

Compiler Optimized Remote Method Invocation

Ronald Veldema

Michael Philippsen

University of Erlangen-Nuremberg
Computer Science Department 2
Martensstrasse 3 • 91058 Erlangen • Germany

veldema@cs.fau.de

philippsen@cs.fau.de

ABSTRACT

We further increase the efficiency of Java RMI programs. Where other optimizing re-implementations of RMI use pre-processors to create stubs and skeletons and to create class specific serializers and deserializers, this paper demonstrates that with transformations based on compile time analysis an additional 18% performance gain can be achieved over class specific serializers alone for a simple scientific application.

A novel and RMI-specific version of static heap analysis is used to derive information about objects that are passed as arguments of remote method invocations. This knowledge of objects and their interrelations is used for three optimizations.

First, dynamic introspection and/or (recursive) dynamic invocations of object specific serializers is slow. With knowledge from our heap analysis, the marshaling of graphs of argument objects can be inlined at the call site. Hence, many method table lookups and skeleton indirections of previous approaches can be avoided and less protocol information is sent over the network.

Secondly, because object graphs may be passed as RMI arguments, cyclic references need to be detected. With our heap analysis, we can detect if there is no potential for cycles and hence, if cycle detection code can be left out of the serialization and marshaling codes.

Finally, object arguments to remote methods cause object creation and garbage collection. Heap analysis and an RMI-specific version of escape analysis allows the reuse of object graphs created in earlier remote invocations.

1. INTRODUCTION

Java contains a mechanism to transparently call methods on objects allocated on remote machines, named Remote Method Invocation (RMI). While RMI allows for easy distributed programming, its efficiency is a problem when using a simple, naive implementation.

Because this paper requires intimate knowledge about RMI's functionality, we start with a step-by-step walkthrough of a single RMI. For the purposes of this walkthrough we will use the example in Figure 1. When executing a call to `<Example_reference>.foo`,

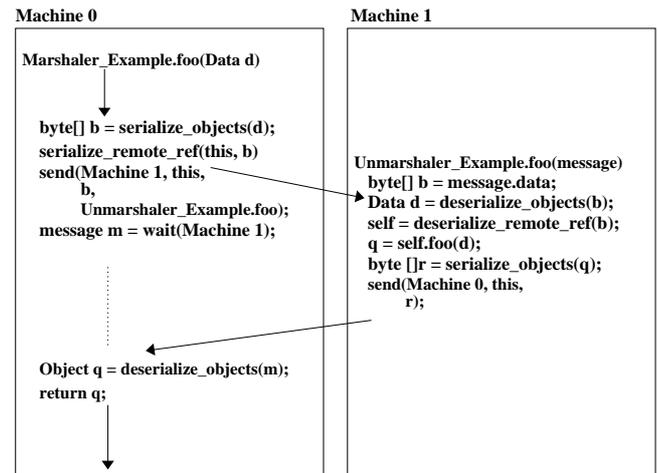


Figure 1: Simplified RMI example.

instead of directly invoking `foo`, a generated *marshaling* method is called (`Marshaler_Example.foo`). It calls the *serialization* routines for each of its arguments to convert them to a sequence of bytes. In addition to the argument objects, all the objects they are referring to are converted into a byte representation as well.

The resulting array of bytes is sent over the network to be unpacked by the counterpart of the marshaling routine:

`Unmarshaler_Example.foo`. `Unmarshaler_Example.foo` calls a de-serialization routine for each of the parameters to recreate copies of the argument objects. Because a to-be-serialized object may contain a reference to itself or to a previously serialized object, a hash-table is maintained to allow self-referring data-structures to be serialized. The `'serialize_objects'` function will therefore, before appending the fields of an object to the message, check if the object has already been serialized and if so insert a 'handle' to the previously serialized object in the message.

After copies of the arguments have been reconstituted, a new thread is created to invoke the user's code, in this case `foo` is invoked. After `foo` has finished, the return values are serialized to another array of bytes which is sent back to the invoking machine. The invoking machines performs de-serialization to reconstitute a copy of the return value.

If the remote object upon which the RMI is performed is (accidentally) located on the same machine as the invoking machine, the parameter and return value objects are cloned. This ensures that the same parameter passing semantics are observed regardless of the location of the called object.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Supercomputing '03 Phoenix AZ

Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

However, the standard implementation from SUN has several efficiency problems:

- the serialization and de-serialization implementations are slow;
- the network protocol is very heavy weight sending much unneeded data for many applications, mainly because too much type information is sent for each transferred object;
- the cycle detection hash-table is always created, even if it is clear that the objects to be serialized will not contain cycles;
- the object allocation and deallocation costs are unnecessarily high due to the de-serialization process.

In this paper we attack the causes of this overhead: the cost of the serialization process, the cost of cycle detection, and the object allocation and deallocation costs.

The first problem is the cost of introspection in the serialization process: examining an object's layout to locate normal fields and references to other objects. Whenever a reference field is found inside an object, the serialization process is recursively invoked for the referred-to object. This process can be sped up by generating a serialization routine for each class, as has been implemented in the KaRMI [15] and Manta [12] projects. But still, whenever a reference is found inside an object, the serialization routine is called indirectly and type information is sent anew.

By performing heap analysis, we can often detect what type of object is pointed to by a reference field at compile time and generate specialized code to serialize the fields of the pointed-to object greatly reducing method invocation overhead. Plus, it is no longer necessary to send type information over the network for the referred-to object. Moreover, serialization code can be inlined at the RMI call site – often even for referred-to objects.

