University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Dynamic performance optimisation of distributed Java applications

Kwok Cheung Yeung

March 2004

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering of the University of London

Abstract

This thesis describes a novel approach to automatically optimising the performance of distributed Java applications that make use of Remote Method Invocation (RMI) while preserving the original application semantics.

The key enabling optimisation is call aggregation, where the execution of remote calls is delayed for as long as possible on the client side until a dependency forces their execution. The delayed calls are sent over to the server as a single unit, along with metadata describing the remote calls. This reduces the number of network transfers, and the increased context information in conjunction with the metadata enables the application of cross-call optimisations such as data sharing and dead-variable elimination. Other optimisations include server forwarding, which can reroute data communications to exploit fast connections, and plan caching, which is used to reduce communication overheads. Experimental evaluation suggests that the performance of applications under these optimisations can be comparable to that of implementing the aggregation and forwarding optimisations by hand.

The Veneer virtual Java Virtual Machine (vJVM) is presented as a flexible platform on which to base the RMI optimisations by providing a level of control over an executing program close to that of a customised interpreter while still running on a standard JVM. Veneer can intercept selected methods of a program and delegate the process of execution to a user-defined *executor*, which is essentially a simple interpreter that may deviate from normal execution while executing the method if required.

There are many circumstances in which the optimisations might alter the semantics of a RMI program, and practical ways to detect and correct this are investigated. A simple logical framework on which to reason about the optimisations has been developed, and used to show that call aggregation is safe provided that certain conditions are met.

Acknowledgements

I wish to thank the following people for their help during my career as a PhD student at Imperial College:

- My supervisor, Paul Kelly, for his guidance and support throughout this work
- Sarah Bennett, for her help in testing Veneer and the DESORMI optimisations during the development process, and for her subsequent bug reports (which were often alarmingly long in length...)
- Olav Beckmann, for helping me getting settled into my first year as a PhD student
- The rest of the Software Performance Optimisation group, for all the discussions and reading groups that have considerably broadened my knowledge of computing
- My parents, for both moral and financial support throughout the years
- My uncle, for igniting my interest in computing and the scientific disciplines in the first place, and for the encouragement throughout the years
- The rest of my family, for simply being there for me
- The Sable research group at McGill University, for producing the Soot framework, which has saved me from enormous amounts of work
- My PhD examiners, Alan Dearle and Wolfgang Emmerich, for their advice on improving this thesis
- The Engineering and Physical Sciences Research Council (EPSRC), for financial support with a research studentship and grant (GR/R 15566)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The focus of the computing industry has changed dramatically during the past decade, from computers working as independent units to computers working together as part of a group to achieve some overall task. This move has been fuelled largely by the growth of the Internet, which has provided a wide-area network of unprecedented size and pervasiveness, offering new opportunities in many diverse areas. However, in order to exploit the Internet, one must develop distributed applications to use on it.

There are many abstractions available that make distributed programming easier by using familiar concepts from conventional single-machine programming, such as objects with callable methods. However, none of them are able to maintain the illusion for long in terms of performance, since invoking a method over a network is orders of magnitude slower. Developers must therefore make remote calls sparingly in order to achieve good performance, which typically forces applications into conforming to one of several performance-oriented design patterns, which may or may not be a natural fit for the application logic.

The goal of the work presented in this thesis is to change this, so that developers *can* write programs as if they were meant to run on a local machine, and program would be automatically and transparently transformed into one that is comparable in performance to an equivalent application that was designed with performance in mind from the start.

1.1 Programming for networks

The Internet is built on top of the Internet Protocol (IP), which provides a simple means for one computer to deliver data to another at a specified address, with additional protocols such as TCP providing the notion of a connection between hosts. Practically all modern systems have since adopted these protocols as their core network protocols.

These protocols by themselves already provide enough functionality for developers to write programs that can communicate with other programs running on separate hosts by means of raw data exchanges. However, this is still too low-level for some applications, especially those with less specialised and fixed purposes. This has led to a variety of abstractions that provide for easier development of general distributed applications.

One popular abstraction is that of a Remote Procedure Call (RPC), which provides the illusion of one process being able to call a procedure in a second process that is running on a different host. With the increasing popularity of Object-Oriented Programming (OOP)

and component-based programming, the concept of RPCs has been extended to the object domain, enabling the methods of objects to be invoked remotely. Examples of such systems are the Common Object Resource Broker Architecture (CORBA) [39], DCOM (Distributed Common Object Model) [41], .NET Remoting [61] and Remote Method Invocation (RMI) [63].

1.2 Problems with remote object programming

The creators of CORBA, DCOM, Remoting and RMI have put considerable effort into making calls to remote methods behave as closely as possible to calling local methods, but that does not mean that one can write a program exactly as if the remote methods *were* local.

One problem is that the semantics of remote calls differ from those of local calls in several aspects:

- There is always the possibility of network failure
- Arguments to remote methods are passed by value (since the remote method operates in a different address space), whereas local methods offer at least the possibility of passing the arguments by reference

These semantic differences are stated in the specifications, and the programmer is expected to handle these explicitly.

However, a potentially more serious problem is performance. The cost of a remote call may be several orders of magnitude greater than a local call, since in addition to the time taken for the callee to execute, there is also the time consumed by network latency and the time taken to marshal data across the network[1]. Whereas the semantic differences may necessitate localised changes or additions to an existing code-base, solving the performance problem may require the restructuring of the entire application.

Traditional OOP often leads to a proliferation of small methods and objects, with accessor methods (i.e. `get` and `set` methods) for every field, one object per conceptual entity, and so on. If this approach is taken with remote objects, then the result will typically perform poorly, since the network boundary is being crossed frequently. Performance can be drastically improved by reducing the number and size of network transactions made by the distributed application. This is normally done by structuring the application in ways that are covered later in Chapter 2.

The structuring of an application is obviously best done at design-time, before any code has been written. Restructuring an existing application is much more problematic, especially if it has already been widely deployed. It requires source-code access, is often tedious[2] and new bugs might be introduced into the code during the process.

[1] Marshalling is the process by which data is transmitted from one host to another.
[2] Automatic refactoring tools can help here.

1.3 The current state of distributed program optimisation

In many respects, the situation with remote-calls is analogous to the early days of structured programming, where procedure calls had a relatively high overhead, and developers wanting optimal performance either avoided them altogether in favour of gotos or subroutines, or used them sparingly. However, this is no longer a major issue, due to the vastly improved performance of modern processors and compilers. Fast modern CPUs with the aid of features such as branch prediction and speculative execution have dramatically reduced the overhead of procedure calls, while modern compilers, with features such as inlining and interprocedural data analysis, can often eliminate procedure calls altogether or at least remove the barriers posed to code optimisation by the call boundary.

Network speeds are also increasing, though at a considerably slower rate compared to CPU speeds. This means that the network overhead of remote calls will tend to decrease as time progresses, but communicating across networks is still far slower than local communication within a single system.

Most approaches to optimising existing distributed programs involve fine-tuning the remote call mechanism and the underlying communication protocol, so that each call takes as little time as possible. However, these optimisation schemes often have limitations, such as having a restricted context in which the optimisations are valid. These techniques are addressed in Chapter 2.

Relatively little investigation has been done into trying to apply the principles of modern optimising compilers to distributed programs as a whole, to improve performance at the application level by utilising knowledge of the structure and state of the programs running on all communicating hosts. So far, the closest attempts have been in the area of remote-object caching (see Section 2.2.5). There are a few possible reasons for this.

- Code visibility — in order to optimise code, one must have access to it. This is clearly not the case in a distributed environment, since the client never sees the body of the remote method that is executed on its behalf at any point. Similarly, the server is unaware of the context of the client in which the remote method is being invoked.

- Versioning — in a generalised distributed environment, there may be many different types of client programs, and many instances of each of those types. On the server, a given remote object might be accessed simultaneously by several different clients at any point in time. It would obviously not be acceptable to modify the behaviour of the remote object in order to optimise the performance for one particular client, or even for one particular *class* of client, if that would break the other clients.

- Maintainer separation — clients and servers might not be created and/or maintained by the same parties. This means that proposals for improvement from another party will often take a long time to be processed, with no guarantee of acceptance.

- Failure — local method calls are very reliable, and the possibility of unexpected failure can essentially be ignored. However, with remote calls, the possibility of sudden failure is always present, and must be accounted for. This means that remote call

optimisations must respect the original behaviour under failure conditions as well as normal execution, limiting the possible scope for optimisation.

The current state of affairs is clearly undesirable from the software engineering perspective — in order to achieve good performance, the design of distributed applications must revolve around the performance issues, rather than issues such as maintainability, simplicity and clarity. Extra effort is also needed on the part of the developers, since they must spend time in learning and applying high-performance design patterns.

At present, these problems are tolerated in the developer community. This is because most distributed applications are either developed in-house, or have a narrow, clearly-defined role to play, thereby limiting the number of remote methods that need to be provided. However, this may well change in the future as concepts such as web services and grid computing gain wider acceptance, and servers begin to offer a wide range of services to the public. A client may well wish to compose different services from different servers together to achieve some overall task. In that scenario, the current conditions will eventually break down, and the performance and maintenance problems will come into the foreground.

1.4 Goals

The ultimate goal of this thesis is to do for distributed programs what optimising compilers have done for local programs — so that developers can, within reason, write code as they see fit, and the compiler will automatically modify the code to achieve a high level of performance without changing the overall behaviour. To this end, the following problems are investigated:

- Reducing the number of times that the network boundary is crossed
- Reducing the amount of data that is sent over the network
- Routing around slow connections
- Overcoming the optimisation barrier posed by the network in order to allow code optimisation to take place
- Maintaining the semantics and security of the original code as far as possible

1.5 A new approach

Java [6, 38] Remote Method Invocation has been chosen as the target for optimisation primarily because it is much smaller, simpler and cleaner when compared to technologies like CORBA or DCOM, and hence is easier to work with and manage. Furthermore, it forms the base upon which Enterprise JavaBeans [65] (Sun's distributed component-based architecture) is built, which is becoming increasingly important in commercial applications. However, the principles that are explored in this thesis should be applicable to other distributed architectures as well, although the dynamic nature of the optimisations make similarly dynamic systems such as Java, Python or .NET more suitable as implementation platforms compared to static systems such as CORBA programs written in C++, which will have to rely heavily on static source-to-source transformations instead.

The core concept is to intercept each remote method invocation made by the client. Instead of executing it immediately, it is placed into a queue of pending remote calls instead, and execution continues. The queue continues to build up until some local operation leads to a condition that necessitates the execution of the delayed remote calls (for example, if the local code uses a value defined by one of the local calls). At that point, the client dispatches the delayed calls as a group to their destinations, where they are executed. Normal execution of the local code may proceed after the remote calls have been executed.

This scheme has a number of benefits. Firstly, sending delayed calls in batches reduces the number of times the network is crossed, thereby decreasing the amount of network latency incurred. Secondly, it also increases the scope of the client context information available to the remote servers, by dealing with groups of calls at a time instead of individual calls. This opens up the possibility of optimising between remote calls by exploiting the relationships between calls.

In addition to the identity of the remote method and the call arguments, the client also sends additional metadata to the server regarding the relationship between the calls within the group, and that of the group with the remainder of the program. This enables the remote server to make optimisations that could not have been performed otherwise. For example, it is possible to avoid returning values that are not subsequently used by the client.

Other optimisations can also be performed, such as rerouting communications to exploit fast connections, and caching frequently occurring groups of calls on the server so that they can be quickly executed when the client encounters the same group of calls again.

1.6 Contributions

The following original contributions are made by this thesis:

- A new approach to automatically optimising distributed applications using Java RMI is presented, based on the principles of delayed evaluation.
- A prototype implementation of the new RMI optimisations was designed and built. A novel optimisation framework was created to support the implementation of this prototype.
- The effect of the optimisations was evaluated using the prototype on a selection of test applications.
- A number of issues are identified, that may lead to semantic problems when the RMI optimisations are applied, as well as possible solutions.
- A theoretical framework on which to reason about the effects of the RMI transformations on the data flow of distributed programs is formulated, and the validity of one of the RMI optimisations is demonstrated using this framework.

1.7 Thesis outline

Existing work related to the Veneer virtual JVM and the RMI optimisations is surveyed in Chapter 2. Each major topic is examined in turn, and the relative advantages and

disadvantages of each are discussed. The relationship of the topic with respect to the work presented in this thesis is also discussed.

The new runtime optimisation framework that was developed to implement the RMI optimisations is presented in Chapter 3. The performance of the framework has been evaluated under some common benchmarks.

The new RMI optimisations are presented in Chapter 4. The various problems that may occur are laid out, along with possible solutions. The optimisations have been experimentally evaluated using a small suite of examples.

A theoretical framework on which to model the RMI optimisations is developed in Chapter 5. The effects of the optimisations, the problems that arise and their solutions are described in terms of the framework.

The thesis is concluded in Chapter 6 by summarising the main contributions of this thesis, and discussing potential future extensions to this work.

Chapter 2

Related work

This thesis covers a broad range of work, including code optimisation, software engineering, virtual machines and distributed systems. This chapter surveys existing work related to the major contributions of this thesis — the DESORMI optimisations, and the virtual JVM platform upon which the RMI optimisations are implemented. Since these are fairly disjoint topics, this chapter is split into two major sections to cover the work related to each of these subjects. For more information on the Soot framework used to implement Veneer and on Java RMI itself, please refer to Appendices A and B respectively.

Related work is grouped by the general approach taken. The approach is introduced, and the general advantages and disadvantages inherent to the approach are discussed, and contrasted with work covered by this thesis. Specific implementations of the approach are then presented.

2.1 Work related to Veneer

This section covers work related to the Veneer virtual JVM, which is described in Chapter 3.

2.1.1 Dynamic program optimisations

There are many different types of dynamic optimisations. For example, many optimisations are only valid under certain conditions. If these conditions cannot be determined statically, then the optimisation can still be used by guarding the optimised code with a runtime check, falling back to the original code if the check fails. One example of such an optimisation is partial evaluation [82, 83], where a function is specialised assuming that certain parameters take on particular values on entry. Naturally, the resulting function is only valid if this assumption is true at runtime, which requires a runtime check to be made before the specialised function is called.

Dynamic optimisation is also used to work around the limits of static data-flow analysis for more aggressive optimisations. Conventional data-flow analyses lose information at program branches, since the combining of data from the various branches must be done conservatively. For example, if one branch only uses a variable A, while the other branch only uses a variable B, then both A and B will be identified as being live at the branch point

despite only one actually being used at runtime. If the branch that will be taken is known beforehand, it may be possible to optimise the program more aggressively by knowing which of A and B are live. Sharma [84] uses this concept to perform accurate data prefetching.

More ambitious dynamic optimisations involve modifying a program on-the-fly. For example, the Sun HotSpot JVM [64] can generate compiled code containing inlined methods, but when classes are dynamically loaded, some of the inlinings may be invalidated. In this case, HotSpot must remove the inlined code immediately and revert back to unoptimised code, although it may be reoptimised later.

2.1.2 Metaprogramming

Metaprogramming is the creation of metaprograms, which are programs that operate on other programs. Compilers and interpreters can regarded as a type of metaprogram. Metaprogramming is an umbrella term that encompasses a large and diverse range of subjects, such as reflection and multi-stage programming.

Veneer is a metaprogramming tool, since it is used to modify the behaviour of existing programs as defined by the executors used to execute application code. As a result, a number of metaprogramming subjects are particularly closely related to Veneer, which are presented in this section.

2.1.2.1 Aspect-Oriented programming

A number of parallels can be drawn between the Veneer execution model and the aspect-oriented programming [45] world. Aspect-oriented programming (AOP) deals with the separation of cross-cutting concerns. For example, when adding support for logging, logging statements need to be inserted throughout a program at various points, which is error-prone and laborious. It is also lacking in modularity, which can be especially problematic if all the logging statements need to be changed in some way.

AOP modularises this process by introducing the concept of aspects as separate program units. An aspect defines the changes needed to implement a cross-cutting concern (known as *advice* in AOP terminology) and the program points that the changes need to be applied to (the program points are known as *join-points*, and are specified by *pointcut designators*). The process of applying the changes defined by an aspect to an application program is known as *aspect weaving*.

The process of fragmenting a method and introducing points at which the executor can regain control is analogous to establishing join-points in the program. The actions performed by the executor can be considered a form of advice, and the decision code in the executor is similar to the pointcut designators of AspectJ [24].

The main advantage of Veneer over existing implementations of AOP on Java is the control that Veneer provides over control-flow. AspectJ provides a limited form of method control-flow by using 'around' advice, where the advice is executed in place of the original point-cut, with the possibility of executing the original code using the `proceed` construct. However, AspectJ does not permit one to manipulate the code at join-points as a first-order object, which was needed for the RMI optimisations. In essence, blocks, in conjunction with the method state, form *closures* that can be passed around arbitrarily.

2.1.2.2 Reflective architectures

Guaraná [71] and metaXa [37] are reflective architectures for Java. Essentially, they provide a means of attaching *meta-objects* to objects, such that any accesses to and from an object are intercepted by the meta-object. The meta-objects can then decide what to do next. For example, if one invokes a method on an object, then the meta-object can perform tasks such as:

- Return a value, without ever calling the intended method
- Modify the parameters to the method before passing control to the method
- Allow the method to be executed but modify the result before it gets back to the programmer
- Redirect the method call to another object
- Redirect the method call to another method

An obvious application for Guaraná and metaXa would be in delaying remote calls, by attaching a meta-object to objects that are potentially remote stubs. However, the only time when meta-objects have control is when the object they are associated with is accessed in some way. At other times, they are powerless. This means that they are not powerful enough to arbitrarily insert force-points inside an executing method even if meta-objects were attached to all objects. This is because simple operations such as `a = b`, where `a` and `b` are both local variables, will not be intercepted since they do not modify the state of any objects. Also, they cannot radically alter control-flow (i.e. arbitrarily jumping from one point in a method to another), or generate new code except in the limited sense of composing existing code together. Furthermore, both systems rely on direct modifications to the JVM. This makes them platform-specific, which is especially undesirable in a heterogeneous distributed environment.

2.1.3 Automatic runtime program optimisation

Runtime platforms that automatically optimise the performance of programs running on them are now mainstream technologies, primarily due to the rapid rise of Java. They typically work by gathering information about a program as it runs, and using this information to focus program optimisations on the problematic areas detected. This process is typically transparent to the user, and requires no manual intervention.

The main problem is that the optimisations carried out by these runtime platforms are usually generic in nature and fail to exploit domain-specific knowledge, although this is not inherent to the approach. By contrast, Veneer does not perform any optimisations per se, but provides a convenient platform on which runtime optimisations can be implemented, which can be generic or domain-specific in nature.

2.1.3.1 HotSpot performance engine

HotSpot [64] is an advanced Java virtual machine developed by Sun that attempts to solve the performance problems of Java using dynamic optimisation. It can collect information

about a program as it is running, and make use of this to efficiently optimise the program on-the-fly.

One way in which this information is used is in deciding what code should be compiled. Since just-in-time compilation is a relatively expensive process, HotSpot starts execution in interpretive mode, and only applies compilation to code that is determined to be frequently executed, so that the improvement gained by compilation outweighs the cost of compilation. Another optimisation is the aggressive inlining of virtual calls, which eliminates method call overhead and exposes more opportunities for code optimisation. The inlined code can be 'de-inlined' when the type of object upon which the call is invoked changes.

The great advantage of HotSpot is that it is completely transparent to the user, and does not require the source code to a program. However, it is limited in the scope of its optimisations to fairly low-level optimisations such as common-subexpression elimination and dead-code elimination — higher-level optimisations such as that being done in this project are generally not possible. It is also highly platform-specific.

2.1.3.2 Dynamo and Mojo

Dynamo [9], DynamoRIO [15] and Mojo [18] are dynamic optimisation systems, with Dynamo running on PA-RISC based workstations running HP-UX, while DynamoRIO and Mojo work on IA-32 architectures running Windows 2000. They operate by executing a program, building up an instruction trace as they proceed. When a particular trace is identified as 'hot' (i.e. often used), the trace is compiled into a fragment and is run through an optimiser. When control later passes to the point marking the beginning of the hot-trace, control is passed to the optimised fragment instead. Fragments that pass control to a point that marks the beginning of another hot-trace are patched so that they jump directly to the next fragment in line instead of returning control to the runtime system.

The underlying principle is practically identical to Veneer, differing only in environment and usage. Dynamo and Mojo both operate at the machine-code level. Machine code is essentially unstructured in nature, while structured-programming is strictly enforced in Java bytecode. For example, no jumping between instructions in different methods is permitted. To obtain the same effect, a new class must be created with the two method bodies joined together and subsequently loaded into the JVM, as opposed to simply appending a jump instruction in Dynamo when linking fragments together. It is also not possible to truly change code that has already been loaded into the JVM, so this must be emulated by generating new code and pointing everything in the direction of the new code. Although the low-level nature of machine code makes program analysis difficult for Dynamo and Mojo, the freedom provided by machine code makes it relatively easy to implement efficiently. With Java bytecode, the situation is reversed.

Dynamo needs to process all the code that a program executes in order to ensure that control is maintained, since there may be constructs such as direct jumps to arbitrary locations, functions that never return, return-address rewriting, `longjmp`s etc. Veneer does not need to do this, hence intercepted code may call original code, and vice-versa. This is possible due to the rigid structure of Java bytecode, where the only interprocedural jumps permitted are method calls. This guarantees that all method calls are guaranteed to return eventually, via either a method return or a thrown exception, at which point control is re-

gained. Object fields are also left in place, so that unintercepted methods can find the data at the expected locations.

Also, whereas Dynamo and Mojo are primarily concerned with extracting the maximum performance out of software, Veneer also acts as an *enabling* architecture that permits programmers to do new things that they would not have been able to without it.

2.1.4 Nested virtual machines

The concept of implementing a virtual machine for one language on top of another is not original to this thesis. It is relatively uncommon though, due to the obvious performance penalty involved.

Examples of Java Virtual Machines implemented on top of another JVM are JavaInJava [87] and Rivet [16]. However, unlike Veneer, they attempt to manually simulate *every* aspect of a JVM, including memory allocation, multi-threading and garbage collection. Unsurprisingly, the result is extremely slow. Veneer attempts to avoid this by delegating as much as possible to the underlying JVM, taking control only when strictly necessary.

The Jikes RVM [4], which is also written in Java, takes a completely different approach in that it compiles the bytecode to machine code which is then executed natively rather than on an underlying JVM. This has the advantage of high execution speed, but sacrifices portability since the output of the compiler is highly platform-specific.

2.1.5 Interpreters

Gagnon [34] classifies interpreters into the following categories based on the method used to dispatch instructions:

- Switching — the interpreter consists of a large switch statement inside a loop. Each branch of the switch statement contains the implementation of one instruction. On each iteration, the opcode pointed to by the program counter is used to select which branch of the switch statement should be taken to execute the current instruction. The program counter is then updated, and the next iteration begins.

- Direct threading — the opcodes that constitute an executable program are replaced by the addresses of the code that implement the instructions. In the implementation of every instruction, after the instruction has been executed, the program counter is incremented, and an address is fetched from the memory location pointed to by the updated program counter. A jump is then made to the fetched address. In this model, each instruction implementation jumps directly to the implementation of the next instruction via an indirect jump.

- Inline threading — this is an extension of direct threading. A buffer is allocated for every basic block of the program. For every instruction in the block, the corresponding code that implements it in the interpreter is copied into the buffer. At the end of the basic block, dispatch code is added to use the contents of the location pointed to by the program counter as the destination of an indirect jump. The original opcode at the beginning of each basic block is then replaced by the address of the newly allocated buffer corresponding to the basic block. In this model, all instruction implementations

> within a basic block fall-through to the next instruction implementation, with the sole exception of the last instruction, which must use an indirect jump to the buffer containing the implementation of the next basic block.

Veneer is unusual in that it exhibits characteristics of all three dispatch models. It behaves similarly to inline-threaded interpreters in that multiple instructions can be grouped together and executed at full-speed with no extra overhead involved in moving between instructions. In fact, Veneer goes beyond inline-threading in that more than one basic block can placed into the same group — the user-supplied runtime policy is responsible for determining where the breaks between groups occur. In Veneer, such instruction groups are referred to as instruction blocks, or just blocks.

Veneer diverges from the threaded dispatch models when moving between blocks. Since Java requires all executable code to be inside methods, blocks are implemented as methods in new classes generated at class load-time. In order to execute the next block, the method encapsulating it must be invoked. However, if this is done from within another block, it may eventually lead to a stack overflow since the method activation records are not removed from the call stack until the end of the method being executed is reached. If the executed code contains loops, then the number of calls made is potentially unbounded.

This problem may be solved using tail-recursive calls, but Java does not currently support this. The workaround used in Veneer is to adopt a continuation-passing style, where each block returns an index that identifies the next block when it finishes. At the top level of execution implemented by a Veneer executor, there is a loop that on each iteration executes the current block and then updates the current block to its successor based on the returned index. This resembles the classic switching dispatch mechanism. However, the pointer to the next block is a reference to a polymorphic object rather than an index to an opcode. Blocks are therefore dispatched by a virtual call rather than by a switch statement, which is closer to the principle of threaded dispatch than to switching since virtual calls are a form of indirect jump. A similar approach was used by Meehan and Joy [62] to compile tail-recursive Haskell functions into Java methods that execute in constant space.

2.1.6 Conclusion

Veneer is a tool that provides an interpretive model of program execution, which facilitates the easy implementation of metaprogramming techniques by modifying the interpreter. This low-level view of a program permits a higher degree of functionality compared to other existing metaprogramming frameworks.

As an interpreter, Veneer uses a mixture of the switched and inline threaded dispatch methods due to the limitations of the Java programming model. This means that performance is much better than a pure switched interpreter, but is lower than the potential performance of an interpreter with true inline threading.

Since Veneer is used as a metaprogramming tool, it is only used to control the execution of a program. It makes no attempt to handle VM functions such as memory allocation or threading, relying instead on the facilities provided by the underlying JVM. This makes Veneer considerably faster compared to other nested virtual machines which attempt to emulate all aspects of the JVM.

