a streamlined and efficient RMI protocol; and the use of fast communication protocols. To understand the performance implications of these optimizations, we compared the performance of Manta with that of the Sun RMI protocol. Unfortunately, current implementations of the Sun protocol are inefficient, making a fair comparison a difficult task. To address this problem, we have built an implementation of Sun RMI by compiling the Sun protocol with Manta's native compiler. We also reduced the overhead of this system for native calls, thread switching, and temporary objects. This system, called *Sun compiled*, achieves better latency and throughput than the Sun JDK and the IBM JIT, so we used this system for most measurements in this article, in particular to compare the Manta and Sun RMI protocols.

The performance comparison on a Myrinet-based Pentium Pro cluster shows that Manta RMI is substantially faster than the compiled Sun RMI protocol. On Myrinet, the null-RMI latency is improved by a factor of 8, from 301 s (for *Sun compiled*) to 37 s, only 6 s slower than a C-based RPC. A breakdown of Manta and *Sun compiled* shows that Manta obtains major performance improvements in all three layers. The differences with the original Sun JDK 1.2 implementation of RMI are even higher; for example, the null-RMI latency of the JDK over Myrinet is 1316 s, 35 times as high as with Manta. The throughput obtained by Manta RMI is also much better than that of *Sun compiled*. In most cases, the Sun protocol performs unnecessary byte-swapping, resulting in up to three times lower throughput than Manta's.

Although such low-level latency and throughput benchmarks give useful insight into the performance of communication protocols, a more relevant factor for parallel programming is the impact on application performance. We therefore implemented a collection of six parallel Java programs using RMI. Performance measurements on up to 32 CPUs show that five out of these six programs obtain high efficiency (at least 75%) with Manta, while only two applications perform well with *Sun compiled*. Manta obtains up to 3.4 times higher speedups for the other four applications.

In conclusion, our work has shown that RMI can be implemented almost as efficiently as Remote Procedure Call, even on high-performance networks

like Myrinet, while keeping the inherent advantages of RMI (polymorphism and interoperability). Also, we showed that an efficient RMI implementation is a good basis for writing high-performance parallel applications, although the lack of broadcast support complicates programming of some applications. Adding support for broadcast to RMI by replicating shared objects is a topic of our ongoing research [Maassen et al. 2000].

REFERENCES

ALEXANDROV, A. D., IBEL, M., SCHAUSER, K. E., AND SCHEIMAN, C. J. 1997. SuperWeb: Research issues in Java-based global computing. *Concurrency: Pract. Exper. 9*, 6 (June), 535–553.

ANTONIU, G., BOUGÉ, L., HATCHER, P., MACBETH, M., MCGUIGAN, K., AND NAMYST, R. 2000. Compiling multithreaded Java bytecode for distributed execution. In *Proceedings of the Euro-Par 2000*. LNCS 1900 Springer, München, Germany, 1039–1052.

ARIDOR, Y., FACTOR, M., AND TEPERMAN, A. 1999. cJVM: a single system image of a JVM on a cluster. In *Proceedings of the 1999 International Conference on Parallel Processing* (Aizu, Japan).

BAL, H., BHOEDJANG, R., HOFMAN, R., JACOBS, C., LANGENDOEN, K., RÜHL, T., AND KAASHOEK, M. 1998. Performance evaluation of the Orca shared object system. *ACM Trans. Comput. Syst. 16*, 1 (Feb.), 1–40.

BAL, H., BHOEDJANG, R., HOFMAN, R., JACOBS, C., LANGENDOEN, K., AND VERSTOEP, K. 1997. Performance of a high-level parallel language on a high-speed network. *J. Parallel Distrib. Comput. 40*, 1 (Feb.), 49–64.

BALDESCHWIELER, J., BLUMOFE, R., AND BREWER, E. 1996. ATLAS: An infrastructure for global computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*. ACM, New York, NY.

BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (*SOSP-15*). ACM, New York, NY, 267–284.

BHOEDJANG, R., VERSTOEP, K., RÜHL, T., BAL, H., AND HOFMAN, R. 2000. Evaluating design alternatives for reliable communication on high-speed networks. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-9*, Cambridge, MA).

BLACKSTON, D. AND SUEL, T. 1997. Highly portable and efficient implementations of parallel adaptive N-body methods. In *SC'97*. online at http://www.supercomp.org /sc97/program/ TECH/BLACKSTO/.

BODEN, N., COHEN, D., FELDERMAN, R., KULAWIK, A., SEITZ, C., SEIZOVIC, J., AND SU, W. 1995. Myrinet: A gigabit-per-second local area network *IEEE Micro 15*, 1 (Jan.), 29–36.

BREG, F., DIWAN, S., VILLACIS, J., BALASUBRAMANIAN, J., AKMAN, E., AND GANNON, D. 1998. Java RMI performance and object model interoperability: Experiments with Java/HPC++ distributed components. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing* (Santa Barbara, CA), ACM, New York, NY.