To increase the precision of the analysis, (un)marshalers are generated on a per call site basis instead of creating a single marshaler and unmarshaler per callee. The serializers for objects are likewise generated on a per call site basis as the size and type of the objects passed to a method can vary per call site.

The second cause of RMI's overhead is due to the need for cycle detection. Cycle detection is used in RMI to allow cyclic data-structures to be (de)serialized. Whenever a reference is encountered that points to an object that has already been serialized, a handle to the previously serialized object has to be inserted instead of re-serializing the object again. The costs involved in cycle detection are thus: the creation and deletion of a hash-table, adding every single object reference to that hash-table and finally, checking if an object has already been serialized.

Based on heap analysis we can often determine at compile time whether or not the object graphs to be serialized may be cyclic at all and only then add code to the serializers to perform cycle detection.

The third problem is the cost of object allocation and garbage collection due to the objects that are created by de-serialization of argument and return value object graphs. We solve this problem by reusing objects that have been created by in earlier remote method invocations.

Our testbed for experimentation with RMI is based on JavaParty [16]. In JavaParty, classes can be marked with a *remote* keyword to allow all methods of the class to be remotely invocable by means of RMI. The underlying details of remote object placement, remote thread allocation, and exception management that are visible in normal RMI are hidden in JavaParty. This simplifies the process of compiler analysis tremendously.

Because the original JavaParty implementation does not easily support the more advanced compiler analysis required for the optimizations proposed here, JavaParty has been re-implemented in the Manta system. The new system is henceforth referred to as Manta-JavaParty. Similar to [15] and [12], Manta-JavaParty replaces Sun's

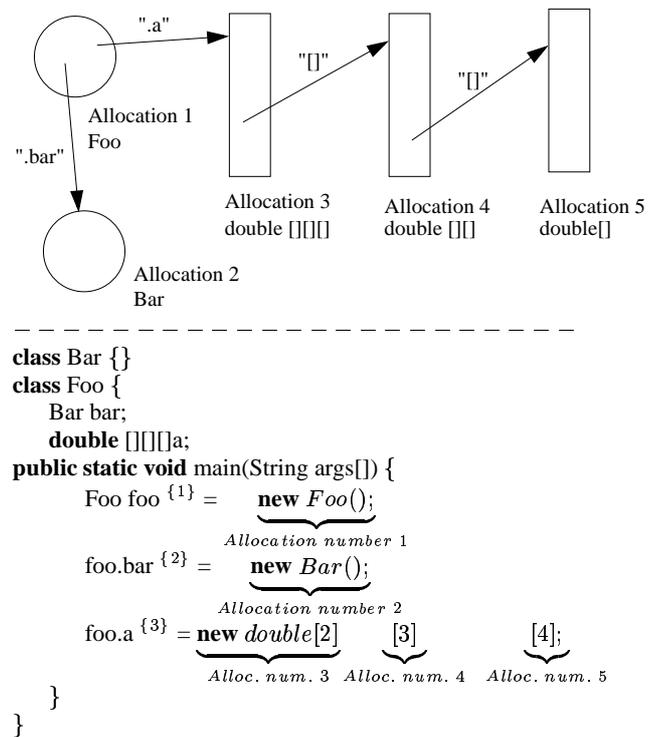


Figure 2: Example Heap analysis.

heavyweight RMI network protocol by a lightweight protocol.

2. HEAP ANALYSIS

For all three of the optimizations proposed in this paper, detailed information is required about what types of objects are passed to and returned from RMIs at runtime. Furthermore, it is not sufficient to know the types of the arguments themselves but also what types of objects they are referring to (recursively). For example, for cycle detection elimination we need to know what objects are passed to an RMI and what their relations are, specifically, does object X (indirectly) contain a reference to itself?

To supply this information we employ heap analysis. Heap analysis tells us if an object allocated at one object allocation site *may* contain a reference to an object allocated at another object allocation site. Heap analysis also gives us the set of object allocation sites to which global variables, local variables, arguments, or parameters may refer.

The implementation described here is a variation of the heap analysis described in [8] but extended to handle the parameter semantics of RMI. As described in the introduction, parameters and return values are cloned during the (de)serialization process. To arrive at the most precise representation of the runtime heap, the heap approximation needs to reflect the cloning process by cloning those parts of the heap graph that are passed by RMIs.

An example heap graph is shown in Figure 2. Here an object (of type *Foo*) is shown that holds two references, a reference to another object (of type *Bar*) and another reference to a three dimensional array of doubles. The code shows the allocation site numbers. In addition, the variables are attributed with a list of allocation site numbers that construct the heap graph. The numbers refers to the code sites where the objects stored in the variables may have been created. Note that the array of arrays is not represented with six nodes (for 2×3 double arrays of size 4) in the graph as the nodes

represent *object allocations sites* and not actual objects in the heap. Conceptually, the compiler constructs the heap graph as follows:

1. convert all code to SSA form [6];
2. assign to each object allocation site a unique number (allocation site number) and create a new node (allocation site number, type) for it in the heap graph;
3. perform data-flow analysis on the allocation site numbers: for each assignment ' $a = b$ ', assign the set of allocation site numbers associated with ' b ' to ' a '. For each ϕ assignment ' $a = \phi(b \dots z)$ ', assign to ' a ' the union of the sets of allocation site numbers associated with $b \dots z$. For each call instruction ' $a = \text{foo}(b \dots z)$ ', copy each argument's set of allocation site numbers associated with the argument to the callee's formal parameter. In addition, copy the union of all the allocation number sets of all the return statements in the callee to ' a ';
4. whenever encountering a field assignment: ' $a.b = c$ ', assign the set of allocation site numbers associated with ' c ' to the ' b ' field of all objects associated with the allocation number set belonging to ' a ';
5. whenever encountering a field load: ' $a = b.c$ ', perform the reverse operation of step 4;
6. data-flow continues until a fixpoint is reached (steps 3 to 6).