2.2 Work related to DESORMI

This section covers work related to the DESORMI optimisations, which are presented in Chapter 4.

2.2.1 Design patterns

There are a huge number of ways to write a distributed application. However, programming practices that lead to good applications and to bad applications tend to occur repeatedly. These practices are codified into what are known as 'patterns' and 'anti-patterns' respectively. There are many works on design patterns, such as [5, 7, 59].

Design patterns play a vital role in enterprise development, since the design of an application has such a drastic impact on its performance. Examples of performance-related patterns are:

- Command objects — a command object is an object that contains a script along with some data that is to be executed by the server. The server needs to provide a remote method that accepts a command object, and when this method is called, the server executes the script to completion before returning. Since command objects can contain multiple commands, one can execute many commands on the server-side using only a single remote call. The IBM SanFrancisco project uses this approach to speed up remote calls [20].

- Session façades —the session façade pattern is probably the most commonly used design pattern of all. In a sense, it is the opposite of the command-object pattern — instead of the client sending the server a set of operations to be executed, the server offers a choice of predefined sets of operations for the client to choose from. Session façades may therefore be regarded as 'pre-bottled' batch scripts that are explicitly provided by the server. There are numerous advantages to using session façades. The 'scripts' are already situated on the server, so there is no need to transport them across the network explicitly — all that is required is the identity of the façade method. Also, since the façades are explicitly provided by the server and are limited in number, there are likely to be fewer security holes in the server than if arbitrary scripts were accepted from the client.

- Value objects —methods to access fields of an object (i.e. get and set methods) are very common in object-oriented programming in general, and compulsory in RMI since there is no way to set fields on a remote object directly. However, it is very inefficient to make many remote calls just to get or set the contents of multiple fields in the remote object. Value objects are essentially a specialised type of command object used to 'bundle' the set of data to be sent or retrieved so that multiple get or set operations can be performed using a single remote call.

- Session state — a remote object can store state between method calls so that the client does not have to resend previously sent data. The main disadvantage is that it adds additional complexity due to the need to manage the storage of state, especially when multiple clients access the same object.

These patterns operate on the same general principle — to reduce the amount of traffic over the network by lowering the number of calls made or the amount of data sent in each call. These are the same principles that the DESORMI optimisations are based around. Indeed, the DESORMI optimisations can be viewed as a way to dynamically restructure a sub-optimal program to implement these design patterns automatically.

2.2.2 RMI protocol optimisations

RMI was designed for operation across large, unreliable networks where the servers and clients may be updated at any time. This means that the RMI protocol is rather verbose in order to deal with the various situations that may arise. It is therefore possible to achieve better performance with RMI by reducing the amount of information sent on each call.

The main problem with this type of optimisation is that it inevitably reduces the resilience of RMI. It can also be argued that the relatively minor reductions in the amount of data sent are unimportant in the long term, since there is no practical limit on the amount of bandwidth one can have (provided that one is willing to pay for it), and the cost per unit of bandwidth is steadily decreasing in any case.

The DESORMI optimisations also aim to reduce the amount of network bandwidth used in addition to reducing latency. This is performed by making use of prior knowledge gained during previous remote calls. For example, call aggregation enables the sharing of data between calls, while plan caching makes use of call patterns that have been encountered before to effectively compress the remote plan description.

DESORMI shares the same weakness as the other examples of this approach in that it is more fragile than the original RMI mechanism, although it manifests in a different way. This is due to the networked hosts relying more on implicit knowledge of each other. For example, if the plan cache was corrupted on a server, it might execute the wrong set of remote calls in response to a request to execute a cached remote plan from a client.

2.2.2.1 Dropping support for unneeded RMI facilities

One example of RMI protocol optimisation is UKA serialisation [75]. This work was done in the context of using RMI as a communication mechanism between nodes for high-performance parallel programs, so during a single run of the entire parallel program, the code at each node will not change. Support for versioning may therefore be safely dropped within this context. Two main techniques were used for this:

- Slim type encoding — when an instance of a class is first serialised by the default mechanism, a description of the class layout is written (i.e. the names and types of the fields) so that the instances can be reconstituted when deserialised later. UKA serialisation skips this, reasoning that the same bytecode for the class should be available to both client and server. Instead, it only writes the textual name of the class, which is considerably shorter and faster than the complete type information.

- Partial resets — the output streams used to serialise objects are reset between remote calls, which flushes the object type information as well as the hash-table used to keep track of objects that have already been serialised. UKA serialisation resets only the

hash-table between calls, and not the type information. Type information therefore does not need to be sent again between calls.

UKA serialisation does not support automatic object serialisation, which is a major inconvenience. However, since the target is for high-performance programs, it would be better for the serialisation routines to be made explicit anyway to eliminate the reflective overhead required for automatic serialisation. This limitation is likely due to the order of fields returned by the `getFields` method in the reflection API being non-deterministic, and so the receiver will not be able to determine the order in which the fields were originally serialised. However, since UKA serialisation assumes that the classes are identical on server and client, it should have been possible to sort the fields lexicographically by name and type, and force serialisation and deserialisation to occur in that order.

Although UKA serialisation offers good performance, it cannot be recommended in a general, loosely-coupled distributed environment. Versioning issues *will* arise, since both clients and servers are constantly changing by upgrading their JVMs and class libraries, modifying their programs etc. Under these conditions, interoperability would break down sooner or later with the UKA serialisation approach.

2.2.2.2 Using less expensive network protocols

RMI is based on the TCP networking protocol, which offers a reliable transport channel between two endpoints. However, when operating on reliable networks, it may be more efficient to use a less reliable protocol such as UDP, since it has less overhead.

An implementation of this approach can be found in the KaRMI framework [69], which is a 'lean and fast' reimplementation of the JDK 1.2 RMI specification. Unlike the monolithic implementation found in the standard JDKs, KaRMI permits the implementation of the three main layers of RMI (stub, reference and transport layers) to be changed in order to best suit the architecture on which the program is running. It has several advantages compared with the implementation supplied by the JDK:

- Support for communications protocols other than socket communications built on top of TCP/IP
- Takes short-cuts when a remote-object residing on the same host as the client is called via RMI
- Reduced reliance on hash-tables
- Less debugging code
- Distributed garbage collection algorithm may be altered to suit the situation at hand

The ability to change the layers in KaRMI leads to several possible optimisations, one of which is to exchange the default TCP transport layer for one which uses UDP instead for a lower communication overhead. However, KaRMI cannot deal with applications that make use of the fact that the JDK implementation is built on top of sockets by explicitly giving the port numbers to which to send objects. Programs that rely on undocumented classes of the JDK RMI implementation will obviously be broken by using KaRMI.

Another approach is implemented by R-UDP [46], which is a modification of standard RMI rather than a complete reimplementation. It uses the unreliable UDP transport protocol as opposed to TCP, trying to exploit properties specific to RMI in order to produce an efficient communications protocol. In particular, it exploits the request-response nature of RMI — since a remote call will always produce a response, then the equivalent of the ACK signal in TCP that acknowledges the successful receipt of a number of packets may be 'piggy-backed' on top of the reply, thereby cutting down on the overall number of low-level signal exchanges.

Unfortunately, in practice R-UDP turns out to be about to be about twice as slow as standard TCP communications for making RMI calls. This was attributed to use of multiple threads to handle transmission, retransmission and acknowledgement, with the requirement for synchronisation between threads placing extra overheads on the system.

2.2.3 Specialising object serialisation to RMI

One of the reasons for the inefficiency of the RMI protocol is that the marshalling of data to and from the remote object is handled using Java object serialisation. The object serialisation API in Java exists separately from RMI, and as such is rather generalised, being designed for flexibility and convenience rather than performance.

For example, when automatic serialisation is employed, reflection is used to first discover the structure of an object, and then to write data to or read data from the object in an interpretive fashion. Even providing explicit `readObject` and `writeObject` methods does not avoid the use of reflection, since reflection is also used to discover the presence of these methods, and to serialise superclasses.

Since marshalling can take a substantial portion of the time required for an RMI call, especially when operating over fast networks, it makes sense to optimise the serialisation process with respect to RMI. However, although optimising serialisation can provide significant gains, it is also of questionable importance in the long term since the time taken to serialise data is proportional to the speed of the CPU and the quantity of data to be serialised. Since the speeds of CPUs are increasing at a far quicker rate than those of networks, the proportion of time spent in marshalling will become more and more insignificant as time progresses.

The DESORMI optimisations are orthogonal to this form of optimisation since DESORMI does not make any attempt to optimise the underlying serialisation mechanism, although it attempts to use it as efficiently as possible. Instead, it focuses on improving the pattern of communication patterns between network hosts, using standard RMI to communicate when necessary.

2.2.3.1 Manta

An example of specialising object serialisation is Manta, which is a native compilation system for Java that among other things contains an efficient implementation of RMI [58, 57]. In keeping with the theme of native compilation, the majority of code in the critical path of RMI is precompiled as well.

Manta uses the Panda library to handle low-level communication rather than the multi-layered streams system provided by the Java library. At compile time, marshalling code is

generated for all available classes that are used as arguments to remote calls. This code explicitly reads/writes the contents of the class from/to the buffers provided by Panda, thereby providing automatic object serialisation without the overhead of reflection incurred by the default Java mechanism. The Manta equivalent of method stubs and skeletons also read and write directly into Panda buffers, making use of the compiled marshalling code to serialise the arguments of method calls.

Marshalling code is generated at runtime only for classes that have been dynamically loaded. This only needs to occur once for every new class, since the generated code is retained. Note that dynamic inspection of the class does not occur at any point during an RMI call regardless of whether the class was dynamically loaded or not.

Manta also offers some other RMI optimisations:

- Arrays of primitives are copied into message buffers using a direct memory copy.
- Remote methods that can be conservatively determined to be non-blocking are serviced by the thread that receives incoming connections rather than delegated to a new thread to eliminate the overheads of context switching.
- Full type information for any class is only sent once to a given host. On subsequent sends, a short, host-specific type ID is sent instead.

One weakness of Manta RMI is that it can only reach its full potential when communicating between Manta systems. Manta *does* provide interoperability with standard Java RMI, but most of the benefits are lost. Manta is not in widespread use, but this should not matter for running tightly-coupled parallel programs. However, as with most Java implementations built from scratch, Manta implements Java 1.1, which is far behind in terms of functionality compared to the latest offerings from Sun and IBM, and will limit its acceptance.

2.2.3.2 Optimising CORBA

The work done on optimising CORBA by Gokhale and Schmidt [36] relies upon improvements to the algorithms used in the free SunSoft implementation of the CORBA IIOP (Internet Inter-ORB Protocol). Improvements include:

- Inlining of frequently called methods
- Precomputing and storing frequently-used information
- Specialising generic methods with regard to caller context
- Eliminating needless waste
- Splitting large general functions up into smaller, specialised ones for better cache use

This main problem with this work is that it is far too implementation specific, since the improvements made apply to that one implementation and no others, although the principles are widely applicable. Indeed, it seems more like an exercise in program optimisation in general, since the optimisations are not specific to the distributed context in which the program will be used.

2.2.3.3 Flick

Flick [30] is an IDL compiler that attempts to speed-up remote calls by applying techniques that are found in optimising compilers for traditional languages such as Fortran or C, plus a few domain-specific ones, to the compilation of stubs and skeletons. It performs optimisations such as:

- Avoiding unnecessary tests for sufficient buffer space by analysing the storage requirements of messages at compile-time
- Efficient management of memory allocated for parameters
- Block-copying instead of component-by-component copying
- Transport specialisation
- Code inlining

2.2.3.4 Compiler optimised RMI

A recent paper [91] describes an approach using a form of heap analysis [35] to optimise the performance of RMI as used in JavaParty [76] in three ways:

- If the analysis can detect the exact type of a call parameter, it can generate specialised inline code to marshal it
- If it can be proven that there are no cyclic references in objects passed as arguments, then cycle detection code is removed from the serialisation process
- Space allocated for arguments and return values from previous remote invocations is reused if escape analysis indicates that the space does not escape from the caller thread

The first two optimisations are per-call optimisations that cut down on the amount of work that needs to be done to serialise call arguments. The third optimisation is interesting in that it is a rare example of a RMI optimisation that spans more than one call. However, it is far less ambitious than the optimisations presented in this thesis in that it only implements a form of pooling — it only reuses space allocated for objects, and not the actual *value* of the objects.

2.2.4 Asynchronous remote calls

One obvious way of improving the performance of distributed programs is to employ parallelism, where the client performs some other task while waiting for a remote call to finish. Systems such as .NET Remoting and CORBA explicitly provide support for this, while RMI does not. However, the same effect can be produced in an ad-hoc manner by using the standard Java mechanisms for multi-threading.

The main conceptual difference between this and the DESORMI optimisations is in the way they deal with latency. By using asynchronous calls, latency is hidden by performing work in time that would otherwise be wasted. For this to be effective, there must be work available locally that can be done while the remote call is in the process of executing, since the latency is still present. However, this local work must generally be independent of the

work being done remotely. Such work cannot always be found, and in any case requires explicit code rescheduling on the part of the application developer, which may complicate the structure of the program. If no local work can be found to execute in parallel with the remote call, then asynchronous calls are usually slower than synchronous calls due to added administrative overheads.

By contrast, the DESORMI approach actually reduces the overall amount of latency incurred, rather than hiding the consequences of it. It does not rely on having work that can be done independently of the remote call for efficiency, although it can also exploit independent local code by aggregating calls after it, which helps to reduce latencies further.

The relative performance of the two approaches depends very much on the application being optimised. If useful local work can always be found during a remote call, then asynchronous calls will perform better since DESORMI is inherently serial rather than parallel. However, if parallelisation is not always possible, then DESORMI will perform better since the time wasted due to latency is reduced. A scheme to merge the two types of remote call is proposed as future work in Section 6.2.12.

2.2.4.1 Asynchronous RMI

Asynchronous RMI (ARMI) [79] is an RMI variant which provides explicit support for asynchronous remote calls. This is done by the client explicitly providing a *mailbox* into which the server can drop completed results into. Every remote call results in a *receipt*, which acts as a key into the mailbox. The server uses it to deposit the finished result of the call, while the client can use it to retrieve the result.

ARMI is an explicit mechanism that modifies the semantics of RMI, which may make it unattractive to move existing RMI programs onto, since the changes that need to be made might not be obvious. A trivial way of maintaining semantics would be to explicitly wait after each call for the result to arrive in the mailbox before continuing, but this has been shown to behave worse than standard RMI due to the extra administrative overhead. However, it may provide a good point from which to start.

2.2.5 RMI object caching

The concept of caching a remote-object locally is covered in the same paper as R-UDP [46]. The basic idea is to keep copies of previously accessed remote-objects on the same same host as the callee, such that subsequent remote calls will be received by the cached copy instead of the 'true' remote object.

When a write occurs to a local cached object, the client communicates with the server hosting the real remote object, which in turn sends out invalidation messages to all hosts that have a cached copy. When the invalidation is complete, the client may continue with its operation. Changes are *not* written back to the server. When another client attempts to access read invalidated data, it makes a request to the server, which may have to fetch it from the client that holds the latest state of the object.

This works well provided that most operations on cached objects are reads. A write operation incurs high penalties for all users of the cached object, since the client has to wait for invalidation to finish before proceeding. The first request for invalidated data will also incur an extra delay as the server fetches it from the client performing the update.

Essentially, extra timing dependencies have been introduced where there were none before, with performance limited by the speed of the slowest link. Also, this work does not appear to consider the size of the object state in question. If the object state is large, then the cost of transporting it around may outweigh the benefits gained by having a local copy.

A later implementation of remote-object caching [29] deals with this problem by implementing the notion of *reduced objects*, where only a subset of the remote-object state is cached on the client. The subset that is cached depends on the properties of the invoked methods — for example, if a called method only accesses immutable variables, then those variables can be cached on the client without needing to deal with consistency issues.

Other potential weaknesses of object caching include:

- It takes time to transport bytecode across a network and to load it into a virtual machine. This time may outweigh the speedup gained by local execution if the cached object is rarely used.
- The balance of work between the client and server is drastically changed, with the client taking on more work. If the original purpose for distributing an application was to offload work from the client, then moving it back would obviously defeat the original intent.
- The host that contains the most up-to-date state of a remote object will shift around as clients write to the object. Moving the master copy of an object onto a potentially unreliable client is dangerous, since the client may fail, causing the most recent state of the object to be lost. A malicious client may even deliberately corrupt the state in an attempt to disrupt the overall system.
- Clients will be able to inspect the implementation of the remote object, which may be a problem if the implementation contains sensitive code.

2.2.6 Performance tuning systems

One of the best ways of optimising the performance of an application is to profile it first. Profiling provides information regarding program behaviour such as time spent in different methods or memory usage, which helps developers to efficiently find the location, severity and type of performance problems in a program. There is a large body of work in this area, and profiling tools are also widely available. Examples of commercial profiling tools for Java are JProbe [44] and Optimizeit [73].

However, while profiling tools can help to locate performance problems, they cannot actually solve them, since they are primarily programming aids. A further development of profiling is to use the collected information to automatically optimise a program as it runs. Systems exist that perform this at various levels of abstraction.

At one end of the abstraction scale are high-level systems such as AutoTune [28], which collect performance data for an application such as a web server via a system of monitoring agents. These agents monitor the workload and performance level of an application, and modify adjustable parameters of the application to dynamically optimise performance as the workload changes. By using a feedback system where the consequences of a parameter change are correlated with its results, it is possible for such systems to learn how to maximise the performance of the application.

At the other end of the scale are systems that operate at low levels of abstraction, such as Dynamo and the Sun HotSpot JVM discussed earlier in Section 2.1.3. These typically insert profiling hooks into the program code to measure certain performance characteristics, and after gathering enough information, they modify the actual code executed by the program in order to improve performance based on the measured information.

The high-level approach has various advantages. It is usually safer, since it operates on a predefined set of parameters, and so any mistakes made by the optimisation system are unlikely to be disastrous. The tuning parameters are also likely to be domain-specific with a well-defined effect. However, some of the burden of optimisation is now placed on the developers of the application since they must provide these tunable parameters.

The low-level approach has the advantage in that it is usually transparent from the point of view of the developer, who does not need to do anything special to enable optimisation. However, generalised systems such as HotSpot are limited in the scope of their optimisations since they tend to be generic and conservative in nature.

At present, the RMI optimisations covered in this thesis do not rely on any form of profiling. Instead, the optimisations are applied at every available opportunity. However, it is possible to extend at least the server forwarding optimisation to make use of profiled information. This is discussed as future work in Section 6.2.6.

2.2.7 Delayed execution

The general concept of automatically aggregating a large number of small operations into a smaller number of large operations for more efficient processing is very old and ubiquitous, occurring at all levels in a computer system from the underlying hardware to the operating system in the form of buffering. However, while it is fairly easy to automatically buffer low-level operations because they typically behave in a uniform manner and work within a fixed context, it is considerably harder to apply this to application programs due to the high-level dependencies that occur between instructions and the sheer number of possible instruction mixes.

There is also a connection with functional languages featuring lazy-evaluation such as Haskell, in that work is only done when required, and at the last possible moment, thereby gathering a higher degree of context information with which to optimise the operation. However, in functional languages, it is possible to avoid doing work altogether if it is not ultimately needed (which allows one to write elegant programs that involve operating on infinite data structures). Since Java is an imperative language with side-effects, it can be very difficult to determine whether operations are needed, and so all operations are executed eventually. The presence of I/O and multi-threading only serve to compound the problem.

The remainder of this section deals with existing work that shares the same basic philosophy of delaying operations for as long as possible to accumulate context information in order to find better optimisation opportunities at the application level.

2.2.7.1 Delayed RMI

Delayed RMI (DRMI) [60] was the predecessor of this project, and essentially had the same aims and the same basic technique as this project, but was much more primitive in many ways.

```
d_RemoteObj d_r = (d_RemoteObj) r;

d_int     d_x = d_r.f(new d_int(a));
d<String> d_y = d_r.g(d_x);
d<String> d_z = d_r.h(d_y);

d_z.claim();
force();

z = d_z.getValue();
```

Figure 2.1: An example program fragment using DRMI

The main difference is that it relies on the manual specification of where the delaying of remote calls should take place. An example of an RMI program transformed to use delayed RMI is shown in Figure 2.1. This code fragment calls methods `f`, `g` and `h` on remote object `r`, passing the output of each method as the input to the next. Some points to note are:

- The remote object `r` (which is assumed to be of type `RemoteObj`) is enclosed in a delayed wrapper of type `d_RemoteObj`. This wrapper is generated offline using a separate code-generation tool.
- The return values and arguments of the delayed methods must be in delayed form. This means that values that are not generated by other delayed methods must be wrapped. Wrappers for value types are explicitly defined, while wrappers for reference types are instantiated at compile-time from Generic Java [12, 13] templates.
- The claim method is invoked on `z` to indicate that it is the only result that is wanted. When the delayed calls are forced, only the value of `d_z` is updated from the server. This value can subsequently be fetched using the `getValue` method.

In addition to the changes to the client, the remote interface and the remote object need to be changed to inherit from `drmi.Remote` and `drmi.RemoteObject` respectively, and the stubs must be recompiled using a specialised stub compiler.

The main advantage of DRMI over the DESORMI framework described by this thesis lies in its explicit nature, since the developer is generally in a better position to know whether a delayed call sequence will be valid or not in a given context compared to the runtime system presented in this thesis, which has to make some conservative assumptions due to limited information. Even if delaying calls is invalid, the developer might be able to work around any changes in program behaviour due to the modified semantics. DRMI also has less overhead, and so may be capable of performing better.

The main disadvantage of DRMI also lies in its explicit nature, since it takes some effort on the part of the programmer to incorporate. The effects of callbacks, exceptions etc. are not handled by the DRMI library, and it will be up to the developer to compensate for these if necessary. The DRMI library is also intimately tied to the implementation of RMI provided by version 1.1 of the Sun JDK, and will not work with later versions without changes.

2.2.7.2 Batched Futures and Promises in Thor

Batched futures [11] are an optimisation that was developed on an object-oriented database system called Thor [55]. Results returned from Thor are either handles to objects or basic values such as integers. Batched futures take advantage of this existing programming infrastructure to implement a form of lazy evaluation, where calls that return a handle are delayed on the client side instead of being executed immediately. These calls return a special handle, called a *future*, which may be used in other calls. Futures act as 'stand-ins' for entities that might not have been created yet.

The batched calls are evaluated when a call that returns a basic value (which is the only way to retrieve raw data in Thor) is encountered, or a transaction is committed. At this point, the delayed calls are sent as a single group to the server to be evaluated, and the return value of the call that led to the evaluation is returned.

In general, the performance increases as more calls are batched, since the number of communications between client and server is reduced. However, Boyle reports the average batch size for a real-world benchmark to be at 2.33 calls per batch, for an average speedup of about 1.7, which is fairly low. One reason for this low batch size is due to the high frequency with which methods that return basic values occur, which is not surprising since these are the means by which actual information is obtained from Thor.

Zondervan [98] extends this system with promises [56], which applies the same technique to values as well. The Thor implementation uses tagged unions as promises — when the promised value is available, it contains that value and is tagged as such, otherwise it contains a future that refers to the value. Since value-returning methods were originally the triggering mechanism for batched futures, the programming interface must be changed to accommodate promises. This was done by creating alternative sets of access methods that accept and return promises instead of values. These alternative methods will not lead to the execution of the delayed calls, whereas the original value-returning methods will.

2.2.7.3 Data placement optimisation

Beckmann [10] uses concepts similar to those detailed in this thesis to implement a data placement optimisation framework for parallel architectures. It also relies on delaying the execution of potentially expensive operations until the results are definitely needed, such that opportunities for inter-call optimisations, that would otherwise have been lost, can be exploited.

However, this work differs from the DESORMI work in several aspects:

- The emphasis is different — the RMI optimisation tries to reduce the number and size of network transactions and to create opportunities for low-level optimisations where none previously existed, whereas the data placement optimisation tries to select an optimal data placement scheme for a set of operations that minimises the overall runtime cost, using a collection of prebuilt components
- The execution of delayed operations in the data placement work is completely data driven, whereas with the RMI optimisation, execution is both data and control driven due to the possible presence of side-effects in delayed operations

- The data placement work deals with data in terms in primitives and arrays only, whereas the RMI work also has to deal with more complex structures such as complex objects, aliasing, remote callbacks etc.

2.2.8 Conclusions

There is a large volume of work related to optimising the performance of RMI. The techniques tend to be variations of one or more of the following themes:

- Overlapping computation
- Compiler optimisation of the code path between the remote call and the remote method
- Caching state to avoid future network transfers
- Taking advantage of special scenarios to make assumptions that would not be guaranteed to hold in normal circumstances

Although most of these approaches do not conflict in principle, the implementations generally do, making it nearly impossible to use more than one at the same time without considerable reworking.

From the perspective of an end user who merely wishes to speed up existing RMI applications, the same issues show up repeatedly when employing the techniques discussed:

- Source code modifications required — many approaches require the source code to be modified. This may be a problem with end users who may not even have the source code. Even if the source code is available, it takes time and effort to locate and modify the code unless some automated tool is provided. It is also another potential source of errors.
- Outdated Java implementation — systems that have been built from the bottom-up such as Manta inevitably implement an extremely out-of-date version of Java (usually comparable to Sun JDK 1.1).
- Invalid assumptions — some approaches make assumptions about the environment in which the applications using them will be run in. For example, UKA serialisation discards most of the type information, assuming that classes do not change during any execution of an application. This assumption is clearly not true when applied to long-running servers that could run continuously for months or even years while the clients using it are updated regularly.
- Focus on individual calls — most approaches focus on making individual remote calls fast, with little regard as to the context in which that call is made. Although an application that makes many remote calls will speed up under such optimisations, it will still perform poorly compared to an equivalent application that makes fewer remote calls under the same optimisation.

The approach taken by this thesis to resolve these problems is to build a flexible runtime system into which RMI optimisations can be easily incorporated. This system should apply the optimisations automatically on applications running under this system. It should be

layered on top of the Java runtime rather than built from scratch, so as not to be left behind in terms of the JDK. The optimisations should also preserve the original RMI semantics as far as possible.