BROWN, A. AND SELTZER, M. 1997. Operating system benchmarking in the wake of Lmbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the 1997 Conference on Measurement and Modeling of Computer Systems* (*SIGMETRICS*, Seattle, WA), 214–224.

BURKE, M., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M., SREEDHAR, V. C., SRINIVASAN, H., AND WHALEY, J. 1999. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA), ACM, New York, NY, 129–141.

CARPENTER, B., FOX, G., KO, S. H., AND LIM, S. 1999. Object serialization for marshalling data in a Java interface to MPI. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA), ACM, New York, NY, 66–71.

CHANG, C.-C. AND VON EICKEN, T. 1998. A software architecture for zero-copy RPC in Java. Tech. Rep. 98-1708, Cornell Univ., Sept.

CHANG, C.-C. AND VON EICKEN, T. 1999. Interfacing Java with the Virtual Interface Architecture. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA), ACM, New York, NY, 51–57.

CHRISTIANSEN, B., CAPPELLO, P., IONESCU, M. F., NEARY, M. O., SCHAUSER, K. E., AND WU, D. 1997. Javelin: Internet-based parallel computing using Java. *Concurrency: Pract. Exper. 9*, 11, 1139–1160.

CULLER, D., DUSSEAU, A., GOLDSTEIN, S., KRISHNAMURTHY, A., LUMETTA, S., VON EICKEN, T., AND YELICK, K. 1993. Parallel programming in split-C. In *Supercomputing*.

FOSTER, I. AND KESSELMAN, C. 1998. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann.

GETOV, V. 1999. MPI and Java-MPI: Contrasts and comparisons of low-level communication performance. In *Supercomputing*. Portland, OR.

GETOV, V., FLYNN-HUMMEL, S., AND MINTCHEV, S. 1998. High-performance parallel programming in Java: Exploiting native libraries. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM, New York, NY.

GRAY, P. AND SUNDERAM, V. 1997. IceT: Distributed computing and Java. *Concurrency: Pract. Exper. 9*, 11 (Nov.).

HAGIMONT, D. AND LOUVEGNIES, D. 1998. Javanaise: Distributed shared objects for internet cooperative applications. In *Proceedings of the Middleware'98 Conference* (The Lake District, England).

HIRANO, S., YASU, Y., AND IGARASHI, H. 1998. Performance evaluation of popular distributed object technologies for Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*. Online at http://www.cs.ucsb.edu/conferences/java98/.

HU, Y., YU, W., COX, A., WALLACH, D., AND ZWAENEPOEL, W. 1999. Runtime support for distributed sharing in strongly typed languages. Tech. Rep., Rice Univ. Online at http://www.cs.rice.edu/~willy/TreadMarks/papers.html.

HUTCHINSON, N., PETERSON, L., ABBOTT, M., AND O'MALLEY, S. 1989. RPC in the x-Kernel: Evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating System Principles* (Litchfield Park, AZ), 91–101.

IZATT, M., CHAN, P., AND BRECHT, T. 1999. Ajents: Towards an environment for parallel, distributed and mobile Java applications. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA), 15–24.

JOHNSON, D. AND ZWAENEPOEL, W. 1991. The Peregrine high-performance RPC system. Tech. Rep. TR91-151, Rice Univ., Mar.

JOHNSON, K., KAASHOEK, M., AND WALLACH, D. 1995. CRL: High-performance all-software distributed shared memory. In *15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO), ACM, New York, NY, 213–228.

JUDD, G., CLEMENT, M., SNELL, Q., AND GETOV, V. 1999. Design issues for efficient implementation of MPI in Java. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA), ACM, New York, NY, 58–65.

KAASHOEK, M., ENGLER, D., GANGER, G., BRICENO, H., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. 1997. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, 52–65.

KARAMCHETI, V. AND CHIEN, A. 1993. Concert—efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing'93* (Portland, OR), 15–19.

KELEHER, P., COX, A., DWARKADAS, S., AND ZWAENEPOEL, W. 1994. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 Usenix Conference* (San Francisco, CA), 115–131.

KRALL, A. AND GRAFL, R. 1997. CACAO—A 64-bit JavaVM just-in-time compiler. *Concurrency: Pract. Exper. 9*, 11 (Nov.), 1017–1030. Online at http://www.complang.tuwien.ac.at/andi/.

KRISHNASWAMY, V., WALTHER, D., BHOLA, S., BOMMAIAH, E., RILEY, G., TOPOL, B., AND AHAMAD, M. 1998. Efficient implementations of Java RMI. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems* (*COOTS'98*, Santa Fe, NM).

LANGENDOEN, K., BHOEDJANG, R. A. F., AND BAL, H. E. 1997. Models for asynchronous message handling. *IEEE Concurrency 5*, 2 (April–June), 28–38.

LAUNAY, P. AND PAZAT, J.-L. 1998. The Do! project: Distributed programming using Java. In *Proceedings of the First UK Workshop on Java for High Performance Network Computing* (Southampton, England).