However, this simple algorithm is not sufficient when applied to RMI programs as the serialization process creates deep copies of RMI arguments and this information is not yet represented in the above heap graph. Consider the following example: inside a remote method a parameter is modified. Without observing the RMI parameter semantics the modification would be visible in the caller as well.

A naive (but wrong) solution to this problem is to create a deep copy of the heap graph passed to, or returned from, a remote call. During the cloning process of the heap graph new and unique allocation site numbers are assigned to the new heap nodes. Roots of cloned heap graphs are then associated with the RMI parameters and return value. The problem with this naive handling of remote call parameters are loops in the data-flow. This problem can best be described using the example in Figure 3. An allocation number (2) is associated with object ' t '. During allocation number propagation ' t ' is found as an argument to *foo*. Because *me.foo* is a remote call instruction, the heap graph is cloned and t 's formal parameter ' a ' is assigned a new unique allocation number (3) which is passed to *foo* for further data-flow. In *foo*, ' t ' (now parameter ' a ') is propagated to the return instruction. This again causes cloning and a new unique allocation site number (4) to be associated with the return value of *me.foo*. For the second iteration of the loop the set of allocation site numbers of ' t ' is $\{2,4\}$. Simply cloning the heap graph and assigning new allocation site numbers will cause the set associated with ' t ' to grow in each iteration. Thus, the heap analysis will never reach a fixpoint and it will not terminate.

To stop the data-flow after the first cycle, we change the implementation of the allocation number from a single integer to tuple of two integers: a logical allocation number that changes when flowing from (or to) an RMI and a physical allocation number that remains fixed throughout the data-flow process. Whenever an allocation site number is now propagated over a remote call, it is first checked to see if the physical allocation number has already been propagated to that remote function and only if that is *not* the case, the logical allocation site number is cloned and the pair is passed to the remote function.

```

remote class Foo {
    Object foo(Object a {3,5,7,9,11,etc}) {
        return a;
    }
    static void zoo() {
        Foo me{1} = new Foo();
        Object t{2} = new Object();
        for (int i = 0; i < 100; i++) {
            t{4,6,8,10,etc} = me.foo(t{2,4,6,8,10,etc});
        }
    }
}

```

Figure 3: Data-flow problem with remote call instructions.

```

remote class Foo {
    Object foo(Object a {3,2}) {
        return a;
    }
    static void zoo() {
        Foo me{1,1} = new Foo();
        Object t{2,2} = new Object();
        for (int i = 0; i < 100; i++) {
            t{4,2} = me.foo(t{2,2}, {4,2});
        }
    }
}

```

Figure 4: Data-flow problem with remote call instructions, the solution.

This is demonstrated in Figure 4, the allocation numbers have been replaced by tuples of (*logical allocation site number, physical allocation site number*). The cycle of Figure 3 is now stopped because the physical allocation number of object 't' remains to be 2, so straight after the creation of {4, 2} no further tuples are created. After heap analysis the physical allocation number of the tuple is no longer used. Its only purpose was to stop the cycle in the data-flow propagation.

3. OPTIMIZATIONS

3.1 Call Site Specific RMI Code Generation

In traditional RMI programs those methods of a class that should be remotely invocable are placed in a remote interface. All calls to a certain remote method then have to be directed towards the interface method that hides the implementation of the marshaling code. The advantage of this implementation is the simplicity of the implementation; a simple translator program (rmic) can translate the methods mentioned in the remote interface simply by examining the method prototypes. The other effect of this implementation strategy is that at all call sites the exact same subroutine for marshaling is called. With that approach, only the serialization code within the stub can be optimized.

At least two projects have increased RMI efficiency using specialized serialization code on a per class basis. This simplifies an RMI implementation as the compiler needs only to iterate over the fields of a class once to generate an appropriate serialization function.

There are a number of advantages to create call site specific serializers instead of class specific serializers. Generating serializers specific for a given call site gives the compiler more opportunities for specialization as data-flow and heap graph information is available on a per call site basis.

As a simple example of the advantages of call site specific RMI code generation consider a situation where the return value of an RMI is ignored. Without call site specific optimizations, the return value would be needlessly sent over the network. With call site specific code generation, the return value can be ignored at the sender. Instead, only a small acknowledgment is sent to inform the caller that the calling thread can continue its execution.

A further advantage of call site specific code generation is that the generated code can be further specialized to the arguments passed to the RMI. Namely for the generated code we can, at compile time, differentiate between two different derived classes or between different array sizes that are passed as arguments.

The example in Figure 5 shows a method 'foo' that is called with two different object types as argument. Figure 6 shows the generated code for method 'go'. For each of the two call sites a separate marshaler is created. The first marshaler copies the int field directly into the message. The second marshaler follows the reference field 'p' in class 'Derived2' and copies the int field of the object pointed to by 'p' into the message. Please note that the pseudocode may suggest that the marshalers are generated in Java, they are in fact generated directly in the compiler's intermediate language. Also note that the message object is also not allocated on the heap as the pseudo code suggests but rather allocated on the call stack.

