Fast Parallel Java

J. Maassen and R. van Nieuwpoort
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

August 25, 1998

vrije Universiteit *amsterdam*

Contents

1 Introduction

There is growing interest in using Java for high-performance parallel programming [1]. Java's clean and type-safe object-oriented programming model makes it attractive for writing reliable, large-scale parallel programs. Unfortunately, there are also many obstacles that make Java a less than ideal choice.

Some problems concern sequential language constructs, like floating point performance, lack of complex numbers and inefficient multidimensional arrays, that possibly can be overcome through careful language revisions. A more challenging problem for parallel processing is the bad performance of Java's primary interprocess communication mechanism: Remote Method Invocation (RMI).

An empty remote method invocation takes about 1200 microseconds on a Myrinet network with a latency of 30 microseconds, and about 1800 microseconds over a Fast Ethernet network with a latency of 230 microseconds.

It is clear that if Java is ever to become successful in the area of parallel processing, its communication performance must be improved dramatically.

Besides the communication performance problems, the programming model of RMI is not designed for parallel programming, but for distributed programming. It is the task of the programmer to explicitly set up connections before RMI can be used. In a truly parallel system, the programmer should be bothered as little as possible with setting up connections.

Our goal is to build a Java runtime system that provides a highly efficient, easy to use communication mechanism similar to RMI (i.e., based on the idea of invoking methods on remote objects), but without restricting ourselves to the specific choices made by the current RMI design or implementation.

The system uses a native Java compiler rather than an interpreter or Just In Time compiler (JIT). This compiler, Jcc, was implemented by Ronald Veldema, as part of his Master's thesis [24]. This compiler generates information that the runtime system uses to speed up RMI, serialization, and garbage collection. The runtime system was designed from scratch to implement remote method invocations efficiently, avoiding the high costs of serialization and thread management. Jcc RMI is implemented on top of Panda, a highly efficient communication layer for Myrinet [6] and Fast Ethernet. The best round trip RMI latency obtained by the system is 35 μsec on Myrinet, and 235 μsec on Fast Ethernet, using a benchmark that invokes an empty method on a remote machine. The Jcc runtime system has a throughput of up to 20 MByte/s for method calls with an array of a primitive type as argument and no return value.

In this thesis, we describe the design of this runtime system. In Section 2, we first look at related work. The general design of our system is described in Section 3, followed by a more detailed description of RMI, threads and garbage collection in Sections 4 to 7. Finally, in Section 8, we present the results of several low-level benchmarks for our runtime system, and performance measurements of some parallel applications.

2 Related work

JavaParty [17] provides a mechanism for parallel programming on distributed memory machines. The *remote* keyword is used to identify which objects can be called remotely. The JavaParty compiler generates the appropriate Java code needed to implement the remote method invocations. It also supports object migration. Allthough JavaParty significantly reduces the effort required to write a parallel Java program, it is implemented in Java using RMIs, causing it to have a comparable peformance.

Titanium [27] is a Java based language for high-perfomance parallel scientific computing. It extends Java with features like immutable classes, multidimensional arrays and an explicitly parallel SPMD model of communication. The Titanium compiler translates Titanium into C. Allthough Titanium supports both shared-memory and distributed-memory architectures, Titanium is not garanteed to run efficiently on distributed-memory systems.

IceT [10] 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.

Java/DSM [28] implements a JVM on top of ThreadMarks, an existing distributed shared memory system. ThreadMarks is used to share data between JVMs running on different machines. This approach has the advantage that no explicit communication is necessary, and very fine grained distribution of data is possible (an array can be distributed over multiple machines). However, this approach does not exploit the Object Oriented model of Java, and does not allow the programmer to make decisions about the parallelism in the program.

The *Javelin* [8] system supports internet based parallel computing using Java. It does so, by running Java applets in web browsers. Therefore, the applications will be interpreted or compiled just in time. As the performance of Java interpreters and JITs is still lower than the performance of compiled java code, running the parallel programs as applets in web browsers restricts the use of systems like Javelin. Another problem is that web browsers use RMIs over TCP/IP, typically over slow Ethernets, so communication latencies are high. The result of this is that Javelin can only be used for running embarrassingly parallel programs. An advantage of running applets is that security is guaranteed.

In a paper in *concurrency: practice and experience*, R.R. Raje et al. describes an implementation of java RMI, called ARMI, which supports asynchronous method invocations, see [18]. The implementation uses the standard Java RMI system and gains a speedup of about 30 % for asynchronous calls. This proves that the concept is promising. Asynchronous RMIs will be a topic for future investigation for the Jcc compiler.

3 Overview of the runtime system

To execute parallel Java programs efficiently, fast communication is essential. The standard Java libraries offer network communication in the form of Datagram (UDP) and SocketIOStream (TCP) classes, and a higher level communication method, Remote Method Invocation (RMI) [29].

None of these libraries provides fast communication. Most of Java's libraries (including the network communication) are written Java. This causes essential parts of the communication code to be interpreted, or, at best, to be compiled by a JIT compiler at runtime. To access the system libraries, the Java Native Interface (JNI) [21] is used. This imposes extra overhead, because Java objects may have to be 'translated' into normal (C) data structures.

Besides lacking speed, none of these libraries were designed to offer adequate parallel programming support. Datagram and SocketIOStreams are too low level to use. They only offer the means to exchange data. The programmer must create his own communication protocol, and all communication must be programmed explicitly. RMI is easier to use, but is very client-server oriented. The programmer must define communication interfaces and *export* server objects to an object *registry* to enable client programs to *bind* to the server. Exporting and binding of objects are a major source of programming overhead; a significant part of the source code is dedicated to setting up communication, and checking for exceptions.

JavaParty [17] offers a better solution for parallel programming. In JavaParty, a number of computers can be used as a single parallel platform. A running Java program can create *remote* objects on other machines. Remote objects are identified by simply adding the *remote* keyword to a Java class. Normal method invocations can be used on these remote objects, and no explicit binding or exporting of objects is required. Unfortunately, JavaParty is uses Java RMI to implement communication, and therefore inherits all it's performance problems (see Figure 1).

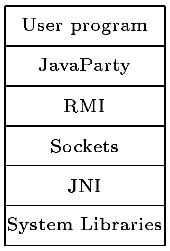


Figure 1: The layers in a JavaParty program.

Performance can be gained by directly compiling a Java program to a binary executable, thus avoiding the overhead of interpretation or JIT compilation of the communication code. Even more performance can be gained by collapsing the layers of Figure 1, and implementing a simpler communication mechanism.

By directly compiling a Java program to a binary executable, some of the platform independence of Java is lost. However, the platform independence of Java is not merely achieved by using byte-code executables. A strict language definition and set of standard libraries ensure that a Java program will encounter the same programming environment, regardless which system the programmer uses. Thus, provided that the source code of a Java program is available, a form of platform independence is also achieved.

3.1 Design

Figure 2 illustrates the design of our Java runtime system. A native runtime system is built directly on the system libraries. This native runtime system contains the basic functionality required to run Java programs: memory management (including garbage collection), thread support, locks, I/O etc. Section 5 gives a short overview of the implementation of Java Threads, locks, and monitors. Section 6 describes the garbage collection techniques used in the runtime system.

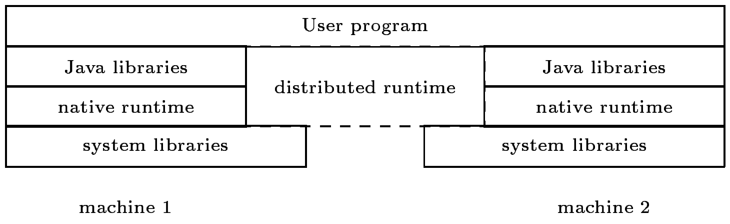


Figure 2: The Java runtime system.

The Java libraries contain the mandatory Java packages *java.lang*, *java.io*, *java.util* and *java.net*. The Java window libraries in *java.awt* have not been implemented yet.

The distributed runtime system resides next to the native runtime system and Java libraries. It is partly implemented in native (C) code, and partly implemented in Java, for easy user access. Section 4 describes the implementation of the distributed runtime system.

The distributed runtime system uses the Panda libraries for communication [4]. The Panda libraries provide communication, threads, locks and condition variables and timers to the user. It features advanced communication primitives, like reliable multicasting and an RPC with at-most-once semantics. Panda is developed at the Vrije Universiteit Amsterdam.

3.2 Java objects in memory

In this section, we will give a short description of the memory layout of objects used by the Jcc compiler and our runtime system. See [24] for an in-depth explanation of the Java object.

3.2.1 Objects and arrays

The memory layout of a Java object consists of an *object header*, containing information necessary to use the object, followed by the data of the object (see Figure 3). A Java array contains some additional information. To be able to index the array, the size of the array elements is recorded in the *element size*

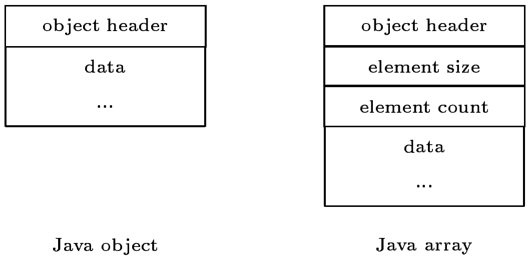


Figure 3: The object and array layout.

field. The *element count* field stores the number of elements in the array and is used for array bounds checking.

An object header starts with a pointer to the *virtual method table.* This table contains pointers to all the methods of this object's class and is extended with type-dependent constant data, such as a table of reference offsets (see Section 6.3.1), and a table of parent type identifiers. A vtable is type-specific, and only one copy of each type of vtable is present in memory, so objects of the same type use a single vtable. See [24] for more information about the Java vtables.

In Java every object can be used as a lock in a *synchronized* statement, or as a monitor using *synchronized* methods. To enable this behavior, a lock must be associated with every object. The lock field is used to store a pointer to the lock of this object (see Section 5).

Some Java objects are able to run in their own thread. A *Thread* object is able to run any Java object implementing the *Runnable* interface. When the *start()* method of the Thread object is called, a new thread is created by the runtime system to run the code of the Thread object. The thread field in the object header is used to store a pointer to the thread information needed to run the Thread object (see Section 5).

The flag field is used to describe basic properties of the object. An *array bit*, for instance, is set if the object is actually an array, and the *finalized bit* is set when the object's *finalizer* has been called by the garbage collector (See section 6.3.1).

Finally the type identifier and the size of the object are also stored in the object's header.

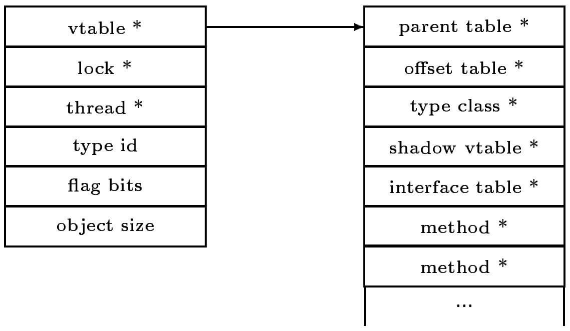


Figure 4: An object header and vtable.

3.2.2 Object stubs

Besides the normal Java objects and arrays, the runtime system also supports *object stubs*. An object stub can be used to represent an object, without actually containing the data of the object. In the distributed runtime system object stubs are used to implement *remote objects* (see Section 7.2.2).

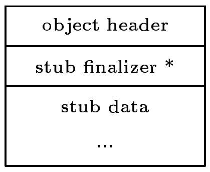


Figure 5: The object stub layout.

An object stub starts with a normal object header, containing the same type identification and virtual method table as the normal object. The *stub bit* is set in the flag field to indicate that this object is really a stub. When a method of this stub is called, the call is rerouted to the *shadow vtable* of this stub. The shadow vtable is contained in the normal vtable of the object (see Figure 4). It contains alternative implementations of the methods in the vtable. For instance, in the stub of a remote object, the shadow vtable contains methods which translate the call to a remote call, sending all parameters over the network to the real object.

The *stub finalizer* is used by the garbage collector when the stub is removed from memory. It enables the stub to free external resources before it is removed (see Section 6).

4 Remote method invocation (RMI)

The goal of Jcc is to support fast and easy parallel computing on a network of distributed machines. In order to allow parallel programming in Java, we need to have a distributed runtime system connecting the different CPUs. This runtime system will provide the compiled Java code with an interface to do remote method invocations.

4.1 Parallel programming with RMI

Parallel programming can be done with Java RMI [23], but this has a few drawbacks. RMI is designed for distributed programming, and is therefore not really suited for parallel programming. Object placement, for instance, is difficult because remote objects may only be created on the local CPU. For client-server applications this is not a problem, but for parallel programming it is.

Another problem with the use of RMI for parallel programming is error handling. The programmer is forced to write exception handling code for every remote invocation, because the system may throw a "RemoteException", indicating communication failure. For distributed programming, this is the desired behavior, because fault tolerance is required. For parallel programming this is usually not needed, because reliable communication hardware or reliable protocols will probably be used.

One attempt to solve these and other problems is the JavaParty [17] system, a Java extension for parallel programming, which is built on top of Java RMI. Jcc uses almost the same programming paradigm as the JavaParty. The implementations of Jcc and JavaParty differ considerably, however, as Jcc RMI is not built on top of Java RMI. Also, JavaParty uses an interpreter, whereas Jcc uses a native compiler. Another difference is that JavaParty supports object migration, while Jcc and Java RMI do not.

In the Jcc system, a security manager is not present. This is intentional because our goal is high performance computing, not running secure applications in browsers. Besides hindering performance, security may sometimes also hamper ease of use, another design goal of our system.

Heterogeneity is also not supported by the Jcc system at this moment. Thus, it is not possible for big endian machines to communicate with little endian machines. For clusters of identical machines, this is not a problem. The implementation of heterogeneous communication, and the measurement of the performance penalty that comes with it, is an area which requires future work.

Java RMI has both heterogeneous communication and a security manager. Because JavaParty is built on top of RMI, it also has these features.

In both the Jcc and the JavaParty programming models, the only thing needed to convert a normal object into a remote object is to add the *remote* modifier to the class declaration. This greatly simplifies the conversion of sequential to parallel programs.

To illustrate the difference between Java RMI and our programming model, a simple "helloworld" application is shown in Figures 6 and 7 for Java RMI and in Figure 8 for Jcc RMI. Another indication is the number of words in the parallel applications we used to bechmark the runtime system, this is shown in Appendix B.2.

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

public interface HelloServer extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException;
}

public class HelloServerImpl extends UnicastRemoteObject implements HelloServer {

    public String sayHello() throws RemoteException {
        return "Hello, world!";
    }

    public static void main(String args[]) {
        // Create and install a security manager.
        System.setSecurityManager(new RMISecurityManager());

        try {
            HelloServerImpl server = new HelloServerImpl("HelloServer");
            Naming.rebind("//" + InetAddress.getLocalHost() + "/HelloServer", server);
        } catch (Exception e) {
            System.out.println("HelloServerImpl error: " + e.getMessage());
        }
    }
}
```

Figure 6: Hello world using RMI, server side.

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

class HelloClient {
    public static void main(String arg[]) {
        try {
            HelloServer server = (HelloServer)Naming.lookup("//" + getCodeBase().getHost() + "/HelloServer");
            System.out.println(server.sayHello());
        } catch (Exception e) {
            System.out.println("Hello exception: " + e.getMessage());
        }
    }
}
```

Figure 7: Hello world using RMI, client side.

```
remote class Hello {

    public String sayHello() {
        return "Hello, World !";
    }

    public static void main(String arg[]) {
        // Create new remote objects on CPU 4.
        RuntimeSystem.setTarget(4);
        new Hello().sayHello();
    }
}
```

Figure 8: Hello world using Jcc.

4.2 Java extensions

The most important Java extension is the introduction of the “remote” keyword. We believe that a new keyword is easier to use and learn than using non intuitive “magic” interfaces such as java.rmi.Remote, as used with Java RMI. When a class is tagged with the new *remote* keyword, every call to a method in this class will be translated into a remote procedure call instead of a local call. In Jcc, the only place where a remote keyword can be legally placed is in the list of class modifiers. This is done because this is enough to allow parallel programming, and we want to modify the Java language as little as possible.

4.2.1 Remote object creation

Java RMI uses a centralized registry to locate remote objects. This central registration of remote objects is needed because Java RMI has a somewhat different programming model than Jcc's remote calls. When using Java RMI, objects accepting remote calls will always be created on the local CPU, because this is logical for client-server based computing. When other objects want to invoke methods of the remote object, they must get a remote reference to it. One way to acquire this reference is to ask the registry for it.

In the Jcc programming model, remote objects may be created on any CPU known to the runtime system, including the local CPU. It is thus possible for code being executed on CPU 1 to create a remote object on CPU 4. There is no centralized naming service to find and create remote objects. Host lookup is instead performed by the runtime system, which maintains a list of hosts willing to accept RPCs. Targeting a specific CPU is performed by invoking *RuntimeSystem.setTarget(processorNumber)*. Which machine is bound to a given processor number is a task of the underlying runtime system. The distributed runtime system can also place objects automatically using a number of policies. At this point, only round robin placement is implemented. When required, the user may implement other scheduling policies using the *setTarget()* method.

4.2.2 Variables in remote classes

In Jcc-RMI, the only way to access the state of a remote object is by method invocation. This means that one cannot access the fields of a remote class from outside the class. If an attempt is made, a compile time error occurs. Some examples of this behavior are shown in Figure 9. Both standard Java RMI and JavaParty have the same behavior as Jcc in this respect. With Java RMI this follows from the language definition. All methods which may be called remotely are specified in an interface. Variables cannot be declared in an interface and can thus not be accessed remotely.