Chapter 3

The Veneer virtual Java Virtual Machine

This chapter presents the Veneer virtual Java Virtual Machine (vJVM), on which the RMI optimisations are built. Veneer provides a flexible framework for the programmatic modification of Java programs at run-time. It sits between the application and a standard JVM, intercepting the control flow of the application. The user of the framework can write simple interpreters called *executors* that execute a representation of the method body (known as a *plan*), deviating from the normal course of execution if necessary.

Some knowledge of low-level Java programming is assumed for this chapter. An overview of this subject is given in Appendix A.

Work that makes use of Veneer in the context of dynamic instrumentation has been presented at the Workshop on Performance Analysis and Distributed Computing (PADC 2002) in the paper 'Dynamic instrumentation for Java using a virtual JVM' [97], and at the 19th Annual UK Performance Engineering Workshop (UKPEW '03) [14] in the paper 'Search Strategies for Java Bottleneck Location by Dynamic Instrumentation'. The second paper has been published as a journal paper in IEE Proceedings — Software.

3.1 Motivation

Optimisations found in mainstream compilers such as GCC are usually of the static variety, where a block of code is replaced by another that performs better but produces equivalent results in all possible cases. Examples of such optimisations are strength reduction and common sub-expression elimination, which are thoroughly explored in the standard compiler texts [2, 68]. However, many modern optimisation techniques are dynamic, in that they require some form of runtime support.

The RMI optimisations that are covered later in Chapter 4 are inherently dynamic in several respects:

- A runtime check needs to be made to determine whether a call is truly remote
- The calls that are delayed and the points at which they are forced depend on the runtime path taken through a program

```
if (x) then
    delayedA = true;
else
    delayedA = false;
    A;

if (y) then
    delayedB = true;
else
    delayedB = false;
    B;
C;

if (delayedA) then
    A;
if (delayedB) then
    B;

D;
```

Figure 3.1: Rewriting a program to delay block A if x is true and block B if y is true

- Code sections containing adjacent remote calls are replaced on-the-fly for better performance

These optimisations require the embedding of control logic into the original program. The initial attempt at this used the static program rewriting approach, which is covered in the next section. However, the difficulties in implementing this eventually led to the virtual JVM approach, which is covered in detail in the rest of this chapter, starting from Section 3.2.

3.1.1 The static program rewriting approach

The first attempt to implement the RMI optimisations was based on the obvious approach of static program rewriting, where a program is used to rewrite the binary of the application program offline. The dynamic optimisations are incorporated into the application program, so that the optimisations will take effect every time the application binary is executed.

Since the concept of delaying remote calls is central to the RMI optimisations, this problem was tackled first. Consider the following program composed of four code blocks:

```
A;
B;
C;
D;
```

The actual contents of the blocks are not important here. Now suppose the program is to be modified so that A is conditionally delayed based on the value of some variable x, and B is similarly delayed based on some variable y, with both conditionals set elsewhere in the program. There is a force point (i.e. a point at which any remaining delayed blocks must be executed) between blocks C and D. The resulting code should look something like that in Figure 3.1.

```
if (x) then
    delayedA = true;
else
    delayedA = false;
    A;

if (y) then
    delayedB = true;
else
    delayedB = false;
    B;
C;

// Conditional force
if (shouldForce()) then
    if (delayedA) then
        delayedA = false
        A;
    if (delayedB) then
        delayedB = false
        B;

D;

// Unconditional force
if (delayedA) then
    A;
if (delayedB) then
    B;
```

Figure 3.2: Rewriting a program with conditional force points

The delaying of blocks A and B is implemented by placing guards around them so that they only execute if x and y respectively are false, and a flag is set indicating whether or not the block was delayed. At the points where the delayed blocks are to be executed, the corresponding flag is tested, and if true, the block is executed. This mechanism is simple to implement, but becomes increasingly complicated as the targeted program becomes more complex.

3.1.1.1 Conditional force points

The program shown in Figure 3.1 has a force point between C and D that always causes any remaining delayed blocks to be executed. Now consider what would happen if that force point was conditional on the result of some function *shouldForce* that cannot be computed statically. A new unconditional force point is also added after block D to force any calls that have passed block C. The program will now look like that shown in Figure 3.2. The main difference is that the flags noting the delayed status of the blocks need to be reset when the conditional force occurs, since failing to do so would result in the delayed blocks being executed twice when the second force point is reached. The flags do not need to be reset at the unconditional force point because it is implicitly known that A and B have been forced beyond that point.

The slight increase in complexity in itself is not a cause for concern. However, consider a program that contains x delayable blocks and y force points. At each force point, the code must be able to execute every possible block that may still be delayed at that point. In general, this means that for every block B that may be delayed, a force point F must be capable of executing it provided that:

- F is reachable from B
- The path from B to F is not dominated by an unconditional force point (i.e. it is not guaranteed that an unconditional force point was hit before reaching F)

This requires some extra program analysis. In the worst case, every force point must be able to execute every delayed block. Assuming each delayed block is of size bs, this could lead to the size of the program increasing by $x \times y \times bs$, not including the additional `if` statements and flag updates.

A simplification is to forgo the reachability analysis and simply make all force points capable of executing all possible delayed blocks, regardless of whether or not it is possible for a particular block to be delayed at that point. If this is done, then the flags must be updated at the unconditional force points too, since the implicit knowledge of possible remaining delayed blocks is no longer used. This will obviously result in the maximum possible size increase for the program.

3.1.1.2 Branches

Branches present an additional problem if force points lie within the branches. For example, consider the following program:

```
A;

if (cond) then
    B;
else
    C;

D;
```

Suppose that block A is to be delayed, and an unconditional force point is placed after block B but *not* block C. At the point just before block D, it is ambiguous as to whether or not block A was forced to execute. In essence, the unconditional force point has become conditional by being part of a branch, although the conditional is now part of the original program rather than being introduced. This means that the force point must also update the flag indicating the status of block A.

3.1.1.3 Loops

Loops present a far greater problem in that the same block may be encountered and delayed many times before being forced to execute.

```
while (cond) do
    if (delayA()) then
        queue.add('A');
    else
        A;
    if (delayB()) then
        queue.add('B');
    else
        B;

// Force delayed blocks
for id in queue do
    case (id) of
        'A': A;
        'B': B;
queue.flush();

C;
```

Figure 3.3: Rewriting a program to delay blocks contained in loops

Consider the following program:

```
while (cond) do
    A;
    B;


C;
```

Suppose that blocks A and B are conditionally delayed, and a force point is placed just before block C. Since the number of times that a loop executes is not always known beforehand, the number of delayed blocks is potentially unbound. Furthermore, it is not guaranteed that the same blocks will be delayed on each iteration. Instead of individual flag variables, a dynamic data structure such as a queue is needed to store the sequence of delayed blocks.

The code required to deal with the example program is shown in Figure 3.3. The queue is used to store the identity of a block as it is delayed. When the force point is encountered, the entire queue is processed by executing the block associated with each identity in turn.

3.1.1.4 State

In general, a block of code B acts on a set of data D. However, D might change between the point at which B was delayed and the point at which it is eventually executed. D must therefore be preserved along with the indicator that B has been delayed, and used in conjunction with B when it is finally executed.

3.1.1.5 Factoring out code blocks

It is generally undesirable to fully inline the blocks at each force point since that can drastically increase the size of programs. An alternative is to factor out the blocks into units that

may be called from multiple points. Java has support for this in the form of methods and subroutines.

If code is factored into a separate method, then local variables used by the block need to be passed into the method, and the variables defined by the block need to be copied back into the caller (since Java passes arguments by value[1] and does not support taking the address of local variables). However, the method call and the copying of variables to and fro incurs a certain amount of overhead.

The alternative is to place the code into a subroutine. This is much faster, but subroutines have numerous restrictions. Subroutines cannot be called recursively, and the operand stack must be at the same level for every subroutine call. Also, variables used by the subroutine must also be located at the same point in the operand stack or local variable slots for every subroutine call.

The method approach was ultimately used since it offers better flexibility, although at the expense of some performance.

3.1.2 The virtual JVM approach

By inspecting Figures 3.1–3.3, it can be seen that the programs incorporating the block delaying are considerably more complex compared to the original program, introducing new control flow and data structures.

It has proven frustrating to write and debug the program transformation code due to the extra level of indirection involved (since code is being written to generate other code, rather than being written directly). Turnaround time is long, and even simple errors might not be caught until the transformation is actually applied to a test program (which will also have to be re-compiled on the next iteration). Debugging information is usually destroyed or invalidated during transformation, severely limiting the usefulness of tools such as `jdb`.

It is also very easy to modify bytecode such that the JVM verifier would no longer accept it. The standard verifier is rather unhelpful with regard to debugging, since it prints out vague error messages such as 'incompatible argument to function' without any additional details such as the exact location of the error or the type of incompatible argument was being provided. The only information provided is the class and method in which the error occurs. The verifier error messages also appear to be undocumented. This situation can be improved somewhat by using the standalone JustIce bytecode verifier (included with the BCEL distribution), which provides more detailed error descriptions and error locations. However, this verifier often generates false negatives, sometimes rejecting even code that was generated directly by the standard `javac` compiler. This is due to differences in the interpretation of the virtual machine specification [54] by the creators of JustIce and `javac`.

The virtual JVM approach was initially developed to address these problems.

3.1.2.1 A new approach

Much of the complexity of the program rewriting approach is due to the need to 'wrap' control code around the existing code, of which there are innumerable variations. The

[1] This applies equally to object references, since the *reference* is passed by value, while the object remains in the heap.

```
while (currentBlock) do
    if (shouldForce(currentBlock)) then
        for block in queue do
            block.execute();
        queue.flush();

   if (shouldDelay(currentBlock)) then
        queue.add(currentBlock);
        currentBlock = getNextBlock(currentBlock);
    else
        currentBlock = currentBlock.execute();
```

Figure 3.4: Conditionally delaying and forcing code using the virtual JVM approach

solution proposed is to effectively turn the situation around, and wrap the existing code around the control code instead.

The virtual JVM approach arose out of the ideas developed in the static rewriting approach. The refactoring of code blocks into separate methods (Section 3.1.1.5) means that portions of the original method can be executed at will outside of the method body itself. This means that it is possible to reproduce the effect of a method by first storing the control flow information between blocks, and then writing a method that calls the method blocks sequentially, using the saved control flow information to decide which block to execute next.

This approach has several advantages:

- It is now possible to write the delaying and forcing mechanism directly in Java, as opposed to writing code that interleaves it with existing code, which considerably simplifies and accelerates the development process.
- It permits source-level debugging using the standard tools.
- It is easy to modify the behaviour of a block by generating a new version of it, and redirecting all control-flow to and from the original block to the new block.

The equivalent of the conditional delaying and forcing mechanisms demonstrated in Sections 3.1.1.1–3.1.1.3 using the virtual JVM approach is shown in Figure 3.4. The code is fairly straightforward. On every iteration of the main loop, a check is made to determine whether delayed blocks should be forced before the current block is handled. If so, then every delayed block in the queue is executed in sequence, and the queue is then flushed. The current block is then processed by checking whether it should be delayed. If so, then the current block is placed on the delayed block queue, otherwise it is executed immediately. The next iteration proceeds with the successor of the current block.

Unlike the previous examples, this code is generic, and is applicable to any program. It is identical in terms of functionality, but is also marginally less efficient because:

- Tests are being performed for every block, even when the result is known statically.
- Control-flow between blocks is explicitly handled, rather than relying on the normal flow of execution to advance from one block to another.

3.1.2.2 Veneer

The dynamic optimisation framework presented in this chapter is an evolution of the static program rewriting approach, which offers greater flexibility both for developers implementing optimisations on it and for users running applications on it.

Although considerable effort has been expended in developing this tool, it was a once-off effort that has paid off by making it very easy to experiment with dynamic optimisations, especially those involving some form of code-motion. The optimisation framework is referred to as a 'virtual virtual machine' because it outwardly behaves like a virtual machine, yet performs most of its work by delegating to the virtual machine that it is running on itself. It has been named 'Veneer', since it interposes a thin, programmer-friendly layer between the developer and the underlying JVM.

3.2 Overview

Veneer can be used as a drop-in substitute for a standard JVM, and is transparent from the perspective of the end user. When an application is first presented to the optimisation framework, it is analysed to determine the points at which execution needs to be intercepted in order to produce the required effect according to a user-defined *runtime policy*. Each method that contains at least one interception point is *fragmented* around these points. These methods are referred to as *intercepted methods*.

Fragmentation produces a number of data structures, which are illustrated in Figure 3.5. The method body is broken up into smaller callable methods called *method blocks*. A data structure known as an *execution plan* is built up that represents the control-flow graph of the original method, with the generated fragments linked to the nodes within this graph. Any additional data that can be calculated statically (such as variable liveness information) is added at this stage as *metadata*. All plans associated with methods in the same class are gathered together into a structure known as a *plan-set*.

The original body of an intercepted method is removed completely, and replaced with a *stub* that passes control to a user-defined *executor*. Executors are effectively simple interpreters written in Java. An executor can execute a method by traversing the execution plan, executing the code associated with each node that it encounters. Local variables are encapsulated in a *state* object that is passed between blocks.

The power of an executor lies in the fact that it can deviate from the standard behaviour. For example, it may access the local variables via the state object, inspect and modify code belonging to the method, communicate with other executor instances, modify the control flow of the method etc.

3.3 Veneer structures

Veneer operates using many different types of objects, which have been briefly introduced in Section 3.2. This section explores these structures in greater depth.

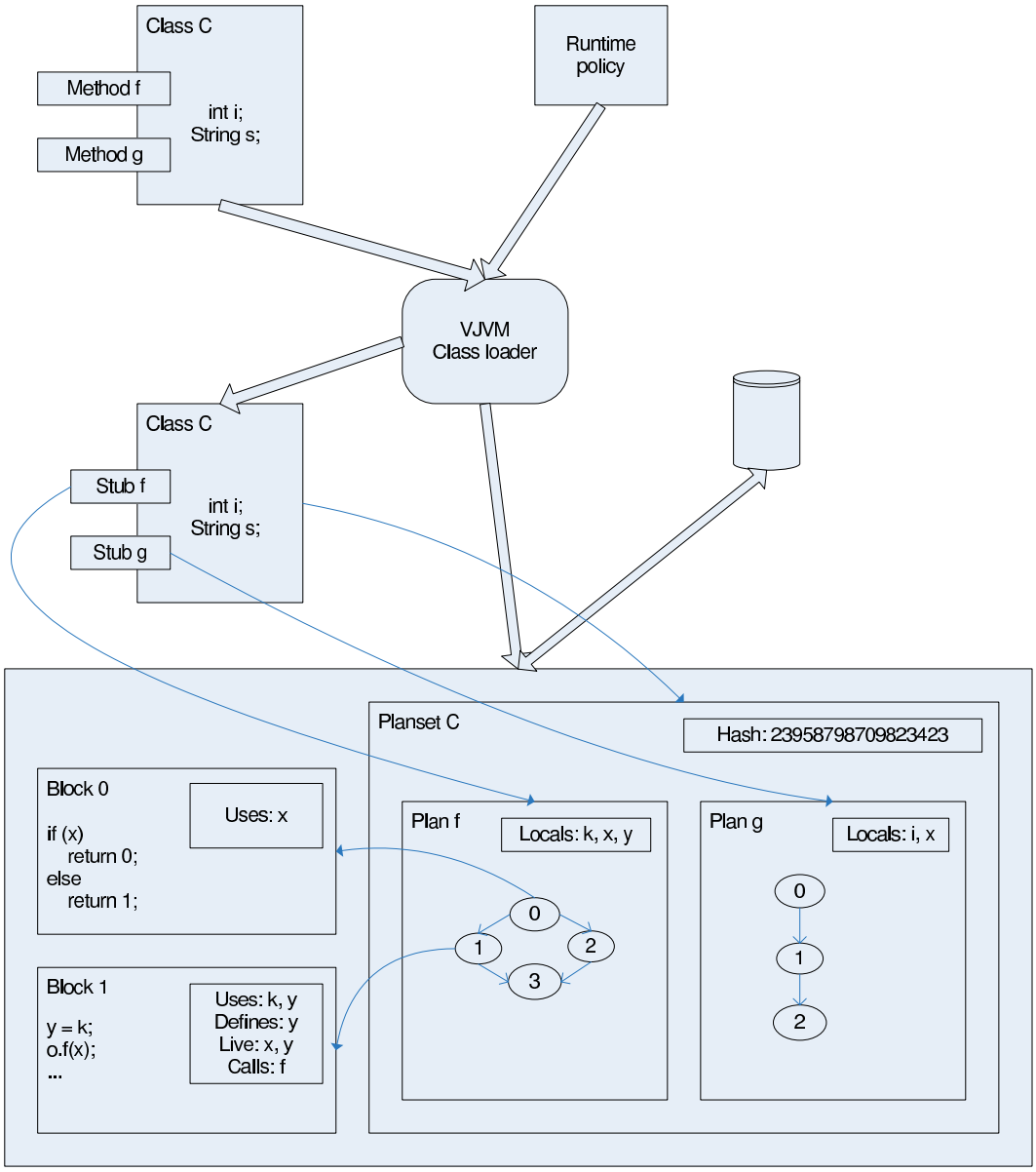


Figure 3.5: Data structures used by Veneer — the Veneer custom class loader accepts an application class and a runtime policy, and produces a plan-set for the class and a new class containing stubs which are used in place of the original methods. The plan-set is cached onto persistent storage. Each intercepted method is associated with a plan, which in turn is composed of code blocks. Each structure contains related metadata.

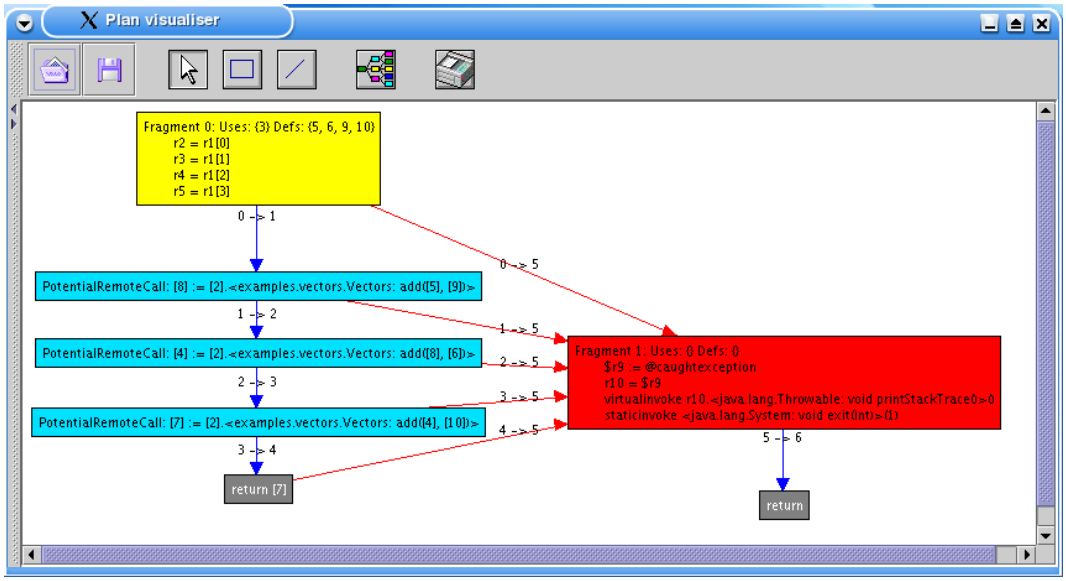


Figure 3.6: Visualisation of a Jimple plan fragmented for RMI optimisations — this represents a fragmented plan for the vector arithmetic example used in Section 4.8.2. Yellow blocks represent starting blocks, grey blocks represent finishing blocks and red blocks indicate exception handlers. Other blocks are coloured blue.

3.3.1 Plan-sets

A plan-set is a collection of plans that represent methods belonging to the same class. Plan-sets are used mainly to store metadata that concern entire classes. In particular, it stores a cryptographic hash of the processed class file that is used to detect changes in the original code.

3.3.2 Execution plans

The two main types of execution plan currently available are Jimple [90] plans and bytecode plans. Jimple plans represent the method body in the Jimple intermediate representation as generated by the Soot [88] framework (see Section A.3.1 for more details), while bytecode plans represent the method in bytecode form using classes from the BCEL library (see Section A.3.2). At present, the Jimple representation is much more mature compared to the bytecode version. The type of plan representation is determined by the fragmentation policy. The plan type also determines the types of executor that may be used to execute the plan — some may be generic, while others are specific to a particular representation. The RMI optimisations are specific to the Jimple representation.

Regardless of the representation, plans consist of a graph representing the method body, and metadata regarding the method. The arcs of the graph denote control-flow between the nodes, while the nodes themselves are executable code-blocks. A visualisation of a Jimple plan using a tool based on OpenJGraph [72] is shown in Figure 3.6.

Plans can be modified at runtime — modifying the arcs between blocks effectively changes the method control flow, and modifying the blocks effectively modifies the program code that will be executed.

3.3.3 Code blocks

Code blocks are encapsulated sections of code that may be executed at any time by the executor. They can have only one entry point, though they can have multiple exit points. Code blocks fall into two main categories — parameterised blocks and fragments.

Parameterised blocks usually represent single statements, such as a call to a method, an `if` statement, or an assignment statement, and are usually interpreted. The behaviour of these blocks may be modified at run-time by adjusting their parameters.

Fragments are fixed sections of code. These usually represent sections of code that the executor writer is not interested in modifying at run-time. Fragments may contain internal control flow (e.g. loops), provided that the scope of these statements lies entirely within the fragment itself. The code for fragments is executed directly by the underlying JVM, and so can run at full-speed.

3.3.4 Method state

The method state is used to hold the local variables used by a method between executions of the various blocks in a plan. The contents of a method state may be inspected and modified at any time.

The type of the state used by a method depends on the type of plan used to represent the method during execution. A Jimple state is manipulated as an array of objects, with one slot for each Jimple variable, while a bytecode state is manipulated as a stack and array pair to represent the operand stack and the local-variable slots of a Java stack frame. Both state types are derived from the `State` class and internally store the method state as an array of objects, with different sets of access methods for each state type.

3.3.5 Executors

A developer may specify how the execution of an intercepted method should proceed by writing a simple interpreter known as an executor. Developers are presented with a simplified execution model, and within the boundaries of this execution model and the Java environment, are free to do anything.

An executor is built by extending the abstract class `Executor`. The relevant parts of this class from the point of view of the developer are shown in Figure 3.7. The main task of the developer is to implement the abstract `execute` method.

When control reaches the `execute` method, various attributes of the base `Executor` class will have been initialised to their starting values. These may be retrieved with:

- `getCurrentPlan` — returns the plan representing the body of the method
- `getCurrentBlock` — returns the current block, which is initialised to the entry-point to the plan.
- `getCurrentState` — returns the current method state, which is initialised with the starting state (i.e. with the value of `this` and the arguments supplied to the method).

To make progress in executing the method, the `execute` method of the current block should be called, passing in the executor as an argument so that it can retrieve any information

```
public abstract class Executor {
    // To be implemented in derived classes
    public abstract void execute() throws Exception;

    // Accessor methods
    public Plan getCurrentPlan();
    public Object getCurrentState();
    public Block getCurrentBlock();

    // Sets the current block to the i'th
    // successor of the current block
    public void gotoNextBlock(int i);

    // Goto the exception handler for the exception e
    public void gotoExceptionHandler(ExecuteException e);

    // Returns true if the method has finished
    public final boolean isFinished();

    // Single-stepping support
    public final void setSingleStepping();
    public final void clearSingleStepping();
    public final boolean getSingleSteppingState();

    // Locking support
    public final boolean lockWasReleased();
}
```

Figure 3.7: The executor base class

needed to execute the block. This will execute the code represented by that block. The `execute` method, if successful, updates the value of the current block to the next block due to be executed by calling `gotoNextBlock` on the executor.

Between block executions, the executor can be programmed to do anything permissible by the Java environment. It also has full access to the Veneer runtime, and can introspect into any of its data structures. For example, it could:

- Inspect and perhaps modify the current state, which would have the effect of examining and modifying the local variables and intermediate results of the method as it is running.
- Set the current block to another block, which will have much the same effect as a `goto` statement would.
- Modify the current plan, which effectively changes the method implementation. This change takes effect immediately, and is seen by all executors that are in the process of executing the same plan.

3.3.5.1 Exceptions

If an exception is thrown by the fragmented program code, then an instance of `Execute-Exception` is thrown from the block, which is a wrapper for the exception that is actually

```
public class BasicExecutor extends Executor {
    public int execute() throws Exception {
        while (!(isFinished() || lockWasReleased())) {
            try {
                // Execute the current block
                getCurrentBlock().execute(this);
            } catch (ExecuteException e) {
                // Pass control to exception handler
                gotoExceptionHandler(e);

                // Propagate exception if no handler
                if (isFinished())
                    throw e.getException();
            }
        }

        return next;
  }
}
```

Figure 3.8: Structure of a basic executor that executes a plan without any changes

thrown. In this case, the next block to be executed should be set to the corresponding handler for the exception by calling `gotoExceptionHandler`. If there is a handler, the next call to `getCurrentBlock` should return it, otherwise the method is marked as being finished, and the wrapped exception should be thrown to be propagated by the method stub later.

An example of a generic executor that executes a method body without performing any other tasks is shown in Figure 3.8.

3.3.5.2 Threading

Since executors are called in place of the method represented by the execution plan, they run in the same thread as the original method — i.e. in the same thread as the caller. However, executors are strictly single-threaded objects in the sense that at most one thread can use any given instance of an executor to execute an intercepted method at any point in time. This is enforced by using a new instance[2] of the required executor type to execute the method plan every time a method stub is invoked.

The single-threaded model does not prevent the executor from interacting with objects that are accessed by other threads, or from spawning new threads itself to perform tasks in the background. However, it is essential to avoid synchronising on objects that are visible from the application program, since this may introduce new timing relationships that were not present in the original program.