In contrast, the version using class specific serialization (see Figure 7) causes explicit invocations of serialization methods. These need to send explicit type information (*write_type*) that can be avoided with the call site specific approach. Another method invocation is caused by the reference field 'p'. The serializer of class 'Derived2' recursively calls another serializer, whereas the call site specific alternative can eliminate that recursive call if heap analysis

```

class Base {}
class Derived1 extends Base {
    int data;
}
class Derived2 extends Base {
    Derived1 p{1} = new Derived1();
                                     Allocation num. 1
}
remote class Work {
    void foo(Base b) {}
    void go() {
        Base b1{2} = new Derived1();
                                     Allocation num. 2
        foo(b1{2});
        Base b2{3} = new Derived2();
                                     Allocation num. 3
        foo(b2{3});
    }
}

```

Figure 5: Source code example.

```

//NOTE: Derived1 is inferred by compiler analysis!
void marshaler_Work.go.1(Derived1 s) {
    Message m = new Message();
    p.writeInt(s.data);
    m.send();
    delete m;
    wait_for_return_value();
}
//NOTE: Derived2 is inferred by compiler analysis!
void marshaler_Work.go.2(Derived2 s) {
    Message m = new Message();
    p.writeInt(s.p.data);
    m.send();
    delete m;
    wait_for_return_value();
}

```

Figure 6: Pseudo code generated for Figure 5 using call site specific code generation.

```

void Work.go() {
    b = new Derived1();
    marshaler_Work.go.1(b1)

    b = new Derived2();
    marshaler_Work.go.2(b2);
}
// compiler inserts this method into class Derived1:
void Derived1.serialize(Message m) {
    write_type(this);
    write_int(data);
}
// compiler inserts this method into class Derived2:
void Derived2.serialize(Message m) {
    write_type(this);
    p.serialize(m); // note: recursive call
}
// NOTE: Derived1 is inferred by compiler analysis!
void marshaler_Work.go.1(Derived1 s) {
    Message m = new Message();
    s.serialize(m); // note: method call
    m.send();
    delete m;
    wait_for_return_value();
}
// NOTE: Derived2 is inferred by compiler analysis!
void marshaler_Work.go.2(Derived2 s) {
    Message m = new Message();
    s.serialize(m); // note: method call
    m.send();
    delete m;
    wait_for_return_value();
}

```

Figure 7: Pseudo code generated for Figure 5 using class specific code generation.

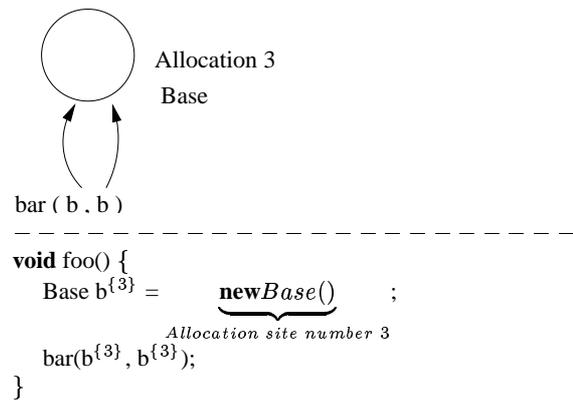


Figure 8: Dynamic Cycle detection is necessary.

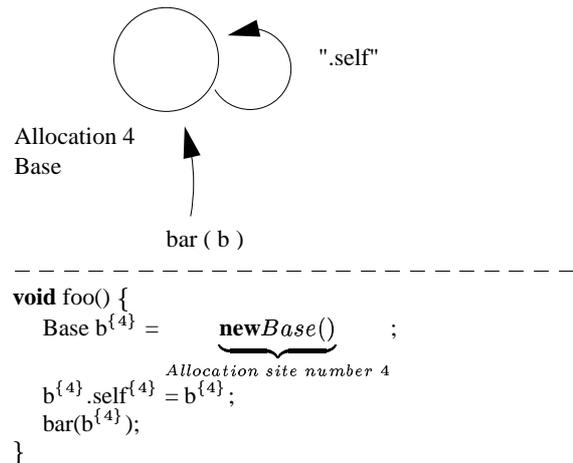


Figure 9: Dynamic Cycle detection is necessary.

guarantees that a reference will unambiguously refer to a certain type at a call site (it may be impossible to inline at another call site).

The deserializer for the unmarshaler of *Derived2.p* (not shown) also has to perform less work in case of call site specific marshaling: it knows what type of object to expect for *Derived2.p* and can directly create an object of type *Derived1*. The class specific unmarshaler would need to unparses the type information from the message and then locate the meta class for class *Derived1* to recreate the object.

3.2 Elimination of Cycle Detection

Serialization normally requires cycle tables/hashes to detect circular references. By properly analyzing RMI parameter passing and combining that analysis with heap analysis we can detect the absence of cycles at compile time.

Using the propagated allocation numbers and the heap graph, conservative cycle detection becomes very simple. Our (conservative) algorithm traverses the heap graphs rooted at the arguments of the call instruction and records the allocation numbers that it has already encountered. Once an allocation number is seen twice, we assume that the argument graph may contain a cycle that requires runtime cycle detection. If no allocation number is seen twice, there cannot be a cycle in the object graph.