It is possible to implement semantics, where access of non-static class fields is allowed. We chose not to do this, because the compiler does not really know the use of the fields. Some fields are never accessed from outside the class, others require synchronized access. Synchronized get/set-methods could be generated for each class field, but this may add unnecessary overhead. The programmer has to provide the wanted get/set-methods when they are needed. This is not a problem, because the use of get/set-methods is considered good programming style for object oriented languages.

Static variables, however, are the exception to this rule, for both Jcc and JavaParty. The location of static variables can be determined at runtime, because they are allocated on one processor only. Static variables may be accessed just like when using normal classes and are shared between all instances of the class on all CPUs. Java RMI does not support static variables.

```
remote class A {
    int var;

    void foo() {
        var = 1; // OK, var is in my class.
    }
}

class test {
    remote int i; // NOT allowed, only classes may be remote.

    public static void main(String arg[]) {
        A a = new A();
        a.var = 1; // NOT allowed, var is a variable in a remote class.
    }
}
```

Figure 9: The remote keyword and local variables.

4.2.3 Parameters to remote method invocations

Object migration (as in JavaParty) is not implemented in Jcc. Neither is it present in standard Java RMI. References to remote objects, however, may be passed around as a parameter to both local and remote method invocations. They can also be used as return value. While *call-by-value* is used by normal parameters to remote calls, remote references are passed *by reference*. This is the same as in the standard Java RMI and JavaParty system. The reason for this is that remote objects cannot be serialized easily, as they potentially reside on another machine.

A performance problem with JavaParty occurs when a method in a remote class calls a method in the same class. Because the called method was defined in a remote class the parameters to the call will be serialized, even though the called method resides in the same class on the same machine. An example program is shown in Figure 10.

```
remote class A {

    void bar(String s) {
        System.out.println(s);
    }

    void foo() {
        String s = new String("this string is passed by value");
        bar(s);
    }
}
```

Figure 10: Calls within remote classes.

The string parameter will now be passed using call by value and not call by reference, as in non-remote classes. This way, when a method in a remote class is called, the parameters to the call will always be passed by value, resulting in clear semantics. As will be discussed in Section 8.3.2, JavaParty programs that use many calls to remote methods within the same class (e.g., a recursive method in a remote class) have a very high serialization overhead. Performance of serialization and marshalling is better in the Jcc system, but some unnecessary overhead remains.

In order to solve this problem, the *local* modifier was introduced. Methods in remote classes may be marked with this modifier to indicate call by reference must be used for parameter passing. This implies that the method may no longer be called from another machine. An example of the use of the local modifier is shown in Figure 11. The difference between methods tagged with the local modifier and static methods is that static methods can only access static variables. Local methods can access all class fields.

Another option to solve the problem would be the introduction of a "by_ref" modifier, which can be put in front of the parameters in the formal parameter list. This allows the programmer to specify for every parameter whether it should be be passed by reference.

```
remote class A {

    int i;

    local void bar(String s) {
        i = 1; // Allowed, 'i' is a field in my class.
        System.out.println(s);
    }

    static void staticBar() {
        i = 1; // NOT allowed, 'i' is not declared in static context.
    }

    void foo() {
        String s = new String("this string is passed by reference");
        bar(s);  // Call by reference will be used, because bar is marked "local".
    }
}

class B {
    A a;

    void faa() {
        a.bar("some text");  // NOT allowed, bar is marked local in class A.
    }
}
```

Figure 11: The local modifier.

4.2.4 Remote classes and inheritance

In Jcc, remote classes can also be inherited from, provided the new sub-class is also remote. A compile time error occurs if the programmer attempts to extend a remote class with a non-remote class. This way, class hierarchies of remote classes can be built and no runtime checking has to be done to see whether a class is possibly remote because it inherits from a remote class. More importantly, the semantics of remote classes remain clear. A strange situation occurs when a remote class is extended with a non-remote class. It becomes unclear whether the programmer is allowed to call the methods of the local subclass from another machine and whether call by value or call by reference is used for the invocations.

In our current implementation it is illegal to call methods from a non-remote base class of a remote class from outside the remote class. An example is shown in Figure 12.

```
remote class A {}

class B extends A {} // NOT allowed, 'B' is not declared remote, while A is.

remote class C extends A {} // Allowed, both A and C are remote.

class D {
    void foo() {}
}

remote class E extends D {
    void bar() {}

    void faa() {
        foo(); // OK, calling local method of our base class.
    }
}

class Test {
    public static void main(String arg[]) {
        new E().foo(); // NOT allowed, foo is declared in a non-remote base class.
        new E().bar(); // OK, bar is remote.
    }
}
```

Figure 12: Inheritance and remote classes.

The A class was not declared remote, so "foo()" may not be called remote. This still holds when A is extended by a remote class. It was defined this way, because there are no marshallers generated for the A class, since A was not marked remote. Marshallers contain code to serialize the parameter list to a call and will be explained in Section 4.4.1. This may easily be circumvented by generating the marshallers for *all* classes, whether they are declared remote or not. We chose not to do this, because the executable sizes would be enlarged significantly by this. It is now clear why "foo()" may be called by the "faa()" method. Both methods are in the same class and may thus be called locally like a normal method invocation, without using marshallers.

4.2.5 Remote exceptions

Remote exceptions are supported in Jcc RMI. When the Java code executed in a remote call throws an exception, that exception will be sent back over the network to the caller of the remote method. An example of using an exception in a remote method is demonstrated in Figure 13. The difference with standard Java RMI is that the runtime system does not throw exceptions when the underlying communication mechanism fails. Thus only exceptions thrown by the Java program have to be caught. When a communication error does occur, the system will just give an error message and exit. For parallel programming this is normally not a problem because reliable communication hardware or reliable protocols will probably be used.

4.2.6 Nested remote classes

In Java, classes may be nested. In Jcc, the programmer is not allowed to make a nested class remote. The compiler will give an error when this is tried. The reason for this is that in Java a nested class can access fields from the parent class. Allowing this would imply that the remote class would require a remote

```
remote class A {
    void foo() throws Exception {
        throw Exception("This is a remote exception");
    }

    void faa() {
    }
}

class Test {
    public static void main(String arg[]) {
        try {
            new A().foo();
        } catch (Exception e) {
            System.out.println("an Exception occurred: " +e.toString());
        }

        new A().faa(); // No exceptions have to be caught.
    }
}
```

Figure 13: Exceptions and remote classes.

reference to a local class, as is shown in Figure 14. When using standard Java RMI, this is not a problem, because the user has only the interface to the remote class. Variables may not be specified in an interface and can thus not be accessed. The other way around, however, is allowed. It is permitted for a normal nested class to access fields of a parent remote class. An example of this is the nested class in Figure 15.

```
class A {
    int a;

    void foo() {}

    // This remote inner class is NOT allowed.
    remote class B {
        int b = a;    // The a field is a variable from the outer A class.
    }
}
```

Figure 14: Remote inner classes.

```
remote class A {
    int a;
    class B {
        void foo() {
            a = 2;
        }
    }
}
```

Figure 15: Inner classes in remote classes.

4.2.7 Casting remote references

It is not allowed in the Jcc system to cast a remote object to a local object. The reason for this, is that the compiler would no longer be able to distinguish between remote and local classes and thus again allow access to fields in the object. This may again be avoided, but expensive runtime checks to find out whether a local object is really a remote object cast to a local object would be needed. When using JavaParty or Java RMI, the programmer is allowed to write a cast from a remote class to a local class. A runtime exception will be thrown when this is done incorrectly.

4.3 Serialization

The Java serialization mechanism [22] offers the possibility of writing arbitrarily complex objects to a stream and reading them back. When doing this, a deep copy is made of the object, meaning that not only the object passed to the write call is serialized, but also all objects this object has a reference to. In this way, an entire tree of objects is written to the stream with one write call. Consider, for example, a linked list. When the first element in the list is written, the entire list will automatically be serialized, because the first element has a reference to the second, which has a reference to the third, etc.

One potential problem springs to mind: What if the data structure to be serialized has duplicates or cycles? We have to prevent objects to be serialized more than once. The solution is not really difficult, but it does require some administration. We maintain a list of objects that have already been serialized. If we encounter a reference to an object that has already been serialized, we just write some offset to the previously written object to the stream. It is evident that we must be able to find out whether an object has been serialized before as fast as possible, therefore we use a hash-table to store the references to previously encountered objects.

4.3.1 Why is serialization important?

Serialization is an important aspect of the runtime system because it is used to implement RMI. Because any object may be passed as a parameter to a remote invocation, it is evident there must be some way to send the object to the remote CPU. Serialization is thus used to implement the call by value semantics of RMI, which we will discuss in Section 4.4.

In most current Java implementations, a large part of the serialization code is written in the Java language itself, which is interpreted (or compiled just-in-time), and thus quite slow. Because serialization is also used for implementing RMI, it is imperative for the performance of a distributed system that it be implemented efficiently. Therefore, in Jcc, the serialization code is completely native, and written in C. For even better performance, a large part of this code is generated at compile time by the Jcc compiler. This has the advantage that we don't have to inspect the object at runtime, asking it what fields it has, because this is already known at compile time.

Another point where our implementation differs significantly from the standard Java implementations is the protocol used for the serialized object layout. We have implemented our own simple and fast protocol, resulting in better performance. This has one disadvantage, as we now are no longer compatible with other Java implementations. The result is that we cannot read a file containing serialized objects according to the Java serialization standard [22], and cannot communicate through RMI with Java virtual machines. The programming interface, however, is still compatible, so any Java source file using serialization can be compiled unchanged with our compiler, as only the underlying protocol differs.

When Java objects use serialization to save state in files, the potential problem arises that the version of a class reading the data is different than the version that wrote the data. To solve this problem, The Java serialization protocol supports versioning. This is not an issue with parallel programming, because all hosts run the same executable. Therefore, versioning is currently not supported in the Jcc protocol. Another difference between the Jcc protocol and the Java protocol is the treatment of strings. In the Java protocol, strings are represented in UTF-8 [26] encoding. The Jcc protocol treats strings just like normal objects. The array of characters containing the string data is serialized like any other array of a primitive type.

4.3.2 Duplicate detection in serialized objects

In order to detect duplicates, the serialization code uses a hash table when writing the objects, to store all references that have been seen so far. Each time we serialize some reference, which may also be an array or an interface pointer, we search the hash table to see if we encountered it before. It is important that this search be fast, as it might occur often when serializing complex data structures. If the reference is not found in the hash table, we add the reference and an index number, which is the current number of entries in the hash table. Next, we call the appropriate function to serialize the object referenced to. If the reference was present, however, an opcode is stored, indicating this reference was already seen, followed by the index number which was stored together with the reference in the hash table.

When deserializing, we use a reference table, which is an array of pointers and a count. Each time we have deserialized and created an object, we put the reference to this object in the table. When we encounter the opcode representing an object which is already seen, we read the following index number. Because objects are serialized and deserialized in the same order, it can now easily be seen that the index number is the index in the reference table, which contains the reference to the earlier deserialized and created object. So instead of trying to deserialize the reference, we just return the reference at the now known index in the reference table.

4.3.3 Generated serialization code

When compiling a source file, the Jcc compiler generates code to serialize and deserialize all classes in the file. The generated C code will directly store the passed object's primitive types into a buffer and call the appropriate generated serializers for fields referencing other objects. Because arrays are objects in Java, but there is no such thing as an Array class, arrays must be treated separately, as there is no code generated to serialize them. Hence, the code to serialize arrays was hand-crafted in C. To get an idea of the generated code, consider this example class "Monkey," shown in Figure 16.

```
class Monkey {
    int i, j;
    double d;
    long[] longArray;
    String string;

    int foo(int p, String s) {
        // do something here...
    }
}
```

Figure 16: A simple Java class.

The C code generated by the Jcc compiler will look like the code fragment in Figure 17. Although this is greatly simplified pseudo code, it does give an indication of what the real code is like. The address of the buffer pointer is passed to every write-call and not the buffer pointer itself, because the write-call increments the buffer pointer by the amount of data that was written to the buffer.

```
void packageClass_Monkey(char* buffer, flush_function_t flush_function) {
    writeInt(&buffer, 345987346, flush_function); // unique class id
    writeInt(&buffer, i, flush_function);
    writeInt(&buffer, j, flush_function);
    writeDouble(&buffer, d, flush_function);
    writeArray(&buffer, longArray, flush_function);
    packageClass_String(&buffer, string, flush_function);
}
```

Figure 17: The generated code to serialize the Monkey class.

Special care was taken to optimize for one dimensional arrays of primitive types. All other arrays contain pointers to arrays or objects. For arrays of primitive types however, the relevant data is stored adjacently. This means that the data may just be written to the serialization buffer with one "memcpy()." Therefore, in our system it doesn't matter whether arrays of integers or arrays of doubles are serialized, both are just copied entirely. The standard Java serialization mechanism always loops over all entries in the array to call the appropriate method to serialize the entry, even for primitive types. Both the code to serialize the array and the code to write the primitive types are programmed in Java and are interpreted (or compiled just in time with a JIT-compiler). It is clear that this method is significantly slower than the method used in the Jcc runtime system. This method has one drawback however. Because the array data is copied to the serialization buffer, the internal representation of the elements is preserved. Therefore, a little endian machine is not able to deserialize data from a big endian machine, and vice versa.

Because RMI is discussed later, see Section 4.4, we will explain our serialization implementation using the Java ObjectInputStream and ObjectOutputStream classes, which allow the serialization of objects to any stream. An ObjectOutputStream is created by calling the class constructor with an OutputStream object as parameter. The OutputStream may be any stream, for instance a FileOutputStream for writing to disk, or a SocketOutputStream for sending objects over the network. We may now call the newly created ObjectOutputStream's writeObject method, with any object implementing the java.io.Serializable interface as parameter. In Java, arrays are also serializable objects, so they can be passed directly, without encapsulating them in an object. An example Java program using object streams on top of a file stream is shown in Figure 18.

From the ObjectOutputStream's writeObject method, a native call to the runtime system is done to serialize the object. The native serialization is always done to some buffer, and not directly to the passed stream. When we run out of buffer space, a user defined native flush function is called, after which the serializing process is continued. When serializing to an ObjectOutputStream, the flush function will do a virtual call back to Java, calling the ObjectOutputStream's flush method. This is needed because we do not know at compile

```
import java.io.*;

class Test implements Serializable {
    int i;
    double d;
    long[] longArray;
}

class TestObjectStream {
    public static void main(String args[]) {
        ObjectOutputStream oos;
        FileOutputStream fos;
        ObjectInputStream ois;
        FileInputStream fis;

        Test t = new Test();

        try {
            fos = new FileOutputStream("filename", false);
            oos = new ObjectOutputStream(fos);

            oos.writeObject(t);
            oos.writeObject(new int[20]);
            oos.close();

            fis = new FileInputStream("filename");
            ois = new ObjectInputStream(fis);

            t = (Test) ois.readObject();
            int[] array = (int[]) ois.readObject();
            ois.close();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            System.exit(1);
        }
    }
}
```

Figure 18: An example Java program using object streams.

time what kind of stream we are serializing to. The ObjectOutputStream class serializes data to an instance of the OutputStream class, which may be extended by another class.

After the flush function is called, the serialization process continues. When we are actually marshalling parameters to a remote call, the flush function is entirely native, and will send a packet to the remote CPUs over the network. The flush function in respect to network packet fragmentation will be discussed in more detail in Section 4.4.5.

The deserialization process is exactly the reverse of the serialization process. Instead of a flush function, the deserialization code uses a *fill* function. It is the responsibility of the fill function to deliver the next data chunk. The fill function will for instance read the next block from disk, when a FileInputStream is used for deserializing.

4.4 Executing a remote call

The primary goal of the Jcc runtime system is high performance. In order to achieve this for remote method invocations, it is important that the call request message is built fast and that it does not have to go through a large number of layers to get to its destination. In the Java RMI implementation, however, many different layers are used, most of which are interpreted (or compiled just in time). JavaParty is built on top of the RMI system and consists therefore of even more layers. Both systems suffer from bad performance.

To solve the performance problem, two techniques can be used. Layers may be collapsed into fewer layers and the layers can be implemented using native code instead of interpreted Java. In the Jcc runtime system we have used both techniques. The layer structure of Java RMI, JavaParty and Jcc is shown in Figure 19. The white layers are written in Java and are interpreted, the grey layers are native code.

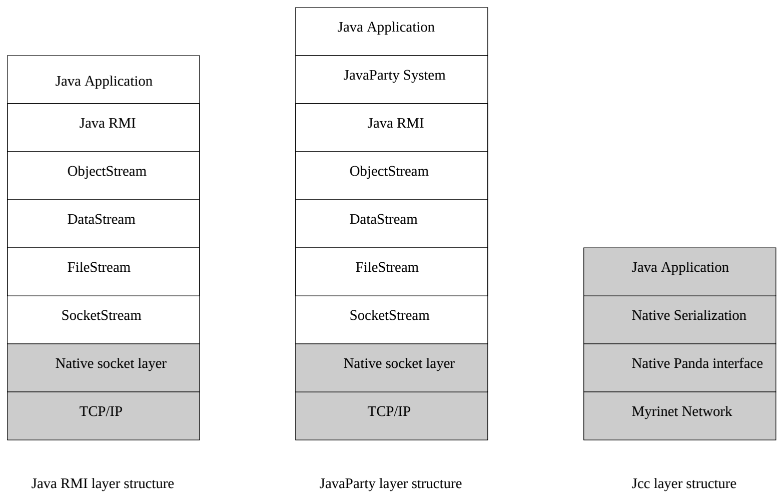


Figure 19: Structure of the different RMI systems.

The Java IOStream layers (the layers “ObjectStream” to “SocketStream” in Figure 19) are passed *four* times for each remote method invocation: twice for packing and unpacking the request message, and twice for packing and unpacking the reply.