LIPKIND, I., PECHTCHANSKI, I., AND KARAMCHETI, V. 1999. Object views: Language support for intelligent object caching in parallel and distributed computations. In *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages and Applications*. 447–460.

MAASSEN, J., KIELMANN, T., AND BAL, H. E. 2000. Efficient replicated method invocation in Java. In *Proceedings of the ACM 2000 Java Grande Conference* (San Francisco, CA), ACM, New York, NY, 88–96.

MAASSEN, J., VAN NIEUWPOORT, R., VELDEMA, R., BAL, H. E., AND PLAAT, A. 1999. An efficient implementation of Java's remote method invocation. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'99*, Atlanta, GA), ACM, New York, NY, 173–182.

MACBETH, M. W., MCGUIGAN, K. A., AND HATCHER, P. J. 1998. Executing Java threads in parallel in a distributed-memory environment. In *Proceedings of the CASCON'98. IBM Canada and the National Research Council of Canada*, 40–54.

MOSBERGER, D. AND PETERSON, L. 1996. Making paths explicit in the Scout operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 153–168.

MULLER, G., MOURA, B., BELLARD, F., AND CONSEL, C. 1997. Harissa, a mixed offline compiler and interpreter for dynamic class loading. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies* (*COOTS*, Portland, OR).

NESTER, C., PHILIPPSEN, M., AND HAUMACHER, B. 1999. A more efficient RMI for Java. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA), ACM, New York, NY, 153–159.

PHILIPPSEN, M. AND ZENGER, M. 1997. JavaParty—Transparent remote objects in Java. *Concurrency: Pract. Exper. 9*, 11 (Nov.), 1225–1242. Online at http://wwwipd.ira.uka.de/JavaParty/.

PHILIPPSEN, M., HAUMACHER, B., AND NESTER, C. 2000. More efficient serialization and RMI for Java. *Concurrency: Pract. Exper. 12,* 7, 495–518.

PROEBSTING, T., TOWNSEND, G., BRIDGES, P., HARTMAN, J., NEWSHAM, T., AND WATTERSON, S. 1997. Toba: Java for applications—a way ahead of time (WAT) compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems* (Portland, OR).

RINARD, M. C., SCALES, D. J., AND LAM, M. S. 1993. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer 26*, 6 (June.), 28–38.

RODRIGUES, S., ANDERSON, T., AND CULLER, D. 1997. High-performance local communication with fast sockets. In *USENIX'97*.

SCHROEDER, M. AND BURROWS, M. 1990. Performance of Firefly RPC. *ACM Trans. Comput. Syst. 8*, 1 (Feb.), 1–17.

SUN MICROSYSTEMS. 1997. *Java Remote Method Invocation Specification*, JDK 1.1 FCS, Online at http://java.sun.com/products/jdk/rmi.

THEKKATH, C. AND LEVY, H. 1993. Limits to low-latency communication on high-speed networks. *ACM Trans. Comput. Syst. 11*, 2 (May), 179–203.

VAN REEUWIJK, K., VAN GEMUND, A., AND SIPS, H. 1997. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Pract. Exper. 9*, 11 (Aug.), 1193–1205.

VAN RENESSE, R., VAN STAVEREN, J., AND TANENBAUM, A. 1989. Performance of the Amoeba distributed operating system *Softw. Pract. Exper. 19*, 223–234.

VELDEMA, R., HOFMAN, R., BHOEDJANG, R., AND BAL, H. 2001a. Runtime optimizations for a Java DSM implementation. In *Proceedings of the ACM 2001 Java Grande Conference*. ACM, New York, NY.

VELDEMA, R., HOFMAN, R., BHOEDJANG, R., JACOBS, C., AND BAL, H. 2001b. Source-level global optimizations for fine-grain distributed shared memory systems. In *PPoPP-2001 Symposium on Principles and Practice of Parallel Programming*.

VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. 1992. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Int. Symposium on Computer Architecture* (Gold Coast, Australia), 256–266.

WALDO, J. 1998. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency 6*, 3 (July–Sept.), 5–7.

WALLACH, D., HSIEH, W., JOHNSON, K., KAASHOEK, M., AND WEIHL, W. 1995. Optimistic Active Messages: A mechanism for scheduling communication with computation. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'95*, Santa Barbara, CA), ACM, New York, NY, 217–226.

WELSH, M. AND CULLER, D. 2000. Jaguar: Enabling efficient communication and I/O from Java. *Concurrency: Pract. Exper. 12*, 7, 519-538.

WOO, S., OHARA, M., TORRIE, E., SINGH, J., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 24–36.

YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., AND AIKEN, A. 1998. Titanium: A high-performance Java dialect. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*. Online at http://www.cs.ucsb.edu/conferences/java98/.

YU, W. AND COX, A. 1997. Java/DSM: A platform for heterogeneous computing. *Concurrency: Pract. Exper. 9*, 11 (Nov.), 1213–1224.