Consider the program in Figure 8. Here the same object is passed twice to method 'bar' thus requiring a cycle check. The algorithm

```

remote class Foo {
    double sum;
    void foo(double a[]) {
        this.sum = a[0] + a[1];
    }
}

```

Figure 10: Escape Analysis Coverage: 'a' never escapes; the array object can be reused.

detects this cycle because both arguments 'b' and 'b' have the same associated allocation site number ($\{3\}$). After traversing the entire heap graph associated with the first argument (which will have resulted in a set of allocation numbers including the allocation number of the first encounter of 'b'), the algorithm will detect the cycle upon re-encountering the allocation number associated with 'b'.

The example in Figure 9 displays an object with a reference back to itself. Following and recording the references from the heap graph associated with parameter 'b' from 'bar' will disclose the cycle caused by the self reference.

Because the heap graph does not denote whether the '.self' field will reference the exact same object or whether it will reference another object allocated at the same allocation site (creating a linked list or a cyclic list), the analysis cannot distinguish between a linked list, a cyclic list, and the example in Figure 9.

3.3 Argument/Return Value Reuse Analysis

One part of the overhead incurred in RMI is the cost of object allocations caused by the de-serialization process of arguments and return values. Of course if the garbage collector and the object allocators are very efficient these costs are low but with sufficient de-serialization the costs can add a few percent to the total execution time.

Consider the following example, on a Myrinet [4] network, a single optimized RMI may cost as little as 40 microseconds and object allocation and deallocation costs about 0.1 microseconds. If the creation of an argument tree of a 100 objects can be saved by recycling the argument tree of a previous RMI, 10 microseconds can be saved, thus reducing the total RMI latency.

Object reuse between RMI calls is, however, only valid in the following scenario: if the argument (and, recursively, any of the objects the argument may refer to) does not escape the remote method.

Object reuse is implemented by performing escape analysis [3, 5, 18]. Escape analysis tells us whether or not an object escapes from a thread, for example, by assigning a reference to the object to a global variable or to a heap location. Escape analysis for RMI is slightly different from normal escape analysis as an object also escapes if recursively any of the objects it refers to escapes. An example of an RMI that is covered by escape analysis is shown in Figure 10. In this example the 'a' parameter is never assigned to a global variable nor is it assigned to a field of another object. Thus can the object safely be reused on the next invocation of 'foo'. In the example in Figure 11, the algorithm fails as 'a' is assigned to a static variable and thus escapes the thread: object 'a' cannot be reused.

4. GENERATED CODE WITH ALL OPTIMIZATIONS ENABLED

To demonstrate the effectiveness of the optimizations described thus far, let us examine the generated code for the simple benchmark application in Figure 12. Its performance will be examined later in the performance section. The generated (un)marshaller for

```

class Data { }
class Bar {
    Data d;
}
remote class Foo {
    static Data d;
    static void foo(Bar a) {
        d = a.d;
    }
}

```

Figure 11: Escape Analysis Coverage: 'd' escapes therefore escapes 'a' as well. Neither the Data-object nor the Bar-object can be reused.

```

remote class Foo {
    public void send(double[][] arr) { }
    public static void benchmark() {
        double [][] arr = new double[16][16];
        ArrayBench f = new ArrayBench();
        f.send(arr);
    }
}

```

Figure 12: 2D array transmission, 16x16 doubles.

the code in Figure 12 is shown in Figure 13.

Our call site specific marshaler/serializer optimization has detected the remote call to 'send' and generated the marshaler and unmarshaler *specifically* for that call. Function names of the generated (un)marshalers are therefore mangled with the containing function name and a sequence number.

Let's look at the marshaler first. The generated marshaler is optimized for shipping the 2 dimensional double array. Regular RMI's introspection or class specific serializers are more expensive. They would have to inspect the arrays to notice that the outer array contains references, follow them to get at the inner arrays which again need to be examined to determine that they contain no references. Only then can each sub array be examined to compute the size of the array's payload. For each array type information is pushed onto the network. Conversely, the unmarshaler needs processing power to interpret the received type information and to hash a type descriptor (a single integer in Manta-JavaParty) to vtable pointers to allow the de-serializing object to be instantiated.

Heap analysis shows that there can be no cycles in parameters to 'send' causing the omission of the cycle hash-table creation, deletion, and usages in the code of both the (de)serializers and (un)marshaler.

Object reuse analysis makes sure that the parameter to 'send' can safely be reused as it does not escape the RMI. The unmarshaler therefore maintains a global variable *temp_arr* to hold the root of the *arr* parameter.

Notice that *temp_arr* is set to null after its examination; this is to guard against multiple threads trying to execute the unmarshaler at a time. In the real code the unmarshalers are all protected with a lock that is acquired when the network message is accepted and released just before starting the RMI's Java code. After the Java code has finished, the lock is re-acquired and released after the message has been deleted. This ensures that at any time only one thread can drain the network as required by our communication software.