The Jcc runtime system uses only four layers to implement RMI. All layers are compiled code, the top level (the user program) is compiled Java code, the bottom three are compiled C code. Note that the Java IOStream classes are not used with Jcc. The generated serialization code was designed to write the serialized objects to a buffer, instead of directly to a stream.

The code to build a message for a remote method invocation request, called *marshalling* code, is generated by our Jcc compiler. To serialize the parameters passed to the method invocation to a buffer, the generated serialization code, as described in Section 4.3.3, is used. This buffer, containing the serialized parameters and a small header, will then be sent over the network using the Panda RPC interface. This interface is shown in Appendix C.

When it arrives at the remote location, the buffer will be deserialized by the generated unmarshalling code. The flow of control in the Jcc runtime system when executing a remote call request is shown in Figure 20. This is the most complex case, when a thread is created to handle the call request. The rounded boxes denote Java code, the grey boxes generated code. The implementation of the Jcc RMI will be discussed in more detail in the following sections.

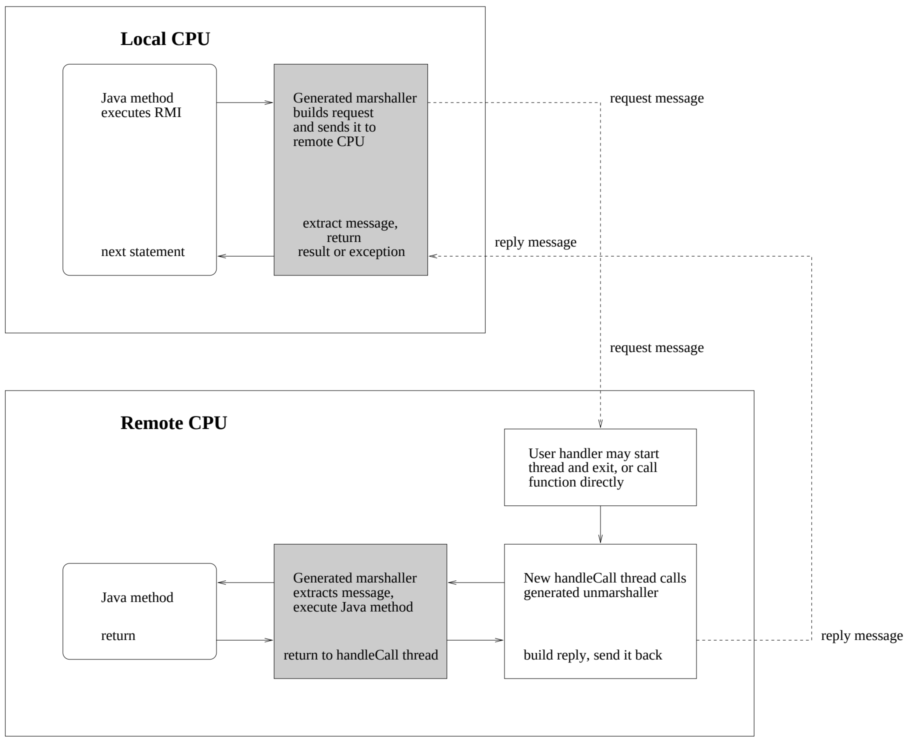

Local CPU
Java method executes RMI
next statement
Generated marshaller builds request and sends it to remote CPU
extract message, return result or exception
request message
reply message
request message
Remote CPU
User handler may start thread and exit, or call function directly
Java method
return
Generated marshaller extracts message, execute Java method
return to handleCall thread
New handleCall thread calls generated unmarshaller
build reply, send it back
reply message

Figure 20: Flow of control of a remote call.

4.4.1 Generated marshallers

The Jcc compiler generates the code to do a remote call, for both the client and the server side. The generated code first creates the hash table and reference table to store all references seen during serialization of the parameters for this call and the return value, so that duplicates can be detected. To achieve good performance, it is important to make the execution paths that are simple and occur frequently as fast as possible. Since remote calls with simple parameters occur quite often, we have tried to optimize these calls. Therefore, the tables to detect duplicates are only created when at least one of the parameters to the call is an object or array. Because the marshalling code is generated, we can detect at compile time whether a remote method has only parameters of primitive types. When only primitive types are passed to the remote call, there is no possibility of duplicates, and thus no tables are needed. There are more optimizations like this possible, for instance when only primitive arrays are passed, but these are not yet implemented. The creation of the tables is expensive, because they have to be allocated and also cleared in the case of the hash-table, so the optimization for the case of simple parameters does improve overall performance, as we will see later. The hash-table has to be cleared because the entries are not stored adjacently, but are scattered over the table. Thus, all slots in the table have to be set to zero at the start of the marshalling process, to indicate the slots are empty.

After the creation of the tables, a buffer is allocated to hold the serialized parameters, and the header that is put in front of them. This header contains five fields, as shown in Figure 21.

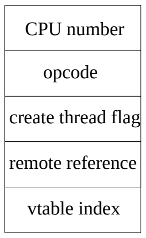


Figure 21: The header of a call message.

The *CPU number* is an integer, that identifies the CPU doing the call. It is needed by the remote CPU to send a reply message. The possible *opcodes* are displayed in Table 1. For a remote call, the call-opcode is put in the header, to indicate the message that follows is a remote call request. Because remote method calls and remote constructor invocations (opcode "new") are treated exactly the same in our system, we will proceed to describe the implementation of remote calls.

opcode	meaning
quit	Stop runtime system, exit program.
new	The message that follows is a remote new request.
call	The message that follows is a remote call request.
return_void	This is a reply from a void method invocation.
return_exception	The remote method or constructor has thrown an exception.
result_new	The message that follows is the reply from a remote new.
result_call	The message that follows is the reply from a remote call.
gc_call	Used for distributed garbage collection.

Table 1: Possible opcodes in message headers.

The *create thread* flag is used to optimize remote calls to methods that are guaranteed to terminate quickly. For such methods, the runtime system does not create a thread to handle the request. There are two cases in which a thread will be created: the method contains a loop or executes a call. Note that this is conservative. When there is a possibility the method will block (the call might be "wait()", or call "wait()" indirectly), or consume a lot of time, a thread is created. This is done because the message may be received by the client in a signal handler. The user handler will be described in more detail in Section 4.4.2.

When the create thread flag is set, the runtime system at the receiving side will create a thread, which will handle the unmarshalling of the parameters and the actual invocation of the Java method. When the flag is not set, the user handler will take care of the unmarshalling and the invocation of the Java function. This way, no thread creation or thread switch is needed to do the remote call. Even with a fast thread package, this improves performance substantially. In object oriented languages such as Java, there are many method calls that just return some field in the class, or the result of a simple computation. These calls will benefit from this approach. Again, we tried to make the paths that are simple and occur frequently as fast as possible.

The *remote reference* field in the call header message is the reference to the object on the remote CPU on which we want to invoke the call. This reference is stored in the stub for the remote object, which will be created as a result of a remote new operation (a constructor invocation). The layout of the stubs was described in Section 3.2. We have chosen to directly use the reference to the remote object, because of performance reasons. This has one drawback, however: object migration will be hard to implement with this approach. When object migration is required, an extra level of indirection will be needed. This means an extra lookup, with some performance penalty.

The last field in the header is an index in the object's vtable, where the pointer to the requested method is stored. The object and vtable layout were explained in Section 3.2. Together with the *remote reference* field, the *vtable index* field forms a unique identification for the method to be executed.

After the call-header, the parameters of the remote call will be stored in the message using the serialization mechanism described earlier. Note that the serialization code was also generated, so the entire process of doing the remote invocation is native code. Because the marshalling code has been generated by the compiler, there is no need to inspect the parameters passed to the remote call at runtime. The compiler has created a sequence of calls to the correct generated serialization routines.

When the request message has been built, it is sent to the remote CPU, using the Panda remote procedure call interface. We have also built a runtime system that uses TCP/IP sockets, but the underlying communication channels do not concern us here.

4.4.2 Handling the request

On the receiving side, Panda calls a user defined handler with the request message as an argument. This function may be called from a lower level signal handler, which restricts the behavior of the user handler. The Panda RPC interface is shown in Appendix C. Panda polls the network interface to find out whether a new packet has arrived, when the application is idle. The user handler will only be called from a signal handler when the network interface is not polled. An interrupt is generated when a message comes in and the application has not polled the network for some time. This interrupt will be converted to a Unix signal, which will call the user defined handler. The OpenThreads [12] thread package assures that only one signal handler is active at a time. Multiple user handlers may be active at the same time, however, as the application may be polling. Panda, and the optimizations used in the implementation are explained in [15].

The user handler is not allowed to wait for the arrival of another message, because this message may be received in a signal handler, and only one signal handler may be active at one time. It is illegal for instance, to block on a mutex, when the corresponding unlock operation is done in another remote call. It is also undisirable for the user handler to run for a long time, again because one signal handler may be active at once. Incomming messages must be stalled while a signal handler is running and the apllication is not polling. Therefore, when the Java method may potentially block or run a long time, a thread has to be started to handle the request message, see Section 4.4.1. When the thread is started, the user handler exits immediately. Because the *create thread* flag in the call header is used to determine whether we need to start a thread or not, the call header is at least partly processed in the user handler, to decode the opcode and the *create thread* flag. The *HandleCall* or *HandleNew* function, which will process the request, will either be called directly from the handler or from the newly created thread. In one of these functions, the message header will be decoded further.

Next, the generated unmarshaller will be called. For every method in a remote class, a pointer to the unmarshaller resides in the shadow vtable of the object, see Section 3.2. The remote object reference and vtable-index of the unmarshaller were packed in the call header, so we can find a pointer to the correct unmarshal function.

4.4.3 The generated unmarshaller

The generated unmarshaller will simply decode all parameters to the remote call from the request buffer by calling the correct deserialization routines. When the deserialization is finished, the requested Java method can be called. This is not difficult, because the Jcc native interface is quite simple, as Jcc is a native compiler and not an interpreter. As a result, The Jcc native interface is easier to use and more efficient than the standard JNI [21] (Java Native Interface). For a more detailed description, see [24].

The return value from the Java function (or, when thrown, the exception) will be serialized to a buffer. This buffer will then be sent to the CPU that requested the call, using the appropriate Panda call. The remote CPU has now finished the handling of the request.

The reply from the remote invocation will be received in the generated marshal function where the request originated, because the Panda RPC interface is a synchronous communication primitive. The generated code will deserialize the reply and return the result back to Java. When the reply buffer contained an exception, this will also have to be passed on to the Java code. The Jcc native interface provides the means to throw an exception from native code to Java.

4.4.4 Buffer management

One of the most important and also most difficult issues when implementing remote calls is the management of the buffers needed for serialization and duplicate detection. Allocating a buffer with malloc is expensive, and on some systems (e.g., BSDI) malloc is not thread-safe or signal-safe. The Java language makes heavy use of multi-threading, and what is worse, the request handler function may be called from a signal handler. This implies that buffer management has to be both thread- and signal-safe. To make things even more complicated, remote calls may be nested in our system. This means that the Java code in a remote method may do a remote call itself. The result is that a remote call may be executed from the generated unmarshal function. Another problem is that several request handler functions may be active concurrently when messages arrive at the same time. The conclusion is that not only buffer management, but the entire system must be both thread- and signal-safe.

To protect shared buffers, locks are needed. They are quite expensive, however, and have to be avoided.

Several buffer management policies were examined. We considered allocating one buffer for serializing remote method invocation requests for each CPU in the system. This fails when two threads on the local machine want to communicate with the same remote CPU, since the request buffer cannot be reused until the reply has arrived, because the Panda RPC mechanism is synchronous. Allocating one buffer for each running thread was also considered, but this does not work when remote calls are nested. It is clear that none of these policies is correct. Buffers have to be allocated for each call and cannot be shared. Panda has a fast malloc implementation, but measurements show that a significant amount of time is still spent allocating and freeing buffers. This is the price that has to be paid for a system that is completely thread- and signal-safe. All parts of the serialization and marshalling code are now safe, without a single lock in it. There is one lock in the Panda malloc implementation, however.

An important issue for performance of network throughput and latency is to minimize the number of buffer copies. Memory to memory copies are expensive and must be avoided, especially when the network has a relatively large throughput compared to the memory bus, as is the case with the combination of Pentium Pro and Myrinet network we use. The number of buffer copies was minimized by serializing and marshalling directly from the Java data structures to the buffer, which also holds the call header. The entire message is directly built into one buffer, which will directly be passed to Panda (or written to a TCP/IP socket). For the reverse process, the same holds. Deserialization will extract information out of the Panda buffer, and copy the data directly to the Java data structures.

4.4.5 Fragmentation

Fragmentation is the splitting of the buffers into chunks that will fit in the network packets. The splitting of the serialized data buffer is handled by the flush function. This function is passed as an argument to the serialization and marshalling code, as was described in Section 4.3.3. For the socket version of the runtime system, for example, the buffer is written to a TCP/IP socket when the flush function is called. The buffer may then be reused to serialize the rest of the data. With the Panda version of the runtime system, an optimistic approach was implemented. A Myrinet network, which has a round-trip of about 30 microseconds at the user level, was used. The Panda RPC interface provides a synchronous communication function. When sending a request to a remote machine, the caller will block until the reply message has arrived. Arbitrarily large data buffers may be sent using the RPC primitive. Because of the synchronous communication and the latency of 30 microseconds, it is faster to do a *realloc()* to increase buffer size in the flush function, instead of writing the data to the network. When the system does run out of buffer-space, the flush function doubles the buffer-size, and the serialization process will continue. Reallocating the buffer is faster than 30 microseconds, so this approach is faster than doing multiple network writes. Panda also provides asynchronous, one way communication primitives. When using these, it would probably be faster to send a packet in each call of the flush function. This will be examined in the future.

Another solution would be to traverse the data structures to be serialized in advance, to count the number of bytes that will be used for serialization. This model has some performance overhead, but has the advantage that the required buffer space is known in advance. We chose to take the optimistic approach, and assume that a reasonable sized buffer will probably suffice for most remote method invocations. Comparing both models is a topic for future investigation.

4.4.6 An example

Consider the “RemoteMonkey” class in Figure 22. The “foo()” method may be called from another machine, therefore Jcc generates marshal and unmarshal code for it.

```
remote class RemoteMonkey {
    int i;
    String s;

    synchronized int foo(int i, String s) {
        this.i = i;
        this.s = s;

        System.out.println("i = " + i);

        return i*i;
    }
}
```

Figure 22: A simple remote class.

The generated marshaller for the “foo()” method is shown in Figure 23 in pseudo code. Because “foo()” has a String as parameter, which is an object in Java, a hash-table is created to detect duplicates. Note that the *create thread* flag in the call-header is set. This is done because “foo” contains a method call (“System.out.println()”) and might therefore potentially do a “wait()”, or block on a synchronization statement. When a remote exception is thrown, a reference table must be created, to detect duplicates in the exception object. The programmer may define his own exceptions in Java, so it is not guaranteed that the thrown exception does not contain a cycle. The *writeObject* call will serialize the string object to the buffer at the current position.

Pseudo code for the generated unmarshaller is shown in Figure 24. The call-header is already unpacked when this unmarshaller is called. Because the *create thread* flag in the call-header was set, this unmarshaller will run in a new thread started by the runtime system. The marshaller itself does not know about this. Note that the *remote reference* field is a valid reference for the machine the unmarshaller will run on. This remote reference will be passed to

```
marshal_foo(int i, class_String* s) {

    alloc_buffers(&inBuffer, &outBuffer);
    hashTable = create_hashTable();

    writeCallHeader(&outBuffer, OPCODE_CALL, CREATE_THREAD, flush_function);
    writeInt(&outBuffer, i, flush_function);
    writeObject(&outBuffer, s, flush_function, hashTable);

    flush(); // Send request.

    opcode = readInt(&inBuffer, fill_function);

    if (opcode == OPCODE_EXCEPTION) {

        referenceTable = create_referenceTable();

        exception = read_object(&inBuffer, fill_function, referenceTable);

        free_buffers();
        kill_hashTable(hashTable);
        kill_referenceTable(referenceTable);

        THROW_EXCEPTION(exception);

    } else {

        result = readInt(&inBuffer, fill_function);

        free_buffers(inBuffer, outBuffer);
        kill_hashTable(hashTable);

        RETURN(result);
    }
}
```

Figure 23: The generated marshaller for the "foo" method.

the unmarshaller as a parameter. When the Java "foo()" method is called, the reference parameter is used as *"this"* pointer.

```
unmarshal_foo(javaObject *this) {
    alloc_buffers(&inBuffer, &outBuffer);
    referenceTable = create_referenceTable();

    i = readInt(&inBuffer, fill_function);
    s = readObject(&inBuffer, fill_function, referenceTable);

    result = CALL_JAVA_FUNCTION(foo, this, i, s, &exception);

    if(exception) {

        hashTable = create_hashTable();

        writeInt(&outBuffer, OPCODE_EXCEPTION, flush_function);
        write_object(&outBuffer, exception, flush_function, hashTable);

        flush(); // Message is created, now write it to the network.

        free_buffers(inBuffer, outBuffer);
        kill_hashTable(hashTable);
        kill_referenceTable(referenceTable);

    } else {

        writeInt(&outBuffer, OPCODE_RESULT_CALL, flush_function);
        writeInt(&outBuffer, result, flush_function);
        flush(); // Message is created, now write it to the network.

        free_buffers(inBuffer, outBuffer);
        kill_referenceTable(hashTable);
    }
}
```

Figure 24: The generated unmarshaller for the "foo" method.

5 Threads and locks

Java uses an object-oriented version of threads. A *Thread* object is used as a wrapper for the native thread implementation. The *start* method of a Thread object creates a new native thread and starts executing the *run* method of the Thread. There are two ways for the programmer to create a new thread. One is to declare a class to be a subclass of Thread, overriding the *run* method. When the thread is started the new run method will be called. The second way is for a class to implement the *Runnable* interface, forcing it to contain the *run* method. To start the thread, a new Thread object must be created, passing the Runnable object as a parameter. When the thread is started the run method of the Runnable object will be called.