3.3.6 Stubs

When a method is intercepted by Veneer, the original body of the method is removed completely. It is replaced by a stub that performs the following tasks in order:

1. Packs the formal parameters into a method state object.

[2] At least conceptually — in practice, executors are pooled for efficiency

2. Fetches a suitable executor for the method from the active policy .
3. Fetches the execution plan for the method.
4. Calls the executor, passing in the plan and the method state.
5. Propagates any thrown exceptions, or return the return value if none were thrown.

Stubs serve as entry-points into Veneer — by calling the stub, the body of the method that has been replaced by the stub is executed by the selected executor.

3.3.7 The runtime policy

The actions performed by the Veneer framework are determined by an active policy selected by the user. A policy is composed of three types of sub-policy. These are:

- The interception policy — this is responsible for determining which classes and methods are intercepted, and what method variants should be generated in the stub for an intercepted method.
- The fragmentation policy — this dictates how fragmentation occurs within the method (see Section 3.4.3). For example, it determines where parameterised blocks and fragments should occur, the type of plan to generate, metadata to attach to the blocks etc.
- The executor policy — this selects the type of executor used to run a particular plan.

3.4 Veneer runtime behaviour

This section presents the aspects of Veneer that lead to the generation of the plans and method stubs. These are presented in the order in which they are encountered from startup.

3.4.1 The bootstrapper

The bootstrapper is a simple program that takes the name of a class as its argument. It then instantiates an instance of the custom class loader `VJVMClassLoader`, and uses that class loader to load the named class. The `main` method on the new application class is then called via the reflection API.

3.4.2 The custom class loader

The custom class loader `VJVMClassLoader` is a sub-class of `URLClassLoader` that attempts to emulate the outward behaviour of the system class loader as far as possible.

When requested to load a class, the custom class loader reads the bytecode using the BCEL [26] library. It then determines whether the class, then each method within the class, should be intercepted using the interception policy. Unintercepted classes and methods are left unchanged by the custom class loader.

If a method is to be intercepted, the original body of the method is removed completely, replacing it with a stub that will execute one of several code variants, as determined by

```
public long f(Object a, int x) throws AnException {
    VJVMRuntime runtime = VJVMRuntime.getRuntime();

    if (runtime.getVariant(<methodID>) == 0) {
        MethodPlan plan = runtime.getPlan(<methodID>);
        Executor executor = runtime.getExecutor(plan);
        State state = plan.newState();

        // Initialise state
        state.setThis(this);
        state.addParameter(a);
        state.addParameterInt(x);

        // Call executor
        try {
            executor.execute(plan, state);
        } catch (AnException e) {
            throw e;
        } catch (Exception e) {
            executor.handleUncaughtException(e);
        }

        // Return return-value
        return state.getReturnLong();
    } else {
        <Original method body>
    }
}
```

Figure 3.9: Example showing the Java equivalent of a method stub containing two variants

the runtime policy. An example of a stub is shown in Figure 3.9. This stub contains two variants — variant 0 passes the responsibility of executing the method to an executor, while variant 1 executes the original method body in-situ.

After the stubs have been generated, a new plan-set is built to hold the plans corresponding to the current class. For each intercepted method, a new plan is generated as described in Section 3.4.3 and inserted into the plan-set.

The fragmentation process generates new classes that contain fragments of code from the original methods. These classes are merged into the class with mangled method names to avoid name clashes. This is done so that the code fragments can access methods and data with private and protected visibility without having to resort to setting everything to public.

Once all the methods have been processed, a final callback is made to the active policy, providing an opportunity for any policy-specific modifications to be incorporated into the bytecode. Finally, the bytecode for the class, which can contain both unintercepted methods and method stubs, is loaded into the JVM using the `defineClass` method.

When an intercepted method is executed, the stub will pass control of the program to the executor specified by the active policy. The executor has full control at the entry and exit of the method, and at the points permitted by the active fragmentation scheme and policy.

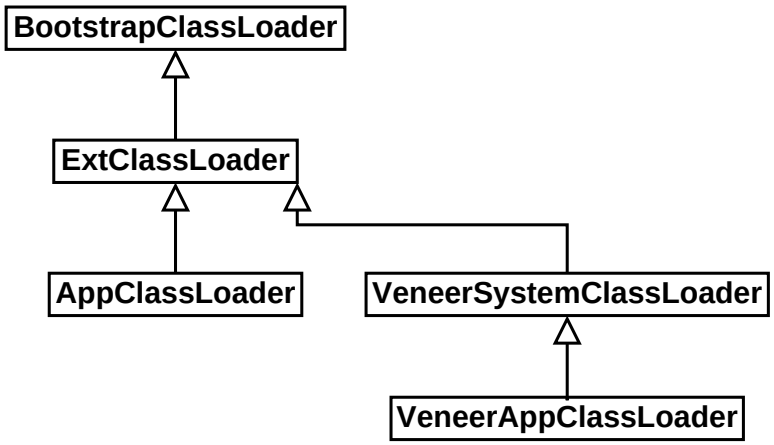


Figure 3.10: The Veneer class loader hierarchy

3.4.2.1 The Veneer class loader hierarchy

The class loaders used by the Veneer framework are arranged in a hierarchy as shown in Figure 3.10. The default application class loader `AppClassLoader` is used only to load in the Veneer system class loader, which bypasses it altogether and adopts the parent of `AppClassLoader` as its parent. The system class loader is responsible for loading the data structures used by the virtual JVM. It is also responsible for loading the Veneer application class loader, which loads in the classes needed by the user application.

The Veneer application class loader is based on the `ClassLoader` class included in the BCEL library. The original BCEL class loader uses the default `AppClassLoader` as its parent. Unfortunately, the BCEL class loader is flawed in some respects. The main problem is that it breaks the Java delegation model, with all the subsequent consequences (see Section A.2.2). It never checks the parent class loader first, but rather, it attempts to identify the class by name. If the class to be loaded is a system class (i.e. `java.*`), then it will delegate to the parent class loader, otherwise it will always handle the class itself. This is understandable, since the BCEL class loader loads classes from the same locations as the parent class loaders, hence delegating to the parent would mean that no classes would ever be intercepted.

This scheme manages to intercept application classes while preserving delegation by bypassing the default `AppClassLoader` altogether, so that when the Veneer application class loader delegates, classes normally handled by `AppClassLoader` will not be found. The Veneer system class loader is responsible only for loading classes belonging to the Veneer framework.

3.4.3 The fragmentation process

When it is determined that the execution of a method is to be intercepted by Veneer (see Section 3.3.7), then the body of that method is transformed into a plan via a *fragmentation* process. All plans for a particular class are gathered together into a plan-set.

The fragmentation process consists of several passes, executed in sequence:

3.4.3.1 Plan creation

A new plan for the method is created. A new parameterised block can be associated with any statement within the method, as determined by the fragmentation policy. If no parameterised block is associated with a statement, then the statement is placed within a *fragment*. Adjacent statements are placed within the same fragment, but a new fragment may be forced by marking the beginning of the fragment with a `Fragment` block.

3.4.3.2 Plan correction

This ensures that the plan is valid, correcting it automatically if not. Currently, the following corrections are implemented:

- If one block attempts to jump into the middle of another, then the destination block is split into two at the target statement.
- Attempts to separate memory allocation and constructor calls will be ignored for Jimple representations (see Section 3.6.3).

3.4.3.3 Control-flow determination

This pass determines the control-flow between blocks. This is done by searching through the statements associated with each block for branch statements. For every branch instruction found, the block that contains the destination statement is located and added onto a next-block array associated with the current block. Some special cases are:

- Return statements are treated as branch statements that jump to a null block.
- Throw statements and method calls are not regarded as branches.
- Statements situated at the end of a block that can fall through to the next statement are regarded as branches to the block containing the next statement.

3.4.3.4 Exception handler determination

This pass determines what exceptions are trapped within each block, and the current handlers for those exceptions. The exception types and references to the blocks of the handlers are stored with the block.

3.4.3.5 Local variable determination

This pass is applicable only for the Jimple representation, and determines the set of local variables visible to the executor. This set consists of all locals defined and used within the method, minus the set of intermediate locals.

Intermediate locals are locals that are defined and used only within a single fragment. These can be found by processing each fragment in turn, extracting all locals defined and used within it. Locals that appear outside of the fragment are removed, as are used locals that are not dominated by a definition of the same local in the same fragment (since this implies a loop-carried dependence that spans calls to that fragment).

3.4.3.6 Parameter determination

This pass fills in the parameters of the parameterised blocks. The information is extracted from the statements associated with each block.

3.4.3.7 Fragment generation

This pass generates a fragment class with a single `execute` method containing the code for a fragment. This method is called by its corresponding fragment block within the plan. The fragment method begins with a prologue that 'unpacks' the current state from the `State` object, so that the stack frame is set up into a form suitable for executing the next fragment. The fragment code is then added to the method. Since the stack frame is set up by the prologue, the original code may be copied almost verbatim. There are two main adjustments that need to be made:

- Branches to statements outside of the current fragment are replaced by code that sets a variable to the index of the next-block array corresponding to the destination block, followed by a jump to the epilogue.
- In exception handlers, a statement that explicitly loads the thrown exception from the method state is generated.

Finally, an epilogue is generated that reverses the effect of the prologue — it stores the current state of the stack frame back into the `State` object.

3.4.3.8 Metadata generation

This pass provides the opportunity to tag the blocks with information that may be computed statically at this point as block metadata. The information needed depends on the current policy. Currently, information such as line numbers and variable definition, use and liveness information are provided.

3.5 Optimisations

This section presents the various strategies that are used to reduce the impact of Veneer on runtime performance.

3.5.1 Fragment merging

In the Jimple representation, all fragments for a method are grouped together into a single 'execute' method as shown in Figure 3.11. This is to optimise the case in which control-flow does not need to be intercepted between fragments — control will only be regained by the executor when a parameterised block or the end of the method is encountered. By using this scheme, the overheads introduced by the executor and the prologue/epilogue are eliminated by jumping from one fragment to another without leaving the fragment method.

Fragments are identified by a unique ID, which are stored by their corresponding block in the plan. After the prologue, a loop is entered which fetches the ID of the current fragment to be executed, then executes the corresponding code. After execution, the current block

```
public void execute(Executor executor) {
    Plan plan = executor.getCurrentPlan();
    Block currentBlock = executor.getCurrentBlock();
    // Unpack locals...

    do {
        int nextBlockID, fragmentID =
            ((Fragment) currentBlock).getFragmentID();

        switch (fragmentID) {
        case 0:
            <Fragment 0>
            <Set nextBlockID>
            break;
        case 1:
            <Fragment 1>
            <Set nextBlockID>
            break; // etc...
        }

        executor.gotoNextBlock(nextBlockID);
        currentBlock = executor.getCurrentBlock();
    } while (!executor.getSingleStep() &&
             currentBlock.getType() == Block.FRAGMENT);

    // Pack locals...
}
```

Figure 3.11: A fragment method

is set to its successor. If the next block is a fragment, then another iteration of the loop is performed using the new block, otherwise the epilogue is executed and the method returns.

The prologue needs to be changed to restore the state of all used variables in all the fragments, and similarly the epilogue needs to store the state of all defined variables. Restoring only the locals used by the starting block will not work since this might lead to variables used in successor fragments remaining uninitialised if those variables were defined before the first fragment. Even if this was not a problem, the bytecode verifier would reject such code, since program paths would appear to exist that lead to the access of uninitialised variables. A similar argument applies to the epilogue.

This slightly cumbersome scheme is needed in order to honour the control-flow as specified by the plan, which may be changed at any time (including the period during which the fragment is running). A *single-stepping* flag is also provided, which reverts the behaviour of the `execute` method to its former behaviour of executing only one fragment at a time.

3.5.2 Plan-set caching

After a plan-set has been generated, it is cached onto persistent storage, along with a checksum of the original class file. Veneer will pick up the cached copy the next time it tries to load that class, provided that the checksum of the class matches the stored checksum. This is much quicker than going through fragmentation on every run, which is highly expensive in terms of both CPU time and memory.

```
public class Clazz_foo_Shim implements Shim {
    public Object invoke(Executor e) {
        ((Clazz) e.getCurrentState().getThis()).foo(e);
    }
}
```

Figure 3.12: Example of a shim class that calls a method `foo` in a class `Clazz`.

3.5.3 Short-circuit return statements

When a return statement is embedded within a fragment block, the return value (if any) is set in the `State` object and the fragment method returns immediately, without going through the checks at the end of each fragment and the epilogue.

This is valid since a return instruction terminates a method, and so there is no need to update the state of the method locals since they will not be used again. However, it may interfere with executors that need to examine the state at the end of a method, since the final state of the local variables is not written back into the `State` object in this case.

3.5.4 Embedded exception handlers

If all the instructions within a method that can throw a given type of exception are contained within fragments, and at least the first statement of the exception handler is also within a fragment, then the jump to the exception handler is handled directly using the standard exception throwing mechanism rather than relying on the executor to catch the exception and propagate it to the correct handler by calling `gotoExceptionHandler`.

3.5.5 Reducing reflection

Originally, when fragment blocks are executed, the fragment method containing the implementation of those blocks is called via reflection because the name of the fragment method is mangled at load-time. However, invoking a method via reflection can be slower by several orders of magnitude compared to the standard virtual dispatch mechanism.

The use of reflection in this case has been avoided by using *shim classes*. A shim class is a simple class that implements the `Shim` interface, which declares a single method `invoke`. An example of a shim class is shown in Figure 3.12.

After the plan-set for a class is loaded, a new shim class is generated for every method that contains at least one fragment block, with the `invoke` method of the shim class containing a call to the fragment method containing the fragment implementations. An instance of the newly generated shim class is created via reflection, then stored in all the fragment blocks of the plan. This shim class instance is subsequently used to invoke the fragment method.

This scheme allows methods whose names are unknown until runtime to be called using generic, statically compiled classes with minimal overhead.

3.5.6 Executor and state pooling

The HotSpot FAQs [42] recommend against the usage of object pooling due to the adoption of generational garbage collection [93]. Nevertheless, considerably better performance can

be obtained by pooling and reusing certain objects used in Veneer, especially executors and method state objects.

3.5.7 Mutable value types

Value types such as `int` or `float` must be wrapped in a value wrapper before being stored in the method state, since the state only stores reference types. Value wrappers such as `java.lang.Integer` are immutable, in order to provide value-like semantics such as thread-safety. However, this means that the only way to 'change' the value of a wrapped value is to create a new instance of the value wrapper and change the reference to the original wrapper so that it refers to the new instance instead.

However, since storing values into the state is a common operation, this can become very expensive as wrappers are constantly being created and garbage-collected as the variable they represent is updated. To combat this, a new set of wrappers have been created. Unlike the native wrappers, the values that they wrap can be changed, so that they can be reused over and over again. The value semantic issues that led to the immutability of the native wrappers are not an issue since these wrappers are only used under strictly-controlled conditions within Veneer.

3.6 Limitations

Since Veneer relies heavily on the underlying Java virtual machine, it is subject to the restrictions imposed by the JVM. The consequences of these restrictions are detailed in this section.

3.6.1 The standard Java library

Veneer cannot intercept classes that lie within the standard Java class library (i.e. classes with names beginning with `java.`), since the `defineClass` method in `ClassLoader` explicitly forbids this. This is partly to ensure consistency, but also for security reasons — if it was possible to redefine the standard library at will, then the security checks embedded within its classes may be subverted.

Fortunately, this is not as problematic as it may seem. Although Veneer cannot intercept the actual methods of the standard Java libraries, it can intercept calls *to* the methods in the standard library, which should suffice in most cases.

3.6.2 Intercepting constructors

The Java specification demands that the first action of a constructor must be to call a constructor of the parent class, or an alternative constructor in the same class. This means that Veneer is unable to intercept this first constructor call, although it can intercept everything that follows. In practice, this is not a problem, since the other constructors can be intercepted in a similar fashion if necessary.

3.6.3 Allocating new objects

In the Java language, whenever a class is instantiated, memory is allocated for the new object and a constructor is called, all in one statement. At the bytecode level, memory allocation and the call to the constructor are two distinct steps.

Problems occur if a method is to be fragmented between a memory allocation instruction (either a new or newarray instruction) and the corresponding constructor call (an invokespecial instruction), since the verifier will not permit uninitialised objects to be passed outside of the method.

A naive solution might be to simply ignore the effect of the new instruction altogether, and perform the operation just before the invokespecial instruction. However, this will fail when copies of the reference to the uninitialised object are made. For example, a common idiom in bytecode is:

```
new C
dup
invokespecial C()
```

This code first allocates memory for a new instance of class C, leaving a reference on the operand stack. The next instruction duplicates the reference on the operand stack. The constructor is then called on the duplicate reference, removing it from the stack in the process. The original reference, which now points to an initialised object, is left on the stack ready to be used.

If the effect of the new instruction is ignored, then the dup instruction will not work properly, since it will have no reference to duplicate. Although it is easy enough to fix in this particular case, there can be any number of copies made of the uninitialised object (both on the operand stack and in local variables) between the new and invokespecial instructions.

In the Soot back-end, the problem is avoided simply by forbidding breaks between memory allocations and constructor calls. If the user requests such break, it will be ignored.

In the BCEL back-end, when a new instruction is encountered, an instance of a placeholder object `UninitialisedObject` is placed onto the stack, which is associated with a unique object ID. This object will get copied and passed around by the program code. This is safe because in a verified program, the code must not manipulate the referenced object in any way until it has been initialised. Storage in local variable slots is also safe since the slots are untyped.

When the invokespecial instruction is encountered, an instance of the class is allocated and initialised in one step. Since a copy of the reference to the `UninitialisedObject` must have been at the top of the stack when the invokespecial is executed, the ID of the object can be retrieved. Veneer can then search through the stack and variable slots, and replace all instances of `UninitialisedObject` with the same object ID with the reference to the newly allocated object. Since the number of slots and the stack size are usually very low, this operation is fast.

3.6.4 Obfuscated code

Veneer is unable to deal with obfuscated code. This is due to the common practice of obfuscators to mangle the names of fields, classes, and methods to names that are illegal in

the Java language, but are nevertheless blindly accepted by the JVM. However, they interact badly with Soot, BCEL and the Veneer custom class loader.

3.6.5 User-defined class loaders

In order to intercept a class, the class must be defined by the Veneer custom class loader. However, if a class loader is defined by the intercepted application and the new class loader looks somewhere outside of the initial class path for the bytecode, then the custom class loader will fail to find the new class and the new class loader will load an normal version of the class.

One way around this might be to intercept the new class loader as it is loaded, changing it such that all calls to `defineClass` are directed to the version in the customised class loader. This will ensure that the loaded class is modified.

3.7 Evaluation

3.7.1 Test setup

Veneer has been evaluated with six benchmark programs. Three of the benchmarks are from the SPECjvm98 suite. These benchmarks have been run as standalone applications rather than as applets as required by the SPECjvm98 benchmark rules, and so these results are not directly comparable to other published SPECjvm98 results.

The other benchmarks in SPECjvm98 have not been used for various reasons:

- `_222_mpegaudio` is obfuscated, and will not work properly with Veneer
- `_205_raytrace` and `_227_mtrt` do not function properly when executed as standalone applications
- The remaining benchmarks do not work properly under Veneer. This may be due to remaining bugs in the fragmentation process.

Veneer was also tested on three other benchmarks:

- Linpack — an old benchmark commonly used to measure floating-point performance
- Tak — a synthetic benchmark used to measure the speed of recursive method calls
- RouteFinder — an application that finds the optimal route between two stations in a railway network using graph algorithms, given constraints imposed on the route (such as disability access at the stations).

The Veneer policies tested were:

- Extensive fragmentation — every method within each application class is fragmented at branches, method calls, returns, exception block boundaries, exception handlers and synchronisation blocks. Branches are placed into parameterised blocks only if they occur after the end of a fragment, while the other types are always placed into parameterised blocks.

Time taken (s)	HotSpot	HotSpot (int.)	Extensive	Extensive (single-step)	Single fragment
_201_compress	12.36±0.082	144.73±0.058	526.61±1.4	622.74±2.7	122.59±0.21
_209_db	21.96±0.0029	76.53±0.023	108.28±0.34	136.38±0.28	25.18±0.014
_213_javac	9.82±0.018	39.15±0.017	231.09±0.38	266.10±1.1	87.47±0.29
Linpack	0.20±0.0039	0.25±0.0059	1.64±0.020	1.60±0.015	1.56±0.0030
RouteFinder	25.37±0.016	365.36±0.098	286.80±4.0	299.38±2.7	86.69±0.34
Tak	1.26±0.0032	11.76±0.0052	110.27±0.073	116.72±0.24	35.62±0.085

Table 3.1: Times for the six benchmark programs under the different test configurations — the mean time is given in seconds with a 95% confidence interval.

- Extensive fragmentation with single-stepping — the same as extensive fragmentation, but fragments always return after being called, rather than proceeding directly to the next block if it is also a fragment.
- Single fragment — the original method body is encapsulated within one large fragment in every method of every application class, such that the executor is only entered at the beginning and end of each method.

All policies are running under the standard executor (similar to that shown in Figure 3.8), which does nothing apart from executing the blocks of the method in sequence.

The tests were run on an Athlon XP 1800+ PC, running on a Linux system with version 2.4.21 of the kernel. The JVM used was Sun HotSpot client JVM 1.4.2_01. The performance of HotSpot running in purely interpretive mode (using the `-Xint` flag) was also tested. The recorded time was obtained using the standard Linux `time` command, and therefore includes the startup time for the virtual machine.

Each test was repeated ten times, and the mean time taken with a confidence interval of 95%. For each test configuration using Veneer, an extra test run was added at the beginning in order to generate the execution plans for the program, which are retrieved from the disk cache in subsequent runs. The additional time taken by this first run in comparison to the cached runs was also noted. The first run is *not* factored into the calculation of the mean execution time.

3.7.2 Analysis

As can be seen in Table 3.1, running Veneer without any optimisations implemented in the executor always results in a slowdown. This is inevitable since Veneer must analyse each class to check for intercepted methods and regenerate the method stubs at class load time. At runtime, there is the overhead of the method stub, executor, fragment prologue etc. that is incurred on every intercepted method call. The JIT compiler of the underlying JVM will also not be able to perform as well since fragmenting the method bodies has the effect of introducing extra barriers to the code optimiser.

The table of slowdowns (see Table 3.2) shows that there is a large range of slowdown factors from 1.15 to 92.34, although the slowdowns for any particular benchmark are roughly of the same order of magnitude. This is still fairly good when compared with the typical three orders of magnitude slowdown reported by the JavaInJava project [87], which implements a full Java interpreter running on a JVM.

Relative execution times (HotSpot=1.0)	HotSpot	HotSpot (int.)	Extensive	Extensive (single-step)	Single fragment
_201_compress	1.00	11.71	42.60	50.38	9.92
_209_db	1.00	3.48	4.93	6.21	1.15
_213_javac	1.00	3.99	23.53	27.10	8.91
Linpack	1.00	1.25	8.11	7.94	7.74
RouteFinder	1.00	14.40	11.30	11.80	3.42
Tak	1.00	9.31	87.23	92.34	28.18

Table 3.2: Normalised execution times for the six benchmark programs under the different test configurations relative to time taken by HotSpot

Extra time taken for first runs (s)	Extensive	Single fragment
_201_compress	30.62	22.16
_209_db	15.68	16.54
_213_javac	123.00	105.67
Linpack	11.15	11.10
RouteFinder	36.55	26.57
Tak	2.32	8.74

Table 3.3: Extra time taken by first-time runs in addition to the mean runtime shown in Table 3.1

At the other end of the spectrum from JavaInJava is the HotSpot JVM running in purely interpretive mode, which is representative of a well-optimised native interpreter. Veneer manages to perform faster than HotSpot in half the test cases when running with the single fragment policy, but only once when using the extensive policy with RouteFinder. This shows that the strategy of delegating fragments of code to the underlying JVM can result in better performance compared to a purely interpretive approach at least some of the time, despite the overheads of having to go through the method stub and executor on every intercepted method.

The time taken by the first run of each new policy with each new benchmark (see Table 3.3) takes considerably longer to complete in order to generate the execution plans for all the methods in the application classes. The additional time taken by the first run over the other runs is approximately proportional to the size of the application.

3.7.2.1 Effect of the policy type

The two policy types represent two extremes — the extensive policy fragments frequently at every basic block boundary, while the single fragment policy avoids method fragmentation. Typical usage patterns will likely fragment at a level between these two policies. For example, the executor for the RMI optimiser only fragments methods that may contain RMI calls, and even then the fragmentation occurs mainly around potential RMI call sites.

Note that the measured times do not represent a lower or upper bound on the achievable execution times. Since every method in every application class is being intercepted, it is possible to do much better by intercepting fewer classes and methods, especially those that are frequently executed. However, it is also possible to do much worse by fragmenting after

	No. blocks called (extensive)	No. blocks called (extensive, single-step)	No. executors called
_201_compress	1,131,003,435	1,312,122,761	225,926,071
_209_db	209,053,605	283,909,528	1,484,131
_213_javac (*)	206,625,894	241,333,622	55,543,198
Linpack	21,257	26,405	10,620
RouteFinder	804,487,222	842,740,542	113,013,770
Tak	222,632,016	238,535,019	63,609,001

Table 3.4: The total number of times that a block is entered from the executor for the two variants of the extensive fragmentation scheme, and the number of times that an executor is entered during the course of a program. For the single fragment scheme, the number of blocks entered is equal to the number of times the executor is entered. (*) The _213_javac benchmark has a non-deterministic element that can cause the numbers to vary by a very small amount (<0.003%).

every instruction in every method.

The performance impact of extensive fragmentation is clearly visible as the slowdown is typically around 3 or 4 times slower than the single fragment policy. The main exception is the Linpack benchmark, which hardly shows any variation between policies. This is probably because Linpack completes so quickly that most of the time is spent in the initialisation of Veneer. The effect of fragment single-stepping is also visible, resulting in a significant slowdown in all but Linpack (which is unreliable due to the startup time).