If an array size is mismatched the size of the cached array, a new array of the correct size is allocated. In most cases, however, as is also the case in this benchmark, the transferred arrays are of the

```

void marshaler(double[][]a) {
    message m = new message();
    m.append_int(a.length);
    for (int i = 0; i < a.length; i++) {
        m.append_int(a[i].length);
        m.append_double_array(a[i])
    }
    m.send();
    delete m;
    wait_for_ack();
}

void unmarshaler(Foo self, message m) {
    // keep the old RMI parameter between calls:
    static double temp_arr = null;
    int a;
    double [][] t;
    // the first RMI for this call site?
    if (temp_arr != null) {
        // see if the cached array is of
        // the right size.
        a = m.get_int();
        t = temp_arr.length == a ? temp_arr
            : new double[a];
        // guard against multithreading
        temp_arr = null;
        // read the sub arrays
        for (int i = 0; i < a; i++) {
            a = m.get_int();
            if (t[i].length != a) t[i] = new double[a];
            read_double_array(t[i]);
        }
    } else {
        // first RMI for this call site.
        a = m.get_int();
        t = new double[a];
        for (int i = 0; i < a; i++) {
            int a = m.get_int();
            t[i] = new double[a];
            read_double_array(t[i]);
        }
    }
    // call the user's Java code
    self.send(t);
    // set the parameter for the next RMI
    temp_arr = t;
    send_ack();
}

```

Figure 13: Pseudo code generated for the array benchmark from Figure 12.

```

class LinkedList {
    LinkedList Next;
    LinkedList(LinkedList Next) {
        this.Next = Next;
    }
}

remote class Foo {
    public void send(LinkedList l) {}
    public static void benchmark() {
        LinkedList head = null;
        for (int i = 0; i < 100; i++)
            head = new LinkedList(head);
        Foo f = new Foo();
        f.send(head);
    }
}

```

Figure 14: Linked list transmission.

same sizes as in previous RMIs.

5. PERFORMANCE

All tests were run on 1GHz Pentium III machines. Each machine is equipped with 256 Kbytes of L1 cache and 1 Gbyte of main memory. The machines are all running Linux (2.4.9), connected by a Myrinet [4] network. We use GM, an efficient user-level communication system, as our communication layer.

Because GM does not support interrupts, we use a blocking kernel thread (GM-poll-thread) that waits inside the kernel for messages. Messages are delivered to Manta-JavaParty's runtime system by upcalls.

GM has been modified to reduce the need for (process-level) thread switches by allowing the runtime system to poll for messages while the GM-poll-thread remains blocked inside the kernel. This allows most messages to be received and processed without kernel interaction by allowing the application to poll the network and subsequently extract and process pending messages. Whenever a network message has arrived and has been pending for more than 20 microseconds the GM-poll-thread will wakeup and process the message. A machine will poll the network whenever there are no runnable threads or while a thread has a data-request outstanding. Polling is performed instead of condition synchronization by the last thread in the system.

In each of the tables showing the performance of the benchmark applications the following legend is used:

- 'class' stands for class specific serialization which is better than dynamic introspection;
- 'site' means that call site specific optimizations were enabled (see Section 3.1);
- 'cycle' means that static cycle detection was enabled (see Section 3.2);
- 'reuse' says that argument and return values were reused if possible (see Section 3.3).

5.1 Microbenchmarks

We will now examine the effects of the three optimizations on two simple benchmarks: sending a linked list of 100 elements and sending a two dimensional array of doubles.

The code for the linked list benchmark is shown in Figure 14. We have run the benchmark routine 100 times and averaged the results to ensure stable measurements. The results are shown in Table 1. The versions with cycle detection enabled do not improve

Table 1: LinkedList: 100 elements, 2 CPU's.

Compiler Optimization	seconds	gain over 'class'
class	161.5	0
site	140.4	13.0%
site + cycle	140.5	13.0%
site + reuse	91.5	43.3%
site + reuse + cycle	91.5	43.3%

Table 2: 2D array transmission, 16x16, 2 CPU's.

Compiler Optimization	seconds	gain over 'class'
class	130.5	0%
site	110.0	15.7%
site + cycle	97.5	25.2%
site + reuse	103.0	21.0%
site + reuse + cycle	91.5	29.8%

execution time because the linked list may contain cycles. Object reuse, however, shows a large gain as per RMI there are 100 object allocations saved. Creating call site specific marshalers/serializers for the call to send the linked list increases efficiency much because a lot of network traffic is saved to transmit type information for each linked list node.

The code for the array benchmark is shown in Figure 12. Like the linked-list benchmark, the benchmark routine is ran 100 times and the times are averaged. The results are shown in Table 2. All optimizations help increase efficiency. The biggest gain is due to the generation of call site specific marshalers/serializers for which the compiler is able to remove much of the typing information that would otherwise be sent over the network.

5.2 LU

LU is an application from the SPLASH-2 benchmark suite [19]. LU factors a dense $n \times n$ matrix into two new matrixes B and N.

The application works by letting each machine update part of the matrix. After each machine has finished its part of the matrix, updates are flushed to machine 0 and a barrier is entered to wait for the other processors to catch up.

Table 3 shows the performance results for the two processor case. Table 4 shows the RMI statistics of the application These statistics were gathered on a separate run of the program with an instrumented runtime system. Class based serialization is slowest as expected. Enabling call site specific serialization and marshaling code helps the most. Eliminating cycle detection is the second biggest gain as all cycle lookups are removed from the code. The remaining two cycle checks are from two RMIs from the initialization of the Javaparty runtime system. As the garbage collector and object allocator used in Manta-JavaParty are already very efficient, the gains from object reuse are small ('site + cycle' versus 'site + reuse + cycle') even though the amount of objects allocated through deserialization is less than a quarter of what it was (from

Table 3: LU: runtime 1024 matrix, 2 CPU's.

Compiler Optimization	seconds	gain over 'class'
class	79.81	0%
site	69.23	13.2%
site + cycle	66.88	16.2%
site + reuse	67.28	15.6%
site + reuse + cycle	64.85	18.7%

Table 5: Superoptimizer: seconds for performing the exhaustive search, 2 CPU's.