In Java, every object can be used as a lock in a *synchronized* statement, or as a monitor using *synchronized* methods. Java locks can be used recursively, enabling the same thread to lock an object a number of times (see Figure 25). If a thread tries to lock an object which is already locked by a different thread, it will block until the lock is released.

```
class LockExample {

    String s = "Lock me";

    void lock() {
        // s is now unlocked.

        synchronized(s) {
            // s is now locked once.

            synchronized(s) {
                // s is now locked twice.
            }

            // s is now locked once.
        }
        // s is now unlocked.
    }
}
```

Figure 25: Using an object as a recursive lock.

If synchronized methods are used, an object will behave like a monitor (see Figure 26). The object is locked when a thread enters a synchronized method, and unlocked when the method returns. This will ensure that at any given time, only one thread is executing a synchronized method. Because the locks are recursive, a synchronized method can call another synchronized method in the same object without causing a deadlock. A thread can block inside a monitor (e.g., to wait for a condition), using the *wait* method. By calling this method, the thread will be blocked and the object unlocked, allowing other threads to call synchronized methods of the object. The *notify* method can be used to wake up a single waiting thread in this object. The *notifyAll* method will wake up all waiting threads. When a thread is notified, it locks the object (blocking if necessary) and continues execution of the synchronized method. Note that the wait, notify and notifyAll methods can only be used inside a synchronized method or synchronized statement.

```
class MonitorExample {

    synchronized void sleep() {

        try {
            wait();
            // The calling thread will now block until it is notified.
        } catch (Exception e) {
            // Don't care.
        }
    }

    synchronized void wakeup() {

        notify();
        // If there is a blocked thread, it will now continue.
    }
}
```

Figure 26: Using an object as a monitor.

5.1 The implementation

Jcc's threads are implemented using Panda threads, which are built on a *user level* thread package called OpenThreads [12]. OpenThreads delivers basic functionality, like thread creation and destruction, and thread priorities.

When the *start* method of a Thread object is called, the runtime system creates a new Panda thread. This Panda thread calls the *run* method of the Thread object, thus starting the Thread or *Runnable* object. Besides starting the thread, the runtime system also records information about the thread, including the Thread object, the Panda thread, and the location of the stack of the Panda Thread. This information is used by the garbage collector (see Section 6.3.2). When the thread exits, this information is removed.

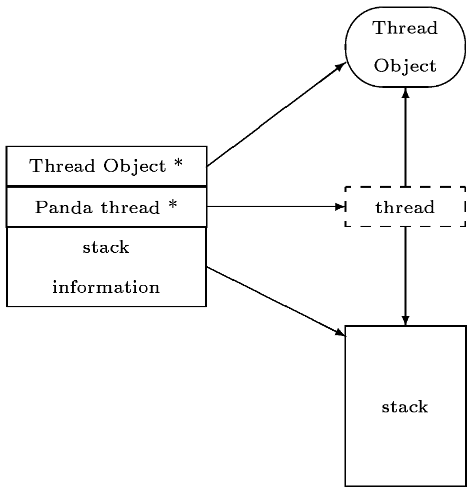


Figure 27: Thread information.

The implementation of the Java lock uses the locks and condition variables Panda provides. A Panda lock is a *mutex*, which can be locked and unlocked atomically. If a thread attempts to obtain a lock which is already taken, it is blocked and inserted into a list of threads waiting for the lock. When the lock is released, a waiting thread is removed from the list and allowed to continue. Using a Panda condition variable, a thread can *wait* for a longer period of time. Other threads can *signal* the condition variable to wake up a single waiting thread, or *broadcast* the condition variable to wake up all waiting threads. Each condition variable is associated with a Panda lock, and the combination acts like a monitor. Before an operation can be performed on a condition variable, the associated lock must be taken. If the operation blocks, the lock will automatically be released.

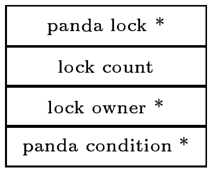


Figure 28: The Java lock.

In addition to a Panda lock and condition variable, the Java lock contains a thread identification to identify the owner of the lock, and a counter to record the number of times the lock is acquired recursively.

When a thread tries to lock an object it checks the *owner* field of the Java lock to see if it already owns the lock. If so, it is sufficient to increase the lock counter. If it does not own the lock, it tries to acquire the Panda lock, blocking if it is already taken, and sets the counter to one. When unlocking, the counter is decreased. If the counter reaches zero, the Panda lock is released and the owner field cleared. Otherwise the Panda lock remains locked.

The *wait*, *notify* and *notifyAll* methods can be implemented directly using the Panda wait, signal and broadcast calls. A wait saves the lock counter and calls the Panda wait on the condition variable in the lock, causing the Panda lock to be released automatically. When the waiting thread is notified (by using a Panda signal or broadcast), the lock is acquired and the counter restored.

6 Garbage collection

Garbage collection is an essential part of a Java runtime system. The Java language does not offer any way of explicitly removing objects. It is the task of the garbage collector to detect which objects are no longer in use, and free the memory occupied by these objects.

Before the memory used by a Java object can be freed, the garbage collector must call the *finalizer* of the object. A *finalizer* is a special method *finalize()* enabling an object to free external resources before it is removed, for example, close a file, remove a window or flush a cache. Although every object in Java has a finalizer, most objects directly inherit the empty finalizer from the Object class. See [20] for a complete description of Java finalizers.

The overhead of the garbage collector can have a major impact on the performance of a program. When very large numbers of objects are created, a significant part of the execution time of the program is used by the garbage collector. Some of this overhead can be reduced by collecting information about objects at compile time. Section 6.3.1 describes the compiler support delivered by *Jcc* to the garbage collector.

In addition to normal Java objects the garbage collector also has to deal with *object stubs*. Object stubs are structures which, viewed by the program, look and behave like normal objects, but don't actually contain the data or methods the real object contains. Object stubs are used to implement *remote objects* (see Section 3.2) and could be used to implement other features, like *persistent* objects.

6.1 Basic garbage collection techniques

Over the years a wide variety of garbage collection techniques have emerged. See [25] for an overview of garbage collection techniques for procedural and object-oriented languages.

An object is considered *garbage* if it is no longer reachable by the running program via any path of reference traversal. *Live* objects are potentially reachable by the running program and are preserved by the collector. A garbage collector consists of two parts:

1. *Garbage detection*, which distinguishes live objects from garbage.

2. *Garbage reclamation*, which removes unused objects.

Many variations of garbage collection are possible, given this basic two-part operation. Two possible ways two implement garbage detection are *reference counting* and *tracing*.

In a reference counting system, each object has an associated count of the references to it. Each time a reference is added (e.g., copied by an assignment) the counter is incremented. When a reference is destroyed the count is decremented. If an object's count reaches zero, the object has become unreachable, and can be removed. In a tracing system the garbage collector periodically traverses the references the program could traverse to determine which objects can be reached. Objects not reached in this traversal are considered garbage and are removed from memory.

A problem with reference counting is *efficiency*. Each operation on a reference introduces overhead. Assignment of a reference implies that one object's count must be decreased while another object's count must be increased. When a reference is passed as a parameter to a function the count must be increased, only to be decreased again when the function returns. Figure 29 shows the overhead of the assignment of two references, $A = B$.

```
if (A != null) {
        decrease count of object referenced by A
}

if (B != null) {
        increase count of object referenced by B
}

A = B
```

Figure 29: Assignment overhead in reference counting.

Cycles are another problem with reference counting. Each object in a cycle will always be referenced by at least one other object in the cycle. The reference count of the objects in a cycle will never return to zero, even if none of the objects is reachable from the program. The memory occupied by these objects can never be reclaimed by simple reference counting. See [25] for an overview of solutions to this problem.

A problem with tracing garbage collectors is the identification of references in memory. The garbage collector must extract each reference to an object from memory, to be able to mark the object as 'live.' One approach is *conservative* garbage collection [7]. In a conservative garbage collector, all of the used memory is scanned. Each word in memory is tested to see if it could be a reference. If a possible reference is found the referenced block in memory is marked and considered 'live.' Note that if the word was actually a non-reference value, it it possible for garbage to be considered 'live', thus decreasing the efficiency of the garbage collector. Although conservative garbage collection is easy to implement and can be added to a system with little effort, the runtime overhead is large. Another approach is to add compiler support to mark each reference in memory, enabling the garbage collector to find each reference with little effort, and preventing a non-reference value to be mistakenly interpreted as a reference.

Another problem with tracing garbage collectors is that the cost of garbage collection is proportional to the amount of memory used. All live objects must be identified and all garbage objects collected. If the number of objects in memory is very large a collection pass could take considerable time. *Incremental* garbage collectors attempt to solve this problem by interleaving the tracing with the running program. After tracing a number of times, all live objects have been marked and the garbage can be removed. The difficulty with incremental tracing is that while the garbage collector is tracing out the graph of reachable objects, the running program may mutate the graph. The change can cause the garbage collector to 'skip' a part of the graph. These 'skipped' objects will be considered garbage and removed (see Figure 30).

The incremental garbage collector must have some way of keeping track of the changes made in the graph of objects between traces. See [25] for an overview of algorithms used to solve this problem.

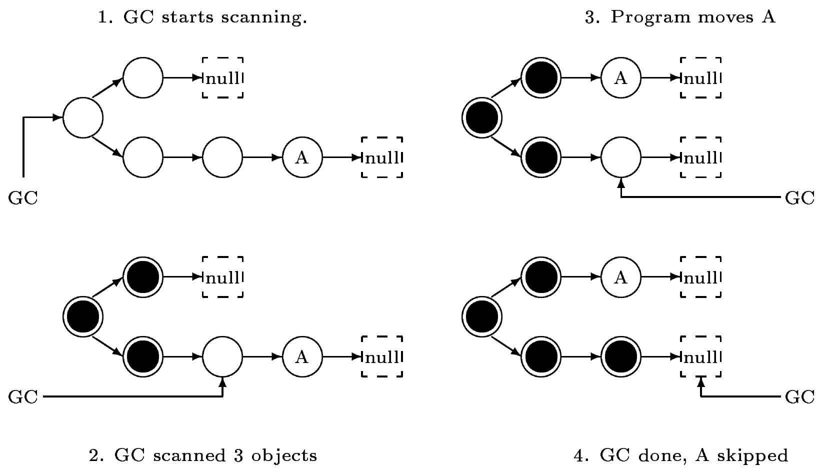


Figure 30: Graph mutation during garbage collection.

6.2 Algorithm

The garbage collection algorithm we use is based on the *mark-sweep* algorithm. The mark-sweep algorithm is a tracing garbage collection algorithm which traces all garbage in a single pass. The algorithm is named for the two phases that are used to collect the garbage. In the *mark* phase, the object graph is traced, marking all the live objects found. In the *sweep* phase, all unused objects are *swept* from memory. We chose this algorithm because it is easy to implement and, with sufficient compiler support, has adequate performance.

6.2.1 Garbage detection

The live objects are the objects that are reachable by the running program. To find these objects, the *root set* must be determined first (see Figure 31). The *root set* is the set of objects that are considered live at the start of the garbage collection. This set includes all objects that do not have to be referenced to stay alive, such as static objects, running threads, and objects referenced from the stack by method variables and parameters. The root set can be used to find all reachable objects.

```
root set = empty
for each object X {
        if X is a thread {
                add X to root set
                for each object Y referenced from the stack of X {
                        add Y to root set
                }
        }
        if X is static {
                add X to root set
        }
}
```

Figure 31: Creating the root object set.

Each object in the root set is a reachable object. In addition, each object referenced from a reachable object is also reachable. The set of live objects can now simply be found by traversing every path of references from the root set. Any object not reachable from the root set is garbage, because there is no way the running program can reach that object. The space used by these objects can now be reclaimed safely (see Figure 32).

```
alive set = empty
for each object X in root set {
        add X to alive set
        for each object Y referenced from X {
               add Y to root set
        }
}
```

Figure 32: Creating the live object set.

6.2.2 Garbage reclamation

After the garbage detection phase, we are left with a set of objects no longer used by the program. For each of these objects, the garbage collector must check if the object can safely be removed.

If an object has not been finalized yet, the garbage collector calls the object's finalizer and returns the object to the used object set. If the object has already been finalized it is removed and the memory is reclaimed (see Figure 33).

```
dead set = (all objects not in live set)
for each object X in dead set {
        if X is finalized {
                remove X from memory
        } else {
                finalize X
                add X to alive set
        }
}
```

Figure 33: Collecting the unused objects.

Note that the object is not immediately removed after calling the finalizer. It is possible for an object to be reachable again after finalization [20]. In the *finalize* method, the object could have been assigned to some global reference or passed as a parameter to some global object, returning it to the set of objects reachable by the running program. Only if an object is finalized *and* unreachable can it safely be removed. The *finalize* method will only be called once for each object. It is the responsibility of the programmer to prevent irregular behavior when a finalized object returns to the set of live objects. See [20] for the complete description of Java finalizers.

6.3 Implementation

6.3.1 Compiler support

To reduce garbage collection overhead, information about objects is recorded at compile time. To aid the garbage collector in determining the root set, the location of global object references in the program is determined by the compiler. The compiler produces an array containing pointers to the locations of the static object references. All static objects can now be found at runtime by simply scanning the entire array and checking each reference for an object (i.e., not null). An example is given in Figure 34.

```
class Monkey {
        static Monkey m;
        static String s = "Banana";
}
```

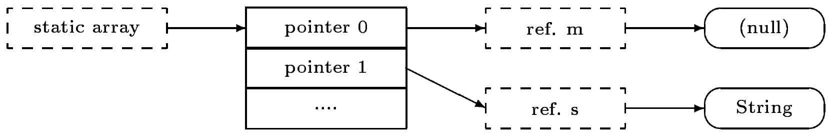


Figure 34: The static reference table.

When a class is compiled, the compiler checks several properties of the class which have an effect on the way objects of this class are treated by the garbage collector. The following properties are recorded in a special flag field in the object :

- Does the object contain any references ?
- Does the object contain a trivial (empty) finalizer ?
- Is the object an array ?

These properties can be used by the garbage collector to optimize the collection. An object without any references does not have to be scanned while determining the set of live objects, since no other object can be reached from it. An object with a trivial finalizer can be removed when it becomes unreachable, without calling the finalizer first. Because the *finalize* method is empty it does not have any effect on the state of the program or the reachability of the object. If an object is an array it will contain a block of data of a single type. If this type is an object type, the compiler will register that the array contains references. Only arrays containing references must be scanned by the garbage collector.

If an object contains references, the location of each reference within the object is recorded in a *reference offset table* for the class of this object. The offset table is saved in the vtable of the object and therefore one table is shared by all objects of the same type (see Section 3.2). When the live object set is determined, the garbage collector knows the location of each reference within each object. It is therefore not necessary to conservatively scan the entire object for possible references (see Figure 35).

6.3.2 Detecting the garbage

To separate the garbage from the live objects the root set must first be determined. In addition to the global object references retrieved from the *static reference table*, running threads and local reference variables are also part of the root set. The garbage collector uses the thread information saved when the threads are started (see Section 5). The thread information contains the location of the thread's stack, and the location of its current stack pointer. This

```
class Monkey {

        int value1;
        Monkey m;
        int value2;
        String s = "Banana";
        int value3;
}
```

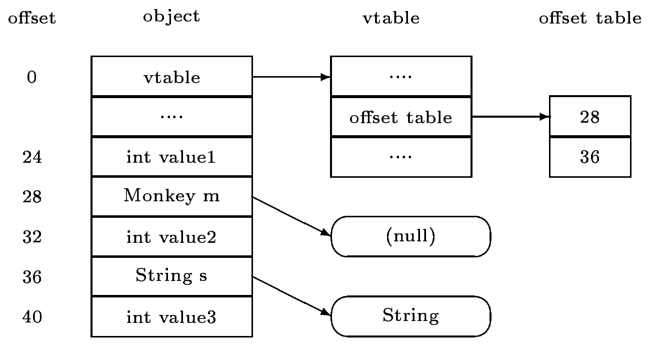


Figure 35: The reference offset table.

allows the garbage collector to determine which part of the thread's stack is in use (see Figure 36). Since the garbage collector runs in a separate thread (see Section 6.3.4), the current stack pointer of all other threads will be saved in memory and are thus up-to-date.

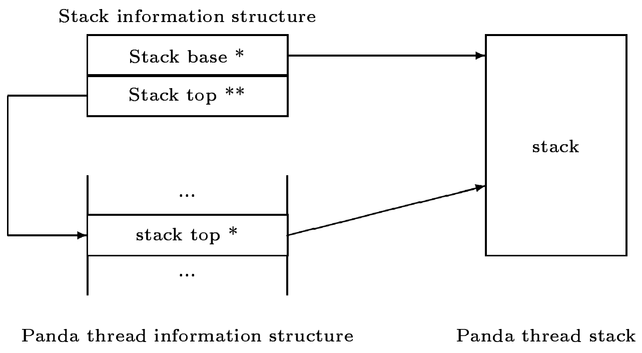


Figure 36: Stack information.

To find the objects referenced by local reference variables, all threads' stacks are scanned. Each word on a stack is checked to see if it contains an object reference. Because all objects are recorded in a hash table the cost of scanning the stacks is reduced.

Note that the stack is scanned *conservatively*, meaning that each word on the stack which could be an object reference is treated as such. It is possible for a normal integer value to mistakenly be interpreted as an object reference. This will affect the efficiency of the garbage collector. Some objects will be added to the root set even if they are not reachable by the running program. The correctness is not affected however. The size of stacks in Java programs is usually quite small, because arrays and objects are allocated on the heap. The extra time involved in conservatively scanning the stack is therefore limited.