The number of times that executors are called and blocks are entered in the benchmark programs are shown in Table 3.4. A massive increase in the number of blocks executed is evident when going from single-fragment fragmentation to extensive fragmentation, and a lesser increase when single-stepping is enabled. Comparing this table with Table 3.1, the number of blocks entered appears to have an approximate correlation with the time taken to execute a program. The main exception is _209_db, which enters over a hundred times more blocks in the extensive policy compared to the single fragment policy, yet is only around four times slower.

3.7.2.2 Call overhead

The Tak benchmark, which was specifically designed to test the speed of procedure calls in a programming language by executing 63,609 recursive calls per iteration, is a pathological case that shows the worst slowdown of all the benchmarks. Also notable is that even when the entire body of the Tak method is placed within a single fragment, the program slows down by a factor of over 28.

This suggests that Veneer has a relatively high method call overhead, since minimal interpretive overhead should be occurring in that scenario. This is due to the overhead of the extra code that is executed between the method stub and the code blocks, which, although heavily optimised, is still large compared to the native dispatch mechanism.

3.7.2.3 Effect of the Veneer optimisations

For purposes of comparison, results for the same set of tests running under an older version of Veneer without the optimisations detailed in Sections 3.5.3–3.5.7 are presented in Tables

Time taken (s)	HotSpot	HotSpot (int.)	Extensive	Extensive (single-step)	Single fragment
_201_compress	12.30	145.38	1190.96	1453.34	483.03
_209_db	22.96	77.69	182.38	255.48	35.57
_213_javac	10.92	40.76	381.30	433.45	150.22
Linpack	0.36	0.41	2.04	2.06	2.01
RouteFinder	25.78	329.24	1092.08	1116.06	859.26
Tak	1.42	12.06	617.67	633.81	469.99

Table 3.5: Times for the six benchmark programs under the different test configurations running on an old version of Veneer prior to the optimisations described in Sections 3.5.3–3.5.7

Relative execution times (HotSpot=1.0)	HotSpot	HotSpot (int.)	Extensive	Extensive (single-step)	Single fragment
_201_compress	1.00	11.82	96.83	118.16	39.27
_209_db	1.00	3.38	7.94	11.13	1.55
_213_javac	1.00	3.73	34.92	39.69	13.76
Linpack	1.00	1.14	5.67	5.72	5.58
RouteFinder	1.00	12.77	42.36	43.29	33.33
Tak	1.00	8.49	434.98	446.35	330.98

Table 3.6: Slowdowns for the six benchmark programs under the different test configurations running on an old version of Veneer prior to the optimisations described in Sections 3.5.3–3.5.7

3.5 and 3.6.

In this set of results, Tak could exhibit a slow down by a factor as large as 446.35, which in the newer set of results was 92.34, representing an improvement by a factor of nearly 5. The best improvement of the newer version over the older version was also found in Tak with the single-fragment policy, where the newer version is over 13 times faster than the old one.

3.8 Alternative approaches to runtime code modification in Java

The challenges of developing a dynamic optimisation framework on the Java platform are very different from those of developing for conventional architectures like PA-RISC (Dynamo [9]) and IA-32 (DynamoRIO [15], Mojo [18]). On the whole, the Java platform is not very amenable to runtime optimisation at a level above that of the virtual machine. The main problem lies in the fact that Java bytecode is far more structured than machine code for real microprocessors, reflecting the structure of the Java language instead. Although this makes it considerably simpler to understand, it is also far more restricted in what it can do.

One of the main tasks required of a dynamic optimisation framework is to provide a ability to replace code on-the-fly. Veneer performs this task by introducing an extra layer of indirection, directing control-flow to a dynamic data structure that can be modified at runtime. Alternative ways of doing this have been considered, but have been rejected due to their numerous drawbacks. These are covered in the remainder of this section.

3.8.1 Class loaders

As discussed in Section A.2, classes defined by different class loaders effectively form a different class name-space. It is therefore possible to load multiple versions of the same class simultaneously [52]. Unfortunately, major problems will be encountered if one tries to use this to substitute one version of a class for another.

3.8.1.1 Accessibility

One problem with this approach is how to access the new version of the class. For example, suppose that an existing class c_1 defined by class loader CL_1 is to be replaced with a new class c_2 defined by a new class loader CL_2, which are used by methods in a class C. In order to preserve the delegation model, CL_1 and CL_2 must be cousins of each other (i.e. they can share a common ancestor, but must not have a direct ancestor-descendant relationship).

The access problem occurs because when C was loaded, it also loads all classes that it is dependent on. If c was also a dependency class, and was resolved to c_1, then all references to c in C will *always* refer to c_1, even after c_2 is loaded. This makes it impossible for c_2 to act as a drop-in replacement for c_1, since:

- If the expression `new c()` appears in C, then it will always be an instance of c_1 that is generated
- Calls to static methods of c will be directed to c_1
- If there are any variables or fields with static type c, then it will be impossible to assign instances c_2 to it since that would be a type-error. This will also make it impossible to invoke methods of c_2 via those variables and fields

The type-assignment problem may be circumvented by making c_2 a descendant of c_1, but this also has its flaws:

- If c_1 is final, then c_2 may not extend it
- If any of the methods of c_1 are final, then c_2 cannot override that method
- c_2 has no access to any of the private methods and fields of c_1

3.8.1.2 Class replacement

In order for the changes to take effect, all instances of c_1 must be replaced with instances of c_2. This is a non-trivial task, since references to c_1 may be hidden within other objects and within other threads.

Another problem is how the state of an old c_1 instance is to be transferred over to a new c_2 instance, since some portions of the state of c_1 may be private. This may be overcome using reflective access to the state, provided that the relevant permissions are granted by the security manager.

Even assuming that a substitution was possible, any methods that are invoked on c_1 before the substitution will continue to run to completion — i.e. the effect of the change is not spontaneous.

3.8.2 HotSwap

Another way of performing the task is to use the recently introduced HotSwap feature of the Java Platform Debugger Architecture (JPDA) [43]. This facility allows a debugger to change class definitions at runtime.

One problem with this scheme is that the changes do not affect methods that were running at the time the changes were made. This makes it unsuitable for optimising long-running loops or methods (such as `main`), which often have the greatest potential for speedup.

Curiously, the designers of the JPDA appear to try to compensate for this by providing the ability to roll-back the caller stack frame using the `popFrames` method in the `ThreadReference` class. However, `popFrames` is not as useful as it may seem, since any fields modified or I/O operations performed by the method will not be reverted by this method, such that the initial state encountered by the modified method will be different from that encountered by the original. Furthermore, any changes previously made to the formal parameters will be visible when the method is re-entered.

Another problem is that this facility is only available via the JPDA, meaning that the application must be run under debugging mode. Normally, this means that the program must be run in interpretive mode. However, as of Java 1.4.0, there is support for 'full-speed debugging', which permits programs to run under the JIT whilst being debugged. However, performing debugging operations on a method such as single-stepping or watching variables will still cause the JVM to fall back to interpretive mode.

3.8.3 Implementation at the virtual machine level

Adding an additional execution layer will inevitably lead to extra overheads. One obvious alternative to reduce this overhead is to implement Veneer at the JVM level.

Platforms such as Java that run programs on virtual machines are particularly amenable to dynamic optimisation, since the virtual machine has full control over how the program is executed. Virtual machines that run in an interpretive mode are especially suited for this task, since the interpreter has the opportunity to affect the course of execution before, during and after every instruction, inspecting the current state of the execution environment to determine what actions to take if necessary. The code to carry out the optimisations may also be written both directly and generically, becoming part of the interpreter rather than the executed program, which helps with the speed and ease of the development process. This means that interpreters can deal with situations as they arise, whereas precompiled solutions must take all possible outcomes into account in advance.

At the other end of the spectrum are virtual machines that rely on Just-In-Time compilation to compile bytecode into native code for execution. Since the virtual machine will lose control of execution while the generated native code is being executed, any dynamic optimisations needed must be incorporated directly into the native code. This means that the JIT compiler must take on the same role as the static program rewriter discussed in Section 3.1.1. However, since JIT compilers usually work on-demand, the JIT compiler may have more context information to work with than an external rewriter if the program has been running for some time before the code section was invoked.

There are two main problems with modifying an existing virtual machine. Modern virtual machines for full-scale languages such as Java are large, complex programs, and the task of

becoming acquainted with their often undocumented internal workings and modifying them is a major task. Modifications made to a virtual machine will also be specific to that one virtual machine, and are unlikely to be portable to others.

3.9 Other uses of Veneer

The Veneer virtual JVM framework has recently been used outside of the RMI optimisation context for which it was originally designed. The Java Utility for Dynamic Instrumentation (JUDI) [97] tool uses the Veneer virtual JVM to provide a means of deploying instruments into a program at runtime by means of a GUI. JUDI works by fragmenting all methods of a program at basic block boundaries. Instruments, in the form of Veneer blocks, can be inserted into the fragmented methods at runtime, and can be removed when instrumentation is no longer necessary. Performance results using JUDI on a fairly old version of Veneer show it to be comparable in performance to using the `hprof` tool, being faster in some programs, and slower in others.

The Java bottleneck locator toolkit (JBolt) [14] is built on top of JUDI, offering a Java equivalent of the Paradyn [66] performance tool, which can automatically find and zoom-in on bottlenecks in a program.

3.10 Conclusion

The Veneer virtual JVM provides a versatile environment for experimenting with runtime code optimisation, especially optimisations involving code motion, at the expense of some runtime performance. The cost depends heavily on the runtime policy used — in general, a program will run more slowly when intercepted methods are often called and block boundaries are frequently reached. The runtime impact can therefore be minimised by careful placement of the interception points.

It should be possible to eventually reduce the runtime overhead down to a reasonable amount, so that even a small amount of speedup resulting from optimisations built on top of Veneer can overcome this performance penalty. Since the penalty is strictly in terms of the number of instructions executed for a given operation, the slowdown should reduce with time as CPU speeds increase.

Chapter 4

Optimising Java RMI

The details of the new RMI optimisations are presented in this chapter. These optimisations are referred to as the Delayed Evaluation Self Optimising Remote Method Invocation (DESORMI) optimisations.

Some knowledge of RMI programming is assumed for this chapter. A brief introduction to this subject may be found in Appendix B. This chapter mainly explores the practical side of RMI optimisation. A more theoretical view is taken in Chapter 5.

Work covered in this chapter has been presented at the 10th Workshop on Compilers for Parallel Computing (CPC 2003) in the paper 'Automated Optimisation of Distributed Java Programs across Network Boundaries' [95] and at the Middleware 2003 conference in the paper 'Optimising Java RMI Programs by Communication Restructuring' [96].

4.1 Modelling performance

In this section, a simple mathematical model is presented for reasoning about the performance of remote method calls.

4.1.1 Assumptions

This model deals with performance in terms of the typical time required for a series of successful calls to remote methods. One-time costs such as object activation and obtaining a remote reference from a registry are ignored, since these are assumed to be amortised over the lifetime of the program. Events such as distributed garbage collection are also assumed not to occur, since they are by nature infrequent and unpredictable events.

These assumptions are not detrimental to the usefulness of the model, since none of these events are affected by the RMI optimisations. Since the model is used to reason about how the RMI optimisations affect the cost of RMI calls, modelling them would simply add unnecessary complexity.

4.1.2 Parameters

The following steps occur for a successful remote invocation to a standard remote object:

1. The arguments of the method call are serialised into a data-stream

2. The serialised arguments, along with the identity of the method being called, are sent to the remote server
3. The server deserialises the arguments
4. The server uses the method identity to call the correct method, passing in the deserialised arguments
5. The return value of the method is serialised
6. The serialised return value is sent to the server
7. The client deserialises the return value

A remote method that throws an exception is also regarded as a successful call — the thrown exception is treated like a return value. Unexpected exceptions, such as those due to network failure, are not covered in this cost model since they are inherently unpredictable.

The costs of a RMI call can be divided into three categories:

- Constant (l) — this is overhead that is incurred on every RMI call, and is independent of the method being called or the supplied data. It is due to:
  - Network latency
  - Stub overhead
- Per byte (b) — this is overhead that is dependent on the size of data transferred between client and server. This is due to:
  - Serialisation and deserialisation
  - Network communication (limited by bandwidth)
- Computation time (c) — this is the time taken by the remote method to actually do the work requested by the client.

4.1.3 Cost model

Using this model, the time required to perform a single remote call, where the amount of data transmitted is s bytes, is:

$$t = l + s \cdot b + c \tag{4.1}$$

For n calls, where the quantity of data and computation time for the ith call are denoted by s_i and c_i respectively, the total time is given by:

$$t = \sum_{i=0}^{n-1} l + s_i \cdot b + c_i \tag{4.2}$$

This model can be simplified by defining $\overline{s}$ and $\overline{c}$ as the mean values of s_i and c_i over the n calls respectively, so that the final cost model is:

$$t = n \cdot (l + \overline{s} \cdot b + \overline{c}) \tag{4.3}$$

This cost model will be elaborated upon as the RMI optimisations are discussed in terms of their effect on the cost equation.

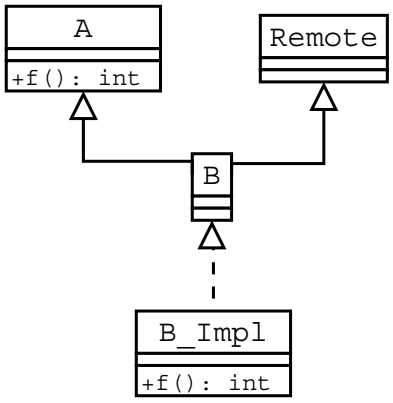


Figure 4.1: Example of a class hierarchy that provides an interface leading to a remote call without implementing `Remote` — the interface `A` does not extend `Remote`, but calling `f` on it will result in a remote call nevertheless.

4.1.4 Limitations

This cost model is oversimplified in many ways. In particular, it ignores the packet nature of most computer networks, which leads to a step-like increase in time as the quantity of data to be transported increases, rather than the linear relationship used in the model.

Nevertheless, this simple model is adequate for the purposes of analysing the effect of the RMI optimisations, since the relative effect of each parameter on the overall time can easily be inferred from the form of the equation — the absolute time is not important in this context. It can also be argued that if the quantity of data transported is large compared to the packet size, then a continuous progression is an adequate approximation of a stepped progression.

4.2 Initialisation

This section deals with the events that happen when the client and server start running under Veneer, and how the client obtains a remote proxy from the server.

4.2.1 Client startup

The client runs under a Veneer policy that only intercepts the execution of methods that may make remote calls, fragmenting around calls that have been statically determined to be *potentially remote*.

Calls are deemed to be potentially remote if they are invoked via an interface, and have `RemoteException` or one of its super-classes on the throw list. A run-time check is later used to ensure that a potential remote call is actually remote. Note that it is not sufficient just to check that the receiver of the call implements `java.rmi.Remote` since the object could be invoked directly instead of via RMI, and some remote calls may be missed if calls are made to a non-remote interface that acts as the superclass of a remote interface — for example, see Figure 4.1.

Under the control of the Veneer framework using this policy, runtime behaviour is unchanged until the first remote call is encountered.

4.2.2 Server startup

The same Veneer policy that is used on the client also runs a remote proxy server on startup, which first registers itself in a naming service via JNDI [48]. This proxy keeps track of all remote objects present on the virtual JVM by inserting a small callback method into the constructors of all remote classes at load time (any concrete implementation of an interface that extends the `Remote` interface is considered a remote object class). When the remote class is instantiated, this callback registers the remote object in the Veneer runtime, and associates the remote object with an ID that is unique for that Veneer instance.

4.2.3 Proxy/object resolution

Clients obtain handles to proxies via JNDI. When a client first encounters a new remote stub, it broadcasts it to all known proxies. The proxy that handles the remote object denoted by the stub will identify itself, returning the ID associated with the corresponding remote object. Remote plans containing calls on that stub will subsequently be sent to the identified proxy. Stub to proxy/object ID mappings are cached on the client for speed.

If none of the proxies claim to handle the new stub, then it is assumed that the remote object resides on a server that does not support the DESORMI optimisations. In this case, a fall-back mode is entered where all calls on the new stub are treated like local code, which is executed immediately when encountered.

Although RMI stubs contain the IP address and port number of the destination server and a unique object ID as part of their internal state, the stub to proxy/object ID resolution has been handled manually for two main reasons:

1. This data is stored in an implementation-dependent class (for example, in the Sun JDK, this information is stored in `sun.rmi.transport.LiveRef`) with no public access methods

2. RMI/IIOP and EJB implementations use different mechanisms for identifying the target of a remote call (for example, RMI/IIOP uses CORBA-style Interoperable Object References). By manually handling the remote target resolution, it is not necessary to reimplement the code to find an appropriate remote proxy for every transport mechanism as long as the stubs support the standard equality and hash code tests.

4.3 Call aggregation

Delaying calls to form call aggregates is the core technique upon which this project is based. It is an important optimisation in its own right, and furthermore can also open up further optimisation opportunities.

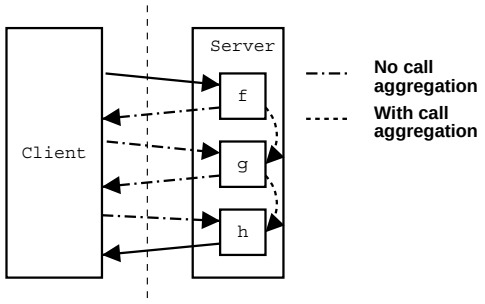


Figure 4.2: Example of call aggregation

Consider the following code fragment:

```
void m(RemoteObject r, int a) {
    int x = r.f(a);
    int y = r.g(x);
    int z = r.h(y);
    System.out.println(z);
}
```

This program fragment incurs three remote method calls, with six data transfers. However, for this example, it is possible to do better:

- Since all three calls are to the same remote object, they can be aggregated into a single large call, such that the number of times that call overhead is incurred is reduced to one (see Figure 4.2).
- x is returned as the result of the call to **f** from the remote server, but is subsequently passed back to it during the next call. The same occurs with the variable y. If the values of x and y were retained by the remote object between remote method calls, then the number of communications could be reduced from six to four.
- The variables x and y are unused by the client except as arguments to remote calls on the remote object from which they originated. x and y may therefore be considered as dead variables from the client's point of view, and there is no need for their value to be passed back to the client at all, thereby further reducing the total number of remote transactions down to just two messages with payloads of size *int*.

4.3.1 Relation to distributed program design

Call aggregation may be considered as an automatic application of several concepts described in Section 2.2.1:

- The command object pattern — the client now sends an object containing multiple commands for the server, so that multiple remote operations can be performed during the course of a single remote call.
- The value object pattern — all data defined and used by the remote calls are transferred to and from the client in a single step.

- Session state — data generated from an earlier remote call in a set of aggregated calls is retained on the server to be used by later calls if necessary, which can effectively make a stateless interface behave like a stateful one while the aggregated calls are being executed.

4.3.2 Effect on the cost model

To recap, the original cost model for n calls was:

$$t_{orig} = n \cdot (l + \overline{s} \cdot b + \overline{c}) \tag{4.4}$$

The effect of aggregating all calls together without any further optimisations is to reduce the call overhead of n calls that of just one call. However, the amount of data transferred per call increases by a constant amount since the identity of the called method and the metadata now need to be transferred as part of the data payload. This is denoted by a constant o.

$$t_{aggr} = l + n \cdot ((\overline{s} + o) \cdot b + \overline{c}) \tag{4.5}$$

The effect of sharing data and dropping of dead return values is modelled by first splitting the quantity of data returned by each call into two components — the data being passed to the method (*sin*), and the data passed back (*sout*).

The effect of sharing data between calls depends on the relationships between the calls. If no data is shared between calls, then there is no improvement at all. At the other extreme, if all the data is shared between all the calls, then the quantity of data is reduced by a factor of n. This is modelled by a sharing factor s, which represents the average proportion of 'fresh' data carried by each call. Low values of s denote high levels of sharing. s is defined as:

$$s = \frac{\text{total quantity of data transferred}}{\text{total quantity of data used from call arguments}}$$

For the calls where the return value is dropped, *sout* does not contribute to the amount of data transferred across the network. This is represented by a factor r, which is the weighted average of the proportion of calls that return a live value.

$$r = \frac{\sum live_i \cdot sout_i}{\Sigma sout_i}$$

where:

$$live_i = \begin{cases} 1 & \text{if call i returns a live value} \\ 0 & \text{otherwise} \end{cases}$$

The values of s and r are bounded by:

$$\begin{aligned} 0 &< s \leq 1 \\ 0 &< r \leq 1 \end{aligned}$$

Bringing all these factors together, the total cost is now:

$$t_{new} = l + n \cdot \left(\left(s \cdot \overline{sin} + r \cdot \overline{sout} + o \right) \cdot b + \overline{c} \right)$$

The optimisations pay off if:

$$\begin{aligned}
t_{orig} - t_{new} &> 0 \\
n \cdot \left(l + \left(\overline{sin} + \overline{sout} \right) \cdot b + \overline{c} \right) - \left(l + n \cdot \left(\left(s \cdot \overline{sin} + r \cdot \overline{sout} + o \right) \cdot b + \overline{c} \right) \right) &> 0 \\
n \cdot l + n \cdot b \cdot \left(\overline{sin} + \overline{sout} \right) - \left(l + n \cdot b \cdot \left(s \cdot \overline{sin} + r \cdot \overline{sout} + o \right) \right) &> 0 \\
(n-1) \cdot l + n \cdot b \cdot \left((1-s) \cdot \overline{sin} + (1-r) \cdot \overline{sout} - o \right) &> 0 \\
\frac{n-1}{n} \cdot \frac{l}{b} + (1-s) \cdot \overline{sin} + (1-r) \cdot \overline{sout} - o &> 0
\end{aligned}$$

This inequality predicts that the optimisation will be more effective if:

- l is large — this usually occurs on a slow network with high latency. The effect of the optimisation is greater in such situations because the time saved by reducing the total number of remote calls made is proportionately greater.
- b is small — this usually represents a high bandwidth network. The higher the bandwidth, the smaller the time needed to transfer the extra data represented by o.
- s is small and $\overline{sin}$ is large — this occurs when there is a high degree of data sharing and a large quantity of data would originally have been transferred. This leads to better performance due to a larger reduction in the amount of data that needs to be sent.
- r is small and $\overline{sout}$ is large — this occurs when most of the return values are dead and a large quantity of data would originally have been returned. Again, this leads to a better result because of the greater reduction in the quantity of data that needs to be returned.
- o is small — o represents the per-call overhead of the optimisation. Obviously, the lower the overhead, the better the performance.
- n is large — as n increases, $\frac{n-1}{n} \rightarrow 1$. This reflects the time saved by avoiding $n-1$ crossings of the network increases, but at the same time, the time taken by the overhead o increases proportionately.

If the effects of the data optimisations are ignored (i.e. if $s = r = 1$) and the inequality is rearranged to express n in terms of the other variables, the following inequality can be derived:

$$\begin{aligned}
\frac{n-1}{n} \cdot \frac{l}{b} - o &> 0 \\
n \cdot l - n \cdot b \cdot o &> l \\
n &> \frac{l}{l - b \cdot o}
\end{aligned}$$

The inequality establishes a lower bound for the number of calls that must be aggregated for the optimisation to pay off. As the time to transport the extra data $b \cdot o$ approaches the

time needed to make an extra remote call l, more calls must be aggregated for the saving in call latency to overcome the overhead of the optimisation. If $b \cdot o$ is greater than l, then aggregation is not worthwhile since it would be cheaper to make the calls separately in the first place.

4.3.3 Client-side implementation

If the executor encounters a confirmed remote call during the course of execution, then it places the call within a queue and proceeds to the next instruction. Sequences of adjacent calls to the same remote object are grouped together into *remote clusters*, which are a type of execution plan that may only contain remote calls to the same remote object. Metadata, such as the set of live variables after each call, is stored as part of the remote cluster.

Calls to other remote objects will not force execution of the delayed calls, but will result in the start of a new remote cluster. The exception to this rule occurs if the target of the call is defined by a previously delayed call, which leads to a control dependency since the object that the call is invoked on is not known until the delayed calls have been executed. This issue is resolved later in Section 4.4 with the introduction of server forwarding.

4.3.3.1 Local code

When a non-remote block is encountered with delayed calls remaining in the queue, a decision has to be made regarding whether or not to force execution of the calls. The decision is made conservatively — calls are forced unless it can be proven that it is safe to continue without forcing the calls. These conditions are covered in more detail in Section 4.6.

When a remote call is delayed across local code, that remote call marks the end of the current remote cluster, and subsequent calls are placed in a new cluster even if those calls are targeted at the same remote object. However, consecutive remote clusters targeted at the same remote object are grouped into *remote bundles*. Remote bundles are another type of execution plan that can contain either remote clusters or other remote bundles. Remote clusters and remote bundles are both types of *remote plan*, and either type may be processed by the server.

When executing local code in the presence of delayed remote calls, there may be anti-dependencies between the local code and the delayed calls. Anti-dependencies occur when variables used by the delayed calls are modified by the local code. This is guarded against by saving the state necessary to execute the current remote cluster just before the local code executes, as described in the next section.

In theory, it is sufficient to copy just the variables that may be touched by the local code and use the state from previous clusters for variables that are unmodified, but due to possible aliasing issues, *all* variables that are needed to execute the cluster are copied conservatively. However, as discussed later in Section 4.6.2.1, even this is not sufficient to deal with all possible cases due to callbacks.

4.3.3.2 Saving delayed call state

The variables required to execute a remote cluster are saved by placing them into an array, then serialising the array of objects into an array of bytes that is associated with the remote cluster. The variables need to be serialised in one step to preserve sharing (see Section 4.6.6).

This serialised state is sent along with the remote cluster when it is sent to the server to be executed later on.

The set of variables that need to be saved are defined as follows. A remote cluster consists of calls $f_1, f_2, \ldots f_m$. Each call f_i uses the variables in the set use_i in its argument list, and assigns to at most one variable, which is contained in the set def_i. The set of saved variables *saved* is defined as follows:

$$x \in saved \iff \exists i \cdot \left(x \in use_i \wedge \forall j \cdot x \in def_j \Rightarrow j > i\right)$$

In other words, only variables that are used in the argument lists of the delayed calls that are not defined by earlier calls are saved, to avoid sending values that will never be used.