Compiler Optimization	seconds	gain on 'class'
class	400.03	0%
site	373.22	6.7%
site + cycle	322.52	19.3%
site + reuse	375.47	6.1%
site + reuse + cycle	322.06	19.4%

348 MBytes to 87 MBytes). Objects allocated *not* due to deserialization are not included in this number.

The columns marked with 'local rpcs' and 'remote rpcs' denote how many RMI's were performed on local and remote objects.

The columns denoted with 'invocations' tell how many calls were made to serialization methods during the serialization process. A notable reduction has been made due to method inlining, some were however rejected for inlining due to method size of either the inliner or inliner.

In total, 18% is gained by enabling all optimizations, 12% of which is due to call site specific serialization code. The rest is due to cycle detection (3%) and object reuse (3%).

5.3 A Parallel Superoptimizer

A superoptimizer is a program that attempts to find the best possible equivalent of a given sequence of instructions by performing exhaustive search over all possible permutations of equal or shorter length (see [13]).

This version of a superoptimizer is based around a single producer thread that produces all possible valid permutations of instructions of up to three instruction length. Each machine runs a tester thread that accepts generated sequences from the producer thread and tests them against the searched for sequence for equivalence. The equivalence test is performed by executing both the search sequence and the permutation sequence with the same set of random input register and memory values. After the execution of both sequences the results are compared for equivalence. If both of the resulting register and memory states are the same, the sequences are deemed equal. Equal sequences are then added to a list which is presented to the user at program termination.

Because the generator is able to generate test sequences faster than the tester threads can test them, queues are inserted in front of each tester thread. The producer thread blocks whenever the queue for a given tester thread is full and unblocks whenever the tester thread has made space available. The producer distributes test sequences in a round robin fashion to the test threads.

RMIs are primarily used by the generator thread towards the tester thread to push test sequences. A test sequence consists of a program object, an instruction array object, and one to three instruction objects each containing three operand objects.

The compiler is able to analyze that the program object is cycle free and is thus able to remove all dynamic cycle checks (see Table 6). The programs themselves are pushed into a queue and are thus not eligible for reuse.

In total, 19% is gained by enabling all optimizations, 6.7% of which is due to call site specific serialization code, 12.7% is due to cycle detection (see Table 5).

5.4 A Parallel Webserver

The parallel webserver is organized as a simple master slave program. The master process accepts webpage requests and forwards the request to one of the slaves for retrieval. The slave searches for

Table 4: LU: runtime statistics 1024 matrix, 2 CPU's.

Optimization	reused objs	local rpcs	remote rpcs	new (MBytes)	cycle lookups
class	0	545.192	538.006	348.14	176.998
site	0	545.192	538.006	348.14	176.866
site + cycle	0	545.192	538.006	348.14	2
site + reuse	132.645	545.192	538.006	87.04	176.866
site + reuse + cycle	132.645	545.192	538.006	87.04	2

Table 6: Superoptimizer: runtime statistics, 2 CPU's.

Optimization	reused objs	local rpcs	remote rpcs	new (MBytes)	cycle lookups
class	0	5250554	5250570	1101	52499065
site	0	5250554	5250570	1101	52499082
site + cycle	0	5250554	5250570	1101	17
site + reuse	2	5250554	5250570	1101	52499082
site + reuse + cycle	2	5250554	5250570	1101	17

Table 7: Webserver: μ s per webpage retrieval, 2 CPU's.

Compiler Optimization	μ s per Webpage	gain on 'class'
class	47.7	0%
site	39.2	17.8%
site + cycle	30.9	35.2%
site + reuse	38.0	20.3%
site + reuse + cycle	29.7	37.7%

the webpage in a hash-table and returns it to the master to forward it to the client. Communication in the webserver application centers around a single RMI: `page = server[url.hashCode()].get_page(url)`. The compiler is able to prove that both the returned webpage and the string parameter are cycle free and is therefore able to remove all cycle checks (see Table 8). The remaining three cycle checks are due to program initialization and the initialization of the JavaParty runtime system. The returned webpage and url string are both determined to be reusable objects further increasing the webserver's efficiency. With object reuse enabled *no* new objects are created after the first webpage has been retrieved (see Table 8, rows with '+ reuse').

Like LU, a notable reduction has been made in the number of invocations to serialization routines (more than half reduced) when applying call site specific marshaling.

In total, 37% is gained by enabling all optimizations, 17% of which is due to call site specific serialization code, 18% is due to cycle detection (see Table 7).

6. RELATED WORK

There have been several projects that have attempted to increase the efficiency of Java RMI. Some projects have improved upon the network protocols while others have improved upon the generation of the serialization routines.

KaRMI [15] is a project to create a more efficient RMI implementation by employing a more compact encoding of types over the network. The implementation described in this paper performs this optimization as well. Due to the layering of the SUN RMI implementation, there are multiple layers of buffering which the KaRMI implementation avoids thus increasing performance. Manta-JavaParty reuses the Panda [2] middleware layer for efficient communication and avoids the Java-IO layers completely.

Manta [12] is a Java environment to compile Java code directly to native code thus achieving high performance. This paper reuses the

Manta compiler to re-implement JavaParty and to implement the optimizations described herein. This paper, however, does not use the RMI implementation from [12] but a completely new RMI implementation. The original Manta-RMI implementation, uses the fact that the compiler knows the object layout to generate specialized serialization routines for each object type and the Panda [2] middleware layer for efficient communication.