A *root list* can now be created containing the root set by adding all static objects and thread objects to the list and scanning the stack of each thread. Each object inserted into the root list is *marked* by setting a few bits in its *flag field* (see Section 3.2). By checking these bits before inserting the object, double insertions are prevented. Each object inserted into the root list is removed from the object hash-table.

After the *root list* is created, the live set of objects is determined by scanning all references in the objects in the root set. Because each object contains a *reference offset table* (see Section 6.3.1) the object does not have to be scanned conservatively. Each scanned object is removed from the root list and added to a *live object hash-table.* Each referenced unmarked object is marked. If the referenced object contains any references itself it must be scanned and is added to the root list. Otherwise it can be added to the *live object hash-table* immediately. This process continues until the root list is empty and each reachable object has been scanned. The *live object hash-table* now contains all reachable objects. All object remaining in the *object hash-table* are garbage and can be collected.

6.3.3 Removing the garbage

Removing the garbage now consists of traversing the *object hash-table* to find the remaining objects. If an object is already finalized or has a trivial finalizer (an empty finalizer directly inherited from Object) it can be removed from memory. Otherwise the object's finalizer is called and the object is added to the *live object hash-table.* If it remains unreachable it will be removed on the next garbage collection pass.

Object stubs will be removed after calling a special *stub finalizer.* A stub finalizer is a function which is attached to the stub when it is created. The stub finalizer serves the same purpose as a normal Java finalizer; it enables the stub to free external resources. A stub for a remote object, for instance, has a stub finalizer which notifies the real remote object that the stub is being deleted (see Section 7).

When a garbage object is removed, the memory used is not released immediately. The garbage object is inserted into an *object cache.* This object cache contains unused objects, sorted by size. When a new object is created, the cache is checked for an unused object of the appropriate size. If such an object is found, it is reused by initializing its data to the proper values. This way the cost of actual memory allocation is saved. When the system runs out of memory the object cache is cleared, enabling the underlying memory allocation system to merge the unused objects back into the heap (See Section 8 for benchmarks of the object cache).

6.3.4 Interaction with the user program

The garbage collector runs in a separate thread. This thread is started when the runtime system is initialized at the start of the program's execution, and sleeps until it is signaled by the program. The program can either directly start the garbage collector by calling *System.gc()* or indirectly by creating a new object. When a new object is created the *new* function tests if the amount of memory used has passed a arbitrarily chosen threshold. If so, the garbage collection thread is signaled and the currently running thread of the program waits for the garbage collection thread to finish. At the end of garbage collection, a new threshold will be calculated, and the suspended thread is allowed to continue.

Because the garbage collector is running in a multi-threaded environment, the mark phase of the garbage collection must be *atomic*. Otherwise, while the garbage collector is tracing the graph of reachable objects, a thread of the running program may mutate the graph. This problem is similar to the problems with *incremental* garbage collection. To prevent this problem, no other thread is allowed to run during garbage detection, to insure that no object references are changed. Once all live objects have been identified, the program threads are allowed to run again. The garbage reclamation does not have to be atomic, because only the garbage collector can reach the garbage objects.

7 Distributed garbage collection

In a distributed system, local garbage collection is not enough, since remote references cross machine boundaries. When an object is only referenced by remote references from other machines, the local garbage collector may conclude that the object is no longer reachable, and remove it from memory. It is the task of the *distributed* garbage collector to prevent the local garbage collector from removing remote objects while they are still in use on other machines.

7.1 Algorithm

The distributed garbage collector uses *reference counting* to record the 'liveness' of the remote objects. Reference counting is one of the easiest distributed garbage collection algorithms to implement. Note that the problem with cycles encountered in local reference counting is also present in distributed reference counting. The distributed garbage collection algorithm described here is not able to detect unused distributed cycles. See [9] and [13] for more advanced implementations of distributed garbage collection.

When a remote object is created its reference count is initialized to one. If the reference to the remote object is sent to another machine, the remote object is notified by sending it a message, and its reference count increased. If an object containing a remote reference is removed, the remote object is notified again and its reference count decreased. When the reference count of an object has reached zero, the object is released by the distributed garbage collector, and will be removed from memory by the local garbage collector on its next pass.

A problem can arise when a reference count to an object reaches zero while a reference to the object is still contained in a message somewhere. When this *reference in transit* reaches its destination, the object it references is no longer valid (see Figure 37).

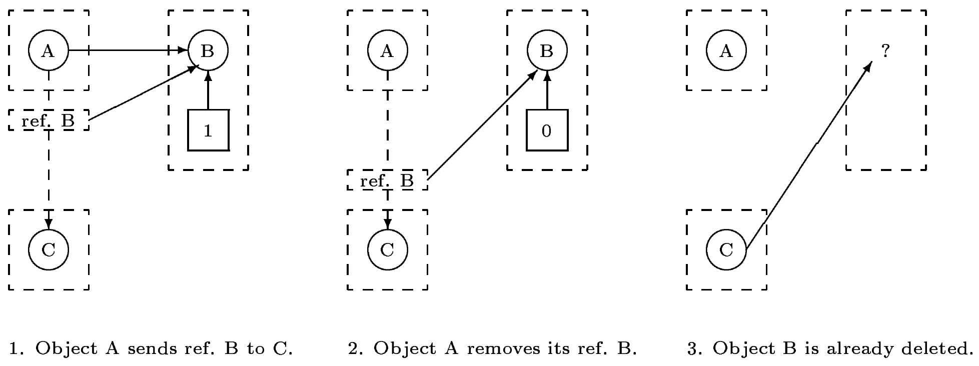
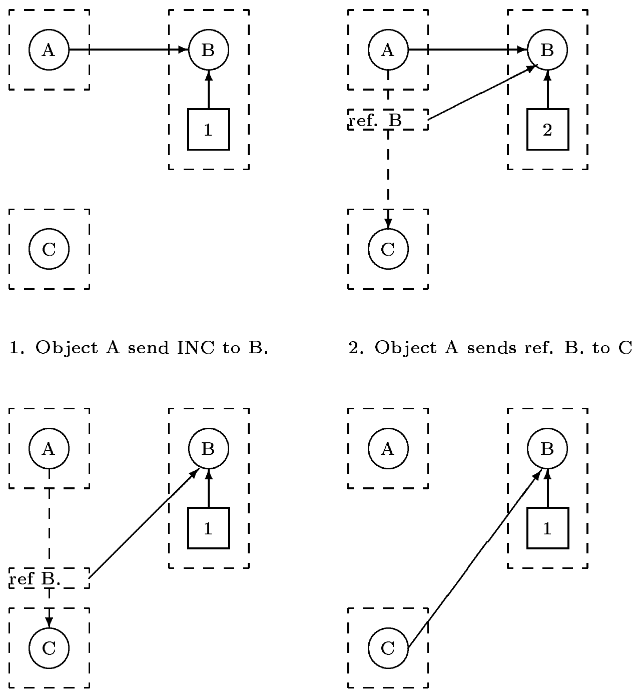

A
B
ref. B
1
C
1. Object A sends ref. B to C.
A
B
ref. B
0
C
2. Object A removes its ref. B.
A
?
C
3. Object B is already deleted.

Figure 37: A reference in transit.

To ensure this problem will not occur, the reference count of an object must include all references in transit. This can easily be achieved by increasing the reference count of a remote object before the remote reference is sent to another machine (see Figure 38). A message is sent to the owner of the remote object to increase the reference count. When the reference count has been increased, the owner acknowledges the message. The remote reference can now be sent to the other machine. Note that this solution has a negative effect on the performance, because an extra message is required for every remote reference sent to another machine.



1. Object A send INC to B.

2. Object A sends ref. B. to C

3. Object A removes its ref. B.

4. C now contains the ref. B.

Figure 38: A reference in transit with increased count.

7.2 Implementation

The distributed garbage collector is a combination of native and Java code. The native code is used to implement remote references and the communication between the distributed garbage collectors. The Java code implements the administrative part of the distributed garbage collection.

7.2.1 Java layer

The distributed garbage collector maintains a *static* hash table, in which it stores information about the remote objects located on this machine. For each of the remote objects a *RemoteObjectCounter* object is created and inserted into the hash table. The RemoteObjectCounter maintains the reference count for the remote object and a reference to the remote object. Because the hash table containing the RemoteObjectCounter is static, it will be included in the root set of the local garbage collector. The hash table and all objects reachable by it will therefore not be considered garbage.

The distributed garbage collector has a number of methods called by the native layer when a message is received, as shown in Table 2.

method	function
addRemoteObject(Object o)	Add remote object o to the hash table.
addRemoteUser(Object o)	Increase the reference count for remote object o.
removeRemoteUser(Object o)	Decrease the reference count for remote object o.

Table 2: Methods in the distributed garbage collection class.

When the reference count in a RemoteObjectCounter reaches zero it is removed from the hash table. Because the remote object is no longer reachable from the static hash table, it will be removed by the local garbage collector on its next pass. See appendix A for the source code of the Java layer of the distributed garbage collector.

7.2.2 Native layer

The native layer uses object *stubs* (see Section 3.2) to represent remote objects on other machines. A remote object stub contains the address of the remote object. A remote object address consists of a CPU number and the memory address of the object on the owner machine (see Figure 39).

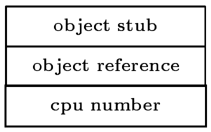


Figure 39: The remote object stub.

The stub finalizer added to the remote object stub will be called by the local garbage collector when the object stub is removed. The remote object stub will be passed as a parameter. The finalizer uses the remote object address in the stub to send a *remove* message to the owner CPU (see Figure 40).

```
void remote_stub_finalizer(stub s)
{
        create dgc message

        message->opcode = remove;
        message->src    = this CPU
        message->data   = s->object;

        send message to s->cpu;
}
```

Figure 40: The remote object stub finalizer.

When a remote object stub is marshalled to be sent to another machine, a similar function is called to send an *add* message to the owner CPU. The owner CPU will send an acknowledgement when the reference count of the object is increased, after which the remote object stub can safely be sent to another machine.

When a CPU receives a message for the distributed garbage collector, it decodes the message and calls the appropriate Java method. The Java layer of the distributed garbage collector can then update the information on the remote object (see Table 3).

opcode	Java method to call	meaning
add	addRemoteUser()	Increase reference count.
remove	removeRemoteUser()	Decrease reference count.
collect	System.gc()	Request to start local garbage collection.

Table 3: Possible opcodes in distributed garbage collection messages.

The *collect* message is used to request the local garbage collector to start on a specific CPU. This can be used to force all machines to start the local garbage collector at the same time, which may improve the efficiency of some distributed programs.

8 Results

8.1 Experimental setup

The system we have tested on has been designed by the Advanced School for Computing and Imaging (ASCI).[1] It is called DAS, for Distributed ASCI Supercomputer. The main goal of the DAS project is to support research on wide-area parallel and distributed computing. (Additional information on the system can be found at *http://www.cs.vu.nl/~bal/das.html*).

The platform is a homogeneous cluster of 128 Pentium Pro processor boards, running the BSD/OS (Version 3.0) operating system from BSDI. Each node contains a 200 MHz Pentium Pro, 128 MByte of EDO-RAM and a 2.5 GByte IDE disk. The Pentium Pro processors have an 8 KByte, 4-way set-associative L1 instruction cache and an 8 KByte 2-way set-associative data cache. The L2 cache is 256 KBytes large and is organized as a 4-way set-associative, unified instruction/data cache.

All boards are connected by two different networks: Myrinet [6] and Fast Ethernet (100 Mbit/sec Ethernet). We have used both the Myrinet and Fast Ethernet network for all Jcc performance tests, and only the Fast Ethernet for the other Java implementations, including JavaParty.

Myrinet is a 1.28 Gbit/sec network. It uses the SAN (System Area Network) technology, consisting of LANai-4.1 interfaces (with 1 MByte SRAM memory) connected by 8-port Myrinet switches. The Myrinet switches are connected using a 2-dimensional torus topology (i.e., a grid with "wrap-around"). The Fast Ethernet network uses SMC EtherPower 10/100 adaptors, KTI Networks KF1016TX hubs, and a single NuSwitch FE-600 switch. The Fast Ethernet network uses segments with 12 or 14 processors each, connected by the switch. The switch has a delay of about 12 microseconds for small packets. The entire system is packaged in a single cabinet and was built by Parsytec (Germany).

Panda [4] on DAS uses Myrinet for communication, using the LFC [5] message passing layer. LFC provides packet unicast, packet broadcast, fetch-and-add, interrupt management and a microsecond timer. Panda is a portable system for parallel and distributed programming. It provides threads, messages, Remote Procedure Call, and group communication. The Panda RPC interface has a latency of 30 microseconds, and a throughput of up to 50 MByte/s with empty reply messages, both on the Myrinet network.

On the DAS we tested the following systems:

- Java sockets on Fast Ethernet
- Java RMI on Fast Ethernet
- JavaParty on Fast Ethernet
- Jcc on Fast Ethernet and Myrinet

For Java sockets, Java RMI and JavaParty the SUN Java interpreter was used (JDK version 1.1.5). JavaParty version 0.97 (August 1997) was used. Unfortunately, we do not have a full Myrinet port of Java Sockets, RMI, and

[1]The ASCI research school is unrelated to, and came into existence before, the Accelerated Strategic Computing Initiative.

JavaParty yet, so the measurements with these systems had to be done using Fast Ethernet and TCP/IP. We have also tried the Kaffe JIT compiler version 0.92, but this version did not support RMI.

8.2 Low level benchmarks

8.2.1 Low level RMI benchmarks

In order to test the performance of the remote method invocation implementation, some low level benchmarks were done. The first benchmark is a Java program which measures the throughput for a remote method invocation with an array of a primitive type as argument, and no return type. The reply message is empty, so the one-way throughput is measured. In Jcc, all arrays of primitive types are serialized with a memcpy(), so the actual type does not matter. The resulting measurements are shown in Figure 41, the actual data in Appendix B.1.

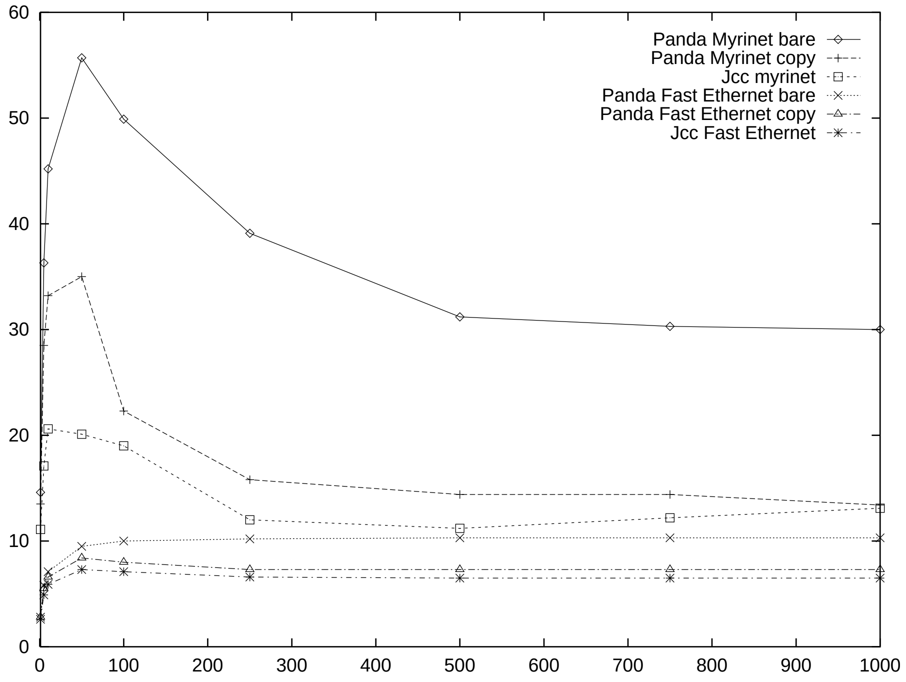


Figure 41: Network throughput in Mbytes/s.

The highest line in the figure the measured throughput of the Panda RPC interface, with the same message size as the remote method invocation. Second comes the throughput of the Panda system, with memory-to-memory copies on both the sending and receiving side. This simulates the Jcc RMI implementation, where, on the sending site, the array has to be copied from the Java array object to the serialization buffer, and on the receiving side from the deserialization buffer into the newly created Java array.

The measurements show that the memory-to-memory copies have a large impact on the performance on the Myrinet network. For array sizes of a hundred kilobytes or more, the throughput is more than halved by the two memory copies on the Myrinet network. On the slower Fast Ethernet network, the impact on performance is much less. The measurements also show that the throughput of Jcc at the Java program level is not much lower than the Panda throughput with two buffer copies. The throughput difference can be explained by the overhead of packing and unpacking the call headers, and the creation of the Java array object.

For the second benchmark, we have measured the time an empty remote method invocation without any arguments takes. Also examined were calls with one to three empty Java objects as parameter. The results are shown in Table 4.

Item / parameters	none	1 object	2 objects	3 objects
Buffers on CPU 0	1.5	1.5	1.8	1.7
Buffers on CPU 1	1.1	1.1	1.3	1.4
Marshalling on CPU 0	0.8	0.9	1.4	1.9
Unmarshalling on CPU 1	1.8	4.2	5.7	6.9
Class descriptor search	—	1.5	2.9	4.4
Object creation	—	1.6	3.0	4.4
Panda RPC	30.0	30.0	30.0	30.0
Total	35.2	40.7	46.1	50.7

Table 4: Jcc remote call performance in microseconds.

All measurements were done by inserting timing calls, using the Pentium Pro performance counters. These counters have a granularity of 5 nanoseconds. An empty remote method invocation takes about 35 microseconds when using Jcc over Myrinet. Only 5 microseconds are added to the latency of the Panda RPC, which is 30 microseconds. When passing primitive data types as a parameter to a remote call, the latency grows with less than a microsecond per parameter, regardless of the type of the passed parameter.