4.3.3.3 Forcing execution

Remote calls are generally delayed for as long as possible to maximise the number of calls aggregated. Forcing of delayed calls is generally done only when it is necessary to preserve the original semantics of the application. These conditions are covered later in Section 4.6.

When execution is forced, the state required to execute the last remote cluster is first saved, if it has not been saved already. The queue of delayed remote plans is then traversed, with remote plans being sent one-by-one to the corresponding *remote proxy* on the server-side via standard RMI invocation to be executed. The proxy call may either return successfully or throw an exception.

If the call returns successfully, then the variables defined by the plan that are still live are copied back into the locals set of the executing method. If an exception was thrown, then the executor goes through the normal process of finding a handler for the exception within the method (the location of the thrower is part of the information contained in the exception), and propagating it up the call chain if one is not found.

4.3.3.4 Method exit

When the executor reaches the end of a method, all pending calls are forced to execute immediately. This is done since the current method may have been called from an unintercepted method, which would proceed after the call returns, oblivious to the fact that the callee has not finished completely. If the delayed remote calls have visible side-effects, then this can lead to events occurring in a different order, changing the program semantics.

Ways to overcome the need for forcing at the end of a method are discussed as future work in Section 6.2.8.

4.3.4 Server-side implementation

Remote plans sent to the proxy are executed by a remote executor, which simply executes the calls one-by-one. Remote calls in remote clusters are executed in sequence, while the contents of remote bundles are recursively traversed in order.

Remote calls are made by invoking the methods on the remote object directly rather than via another RMI invocation for better performance. However, care must be taken due

to the semantic differences between local and RMI calls (see Section 4.6.6).

After every remote method m executed, the set *defs* is updated with the variable that m assigns to, if any. However, before m can be executed, the arguments for m must be supplied to it. The remote executor uses the following algorithm to locate a value for a call argument x:

1. If $x \in defs$, the value from *defs* is used.
2. If x is in the saved state for the current cluster, the value of x from the saved state is used.
3. This step should never occur in the current implementation since partial saved states are not currently used. Nevertheless, if x cannot be found in the saved state for the current cluster, then the saved states for previous clusters are searched in reverse order. The first cluster with a state that contains x is used as the source for x.

When finished, the proxy returns a subset of *defs* back to the client. This subset *return* is defined by:

$$return = defs \cap live_{after}(m_{last})$$

In other words, only defined values that are live on the client after the last remote call is executed (m_{last}) need to be sent back to the client. This liveness information is part of the metadata supplied with the remote plans.

4.3.5 Example

Consider the following fragment of code:

```
x = r.f(w);   // Remote calls to a remote
y = r.g(x,w); // object r

i++;          // Intervening local code

z = r.h(y);   // Another remote call to r

System.out.println(z + i);
```

This code is somewhat contrived, but demonstrates most of the essential features. This will be fragmented by Veneer at each statement. When the executor running this code reaches the first remote call, it attempts to establish a connection with the Veneer instance that is responsible for handling `r`. If successful, this results in a remote reference to a server-side proxy for `r`. If not, it is assumed that the DESORMI is not supported by the server and remote calls are executed normally using the original stub. This type of execution proceeds as shown in Figure 4.3.

If a reference to a server-side proxy is found, then the call to `r.f` is delayed by placing it into a remote cluster. The value of the argument `w` is stored in the cluster. Metadata concerning the delayed call is also stored — this consists of the identity of the client-side

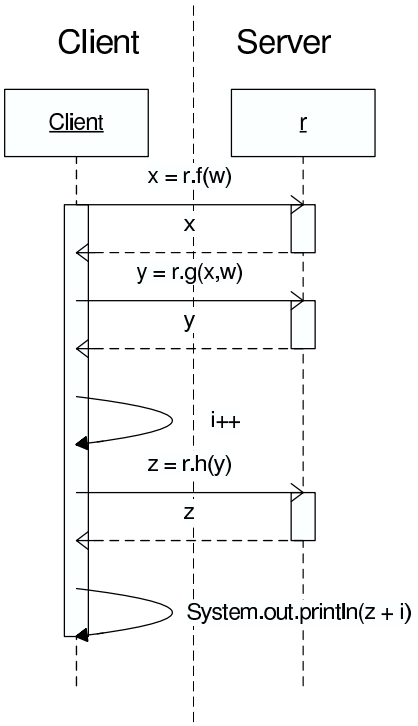


Figure 4.3: Sequence of events without call aggregation

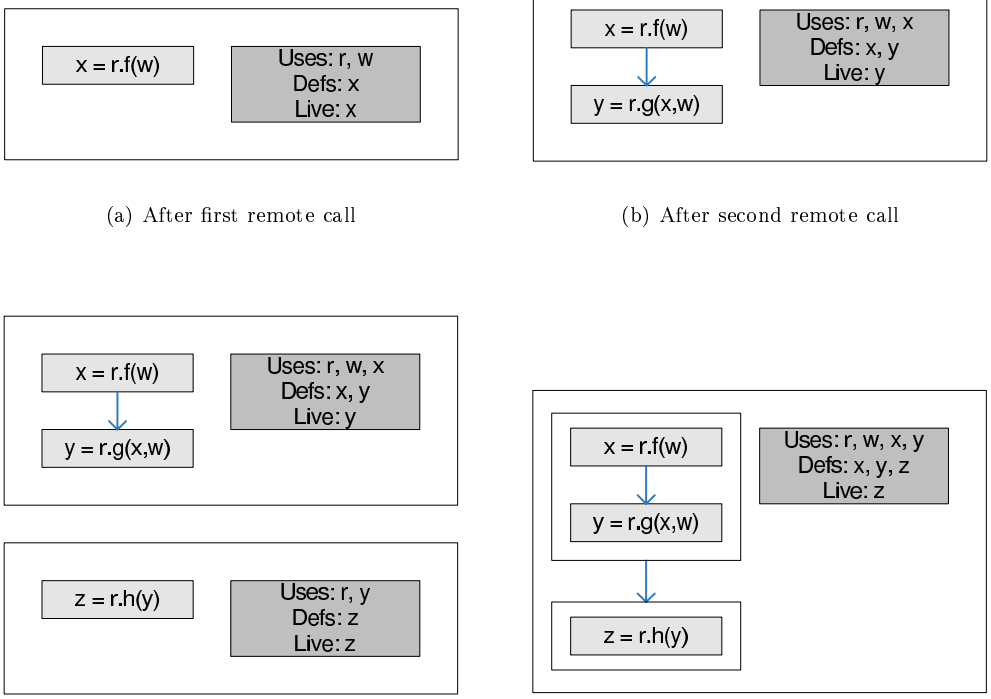
variables used and defined by the call, and the subset of defined variables that are live (i.e. will be used later on by the client). This is illustrated in Figure 4.4(a). The executor will then skip to the successor of the call.

The next remote call `r.g` is delayed in a similar fashion, and is placed into the same remote cluster as the previous call since it makes the call on the same remote object. However, the values of the arguments `x` and `w` are not stored this time, because `x` is defined by the previous call and `w` has already been stored as part of delaying `r.f`. This can be determined on-the-fly by inspecting the metadata associated with the remote cluster before adding `r.g`. After the new call is added, the set of live variables is updated by removing `x` because it is not used after `r.g`. This is shown in Figure 4.4(b).

The next instruction `i++` is local code. The executor must now decide whether the two delayed calls should be executed before the local code can execute. In this case, this is not necessary since the instruction does not need anything defined by the remote calls, cannot throw an exception, and is invisible to any call-backs and exception-handlers that may occur (assuming that `i` is local and exceptions result in the method immediately exiting). The instruction may therefore be executed without the remote calls executing.

The next instruction is a remote call to `r.h`, which is delayed. Although the call is also to the remote object `r`, it is placed in a separate remote cluster from the previous two calls due to the intervening local code. Again, `y` does not need to be stored in the cluster since it is defined by another remote call. Also, `z` is dead after this call, so it is removed from the list of live variables. This is illustrated in Figure 4.4(c).

The last instruction is a local statement that uses the variable `z` defined by the remote call `r.h`. However, the value of this variable is not yet available, since it is defined by a



(a) After first remote call

(b) After second remote call

(c) After third remote call

(d) Prior to remote execution

Figure 4.4: Remote plans during the call aggregation example

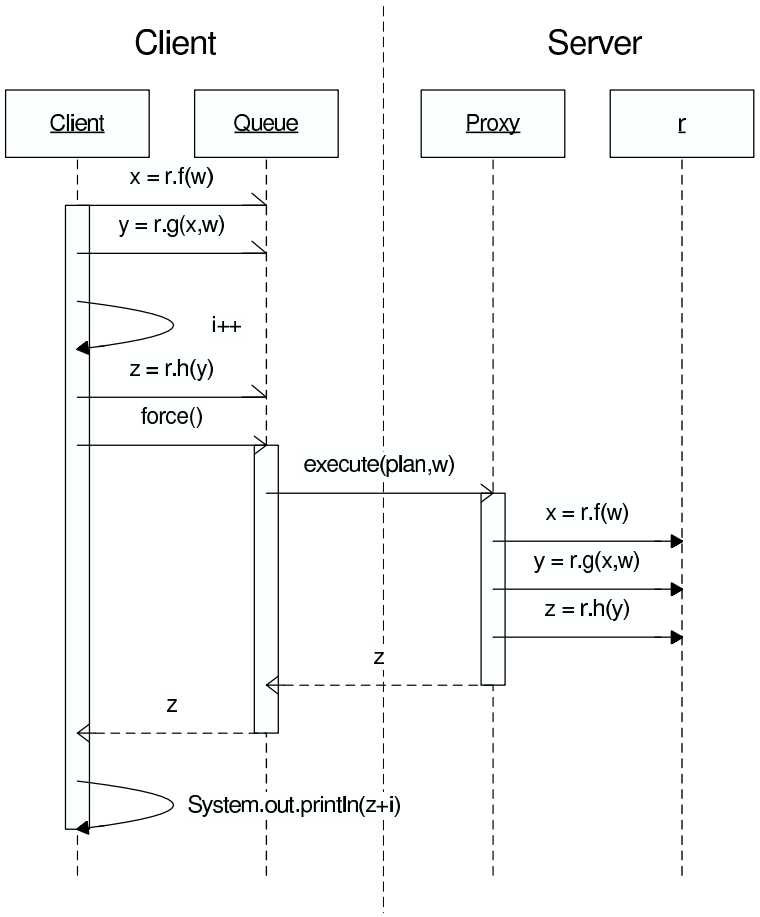


Figure 4.5: Sequence of events with call aggregation

remote call that has not yet been executed. This forces the execution of the delayed calls.

The two remote clusters are first gathered together into a remote bundle since they share the same destination. The metadata for the remote bundle is computed by merging the metadata of its constituents. This is shown in Figure 4.4(d).

The remote bundle is sent to the server-side proxy that is responsible for handling the remote object `r` using the remote reference handled earlier. The proxy makes the remote calls on behalf of the client, extracting the stored value of `w` to execute the first and second calls, and routing the results of prior calls to later calls where necessary. Finally, when all the calls are executed, only the live variables (in this case, `z`) are returned to the client. All decisions made by the proxy can be done efficiently at run-time by performing set operations on the supplied metadata. On the client-side, the returned value of `z` is loaded into the local variable `z`, and the local print statement can proceed.

The overall sequence of events is shown in Figure 4.5.

4.3.6 Implementation optimisations

Several low-level improvements have been made to the implementation that are covered in this section.

4.3.6.1 Transportation of plans

Remote plans must be serialised at the client end before being transported across the network and deserialised on the server. However, due to serialisation being a generalised process, normal serialisation usually results in a byte-stream that includes a considerable amount of extra data.

For example, if an instance of a class is serialised, then the Java runtime will iterate through the class hierarchy of the instance, serialising at every level. At each level of serialisation, the name and version identifier of the class will be placed on the byte-stream, as well as a string that provides the deserialising JVM with a location from which to load the class if it is not already present. Every instance field contained within the class is then serialised in the same manner.

Since the exact types of the transmitted plans and blocks are known, this results in a lot of unnecessary network traffic when transmitting serialised plans over the network. To combat this, several optimisations have been performed.

The classes are made *externalisable* rather than *serialisable*. Externalisable classes are similar to serialisable classes in that they can be transformed to and from byte-streams, but the implementation details differ in two main details:

- Externalisable classes *must* implement two methods `readExternal` and `writeExternal`, which are called when the object is externalised. In serialisation, the runtime must search for and call the optional `readObject` and `writeObject` methods, and serialise the instance fields one-by-one if the methods do not exist. Externalisation is faster because the `readExternal` and `writeExternal` methods are called via the `Externalizable` interface rather than via reflection, which is used for calling the read/write methods in standard serialisation. The automatic serialisation of instance fields (again via reflection) does not occur in externalisation.

- Externalisable classes do not recursively serialise their base classes. This means that the extra class information due to base classes will no longer be present. However, this does means that the `readExternal` and `writeExternal` classes must also be responsible for state that is inherited from a superclass.

However, externalisation will still result in class metadata being written to the byte-stream. Most of this is avoided by having the plans inline the byte-streams of their constituent blocks as much as possible. This is done by manually calling the `writeExternal` methods of the blocks rather than passing the block to the `writeObject` method of the `ObjectOutput` passed in as a parameter to the `writeExternal` method. This will write the contents of the block to the plan byte-stream without introducing a new class header.

This scheme means that the task of identifying the type of block being serialised must be handled manually, so that it can be reconstructed later. This is done using a single byte that is placed just before the block data. This byte corresponds to the value returned by calling `getType` on a block.

4.3.6.2 Resolving methods

The remote-call blocks of the plan need to record the identity of the method that they represent. This information was initially conveyed as a string representing the signature of

the method — i.e. its name, return type, and the types of its arguments.

Unfortunately, this scheme is inefficient, since fully qualified Java class names tend to be long in length, and transmitting them causes a significant increase in network traffic. This has been replaced with an adaptation of the protocol used in the JRMP protocol — the least-significant 64 bits of the SHA-1 hash of the signature string are used as the method identifier instead. These hashes are computed only once and are stored along with a handle to the corresponding method in a lookup-table on both client and server for efficiency.

4.4 Server forwarding

Server forwarding takes advantage of the fact that servers typically reside on fast connections, while the client-server connection can often be orders of magnitude slower. Consider this sequence of calls:

```
x = r1.f();
y = r2.f();
z = r3.f(x,y);
```

The first two methods invoked on *r1* and *r2* are returning objects that are subsequently used as arguments to a method on another remote object *r3*. In this situation, the client is acting as a router for messages between *r1*, *r2* and *r3*. It would be better for *r1* and *r2* to communicate with *r3* directly, such that no constraints are set as to which path is taken between the two servers. Also, if x or y are dead, then they need not be returned to the client.

Forwarding is also necessary for efficient aggregation of factory patterns. e.g.

```
a = r.newObject();
b = a.f();
```

Without forwarding in place, a force is needed after the call to `newObject` because `a` is used as the receiver for the next remote call — without knowing the value of `a`, it is not known where to send the remote plan, or what object to invoke `f` on.

4.4.1 Effect on the cost model

The original cost model for n calls to a single server was:

$$t_{orig} = n \cdot (l + \overline{s} \cdot b + \overline{c}) \tag{4.6}$$

This can be extended to cover a set of calls to two different servers A and B. The per-call and per-byte cost from the client to server A are denoted by l_{cA} and b_{cA} respectively, and the costs from client to server B by l_{cB} and b_{cB} have an expression like the following:

$$t_{orig} = m \cdot (l_{cA} + \overline{s_A} \cdot b_{cA} + \overline{c_A}) + n \cdot (l_{cB} + \overline{s_B} \cdot b_{cB} + \overline{c_B})$$

When server forwarding is introduced, the network connection between A and B is utilised to transfer data. The per-byte and per-call overhead of this connection is denoted by l_{AB} and b_{AB}. Some of the data required by server B originates from the client, and some of the

data returned by server B is needed by the client. This data is now sent to server B via server A, instead of directly to server B. The proportion of the total data sent to server B that is of this type is denoted by f, where $0 \leq f \leq 1$. It is assumed that all the information sent between client and server B via server A is 'piggy-backed' on top of calls to server A, so no extra latency is incurred between the client and server A. A certain amount of overhead o is also incurred with each call to a server. The cost for forwarded calls becomes:

$$t_{forwarding} = m \cdot (l_{cA} + (\overline{s_A} + o) \cdot b_{cA} + \overline{c_A}) + n \cdot (l_{AB} + f \cdot \overline{s_B} \cdot b_{cA} + (\overline{s_B} + o) \cdot b_{AB} + \overline{c_B})$$

Forwarding pays off if:

$$\begin{aligned}
t_{orig} - t_{forwarding} &> 0 \\
(m \cdot (l_{cA} + \overline{s_A} \cdot b_{cA} + \overline{c_A}) + n \cdot (l_{cB} + \overline{s_B} \cdot b_{cB} + \overline{c_B})) & \\
- (m \cdot (l_{cA} + (\overline{s_A} + o) \cdot b_{cA} + \overline{c_A}) + n \cdot (l_{AB} + f \cdot \overline{s_B} \cdot b_{cA} + (\overline{s_B} + o) \cdot b_{AB} + \overline{c_B})) &> 0 \\
n \cdot (l_{cB} + \overline{s_B} \cdot b_{cB}) - (m \cdot o \cdot b_{cA} + n \cdot (l_{AB} + f \cdot \overline{s_B} \cdot b_{cA} + (\overline{s_B} + o) \cdot b_{AB})) &> 0 \\
n \cdot ((l_{cB} - l_{AB}) + \overline{s_B} \cdot (b_{cB} - b_{AB}) - o \cdot b_{AB}) - (m \cdot o + n \cdot f \cdot \overline{s_B}) \cdot b_{cA} &> 0
\end{aligned}$$

As expected, the effectiveness of the optimisation depends mainly on the speed difference between the connection joining the client and server B, and that joining server A and server B, as expressed in terms of the per-byte cost b and per-call cost l. The greater the difference, the better the speedup that will be achieved. The optimisation overhead o and the cost of server A relaying data to and from server B have negative effects on the speedup, and might even lead to an overall slowdown.

Minimising the overhead o and arranging for as little data to be forwarded as possible between the client and server B via server A (i.e. reducing f) can help reduce the impact.

4.4.2 Implementation

Server forwarding is implemented on top of call aggregation in a preprocessing step just before execution on the remote proxies, by permitting remote plans that operate on different remote objects to be grouped together into the same remote bundle. Remote bundles are always sent to the remote proxy that handles the first remote cluster to be encountered during an in-order traversal of the tree rooted in that bundle. When a remote proxy encounters a plan that is handled by another remote proxy, it will forward the nested plan onto that proxy automatically.

The following heuristics are used to decide when to group plans with differing destinations together:

- Plans that are delivered to the same remote proxy should be grouped together
- Plans that are data dependent on one another should be grouped together

The aim is to achieve these goals while preserving the relative ordering of the calls. First, a graph is built from the plans in the remote queue, with an arc between nodes that have a data-dependence or share a remote proxy. Then the plans contained in the remote queue

are processed in sequence, starting from the second plan. At this stage, the queue should only contain clusters and bundles that contain clusters targeting the same remote object. The current enclosing bundle, which is initially unset, is tracked throughout. The algorithm is as follows:

- If there is an arc from the current plan to the previous plan
  1. Create a new remote bundle
  2. Add the previous and current plans into it
  3. Change all next-block pointers pointing to the previous plan to point to the remote bundle instead
  4. Set the current enclosing bundle to point to the new remote bundle
- Else
  - If there is a current enclosing bundle and there are arcs between the current plan and the other plans in the current enclosing bundle
    - Add the current plan to the current enclosing bundle, adjusting next-block pointers as required
  - Else
    - Set the current enclosing bundle to its parent (i.e. the bundle that encloses the current enclosing bundle) and repeat

The algorithm currently gives equal priority to arcs due to co-location and those due to data-dependencies. It is possible to prioritise one type of arc by processing all instances of that type first when traversing through the plan hierarchy, followed by the other type.

4.4.3 Example

Consider the following client code:

```
w = r1.f();
x = r2.f();
y = r3.f(w,x);
z = r4.f(w);
System.out.println(y+z);
```

r1 and *r2* are hosted on the same JVM. The four remote calls are encountered and delayed on the client side, then forced to execute before the print statement, with variables y and z remaining live. Each call would reside in a separate remote cluster since they call different remote objects. The server forwarding algorithm restructures the remote plan as shown in Figure 4.6.

Figures 4.7 and 4.8 show the communication pattern before and after the application of the server forwarding algorithm. Without forwarding, the client is responsible for receiving the values of w and x from the first two calls, and forwarding them to the next two calls. With server forwarding in place, the source of variables w and x is responsible for forwarding the values instead. Also, since w and x are not used elsewhere in the client, they do not need to be returned to the client.

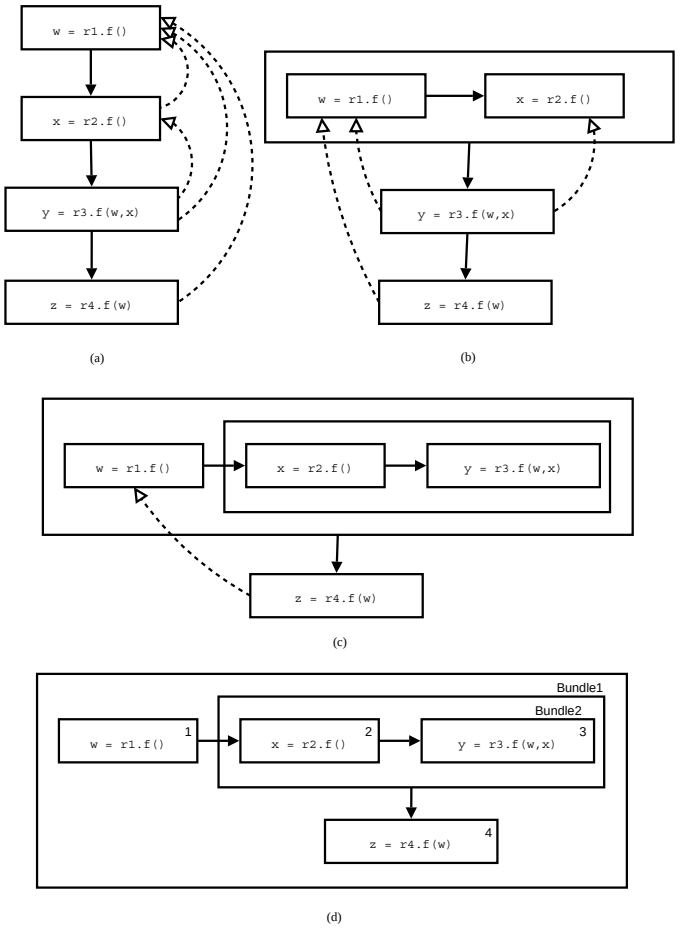


Figure 4.6: Implementation of call forwarding: a) Arcs are placed between the calls to r1–r3, r2–r3 and r1–r4 (due to data dependence), and r1–r2 (due to co-location), b) Current cluster is the call to r2 — a bundle is generated to enclose r1 and r2 due to the r1–r2 arc, c) Current cluster is the call to r3 — a bundle is generated to enclose r2 and r3 due to the r2–r3 arc, d) Current cluster is the call to r4 — no arc to r2 or r3, so the parent bundle is checked, where an arc to r1 is found, and so r4 is added to the bundle containing r1.

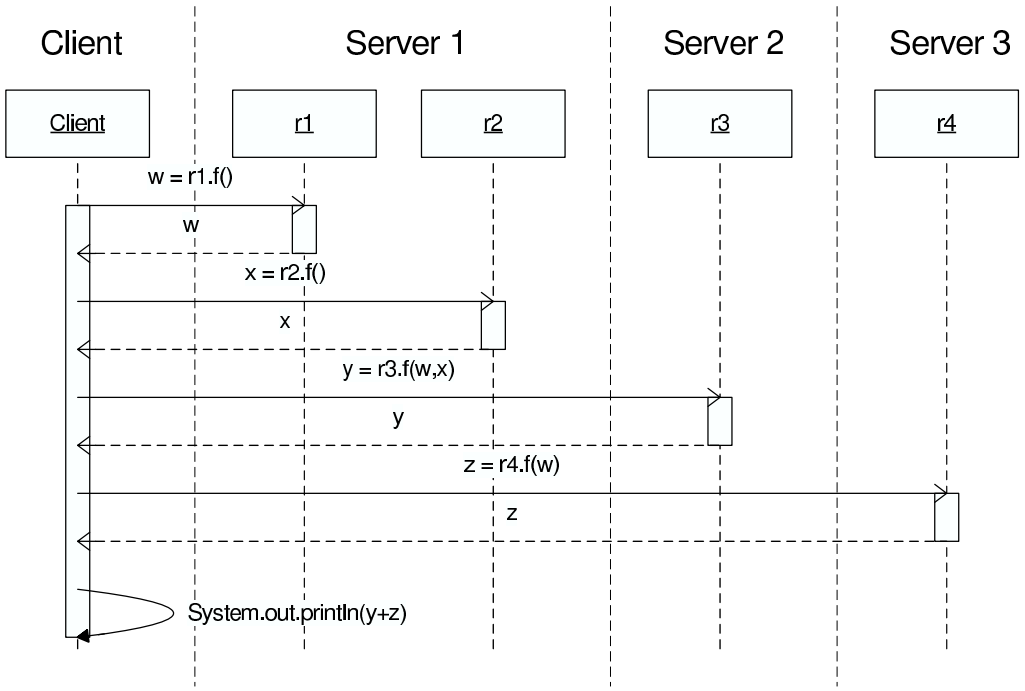


Figure 4.7: Sequence of events without call forwarding

4.5 Plan caching

These optimisations incur a substantial overhead due to factors such as:

- Overhead of the Veneer runtime
- Maintenance of dependence information for delayed calls
- Preprocessing for server-forwarding
- Transmission of remote plans and metadata

The overhead can be reduced by caching plans on both server and client sides. Instead of building up remote plans by delaying calls as they are encountered, the remote calls are replaced with the remote plans built up by delaying those calls previously. When the executor encounters these, it can simply place the remote plan directly onto the remote queue with minimal overhead.