In [11], Java-RMI's TCP based implementation is replaced by an implementation based upon UDP. Furthermore, the system they have developed allows objects to remain cached at the client to reduce the number of network traversals. Methods that can modify the state of an object need to wait until all other cached copies of the objects have been invalidated.

In [10], the (de)serialization routines are generated at runtime for each destination machine type. The purpose of this is to dynamically specialize the serialization routines to the differences between object and data layouts, allowing the de-serialization routines to be simplified dramatically. This also allows object creation during the de-serialization process to be skipped by directly using the objects in the network buffer after a small amount of preprocessing. Our object reuse scheme can be used in combination with their zero copy scheme for increased performance.

In [14], partial evaluation is used to specialize the serialization and marshaling code just after the stubs/skeletons have been generated. Many buffer bounds checks and buffer manipulation routines can then be optimized away. Generating call site specific marshaling as described in this paper is in part an application of partial evaluation. The compiler's information about the arguments to a remote call is used to specialize the code to serialize objects.

Flick [7] is a flexible and optimizing IDL compiler: it compiles a remote interface to a set of skeletons and stubs for a given language. An important feature of Flick is that it also optimizes the generated stubs and skeletons much as a normal compiler would. In our implementation the stubs and skeletons are generated directly by the backend into the intermediate code of the same backend with annotations to enforce serialization specific optimizations for them.

Instead of using RMI there are also several projects using DSM techniques to allow distributed execution with Java. The advantage is that the underlying DSM will perform all communication and no serialization/de-serialization is needed. One such a system is Java/DSM [20] which lets a Java implementation run on top of Treadmarks [9]. cJVM [1] and Jackal [17] follow the same approach as Java/DSM but are fine-grained DSMs where granularity is object based.

Table 8: Webserver: runtime statistics, 2 CPU's.

Optimization	reused objs	local rpcs	remote rpcs	new (MBytes)	cycle lookups
class	0	500.007	500.003	226.94	5.000.004
site	0	500.007	500.003	165.90	3.500.003
site + cycle	0	500.007	500.003	165.90	3
site + reuse	3.499.988	500.007	500.003	0.0	3.500.003
site + reuse + cycle	3.499.988	500.007	500.003	0.0	3

7. CONCLUSIONS

We have introduced and reviewed three compiler based optimizations for RMI:

- call site specific serialization and marshaling routines,
- removal of cycle detection from the serialization process,
- argument/return value object reuse.

Call site specific serialization routines enable the compiler to specialize the generated code to the data sent while enabling the other two optimizations to become more aggressive as well.

Removal of cycle detection code removes a lot of the dynamic hash-table lookups that influence the runtime of the serialization process. The applicability of the optimization is governed by the precision of the representation of the compiler's heap graph. Currently linked lists (containing no dynamic cycles) are mistakenly identified as having cycles. The effects of this optimization are most visible whenever there are many objects to be serialized per RMI.

Argument/return value object reuse analysis attempts to reuse the de-serialized objects if the de-serialized objects are of the correct type and size. The advantages of the analysis are twofold. Firstly, the memory space is reused causing better caching behavior. Secondly, the strain on the garbage collector and object allocator is reduced.

For non-trivial programs most gains come from cycle detection elimination and call site specific optimizations. The gains from object reuse depend on the efficiency of the garbage collector and the object allocator. As in current JVM's (and in Manta-JavaParty) the allocator and garbage collector are highly tuned, the gains are lower as expected and the gains are partly due to better caching behavior.

8. REFERENCES

- [1] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A High Performance Cluster JVM Presenting A Pure Single System Image. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 168–177, San Francisco, CA., June 2000.
- [2] R.A.F. Bhoedjang, T. Ruhl, R.F.H. Hofman, K. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A portable Platform to support Parallel Programming Languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 213–226, San Diego, CA., September 1993.
- [3] B. Blanchet. Escape Analysis For Object-Oriented Languages: Application To Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, Denver, CO, November 1999.
- [4] 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.
- [5] J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape Analysis For Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, Denver, CO, November 1999.
- [6] R. Cytron, J. Ferrante, B.K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [7] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, Las Vegas, NV, June 1997.
- [8] R. Ghiya and L.J. Hendren. Putting Pointer Analysis To Work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, San Diego, CA., January 1998.
- [9] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 Usenix Conference*, pages 115–131, San Francisco, CA, January 1994.
- [10] K. Kono and T. Masuda. Efficient RMI: Dynamic Specialization of Object Serialization. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 308–315, Taipei, Taiwan, April 2000.
- [11] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient Implementations of Java Remote Method Invocation (RMI). In *4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 19–36, Santa Fe, NM, April 1998.
- [12] J. Maassen, R. Van Nieuwpoort, R. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R.F.H. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.
- [13] H. Massalin. Superoptimizer: A Look At The Smallest Program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 22-10, pages 122–127, New York, NY, 1987. ACM Press.
- [14] Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 116–126, November 1997.
- [15] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings*

of the ACM 1999 conference on Java Grande, pages 152–159, June 1999.

- [16] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience (CPE)*, 9(11):1225–1242, November 1997.
- [17] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime optimizations for a Java DSM implementation. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 153–162, Palo Alto, CA., 2001.
- [18] J. Whaley and M. Rinard. Compositional Pointer And Escape Analysis For Java Programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, Denver, CO, November 1999.
- [19] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.
- [20] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *ACM 1997 PPOPP Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.