When passing an object as a parameter to a remote call, the latency does increase considerably, as now some tables must be created by the runtime system in order to detect possible cycles in the passed object. The measurements show that almost all time that is taken by adding an object parameter is spent at the remote side of the call, deserializing the call request. The marshalling of the request on the calling side, however, is affected much less by the object parameters.

The Empty remote call time for Jcc over Fast Ethernet is 230.2 microseconds, while Java RMI takes 1827 microseconds, so Jcc RMI is about 8 times faster. On Myrinet, an empty Java RMI takes 1228 microseconds, while Jcc takes 35, Jcc RMI is thus a factor 35 faster. The relative Java RMI serialization overhead becomes more significant when the latency of the network decreases. For distributed programming, the bad performace of the remote method invocation is not really a problem, but for paralell programming, fast communication is critical.

Thread creation overhead is 16.0 microseconds with the Jcc runtime, so not starting a thread for every call does help a lot. The low unmarshalling time for empty call is due to not creating the tables for duplicate detection. Measurements show that this saves about 2 microseconds for an invocation with primitive types as parameters, so it is clear that this optimization helps.

The *class descriptor search* will find the meta class data for the class of the object parameter, given the unique class identifier. The meta class contains the information which is constant for the class. Amongst other fields, a pointer to the virtual method table of the object is contained in the meta class. This pointer is needed for object creation, as it must be stored in the created object header.

8.2.2 Garbage collection

We have used the following programs to test the performance of the garbage collection, object and array creation and the object cache.

- *GCTest1* measures the time required to create 100,000 objects without saving their references.
- *GCTest2* measures the time required to create a linked list of 100,000 objects.
- *GCTest3* creates a linked list of 100,000 objects and measures the time required to collect these objects.
- *GCTest4* creates a linked list of 10,000 objects. The reference to the list is released and the garbage collector called. This is repeated 100 times, and the total time require is measured.
- *UCSD* is the Garbage collection test of the Java Benchmark created at the University of California [11]. This test creates 200,000 objects, each containing an array of a semi-random size between 1 and 4000 bytes.
- *IDA* is the Iterated Deepening A* application described in Section 8.3.3.

The garbage collection threshold used by our garbage collector in these programs is 4 megabytes. The maximum amount of memory the program was allowed to use was 32 megabytes. All test except IDA use a finalizer in the test object created to count the number of objects removed.

	GCTest1	GCTest2	GCTest3	GCTest4	UCSD	IDA
Java Interpreter						
Time (sec)	0.75	34.70	0.74	51.49	25.17	274.43
Objects removed	97816	0	100000	9992004	198247	-
Kaffe JIT						
Time (sec)	0.30	0.31	0.71	212.23	-	-
Objects removed	0 ?	0 ?	0 ?	0 ?	-	-
Jcc compiler						
Time (sec)	0.38	0.52	0.10	4.50	8.49	79.02
Objects removed	0	0	100000	990000	198821	-
Jcc compiler (no cache)						
Time (sec)	0.38	0.50	0.10	4.04	8.57	100.52
Objects removed	0	0	100000	1000000	199071	-

Table 5: Test results.

Table 5 shows the measured time and objects removed is all the programs. Note that Kaffe does not seem to call the finalizers of the objects and fails to run the UCSD and IDA tests (it threw a "OutOfMemoryError" exception). The

Java interpreter was the only one able to reuse the objects in GCTest1. Our garbage collector was not invoked, because the garbage collection threshold was not passed.

	GCTest1	GCTest2	GCTest3	GCTest4	UCSD	IDA
GC Passes	0	1	3	100	120	399
Objects/arrays :						
Used	100,042	100,042	100,108	1,000,042	400,222	26,382,169
Created	100,042	100,041	100,107	20,041	5,255	72,146
Removed	0	1	100,001	990,001	395,972	26,362,461
Total time required: (sec)						
GC pass	-	0.12	0.28	1.91	0.81	20.18
Creating root set	-	0.00	0.00	0.01	0.01	0.14
Mark phase	-	0.12	0.11	0.01	0.02	2.49
Sweep phase	-	0.00	0.17	1.90	0.78	17.55
Average time required: (msec)						
GC pass	-	117.89	92.11	19.14	6.82	50.57
Creating root set	-	0.13	0.15	0.08	0.11	0.34
Mark phase	-	115.28	36.15	0.09	0.17	6.23
Sweep phase	-	2.47	55.79	18.97	6.54	44.00

Table 6: Garbage collector performance

Table 6 shows the performance of the garbage collector for each of the test programs. Note that in GCTest1 the garbage collector was never invoked. Only in the last three programs, GCTest4, UCSD and IDA, is the garbage collector invoked frequently. In GCTest4 and IDA this is caused by the large number of objects created, and in UCSD by the size of the objects created (arrays with an average size of 2431 bytes). In UCSD the number of objects in memory at garbage collection time is relatively small compared to GCTest4 of IDA. This is reflected in the time required for a garbage collection pass: 6.82 milliseconds in UCSD compared to 50.57 milliseconds in IDA and 19.14 in GCTest4. Most of this time is actually spent in the sweep phase, removing the objects, because the average number of objects removed in each sweep phase is very large: almost 10 thousand objects for GCTest4, and over 66 thousand objects for IDA. The time required to create the root set and mark the live objects is usually much smaller. Only when most objects in memory are 'alive', does the mark phase become expensive. This can clearly be seen in GCTest2 were the mark phase is actually the most expensive part of the garbage collection.

Table 7 shows the overhead of object and array creation, and the performance of the object cache. Because only GCTest4, UCSD and IDA use the garbage collector frequently, they are the only programs using the object cache. The cache hit ratio in these three programs is high, about 98%. Most of the objects are continually reused. The average time required to create a new object or array varies in each of the programs. When a new object or array is created, the entire object or array is zeroed by using *memset()*. The time required depends on its size. The arrays created in UCSD have an average size of 2431 bytes, while the arrays in IDA have an average size of only 16 bytes. This is clearly reflected in the time required to create and zero the array. When objects or arrays are retrieved from the object cache they must be zeroed again to remove the old data. This increases cost of retrieving an array from the cache. Nevertheless reusing objects and arrays from the cache is cheaper than creating new ones. Note that the single retrieval of an object from the cache in GCTest2 and GCTest3 is very expensive, this is probably due to caching effects. When

	GCTest1	GCTest2	GCTest3	GCTest4	UCSD	IDA
Objects/arrays :						
Used	100,042	100,042	100,108	1,000,042	400,222	26,382,169
Removed	0	1	100,107	990,001	395,972	26,362,461
Created object/arrays						
Total	100,042	100,042	100,107	20,041	5,255	72,146
Objects	100,028	100,027	100,063	20,027	3,472	10,338
Arrays	14	14	44	14	1,783	61,808
Average 'new' time (usec)						
Object (new)	3.70	3.68	3.69	3.60	4.78	6.29
Array (new)	21.13	23.18	8.29	22.92	94.30	6.27
Object (cached)	-	5.03	13.59	1.12	1.29	1.52
Array (cached)	-	-	-	-	32.88	1.38
Total 'new' time (sec)	0.37	0.37	0.37	14.48	6.96	37.27
Cache performance						
Hits	0	1	1	980,001	394,967	26,310,023
Misses	100,042	100,041	100,107	20,041	5,255	72,146
Hit rate (%)	0	0	0	98.00	98.69	99.73
Retrieved from cache						
Arrays	0	0	0	0	198,317	22,551,585
Objects	0	1	1	980,001	196,650	3,758,438

Table 7: Object creation performance.

the object cache is used infrequently, each access will cause a cache miss on the machines cache.

8.3 Applications

8.3.1 Successive overrelaxation

Successive Overrelaxation is an iterative algorithm for solving Laplace equations on a grid. The sequential algorithm works as follows. For all non-boundary points of the grid, SOR first calculates the average value of the four neighbors of the point:

$$av(r,c) = \frac{g[r+1,c+1] + g[r+1,c-1] + g[r-1,c+1] + g[r-1,c-1]}{4} \tag{1}$$

Then the new value of the point is determined using the following correction:

$$g[r,c] = g[r,c] + \omega(av(r,c) - g[r,c]) \tag{2}$$

ω is known as the *relaxation parameter* and a suitable value can be calculated in the following way:

$$\omega = \frac{2}{1 + \sqrt{1 - (\frac{\cos\frac{\pi}{totalcolumns} + \cos\frac{\pi}{totalrows}}{2})^2}} \tag{3}$$

The entire process terminates if during the last iteration no point in the grid has changed more than a certain quantity.

The parallel implementation of SOR is based on the Red/Black SOR algorithm as used in Orca [2]. The grid is treated as a checkerboard and each iteration is split into two phases, red and black. During the red phase only the red points of the grid are updated. Red points only have black neighbors and no black points are changed during the red phase. During the black phase, the black points are updated in a similar way. Using the Red/Black SOR algorithm the grid can be partitioned among the available processors, each processor receiving a number of adjacent rows. All processors can now update different points of the same color in parallel. Before a processor starts the update of a certain color, it exchanges the border points of the opposite color with its neighbor. After each iteration, the processors must decide if the calculation must continue. Each processor determines if it wants to continue and sends its vote to the first processor. The calculation is terminated only if all the processors agree to stop. This means a processor may continue the calculation after the termination condition is reached locally.

Figure 42 shows a parallel SOR using four processors. Each processor communicates with its neighbors using two shared buffer (*bin*) objects. A *bin* object can only contain one row of the grid. If a processor attempts to insert a row into a filled bin, it blocks until the bin is emptied by the other processor. After the bin is read, the blocked processor continues and inserts the new row into the bin. If a processor attempts to read from an empty bin, it blocks until the bin is filled by the other processor.

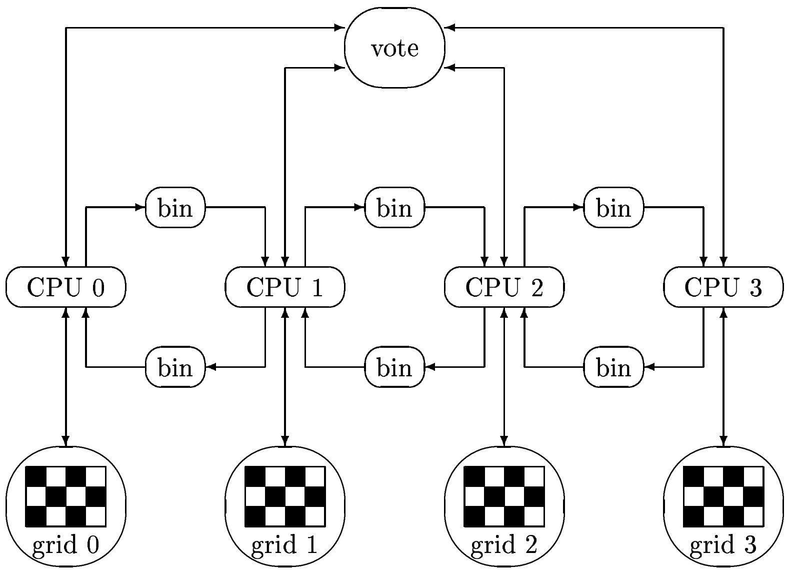


Figure 42: Parallel SOR.

The application was implemented using the four platforms (Java sockets, RMI, JavaParty and Jcc) and tested on the DAS processor pool, using grid sizes of 512x512, 1024x512 and 2048x512 on 1, 2, 4, 8, 12, 16, 24 and 32 nodes. Due to some problems with the runtime system, the JavaParty version did not run on 24 or 32 nodes.

Note that only the Jcc version was tested using both Fast Ethernet and Myrinet. The other platforms use Fast Ethernet for communication.

Tables of the measured values are found in Appendix B and show the total execution time, speedup, calculation time, and the percentage of the execution time spent in communication code, for the programs on the different platforms.

As can be expected, the programs compiled with the Jcc compiler are significantly faster than any of the interpreted programs. For example, the 2048x512

SOR on a single processor takes 1687.1 seconds using the Java sockets version, while the Jcc version only takes 390.0 seconds, approximately 4.3 times as fast. In Figure 50, the calculation time required for 2048x512 SOR on 16 nodes is shown. Note that the Jcc Myrinet version of SOR is faster than the Jcc Fast Ethernet version. Since Myrinet is a switched network, collisions do not occur when the number of messages increases. Fast Ethernet, however, does suffer from collisions. This can be seen clearly in Figure 43. The speedup is computed relative to the *same* program on a single CPU.

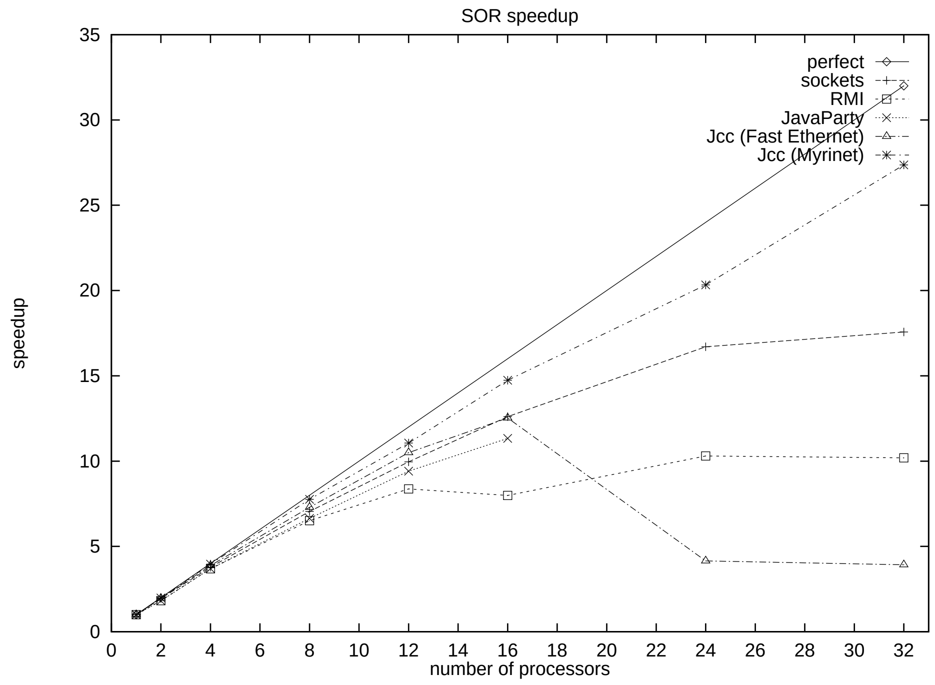


Figure 43: Speedup for 2048x512 parallel SOR

The speedup of Jcc Fast Ethernet collapses when more than 16 nodes are used. As can be seen in Figure 44 the Jcc Fast Ethernet and Jcc Myrinet versions perform equally well on less than 16 nodes. When more than 16 nodes are used, the execution time of Jcc Fast Ethernet approaches the execution time of the interpreted Java sockets version, indicating that the network performance is the problem. The Jcc Myrinet version does not suffer from this problem and shows a good speedup on all tests.

Note that JavaParty performs better than RMI, even though JavaParty uses RMI to implement its communication. This is caused by the different implementation of the communication of SOR in JavaParty and the 'hand crafted' RMI version. The code generated by the JavaParty compiler is more efficient. The sockets version shows the best speedup of the interpreted versions, although the speedup on 32 nodes is only 17.5. Because of its good performance on fewer nodes, the Jcc Fast Ethernet version has the worst speedup on 32 nodes, only 3.9. Due to the advantages of the fast switched network, the Jcc Myrinet version performs the best; a speedup of 27.3 on 32 nodes.

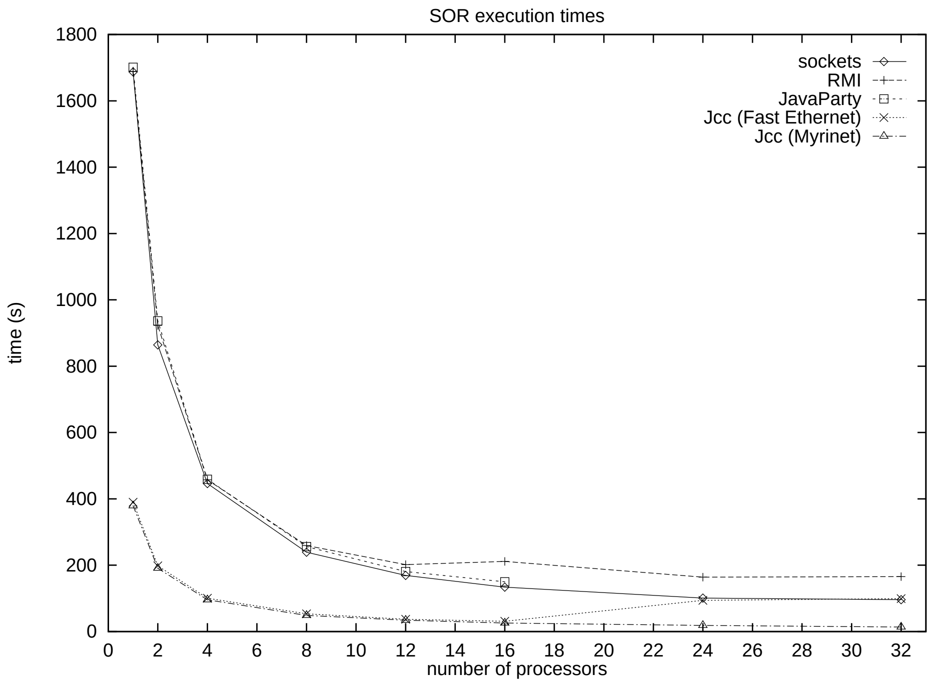


Figure 44: Execution time for 2048x512 parallel SOR

8.3.2 Traveling Salesperson Problem

We have implemented a replicated workers style parallel program for the Traveling Salesperson Problem (TSP), which uses one server (that hands out jobs) and several clients (that execute jobs). TSP is a well known graph searching problem, in which one has to find the shortest path that visits all nodes (cities) exactly once. All nodes share one JobQueue and one global minimum path length, which is replicated on all client CPUs. Jobs consist of a partial path of cities. It is up to the clients to examine all possible paths, beginning with the partial path in the job. If a client finds a better path, the global minimum must be updated. A lower minimum allows the clients to prune more paths, so it is important to inform all processors of the new minimum as fast as possible, in order to avoid unnecessary calculations. This is why minimum updates are broadcast. When a job is finished, the client asks the server for a new one. If all jobs are done, the server may print the global minimum, which is now the length of the shortest possible path.