This can only be done for clusters of remote adjacent calls, and not for remote bundles because the same sequence of remote calls might not occur next time. For example, consider Figure 4.9. During the first iteration, $r.f$, $r.g$ and $r.h$ will be aggregated, but it would not be valid to replace $r.f$ with the resulting remote bundle because the next iteration would result in $r.f$, $r.g$ and $r.i$ being aggregated. However, it is safe to replace $r.f$ and $r.g$ with a remote cluster since these always occur together.

The fact that the server has seen the plan before can also be used to implement a form of data compression. The server can keep a cached copy of the plans that it receives, returning an identifier associated with the cached plan to the client. The client from that point can simply use the identifier to refer to the plan, rather than sending the entire plan every time.

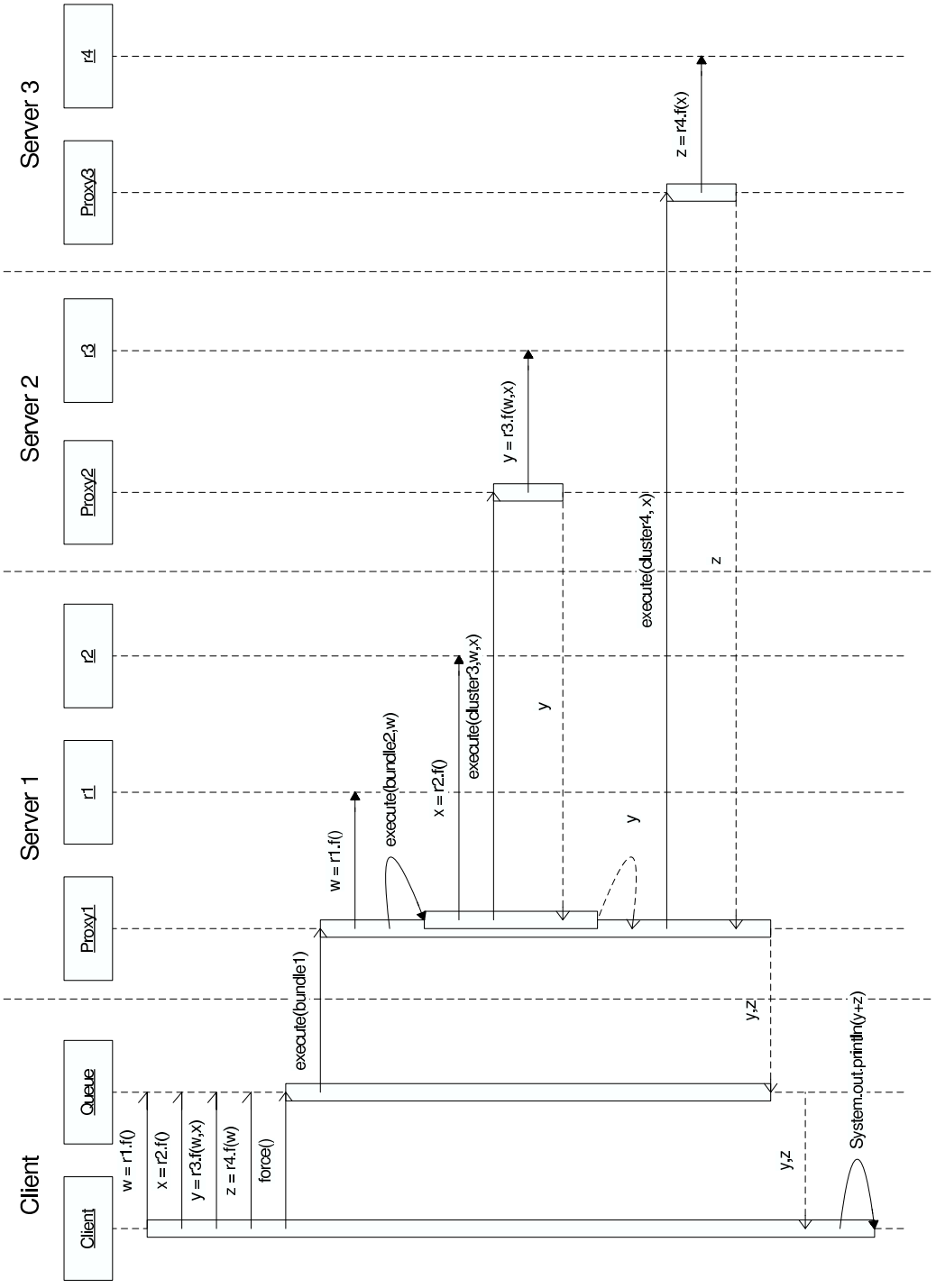


Figure 4.8: Sequence of events with call forwarding

```
for (int i = 0; i < 1000; i++) {
    r.f();
    r.g();
    if (i % 2 == 0)
        x = r.h();
    else
        x = r.i();
    System.out.println(x);
}
```

Figure 4.9: Example of a loop that results in a different remote plan on every iteration

An alternative view of this optimisation is that session façades (see Section 2.2.1) are being built on-the-fly, customised for each client. The identifier of the cached plan becomes the effective 'method name' of the dynamically constructed session façade.

4.5.1 Effect on the cost model

Unlike the other optimisations, caching does not change the communication structure in any way. Caching drastically reduces the amount of data that need to be transferred in order to convey an execution plan to the server in runs after the first run by replacing the full plan description with a simple number instead. This reduces the optimisation overhead that is denoted by o in Sections 4.3.2 and 4.4.1, leading to a better chance of achieving a speedup if there was none before, or a larger speedup to that previously attained.

4.5.2 Client-side implementation

On the client side, a list of newly constructed call clusters is maintained. After the plans are executed, the clusters are incorporated into the method plan, such that for each cluster, all paths leading to the first call in the cluster are re-routed to the cluster, and the successor of the cluster set to the successor of the last call in the cluster. The embedded remote clusters are delayed similarly to remote calls when encountered, though without the processing required to construct the plan.

After a plan is executed, a list of cache IDs is returned by the server proxy. Cache IDs associated with call clusters are assigned directly to the embedded remote clusters. The cache IDs belonging to remote bundles are stored in a global cache, which associates a *cache pattern* with a cache ID. The cache pattern is generated by traversing the remote bundle in pre-order, building up a string containing the cache IDs of the plans encountered during traversal.

The cache IDs for all plans are stored as a hash-map from remote server to the cache ID for that server. In all plans, a handle to the last remote server used and the cache ID associated with that server are retained. If the plan is invoked again on the same server, the cache ID can be reused to avoid a hash-map lookup.

When the plans have been grouped and are about to be sent to the server, the cache IDs are sent in preference to the entire plan whenever possible using the following algorithm, starting at the root of the tree:

- If the plan is an embedded cluster, the associated cache ID from the embedded cluster

is used directly:

  - If the cache ID is found, then that is used in place of the cluster
  - If there is no cache ID, then the entire cluster must be sent

- If the plan is a bundle:

  1. Compute the cache pattern of the bundle
  2. Lookup the cache ID in the global cache

  - If a cache ID is found, then it is used in place of the bundle
  - If no cache ID is found, then the algorithm is recursively applied for each plan in the bundle. The resulting remote bundle is then sent.

4.5.3 Server-side implementation

On the server side, the remote proxy maintains a cache of encountered plans, indexed by an integer identifier. If a remote plan containing uncached entities is executed, the uncached items are cached and an array of cache IDs for the overall plan is returned to the client. Since the remote plan forms a tree structure known by both the server and client during the call, cache IDs are returned to the client as a flat array of integers by performing a pre-order traversal of the remote plan, returning the cache IDs as the nodes are encountered. The client uses this information to allocate the correct IDs to the correct clusters.

4.5.4 Example

Consider the first four iterations of the code in Figure 4.9. Without caching, execution proceeds as shown in Figure 4.10. During each iteration, the plans shown in Figure 4.12 are built up before being forced to execute by the print statement. The entire plan is sent across the network every time the remote calls are forced to execute.

With plan caching in place, execution proceeds as shown in Figure 4.11. The first iteration behaves similarly in that the entire plan must be sent to the server. However, when the aggregated call returns, it returns a sequence of cache IDs, which are applied to the elements of the sent plan as shown in Figure 4.12. These IDs are now associated with these remote clusters and bundles on the client side, and can be used in place of the plan.

During the next iteration, a new remote bundle is built up since the third delayed call differs from the previous bundle. However, the first cluster inside the bundle is identical, and so its ID (which is 1) may be sent in place of the full description of the cluster. When the aggregated remote call ends, it returns more IDs for the bundle, which are again associated with the sent bundle. Note that the ID for the first cluster in the bundle is skipped since the cluster is already cached on both client and server.

On the third iteration, the remote bundle constructed is identical to that of the first. The client identifies this by matching the IDs of the sub-plans contained within the bundle (which are 1 and 2) and searching for cached bundles with the same pattern of cache IDs. Once this is done, the ID for the bundle (which is 0) is sent to the server in place of the whole bundle. No cache IDs are returned since there are no newly cached items.

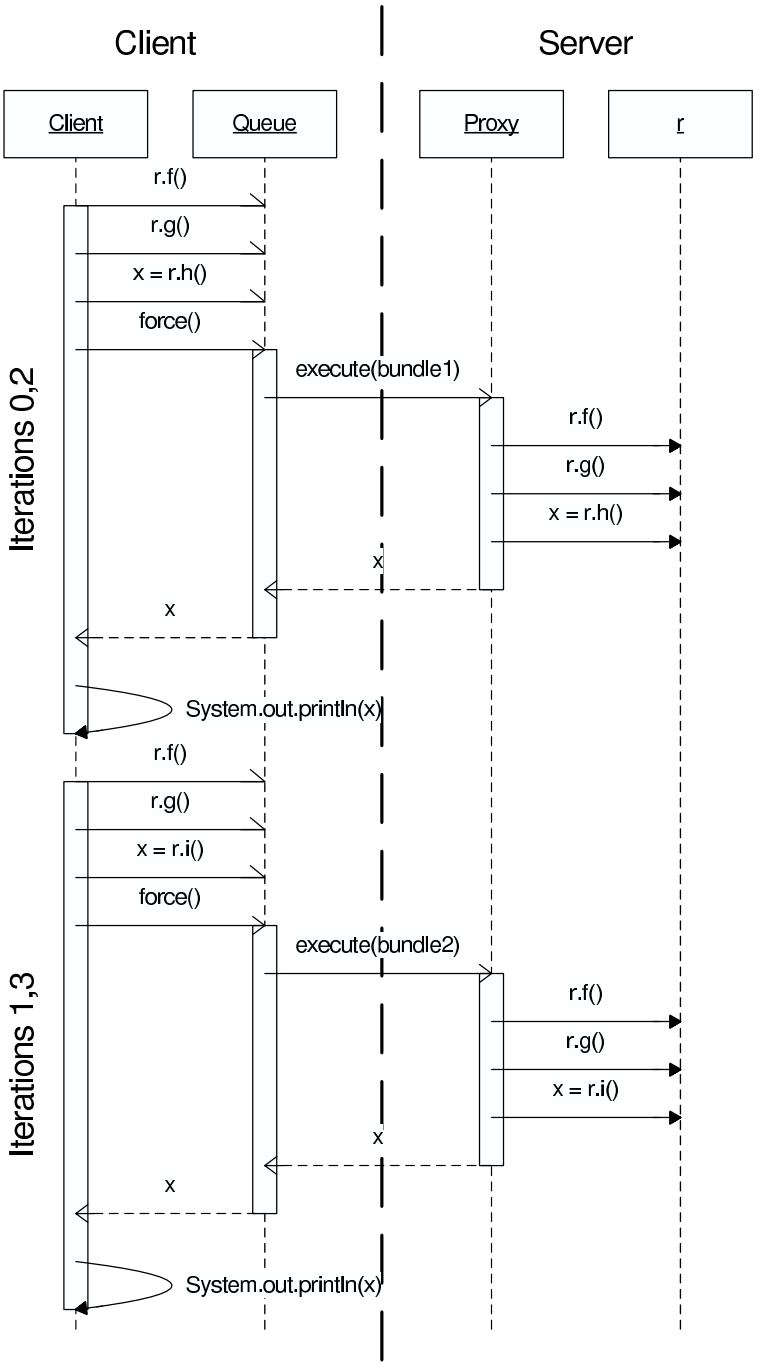


Figure 4.10: Sequence of events without caching

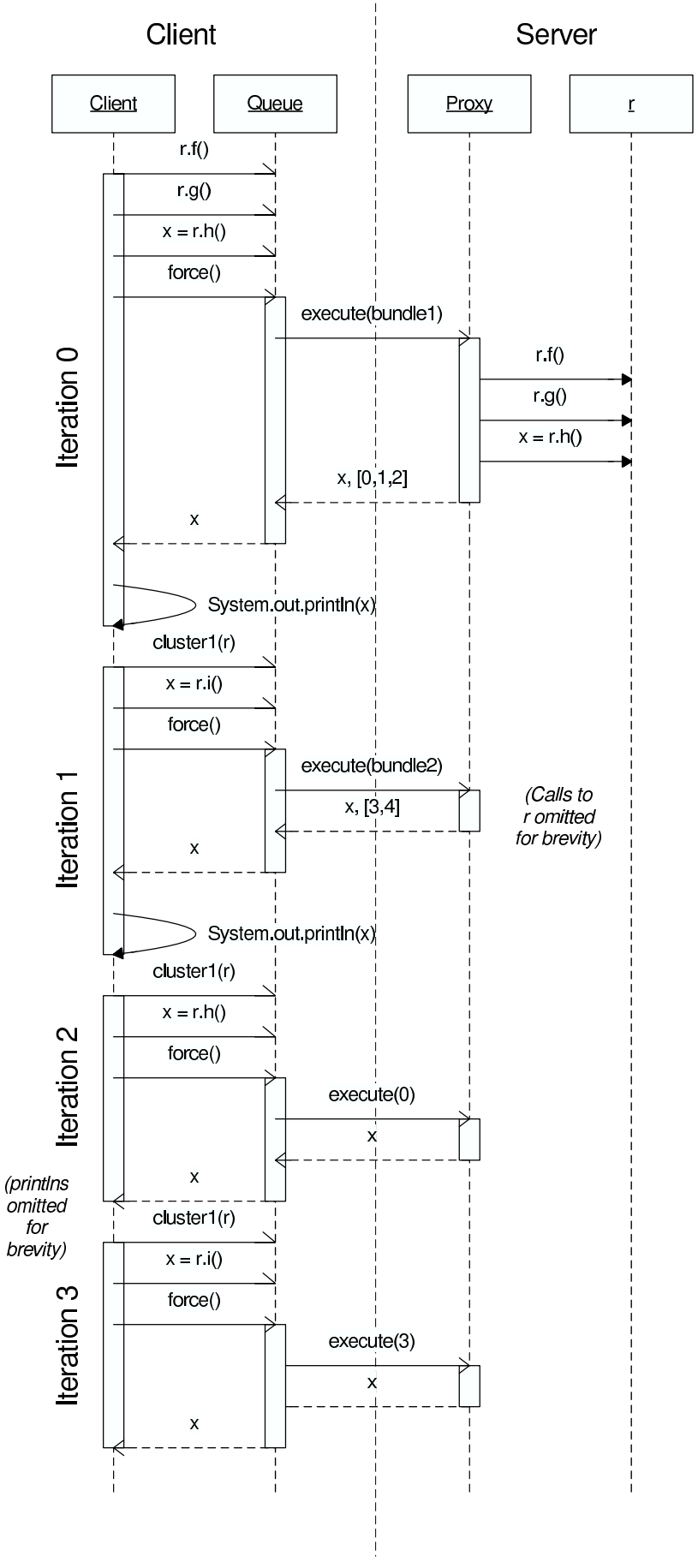
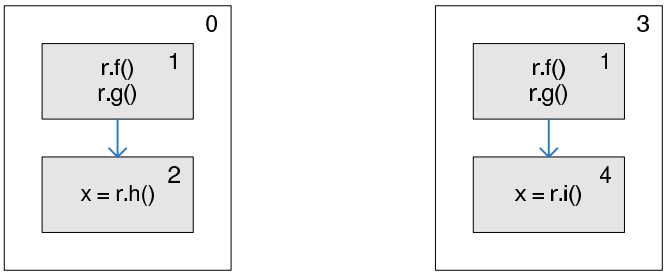


Figure 4.11: Sequence of events with caching



(a) Remote plan during iterations 0 and 2

(b) Remote plan during iterations 1 and 3

Figure 4.12: Remote plans built during the caching example — the cache ID associated with a remote plan is indicated by a number in the corner.

The fourth iteration behaves in a similar fashion. The constructed remote bundle is found to have the same cache ID pattern as the bundle with ID 3, and so the ID is sent in place of the full bundle description. Again, no new cache IDs are returned.

4.6 Maintaining the original program semantics

The DESORMI optimisations affect the semantics of the program in several respects. The main problem is that executing local code with remote calls still pending is to modify the order in which code is executed in a program. Obviously, this could have the effect of changing the overall effect of the program.

This type of problem is solved by forcing the execution of delayed calls before any incorrect reordering can occur. This is analogous to serialising execution in multi-threaded programs. However, as is the case with serialising threads, there is a trade-off between too much and too little forcing. If calls are forced too often, then performance suffers because of the increased number of remote calls made and the limited call context gathered by the runtime system. However, this is balanced by the need for correctness — delayed calls must be forced before the client executes any local code that expects the calls to have been invoked.

One point to note is that remote calls are *never* reordered with respect to each other. When remote calls are forced, all pending calls are forced immediately, and are executed in order. This means that any reordering problems that occur are always due to the interaction between local and remote code. Since server forwarding and plan caching do not change the order in which application code executes, they do not suffer from this type of problem.

This section deals with the specific problems that may arise, and how they may be dealt with. However, note that it always possible to forgo these solutions, in which case the semantics of an optimised program may differ from an unoptimised version, but this can be acceptable as long as the programmer is aware of the differences and takes them into account.

4.6.1 Direct data dependencies

The most obvious condition that necessitates the forcing of calls is when a variable that holds the result of a remote call is used by local code. For example:

```
x = r.f()
System.out.println(x);
```

This case is very easy to detect, since all one needs to do is compute the intersection of the variable define set of the delayed remote calls with the variable use set of the local code. If the result is not the empty set, then a force is required due to data dependence on the return value of a remote call.

This is sufficient even when dealing with object types, because the return value of a remote call is the result of deserialising the byte-stream from the server. This guarantees that for a remote call of the form $y = r.f(x)$, two key properties must hold after the remote call is made[1]:

1. y is the only reference to the root of the returned object.
2. There are no other references to objects that are reachable via y.

The second condition implies that in order to access substructures of the object referenced by y, y itself must be used first. Therefore, to detect uses of the object referenced by y, it is sufficient just to check for uses of y.

This scheme is too conservative in some ways, in that it does not handle aliasing efficiently, since assigning y to another variable constitutes a 'usage' of y, resulting in a force. Ways in which this may be improved are discussed in Section 6.2.9.

4.6.2 Callbacks

When using Java RMI, it is perfectly acceptable for a client to act as a remote server, and vice-versa. This creates the possibility for a callback mechanism, where a call by the client to the server will result in the server calling the client back. This can lead to many possible consistency problems.

4.6.2.1 Problems caused by callbacks

Problems that arise from callbacks can be classified into those resulting from hidden data dependencies, and those from hidden data anti-dependencies.

Data dependencies can arise due to the side-effects of a remote call, which will necessitate a force if local code is dependent on these side-effects. Consider the example in Figure 4.13. In this scenario, the server has managed to obtain a stub that points to the remote object `cRObj` residing on the client. When the client calls `f` on the server, the server makes use of the stub to call the method `changeObj` on the object `cRObj` on the client, which has the effect of modifying the object referenced by `a`. Since RMI calls are synchronous, when `g` is called, the value of `a` *must* have been changed. This causes a problem when aggregating

[1] It *is* possible for a programmer to invalidate these properties by passing a reference from within the object to some external static field or method during deserialisation, but is not considered here since this would be an abuse of the serialisation protocol.

```
// On server:
public class ServerRObj implements ServerInterface {
    ClientInterface clientRef;

    public void f() {
        clientRef.changeObj();
    }

    public void g(SomeObject obj) {
        // Do something with obj
    }
}

// On client:
public class ClientRObj implements ClientInterface {
    private SomeObject obj;

    public A(SomeObject obj) {
        this.obj = obj;
    }

    public void changeObj() {
        obj.change();
    }
}

public class Main {
    public static void main(String[] args) {
        SomeObject a = new SomeObject();
        ClientRObj cRObj = new RObj(a);

        // Export cRObj and get a stub sRObj
        // from the server...
        sRObj.f();
        sRObj.g(a);
    }
}
```

Figure 4.13: Example of code leading to a callback

calls, since if the two calls are aggregated together, then the value of `a` that is sent to the server and subsequently operated on by `g` will be that of the unchanged object, which is incorrect.

Anti-dependencies occur when local code overwrites data that is required by previous remote calls, preventing previous remote calls from being reordered to occur after the local code. This has been dealt with to an extent in Section 4.3.3.1 by saving the values supplied as arguments to previous remote calls, but this is sufficient only if the saved values represent all of the local state that was changed by the local code.

A more insidious problem occurs if a callback is made by a remote call that is reordered with respect to the local code. If the callback (which was originally executed *before* the local code) uses data that is later defined by the local code, then the callback made by the relocated remote call will pick up the values that have been overwritten by the local

code. This is more problematic compared to direct anti-dependencies since it is not always possible to make a copy of local objects. By contrast, variables that are involved in direct anti-dependencies can always be copied since they must be serialisable in order to be passed as arguments in remote calls originally.

The effects of a remote call can be observed by local code only if:

- The client and the server code run on the same JVM
- The remote code makes a callback to a remote method situated on the client, which modifies some data on the client

In the first scenario, it is generally not worth performing call aggregation at all, since there is no network communication is taking place. Such RMI calls should execute almost as quickly as calling them directly, except for the added overhead of serialisation and reflection. Any gain made by aggregating calls in this scenario is very unlikely to overcome the cost of call aggregation.

There are several methods of coping with the second scenario. Obviously, if the client does not export any remote methods, then callbacks cannot occur. However, if callbacks can occur, and local code is encountered with delayed remote calls pending, then it must be decided whether it is legal to delay the remote calls across the local code. There are two main circumstances in which this possible:

- The remote calls do not make callbacks
- The set of data defined by the callback does not intersect with the set of data used by the intervening local code

4.6.2.2 Detecting remote methods that make callbacks

One possible way of coping with callbacks is to try and detect in advance whether a remote method invocation can make a callback at all. If it can, then it can be ensured that execution is forced immediately after the remote call. The detection can be performed by the Veneer instance operating on the server, and the information conveyed to the Veneer instance running the client, where it can be cached for later use.

When examining the body of a remote method, it should be conservatively assumed that any remote call made by the remote method will result in a callback because it is not always possible to know the identity of the callback target in advance, and even if it were possible, there is always the possibility that *that* remote method will eventually result in a callback back to the client.

It is easy to determine if there are potential remote calls within the body of the remote call, since such method bodies will be fragmented under the RMI optimisation policy. However, many method bodies will also call other methods in turn, so every possible method that may be reached from the remote method must be considered. This can be done by applying analyses such as class hierarchy analysis [27] and variable type analysis [86] to each method invocation in the body of the remote method to find the possible receivers of the call, and recursively applying the analysis to the body of each new receiver found until a fixed-point is reached.

There are two problems with this approach. Firstly, the number of reachable methods can be very large. This means that the search to find remote calls can become costly, and the risk of false negatives increases with the number of methods searched due to the inaccuracies of the analysis accumulating. However, there are a number of shortcuts available. If the classes are available offline, then this information may be computed statically and looked up at runtime as metadata. The search tree may also be pruned by noting that:

- The search can stop as soon as any reachable remote call is found — it is not necessary to search the remaining methods that may be called.
- The reachability property is transitive (i.e. if f may call g and g may reach h, then f may reach h). Therefore, if it is already known that g may lead to a remote call, and f may call g, then f may lead to a remote call, and the search can stop.

The other problem is that analyses such as class hierarchy analysis assume that the entire class structure of the program is known. This is not necessarily true in Java, since Java programs can load in new classes at run-time. This problem is compounded in a distributed environment, since servers and clients can dynamically download new class definitions from each other.

Most Java classes are 'open-ended', in that they can be inherited by other classes and have their methods overridden. This means that one cannot in general be confident of accurately determining all the possible targets of a method call, which may lead to remote invocations being missed.

For example, consider a remote method that contains a call to a method f. Assume that analysis determines that the receivers of this call are the implementations of f in classes C and D, neither of which lead to a remote call. The client therefore proceeds to delay the remote call across the local code, safe in the knowledge that the remote call cannot make a callback. However, when the remote code is actually executed, it turns out that the receiver for f is actually class E, which was newly loaded after the class analysis was performed. If $E.f$ performs a remote call, then there is the risk of an unexpected callback occurring.

However, there are cases in which it is possible to precisely identify the receiver of a method call, even in the presence of dynamic loading. These occur when:

- The called method is static
- The called method is private
- For a virtual call on a variable x, the static type of x is final
- For a virtual call of the form $x.f()$, f is implemented in the static type of x, and f is final

It is therefore possible to guarantee that no callbacks will occur in a remote method provided that all method calls made in the body of the remote method and in all the methods reachable from the remote method fall into one of these four cases.

4.6.2.3 Finding the define-use set of callbacks

An alternative approach is to consider the set of data in the current method that may be accessed by remotely accessible methods on the client side. If the callbacks cannot access

any of the data that will be used or defined in the local code that is to be delayed across, then it should be safe to continue delaying remote calls regardless of whether a callback occurs.

However, there are several barriers to using this approach. One problem is that the effects of all methods reachable from all callback methods must be taken into account, leading to the problem of accurately determining all reachable methods as discussed in Section 4.6.2.2.