This approach is nondeterministic, because the calculation time at a node depends on the global minimum, which may be updated by all nodes. So the time a job takes, depends on the speed of the other nodes. If a fast node happens to find a low minimum in a short time, a large part of the search tree can be pruned. This might influence the measurements. This can be circumvented, however, as the algorithm can easily be made deterministic. All that has to be done, is to initialize the global minimum with the shortest path that can be found. No client will be able to find a path shorter than the minimum, so minimum updates will never be done. Measured was the deterministic version of the algorithm, with an input file which contained fifteen cities.

The measurements are shown in Figures 45 and 46. The actual times that were measured are shown in Table 25. We did not measure TSP and IDA* execution times on more than 16 machines, because the problem size is too small. TSP runs for less than three seconds on 16 processors with Jcc, IDA* for less than seven. Also, JavaParty did not start on more than 16 CPUs.

The Jcc version of sequential nondeterministic TSP is about seven times faster than the interpreted Java version. This makes it more difficult to get the same speedups with Jcc as for instance the socket version has, because the Jcc version of program runs much shorter. In this shorter time, the same amount of communication has to be done. The Jcc version is, however, still amost seven times faster than the sockets version and more than fourteen times faster than the JavaParty version on sixteen CPUs, all on Fast Ethernet.

On one CPU, the Jcc version is twelve times faster than the JavaParty version. The bad speedup of the JavaParty version is partly due to the underlying Java RMI system, which has about the same speedup. Still, the Java RMI version is about 1.7 times faster than the JavaParty version. This can be explained by the recursive remote call problem, as described in Section 4.2.3. A part of the TSP program source is shown in Figure 47. The "calculateSubPath" method is a recursive method that implements the TSP problem. Because it is declared inside a remote class, *call by value* is used to call the method. It has only two integers and a small integer array of at most fifteen (i.e., the number of cities) elements. Due to the slow serialization mechanisms in Java, especially for arrays, as described in Section 4.3.3, the overhead to the recursive call is high. The measurements indicate that both TSP and IDA*, which is also recursive

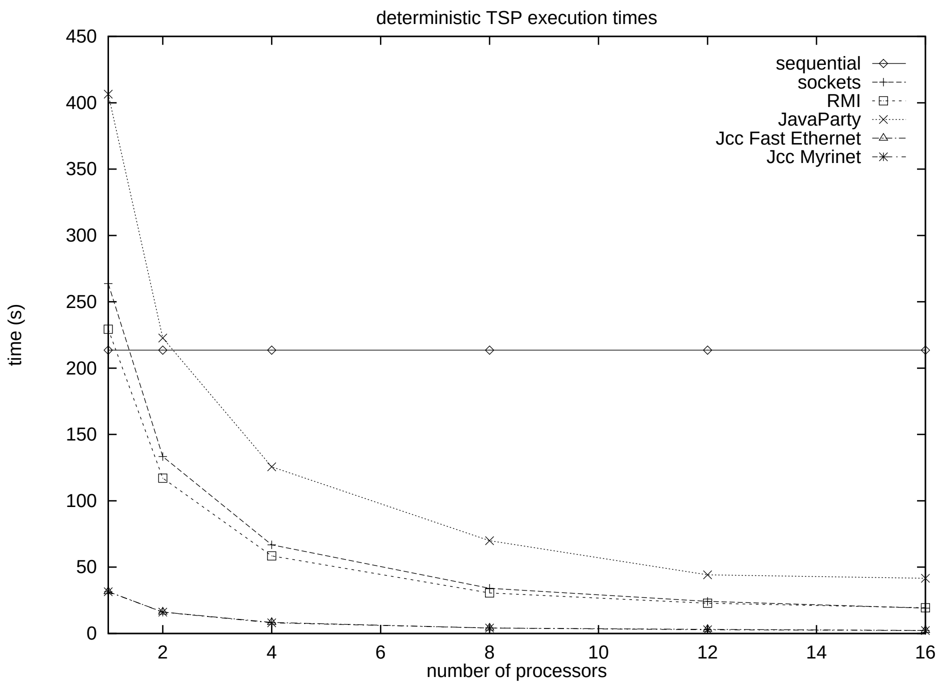


Figure 45: TSP execution time in seconds.

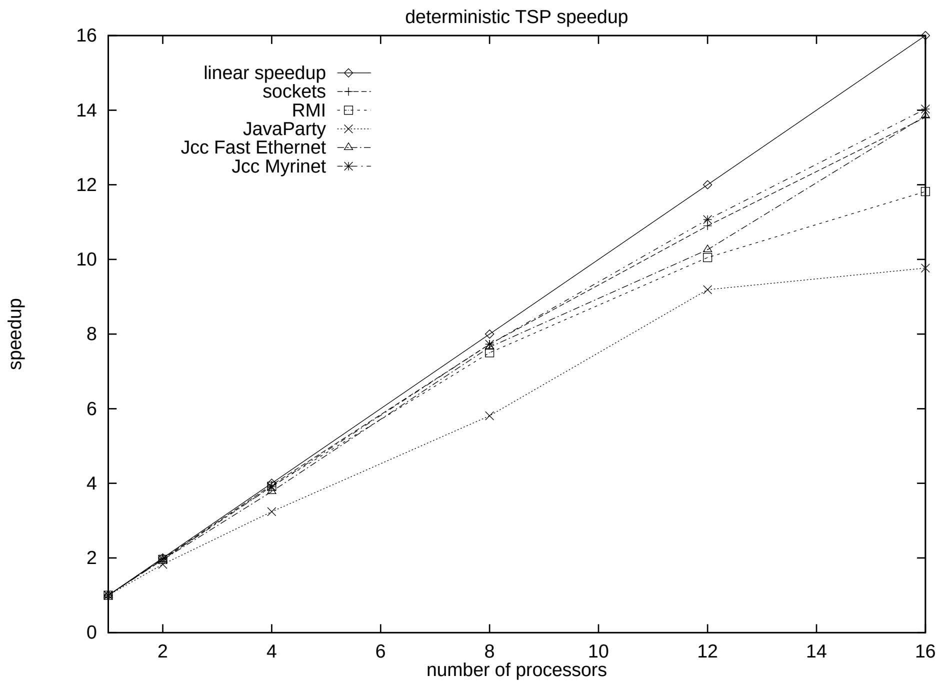


Figure 46: TSP speedup.

```
public remote class Client extends RemoteThread {

    // some methods and variables...

    void calculateSubPath(int hops, int length, int[] path) {
        int me, dist;

        if(length >= min) return;

        if(hops + 1 == nrCities) {
            if(length < min) {
                minimum.set(length);
                min = length;
            }
            return;
        }

        // Path really is a partial route.
        // Call calculateSubPath recursively for each subtree.
        me = path[hops];

        for(int i=0; i<nrCities; i++) {
            if(!Server.present(i, hops, path)) {
                path[hops+1] = i;
                dist = distanceTable.distance(me, i);
                calculateSubPath(hops+1, length+dist, path);
            }
        }
    }

    // more methods...
}
```

Figure 47: Recursion in a remote class.

(explained in Section 8.3.3), suffer badly from this problem.

Because this performance problem was noticed, we introduced the *local* modifier in Jcc, described in Section 4.2.3. When tried, it turned out that the versions of TSP and IDA* with the local modifier, where *call by reference* is thus used for the recursive method call, ran at the same speed as the version where the calls were marshalled, and thus *call by value* was used. This is remarkable, because more code is executed. This is probably due to caching effects, but requires future investigation. The local modifier probably will increase performance when more complex data structures are passed.

8.3.3 Iterative Deepening A*

IDA* is an abbreviation for Iterative Deepening A*, which is a combinatorial search algorithm. IDA* was described by R.E. Korf [14] and integrates iterative deepening with the heuristic control of A* to obtain linear bounds for a heuristic search [16]. A parallel, asynchronous version of IDA* is described in [19]. We started with an existing IDA* version in Orca, described in [3]. We use the algorithm to solve random instances of the 15-puzzle, and as we want a deterministic algorithm, *all* solutions of a particular length are examined.

The application, like TSP, uses a job queue and a shared variable containing the number of solutions found so far. One important difference between the IDA and TSP implementations, however, is that with IDA, all clients can generate new jobs while calculating. The jobs vary widely in complexity, so load balancing is of much more importance than with TSP. The job queue is distributed over all nodes, and work stealing is used to balance the load.

The order of the jobs in the queue is important, because jobs are split up in parts, which are put back at the tail of the queue. So, at all times, the largest jobs are at the front of the queue. Whenever a client tries to get a job locally, the job queue makes sure it gets a small one. When a client is stealing a job from another node, it wants a large one, so it won't have to steal again soon, as this is an expensive operation.

The work stealing algorithm works as follows. First a node looks in its own queue for a job. If there is a job, the node gets it, in LIFO order. If a node does not have a job in its queue, it tries to steal a job from the other clients, in FIFO order. When there are N clients, only log(N) queues are checked, in order to keep the algorithm scalable. When no other investigated client had any jobs left, the stealing node must tell the master it is idle. When all clients are idle, and no solutions are found, we start over with a larger bound. Every time a client adds a job to its local queue, it must announce this to the master CPU, which can then inform all idle clients.

In contrast to the TSP implementation, with our IDA* implementation, we do calculate on the master CPU. This makes the implementation a little harder, but it makes more sense, as the job queue is no longer centralized and residing at the master, but distributed. As a result, the master has less work to do than for TSP, making it worthwhile to participate in the calculation of boards.

The sequential Jcc version is about 3.5 times faster than the Java version, as shown in Table 27. Figures 48 and 49 show that the Jcc version of IDA* also has the best speedup. This is remarkable, because the Jcc version of the program runs only for about seven seconds on sixteen CPUs.

Not only does IDA* only give an indication of RMI performance, it is also a stress test for the garbage collector (see Section 8.2.2).

To give an overview, the execution times of all implemented parallel programs is shown in Figure 50.

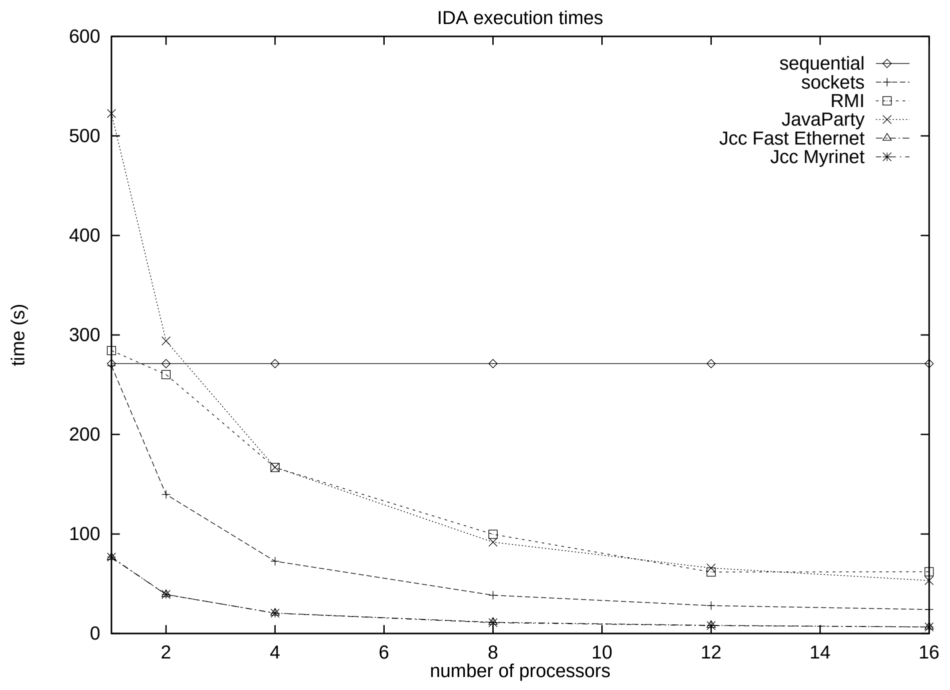


Figure 48: IDA* execution time in seconds.

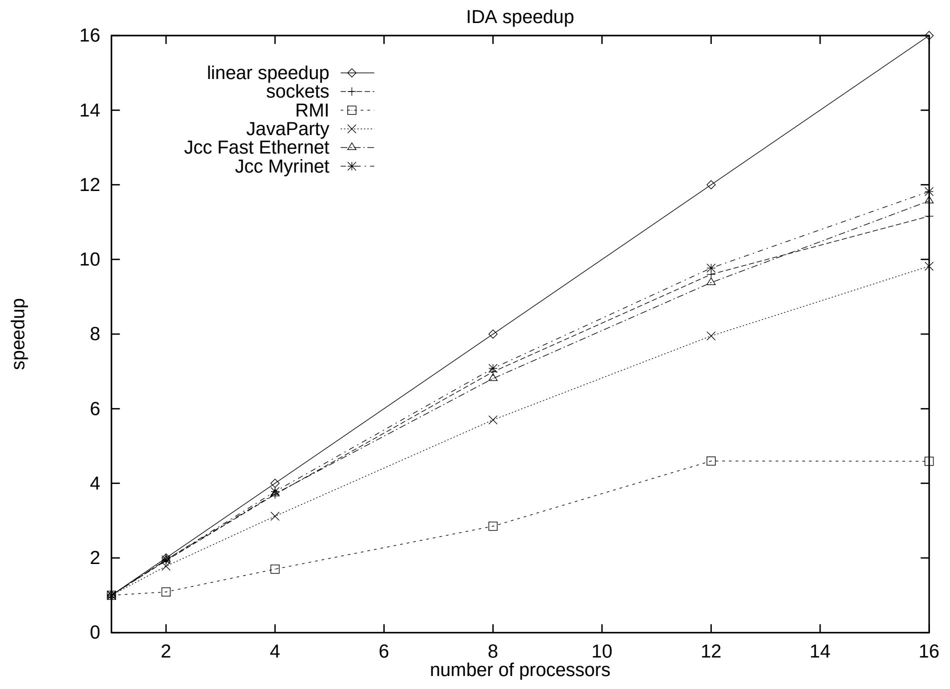


Figure 49: IDA* speedup.

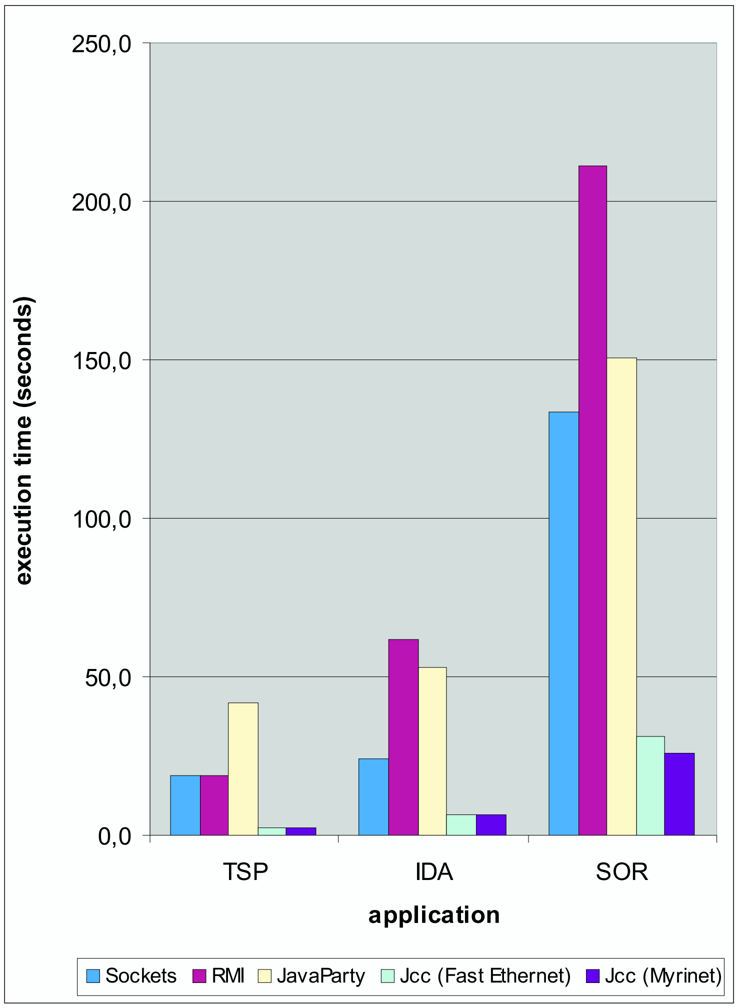


Figure 50: Execution time of the parallel programs on 16 CPUs in seconds.

9 Conclusion

In this thesis, we have given a description of a runtime system used to implement a fast parallel Java system. To implement the runtime system, we have used efficient low level communications software (Panda), and a native compiler for Java (Jcc). Various optimizations were made in the runtime system (using compiler support if necessary):

- Serialization code is generated at compile time (in C).
- A simple and efficient serialization and marshalling protocol which avoids unnecessary copying. It makes only two copies, one at the sender and one at the receiver.
- Simple remote operations are optimized:
  - No thread creation if a method is simple.
  - Prevent duplicate detection when possible.

The runtime system uses a simple and efficient garbage collection scheme based on the mark and sweep algorithm. An object cache is used to make object creation more efficient. A simple reference counting garbage collector is used for distributed garbage collection.

By using this runtime system we have shown that fast communication and and easy-to-use parallel programming is certainly possible in Java, making Java a candidate for high-performance parallel applications.

10 Future work

Future work on this runtime system and fast parallel Java include:

- Support for SUN's RMI.
- Support for heterogeneous environments.
- The use of asynchronous communication primitives for implementing RMI.
- Asynchronous RMI.
- RMI multicast.
- Implement and research the effects of more advanced local and distributed garbage collection algorithms.
- Add other features to the Java language needed for high-performance computing, like efficient multi-dimensional arrays and array operations (which will require language extensions).

11 Acknowledgements

We would like to thank Henri Bal for his support and guidance, Aske Plaat for his support and fifteen (well, ok, 14.8) applepies, Raoul Bhoedjang for proofreading this document, Ronald Veldema for his work on the Jcc compiler, and all other people at the computer systems group at the Vrije Universiteit.

A The distributed garbage collector

```java
import java.util.Hashtable;

class Distributed_GC {

    /* A somewhat trivial distributed garbage collector */

    static Hashtable RemoteObjectHash = new Hashtable();

    static void addRemoteObject(Object o) {
        RemoteObjectHash.put(o, new RemoteObjectCounter(o));
    }

    static void addRemoteUser(Object o) {

        RemoteObjectCounter temp = (RemoteObjectCounter) RemoteObjectHash.get(o);

        if (temp != null) {
            temp.counter++;
        } else {
            System.err.println("ERROR(addRemoteUser) : Unknown remote object reference");
        }
    }

    static void removeRemoteUser(Object o) {

        RemoteObjectCounter temp = (RemoteObjectCounter) RemoteObjectHash.get(o);

        if (temp != null) {
            temp.counter--;

            if (temp.counter == 0) {
                RemoteObjectHash.remove(o);
            }
        } else {
            System.err.println("ERROR(addRemoteUser) : Unknown remote object reference");
        }
    }
}

class RemoteObjectCounter {

    /* Used to count the number of remote users */

    Object ro;
    int counter;

    RemoteObjectCounter(Object ro) {
        this.ro      = ro;
        this.counter = 1;
    }
}
```

B Overview of the results

B.1 Network Throughput

array size (Kbytes)	Panda	Panda + copy	Jcc
Myrinet			
1	14.6	13.5	11.1
5	36.3	28.5	17.1
10	45.2	33.2	20.6
50	55.7	35.0	20.1
100	49.9	22.3	19.0
250	39.1	15.8	12.0
500	31.2	14.4	11.2
750	30.3	14.4	12.2
1000	30.0	14.4	13.1
Fast Ethernet			
1	2.8	2.8	2.6
5	5.8	5.5	4.9
10	7.1	6.6	5.9
50	9.5	8.4	7.3
100	10.0	8.0	7.1
250	10.2	7.3	6.6
500	10.3	7.3	6.5
750	10.3	7.3	6.5
1000	10.3	7.3	6.5

Table 8: Jcc network throughput in Mbytes/s.

B.2 Word count of the application sources

application	sequential	sockets	RMI	JavaParty	Jcc
SOR	4212	17666	17905	10283	9754
TSP	489	1551	1047	775	775
IDA	1036	2837	2204	1562	1562

Table 9: Number of words of the source code.

B.3 Seqential application times

application	time
SOR JDK	1688.44
SOR Jcc	390.00
TSP JDK	213.54
TSP Jcc	32.50
IDA JDK	271.13
IDA Jcc	76.83

Table 10: Execution times sequential applications in seconds.

B.4 SOR

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	348.55	1.00	348.43	.04
2	187.19	1.86	174.67	6.69
4	108.95	3.19	87.18	19.98
8	66.09	5.27	43.37	34.38
12	51.69	6.74	28.98	43.93
16	54.76	6.36	21.99	59.84
24	42.88	8.12	14.72	65.67
32	51.48	6.76	11.35	77.95

Table 11: Results for Socket SOR (512x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	811.53	1.00	811.39	.02
2	422.04	1.92	406.13	3.77
4	227.76	3.56	202.70	11.01
8	128.39	6.32	101.12	21.25
12	94.16	8.61	67.55	28.26
16	76.54	10.60	50.54	33.97
24	65.59	12.37	33.95	48.24
32	60.81	13.34	25.54	58.01

Table 12: Results for Socket SOR (1024x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	1687.11	1.00	1686.95	.01
2	864.24	1.95	844.57	2.28
4	446.14	3.78	421.54	5.52
8	238.99	7.05	210.18	12.06
12	169.38	9.96	140.32	17.16
16	133.78	12.61	105.01	21.51
24	100.99	16.70	70.40	30.29
32	96.00	17.57	52.91	44.89

Table 13: Results for Socket SOR (2048x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	350.99	1.00	350.31	.20
2	186.23	1.88	177.65	4.61
4	103.84	3.38	88.41	14.86
8	65.94	5.32	45.18	31.49
12	47.85	7.33	30.17	36.93
16	46.38	7.56	23.23	49.91
24	71.64	4.89	15.59	78.24
32	94.27	3.72	11.63	87.67

Table 14: Results for RMI SOR (512x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	814.49	1.00	813.68	.11
2	422.66	1.92	410.33	2.92
4	233.00	3.49	207.02	11.16
8	127.03	6.41	103.48	18.54
12	95.71	8.50	69.43	27.46
16	91.31	8.91	53.16	41.78
24	95.70	8.51	35.79	62.60
32	132.50	6.14	26.96	79.66

Table 15: Results for RMI SOR (1024x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	1688.44	1.00	1687.53	.06
2	922.89	1.82	850.25	7.88
4	457.40	3.69	427.29	6.59
8	259.23	6.51	213.91	17.49
12	201.71	8.37	143.64	28.80
16	211.30	7.99	107.92	48.93
24	163.87	10.30	73.46	55.17
32	165.65	10.19	55.50	66.50

Table 16: Results for RMI SOR (2048x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	358.62	1.00	357.92	.20
2	188.38	1.90	178.69	5.15
4	107.36	3.34	90.05	16.13
8	71.61	5.00	46.52	35.04
12	62.23	5.76	31.77	48.94
16	53.20	6.73	24.46	54.03

Table 17: Results for JavaParty SOR (512x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	826.40	1.00	825.61	.10
2	428.08	1.93	414.79	3.11
4	229.24	3.60	208.55	9.03
8	132.97	6.21	106.12	20.20
12	100.06	8.25	71.91	28.14
16	85.37	9.67	55.00	35.58

Table 18: Results for JavaParty SOR (1024x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	1701.20	1.00	1700.10	.07
2	936.60	1.81	855.54	8.66
4	459.05	3.70	432.84	5.71
8	255.98	6.64	219.19	14.38
12	180.69	9.41	147.98	18.11
16	150.03	11.33	112.28	25.17

Table 19: Results for JavaParty SOR (2048x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	80.39	1.00	80.30	.11
2	42.46	1.89	40.67	4.22
4	22.12	3.63	20.02	9.47
8	12.99	6.18	10.19	21.51
12	10.32	7.78	6.57	36.31
16	21.04	3.81	4.92	76.61
24	80.54	.99	3.26	95.96
32	80.27	1.00	2.44	96.96

Table 20: Results for Jcc-FastEthernet SOR (512x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	187.51	1.00	187.42	.06
2	95.96	1.95	94.31	1.72
4	49.11	3.81	46.61	5.10
8	27.32	6.86	24.01	12.11
12	19.83	9.45	15.50	21.87
16	16.68	11.23	11.59	30.51
24	83.21	2.25	7.66	90.79
32	95.53	1.96	5.72	94.01

Table 21: Results for Jcc-FastEthernet SOR (1024x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	390.00	1.00	389.91	.03
2	198.20	1.96	196.56	.83
4	100.34	3.88	97.71	2.63
8	53.46	7.29	49.93	6.60
12	37.11	10.50	32.37	12.78
16	31.10	12.53	24.24	22.05
24	93.91	4.15	16.12	82.84
32	99.35	3.92	12.05	87.87

Table 22: Results for Jcc-FastEthernet SOR (2048x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	78.32	1.00	78.23	.12
2	40.34	1.94	39.10	3.09
4	20.85	3.75	19.50	6.46
8	11.18	7.00	9.64	13.70
12	8.53	9.17	6.37	25.32
16	6.17	12.68	4.71	23.62
24	5.24	14.94	3.11	40.58
32	3.90	20.08	2.33	40.13

Table 23: Results for Jcc-myrinet SOR (512x512)

hosts	time (sec)	speedup	calculation time (sec)	communication (%)
1	379.08	1.00	378.99	.03
2	190.97	1.98	188.81	1.13
4	95.52	3.96	93.91	1.70
8	48.83	7.76	47.00	3.76
12	34.27	11.06	31.53	8.00
16	25.71	14.74	23.61	8.16
24	18.64	20.33	15.52	16.75
32	13.85	27.36	11.54	16.68

Table 24: Results for Jcc-myrinet SOR (2048x512)

B.5 TSP

hosts	sockets	RMI	JavaParty	Jcc Fast Ethernet	Jcc Myrinet
1	263.67	229.30	406.48	31.54	31.50
2	133.34	117.06	222.70	16.15	16.06
4	66.80	59.65	125.28	8.34	8.06
8	34.09	30.59	69.91	4.13	4.07
12	24.19	22.87	44.25	3.08	2.85
16	19.09	19.11	41.61	2.79	2.25

Table 25: Execution times TSP in seconds.

hosts	sockets	RMI	JavaParty	Jcc Fast Ethernet	Jcc Myrinet
1	1.00	1.00	1.00	1.00	1.00
2	1.98	1.96	1.83	1.95	1.98
4	3.95	3.92	3.24	3.78	3.91
8	7.73	7.50	5.81	7.65	7.81
12	10.90	10.05	9.19	10.26	11.17
16	13.81	11.82	9.77	13.85	14.16

Table 26: Speedup TSP.

B.6 IDA*

hosts	sockets	RMI	JavaParty	Jcc Fast Ethernet	Jcc Myrinet
1	269.07	284.30	522.61	76.31	76.83
2	139.68	260.17	293.97	39.10	39.32
4	72.67	166.79	166.93	20.50	20.33
8	38.45	99.63	91.72	11.21	10.86
12	28.04	61.83	65.72	8.14	7.87
16	24.11	62.00	53.22	6.60	6.50

Table 27: Execution times IDA* in seconds.

hosts	sockets	RMI	JavaParty	Jcc Fast Ethernet	Jcc Myrinet
1	1.00	1.00	1.00	1.00	1.00
2	1.93	1.09	1.78	1.95	1.95
4	3.70	1.70	3.12	3.72	3.78
8	6.99	2.85	5.70	6.81	7.08
12	9.60	4.60	7.95	9.38	9.77
16	11.16	4.59	9.82	11.57	11.82

Table 28: Speedup IDA*.

C Panda RPC interface

The RPC module provides reliable RPC with at-most-once semantics. The user handler function is defined as follows:

```
typedef int (*pan_rpc_handler_f)(int ticket, void *request, int size, int len);
```

The receive function gets as arguments a ticket for the reply message, a pointer to the request data, the size of the request data buffer, and the length of the request data. The request handler function returns a boolean specifying whether the data buffer is kept at the handler function level (1) or can be reused at the RPC level (0). If the buffer is kept at the handler function level, it must be released with pan_free.

The RPC layer guarantees that size - len >= MAX(pan_rpc_request_trailer(), pan_rpc_reply_trailer())
The upcall is not allowed to block on a condition synchronization, only on short-term mutex synchronization (lock/unlock). Furthermore, multiple instances of the upcall can be active at the same time.

```
void pan_rpc_init(int *argc, char *argv[]);
```

Initializes the RPC module.

```
void pan_rpc_end(void);
```

Releases all resources held by the RPC module.

```
void pan_rpc_register(pan_rpc_handler_f handler);
```

Registers the request handler function. The registration must be performed before pan_start is called.

```
int pan_rpc_request_trailer(void);
```

Returns the space that the sender has to reserve for a trailer after the user request data. The data in the buffer where the trailer will be put may not be accessed during a call to pan_rpc_trans. The original data in this area is restored when the pan_rpc_trans call is finished.

```
int pan_rpc_reply_trailer(void);
```

Returns the space that the sender has to reserve for a trailer after the user reply data. The data in the buffer where the trailer will be put may not be accessed during a call to pan_rpc_reply. The original data in this area is restored when the pan_rpc_reply call is finished.

```
void pan_rpc_trans(int dest, void *request, int req_len,
   void **reply, int *rep_size, int *rep_len);
```

Sends request message request with length req_len to destination dest. The call blocks until a reply is received. The reply message will be put in reply, with reply buffer size rep_size and reply length rep_len. The reply has to be released with pan_free.

```
void pan_rpc_reply(int ticket, void *reply, int len);
```

Sends a reply message to the originator of the request message with ticket ticket. The reply message will be cleared with pan_free.

References

[1] Java grande forum, 1998. http://jhpc.org/grande/.

[2] M.G. Bakker and P. Dozy. Performance study of parallel programs on a clustered Wide-Area Network. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, August 1997. http://www.cs.vu.nl/~aske/msc97.ps.gz.

[3] Henri E. Bal, Aske Plaat, Mirjam G. Bakker, Peter Dozy, and Rutger F. H. Hofman. Optimizing parallel applications for wide-area clusters. *12th International Parallel Processing Symposium IPPS'98, Orlando, Florida*, April 1998. http://www.cs.vu.nl/~aske/ipps98.ps.gz.

[4] R.A.F. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 213–226, San Diego, CA, USA, September 1993. ftp://ftp.cs.vu.nl:/pub/amoeba/orca_papers/sedms93.ps.Z.

[5] Raoul Bhoedjang, Tim Rühl, and Henri E. Bal. LFC: A communication substrate for Myrinet. In *Fourth Annual Conference of the Advanced School for Computing and Imaging, June 1998, Lommel, Belgium.* ftp://ftp.cs.vu.nl:/pub/tim/papers/asci98.ps.gz.

[6] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software practice and experience*, 1988.

[8] B.O. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K.E. Schauser, and D. Wu. Javelin: Internet based parallel computing using java. *concurrency: practice and experience*, November 1997. http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/finalps/11_chris.ps.

[9] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Proceedings if the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* INRIA Roquencourt, France, 1998.

[10] P.A. Gray and V.S. Sunderam. IceT: Distributed Computing and Java. *concurrency: practice and experience*, November 1997. http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/finalps/12_gray.ps.

[11] B. Griswold and P. Phillps. Uscd benchmark, Aug 1998. http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.

[12] M. Haines and K. langendoen. Platform-independent runtime optimizations using openthreads. In *11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997. http://meru.cs.uwyo.edu/~haines/research/ot/ipps.ps.

[13] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Training distributed garbage, the dmos collector. *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1997. http://www-ppg.dcs.st-andrews.ac.uk/ Publications/PostScript/dmos.ps.gz.

[14] Richard E. Korf. Iterative Deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[15] Langendoen, K. and Bhoedjang, R. and Bal, H.E. Models for Asynchronous Message Handling. *IEEE Concurrency*, Vol. 5(No. 2):28–38, April-June 1997.

[16] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence, structures and strategies for complex problem solving*, pages 133–137, 149–150, 179. The Benjamin/Cummings Publishing Company, inc., 2nd edition, 1993.

[17] M. Philippsen and M. Zenger. Javaparty—transparent remote objects in java. *Concurrency: Practice and Experience*, Vol. 9(No. 11):1125–1242, 1997. http://wwwipd.ira.uka.de/~phlipp/party.ps.gz.

[18] R.R. Raje, J.I. William, and M. Boyles. An asynchronous remote method invocation (ARMI) mechanism for java. *concurrency: practice and experience*, November 1997. http://www.npac.syr.edu/ users/gcf/03/javaforcse/acmspecissue/finalps/16_raje.ps.

[19] A. Reinefeld and V. Schnecke. AIDA* -asynchronous parallel IDA*. In *Proceedings 10th Canadian Conference on Artificial Intelligence, AI'94, May 1994, Banff, Canada*, pages 295–302. Paderborn Center for Parallel Computing, Germany, 1994. http://brahms.informatik.uni-osnabrueck.de/ postscripts/ai_94.ps.Z.

[20] Sun Microsystems Inc. *Java language specification*, August 1996. ftp://ftp.javasoft.com/docs/specs/langspec-1.0.ps.zip.

[21] Sun MicroSystems, Inc. Java native interface specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/jni.ps.

[22] Sun MicroSystems, Inc. Java (TM) Object Serialization Specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.ps.

[23] Sun MicroSystems, Inc. Java (TM) Remote Method Invocation Specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.ps.

[24] R. Veldema. Jcc, a native java compiler. Master's thesis, Vrije Universiteit Amsterdam, August 1998. http://www.cs.vu.nl/~rveldema.

[25] Paul R. Wilson. Uniprocessor garbage collection techniques. Submitted to ACM Computing Surveys.

[26] X/Open Company Ltd. File System Safe UCS Transformation Format (FSS_UTF). X/Open Preliminary Specification, Document Number: P316.

[27] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishmurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. In *ACM 1998 workshop on Java for High-performance network computing.* UCB, February 1998. http://www.cs.ucsb.edu/ conferences/java98/papers/titanium.ps.

[28] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, June 1997. http://www.npac.syr.edu/ users/gcf/03/javaforcse/acmspecissue/finalps/17_yu.ps.

[29] A.D. Zubiri. An assessment of Java/RMI for Object Oriented Parallelism. Master's thesis, University of Alberta, Fall 1997.