Even assuming that it is possible to accurately find all remotely reachable methods, there remains the problem of how to determine whether the effects of the callbacks interfere with the delaying of remote calls. One way might be to use a points-to analysis such as that provided by Spark [50] to find the sets of objects that are accessed by the client and callback code, and test for an intersection. However, given the nature of pointer analyses, this two-pronged approach is unlikely to work for several reasons.

A points-to analysis builds up graphs that describe the objects that variables *may* point to, with the accuracy of the analysis measured by the size of the point-to sets — the more refined the points-to sets are, the more accurate the analysis is considered. The degree of accuracy depends on the context information available to the analysis. If such information is lacking, then a variable may point to anything that is of a compatible type. While even this may be enough for some limited scenarios, the general case requires more information to achieve better accuracy.

One important source of information is the effect of method calls on the points-to graph of the caller (often provided in the form of partial-transfer functions [94]), and this will likely be the main stumbling point of current points-to analyses with regard to the callback problem since remote calls are terminated on the client side with a call to the network I/O libraries on the client. The link between the remote call on the client to the remote method on the server, and from the remote call in the remote method back to the callback method on the client, is therefore missing. This means that context information is missing for both the caller and the callback method. The caller is missing information on the effect of the remote call on the points-to graph, while the callback is missing information on the context in which it was called.

4.6.2.4 Escape analysis

An approach related to points-to analysis that is being considered for implementation is escape analysis [19, 92, 80]. Callbacks by definition must run in a separate thread to the client code, since the client thread will be blocked for the duration of the remote call that led to the callback. This means that in order for a callback to be able to influence the course of execution of the original caller, the data accessed by the caller must be accessible from another thread. If accessed data never escapes from the confines of the current thread, then the effects of the callback cannot be noticed by the caller, making it safe to proceed regardless of whether a callback occurs, otherwise a force must occur.

4.6.2.5 Compensating for callbacks

It might be able to compensate for the effects of callbacks on the arguments of remote calls by using the fragmentation framework to intercept operations that may result in an inconsistency when executing a group of aggregated calls. The following protocol is proposed

as a solution.

Consider a client c. Suppose it aggregates a number of remote calls together and sends the resulting plan p to the server r. The set of objects used within the plan is denoted O_{used}. Now suppose the server r makes a callback to a method *callback* on c while executing the nth remote call f_n, either directly or indirectly.

Since c is under the jurisdiction of the fragmentation framework, it is possible to detect that a remote method has been called while c is still executing a remote call, although it cannot determine if the call was part of a call-chain originating from c itself. It must be conservatively assumed that it is.

If this occurs, the callback method is run to completion, but just before it returns, the executor sends a message to r that includes the updated value of all objects in O_{used} that may have been modified by m (in the most conservative case, this can be all of O_{used}). When r receives this message, it uses the received values to update the corresponding objects in all calls after f_n.

An illustration of this protocol is shown in Figure 4.14.

4.6.2.6 Ignoring callbacks

A case can be made for ignoring the effects of callbacks by noting that between unsynchronised threads, there is no guarantee as to when variables by one thread will be visible to another thread (this is covered in more detail in Section 4.6.5). The caller and the callback cannot be synchronised on the same lock either, since this will result in a deadlock.

Unfortunately, this argument fails in practice since synchronisation blocks will always occur during the path from the caller to the remote callee back to the callback, in the I/O libraries if nowhere else. The end of a synchronisation block has the effect of committing variables to the shared memory, and so the effects of a callback will always be visible.

4.6.3 I/O ordering

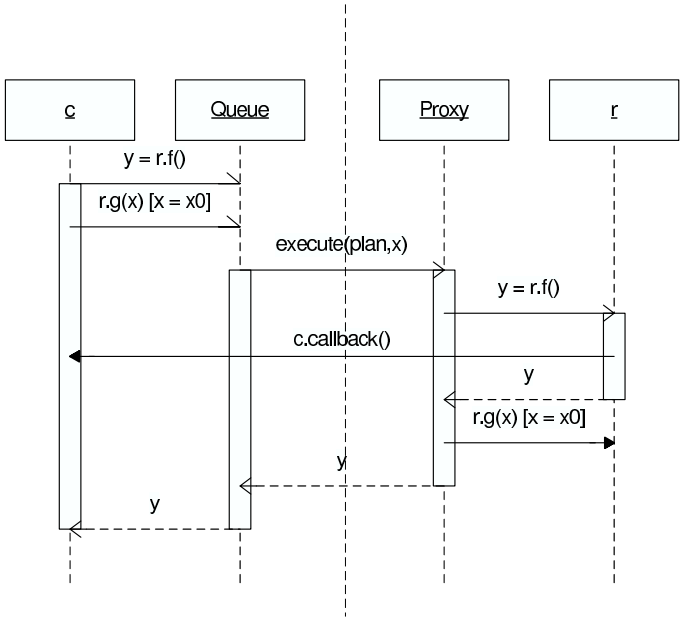
If it is permitted for remote calls to be delayed across local code that performs I/O, then inconsistencies may occur. For example, consider the following code:

```
remoteWindow.println("A"); // Remote call
System.out.println("B");   // Local call
```

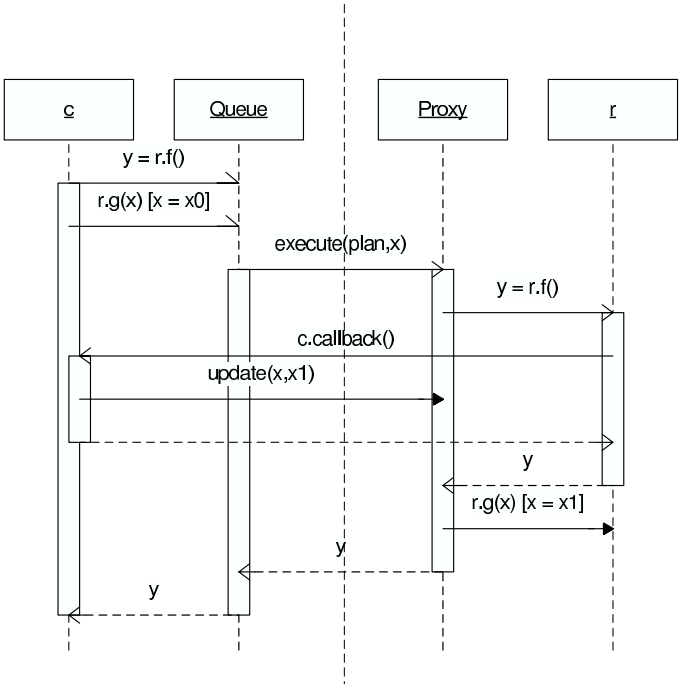
If the user can observe both the output of `remoteWindow` and the local console, then 'A' should appear in the remote window before 'B' does in the local console. However, if the remote call was to be delayed across the local call, then the reverse would occur.

This is solved by forcing all delayed remote calls to execute before any local I/O occurs, which has the effect of creating a form of synchronisation barrier that prohibits the reordering of local and remote I/O. Since remote calls are never reordered relative to each other, any I/O performed by remote calls will always occur in the original order.

This scheme may be overly conservative in some situations, since the relative orderings of I/O might not be observable or important. However, since I/O is by its very nature external, it is generally not possible to determine whether the relative order of I/O between two processes is observable in some way simply by examining the application program.



(a) Without parameter updating



(b) With parameter updating

Figure 4.14: Compensating for callbacks — in the original course of events as depicted in (a), `r` is unaware of the modification to `x` caused by the callback, and proceeds to execute `g` with the old value of `x`. An extra step is needed as shown in (b) — the instance of Veneer on the client side must send an updated copy of `x` to `r` after the callback method has finished executing.

This leads to the problem of determining whether a block of code will perform any I/O. Since I/O requests ultimately end with native code being called, and the effect of native calls cannot be determined from the perspective of the Java runtime anyway, all native calls made via the Java Native Interface [51] must lead to an immediate force. However, the problem of determining which methods are called (covered in Section 4.6.2.2) within the code will arise.

The simplest solution is to make any method call within the local code block lead to a force. However, this is a very conservative approach, and will result in many unnecessary forces. A compromise might be to special-case certain common method calls that are known to be safe, such as string operations. In Section 6.2.4, alternative ways of dealing with this problem are discuss.

4.6.4 Exceptions

Consider the following code:

```
a = 0;
try {
    r.f();
    a = 1;
} catch (Exception e) {
    System.out.println(a);
}
```

Now assume that the call to `r.f` leads to an exception being thrown. The exception handler should print 0, since the code that sets `a` to 1 has not been reached before the exception. However, if the remote call is delayed so that it executes after assigning 1 to `a`, and then it throws an exception, 1 will be printed instead.

This is due to the rearrangement of the code leading to code that should not have been executed actually executing. There is always the possibility of this problem arising when working with Java RMI, since remote calls can always fail due to network problems, and most Java statements are capable of resulting in an exception anyway.

Two approaches to this problem have been considered. The first is to analyse the effects of the local code. If it can be determined that the effect of the code is invisible from the point after the exception handler is reached, then it will be safe to delay across the code. This can be determined by performing a liveness test on the define set of the code block. If any of the defined variables are live during or after any of the exception handlers, then remote execution should be forced immediately. If there is no exception handler, the call-stack will be unwound if an exception is thrown, and so all local variables in the current method can be considered dead. This approach to optimising in the presence of precise exceptions was also used in [40].

Another approach is to think of the incorrect execution of the local code as a form of speculative execution. Correct speculative execution requires the ability to undo operations in case of incorrect execution, which is not provided by Java. Nevertheless, it should be possible to undo simple operations such as assigning new values to variables by keeping hold of the old values, and 'unrolling' when it is discovered that the code should not have

been executed. Undoable operations must be thread-safe, either by ensuring that processed data does not escape from the current thread, or by protecting the code with a lock. This is necessary since another thread could otherwise observe the result of incorrect execution before the unrolling takes place.

4.6.5 Multi-threading

Since threads are an inherent part of the Java platform, optimisations must respect the interactions between instructions executed in separate threads of a multi-threaded program. This section considers the interactions that can occur between threads.

4.6.5.1 Synchronised code

Consider two blocks of code B_1 and B_2 in threads t_1 and t_2 respectively. If B_1 and B_2 both synchronise on the same lock, then the overall execution order is guaranteed to be either $B_1 \rightarrow B_2$ or $B_2 \rightarrow B_1$, with no interleaving of instructions from the two code blocks.

The Veneer policy fragments around the monitorenter and monitorexit instructions that form the boundaries of synchronised code blocks at the bytecode level. When one of these instructions is encountered, any delayed calls are forced to execute before the lock is acquired or released.

If B_1 executes first, then by the time B_2 has the opportunity to acquire the lock and execute, B_1 must have finished executing completely since forcing must occur before the lock previously acquired by B_1 is released. Since the order in which instructions are executed within B_1 is irrelevant to B_2 as long as the end result is the same, this case presents no problem in terms of correctness provided that the execution of B_1 is correct. The same situation applies if B_2 is executed first.

Remote calls are forced before synchronised blocks in case any of the delayed remote calls assume that the lock has not been acquired. For example, suppose that a remote call r that occurs before B_1 makes a callback that synchronises on the same lock as B_1. If r is delayed so that it is executed within B_1, then a deadlock will occur since B_1 holds the lock required to execute r, yet r must be completed before B_1 exits.

4.6.5.2 Unsynchronised code

Consider two blocks of code A and B in threads t_1 and t_2 respectively. If A and B do not synchronise on the same lock, then the instructions in A and B may be interleaved. Any interleaving between A and B is permissible provided that the relative ordering of instructions within a single thread remains the same.

Suppose that a_i and b_i are remote calls occurring in blocks A and B respectively, and the original instruction interleaving was:

$$a_1, b_1, a_2, b_2, a_3, b_3$$

If the call aggregation optimisation is applied, then the resulting instruction interleaving on the client will effectively be:

$$(a_1, a_2, a_3), (b_1, b_2, b_3) \text{ or } (b_1, b_2, b_3), (a_1, a_2, a_3)$$

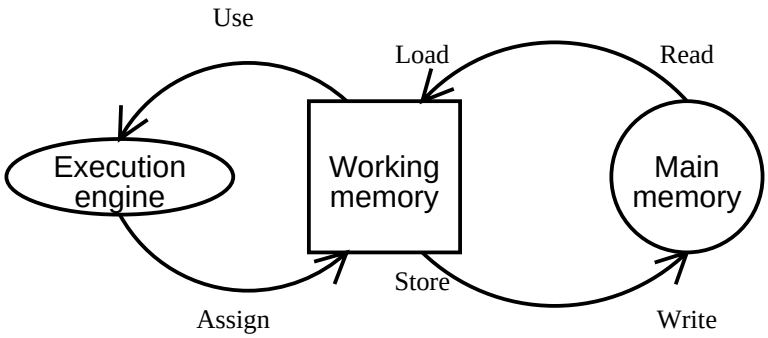


Figure 4.15: Memory actions in the Java memory model

Either of these are valid interleavings, since there is no requirement that the instructions from A and B *must* be interleaved. However, when the calls are actually executed on the server, the execution of the method bodies may well become interleaved again. Again, this is acceptable, since the ordering of calls from A and B relative to calls in the same block will remain the same.

However, when remote calls are delayed across local code, the relative ordering of the local and remote code will change. Since the delaying of remote calls across local code is prohibited if there is any possibility of I/O occurring in the local code, the only way in which another thread can observe this reordering is by the effect of the instructions on data. For example, if a remote call assigns to a variable x followed by local code that assigns to y in the original ordering, then the assignment to x will occur after the assignment to y in the reordered code.

Fortunately, the Java memory model [38, 54] permits this type of reordering to occur. Conceptually, each thread has a private working memory that acts as a buffer between the executing engine that executes the instructions in the thread and the main memory shared by all threads (see Figure 4.15). When a value is generated by an instruction, it is *assigned* to the working memory, and later written back to main memory in a two-phase commit operation where the working memory *stores* the variable and the main memory *writes* it.

In the absence of any synchronisation, there is a non-deterministic period of time between the *assign*, *store* and *write* operations. Although there are certain constraints regarding the relative ordering of memory actions for a single variable, there are no such constraints between actions on different variables. This means that even if variable a is *assigned* before variable b, it is possible that b will be *stored* and/or *written* before a.

Another potential problem might occur if the result of a remote call is assigned to a variable x, and subsequent local code also assigns to x. The first assignment never occurs when the call aggregation optimisation is performed, since x is dead at the point where the remote call originally occurred. The memory model does not help here, since assignments to a single variable must be totally ordered. However, since it is possible for the observing thread to miss the first assignment in the original ordering if both assignments were performed before the observing thread regains control, this is still acceptable.

4.6.5.3 Waits and notifications

There are other forms of synchronisation that can be performed in addition to those provided by the synchronized keyword in Java[2]. For example, consider the following code:

```
// T1:
A;
o.wait();
B;

// T2:
C;
o.notifyAll();
D;
```

Suppose that thread t_1 executes A and then blocks on the object o. Another thread t_2 executes C and awakens thread t_1 by issuing a notification on o. Thread t_1 then proceeds to execute B.

This leads to a dependency between the threads — by the time t_1 executes B, thread t_2 must have finished executing all of C in order to reach the notification that awakens t_1. A problem occurs if C contains a remote call r which is delayed across the notification, since this results in r being reordered with respect to B. This can be resolved by forcing remote calls to execute before notifications, which preserves the relative ordering of the call and the code following the wait statement.

4.6.5.4 Joins

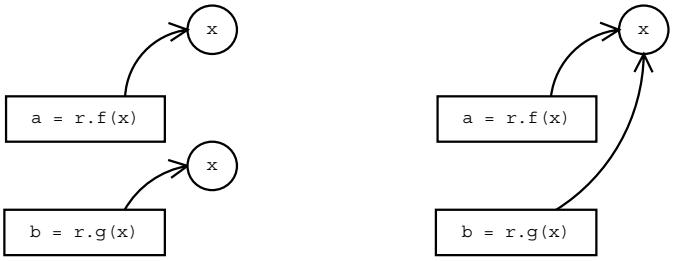
Another form of synchronisation is the join construct, where one thread waits for another thread to die before continuing. This case requires no special handling since threads generally die when their main method finishes. Since delayed remote calls are forced when the end of a method is reached (see Section 4.3.3.4), all remote calls issued by the dead thread are guaranteed to have been executed by the time the waiting thread continues execution.

This mechanism does not work if the thread is forcibly stopped using the `stop` method. Since this method has been deprecated for a long time, no effort has been made to cope with this. However, one possible way of ensuring that all delayed remote calls are forced is to add a new protocol whereby the caller of the stop method must first send a message to the currently active executor in the thread to be terminated. That executor must force any pending remote calls as soon as possible, send an acknowledgement, then block indefinitely. Once the acknowledgement has arrived, the thread may be terminated.

4.6.6 Difference between local and remote semantics

A local call and a remote call differ in the way that they pass objects as parameters. Local calls receive their parameters by reference, whereas remote calls receive them by copy. Consider the following code fragment, where `r` is a remote object:

[2]Synchronised objects are based on Hoare's monitor model.



(a) Without aggregated calls

(b) With aggregated calls

Figure 4.16: Differences in structure sharing

```
a = r.f(x);
b = r.g(x);
```

Using reference semantics, this would be equivalent to:

```
x' = x.clone();
a = r.f(x');
x'' = x.clone();
b = r.g(x'');
```

Note that any changes made to x' by f are not propagated to x or x'', and similarly the effects of g on x'' are not propagated to x (see Figure 4.16(a)). However, by aggregating calls, the original code is transformed to:

```
x' = x.clone();
a = r.f(x');
b = r.g(x');
```

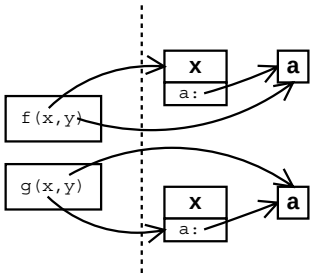
Now, although the effects of f and g on x' still do not affect x, the effect of f on x' *will* affect the functioning of g (see Figure 4.16(b)). It is therefore only safe to aggregate the two calls without copying the parameter if f does not change the value of its parameter.

However, copying is problematic in Java because not all objects implement the `Cloneable` interface that signals that it is acceptable to use `clone` on the object, and of those that do, the default implementation only performs a shallow copy of the object. Unfortunately, neither shallow nor deep copying will provide the correct semantics when dealing with multiple RMI calls with multiple arguments. For example, consider the following code:

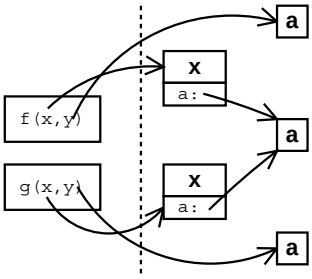
```
x.a = y;

r.f(x, y);
r.g(x, y);
```

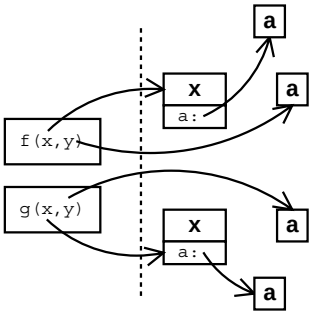
Suppose these two calls are aggregated and sent to the remote server. The server receives one copy of x and y. The arguments received by f are denoted as x' and y', and those



(a) Desired result



(b) With shallow copying



(c) With deep copying

Figure 4.17: Copying data structures used as arguments to RMI calls — (a) The desired result, (b) The result of shallow copying, (c) The result of deep copying

received by g as x'' and y''. If copies of x and y are made, then the following properties should hold for the copies (see Figure 4.17(a)):

$$x' \neq x'' \tag{4.7}$$

$$y' \neq y'' \tag{4.8}$$

$$x'.a \neq x''.a \tag{4.9}$$

$$x'.a = y' \tag{4.10}$$

$$x''.a = y'' \tag{4.11}$$

If x and y are shallow-copied, then equations 4.9, 4.10 and 4.11 will not hold since $x'.a$ and $x''.a$ both refer to the same object as the original y, and neither will be equal to y' or y'' (see Figure 4.17(b)).

If a deep-copy is made of x, then equations 4.10 and 4.11 would no longer hold, since $x'.a$ would no longer refer to the same object as y', and similarly for $x''.a$ and y'' (see Figure 4.17(c)).

4.6.6.1 Local RMI calls

The easiest solution to this problem is to first transport the plan over to the server, and then call the methods within the server via the RMI mechanism. Although making RMI calls to an object on the same host as the client is much faster since there is no need to cross

an external network, the time required for serialising and deserialising arguments remains unchanged, which can cause considerable slowdown.

4.6.6.2 Copying using serialisation

Another way to properly copy parameters is by copying the subset of the state used by the call into an array, serialising the array, then deserialising it again. The values in the new copy of the array may then be used freely by a method without any risk of destroying the original values. Although this still involves an extra serialise-deserialise cycle, it is more efficient than using local RMI calls on the server side since the call no longer has to go through the stub and skeleton.

An alternative that has been tried is to repeatedly deserialise the saved state that was supplied as part of the current remote cluster before every remote call. However, when benchmarked, this generally performed worse since state that was not used in the current remote call was also being deserialised. The excess could not be used in subsequent calls either, since it may contain references to state that *was* used and perhaps modified.

4.6.6.3 Avoiding parameter copying

A parameter to a remote method call need not be copied if any of the following are true:

- The parameter is a value type (which is always passed by copy)
- The parameter is immutable
- All objects reachable via the argument are dead after the call
- The method is guaranteed not to modify the parameter

Checks for a subset of the first three conditions are currently implemented. Value types and common object types that are known to be immutable (such as instances of `String` or the value wrapper types such as `Integer`) are identified as being safe to use without copying.

The notion of 'flat-types' is introduced, which are types that do not contain any references. These include common types such as arrays of primitive types such as `int`. If only flat-types are used as arguments for the current and subsequent calls, then if an argument is dead and is not aliased by any other argument (which can be done simply by checking if any of the other arguments also reference it), then it is safe to avoid copying the argument. A more generalised approach would to use balloon types [3] or ownership types [21] to prove that no aliases to the internal structures of an object can exist.

4.7 Current status of the DESORMI optimisations

Call aggregation, server forwarding and plan caching have all been implemented as a set of policies, executors and extra support classes on top of the Veneer framework. They are fully functional, capable of optimising both adjacent and non-adjacent remote calls, but some of the measures needed to maintain semantic correctness have yet to be fully implemented. Currently implemented are:

- Copying of remote parameters by serialising-deserialising to obtain copy-by-value semantics
- Variables that may be overwritten by intervening local code are preserialised before the local code is entered
- Method calls occurring in local code between remote clusters conservatively forces execution in all cases

The following are currently incomplete:

- Conservative callback detection has been implemented by noting when a remotely accessible method is called while executing a remote plan, but no corrective action is taken.
- Escape analysis has not yet been implemented to detect the possibility of callbacks interacting with local code when delaying remote methods across intervening local code. This is mainly due to the late arrival of pointer-analysis facilities in the Soot framework.
- Variable liveness and exception handler information is available from the Veneer runtime, but is not yet used to check the validity of executing local code in the presence of exceptions.

4.8 Evaluation

It has proven surprisingly difficult to find examples of substantial RMI applications in general, and even more difficult to find ones that stand to benefit from the DESORMI optimisations. The problem with most examples is that they are structured to focus all remote communication into a small number of remote calls per server, leaving little opportunity for any aggregation to occur. This is understandable since it is keeping with the concepts described in Section 2.2.1 for performance, but leaves little to work with.

An example of an unsuitable benchmark is the KaRMI test suite, which contains a few tests that measure the speed at which objects can be sent to a remote server, effectively testing the performance of the serialisation protocol rather than the performance of the program structure. It also contains implementations of a successive over-relaxation (SOR) algorithm and the Salishan problems, which initially seemed more promising, but are examples of parallel algorithms. The main communication patterns involved are data scatter/gather (where data is divided up and different sections sent to different servers for processing, and the processed data is returned to the client and reassembled) and border exchanges (which occur when data from a data fragment assigned to one server is needed to process the data on another server), which offered little that DESORMI could work with since there are no aggregation or forwarding opportunities.

4.8.1 Test procedure

The DESORMI optimisations have been tested with two main examples. The first example is vector arithmetic, which is a simple synthetic benchmark that illustrates the potential of

the optimisations. The second example is a Multi User Domain application, which represents an example of a non-trivial program found in the wild that is naively written and so may benefit from the DESORMI optimisations.

The tests were performed using the Linux version of the Sun JDK version 1.4.1_01 over two network types:

- A private Fast Ethernet network with no other traffic (ping time is 0.1 ms, measured bandwidth is 10.03 MB/s)
- Over the Internet via a slow symmetric DSL connection (ping time is 98 ms, measured bandwidth is 10.7 KB/s).

The client machine in all tests was an Athlon XP 1800+ based PC. The server machines were all Intel Pentium-III based PC machines, but unfortunately vary slightly in configuration, with clock speeds in the range of 500–700 MHz. The exact configurations used are given with the test descriptions. All machines were lightly loaded during the tests.

The client runs on the Sun HotSpot JVM in client mode, which is restarted with each trial. Servers run Sun HotSpot in server mode, and are not reset between trials. For each test, several dry runs are made to give the server the opportunity to 'warm up' before any measurements are made.

For each set of parameters with each test, 3 trials of 1000 iterations were performed, and the mean time per call taken as the result. Since the number of samples is large, the confidence interval is small and is not shown to avoid clutter on the graphs. The times were measured by recording the value of the time-stamp counter of the CPU.

4.8.2 Vector arithmetic — call aggregation

The DESORMI optimisations have been evaluated a simple synthetic benchmark in which the server object provides a single method takes two equal-sized arrays of type `double`, adds them together, and returns the resulting array. In order to test aggregation, the client application executes a sequence of remote calls of the form:

```
tmp1 = r.add(v0, v1);
tmp2 = r.add(tmp1, v2);
result = r.add(tmp2, v3);
```

This benchmark makes it possible to easily observe the effects of varying the size of the data, the number of calls aggregated and various parameters of the framework.

4.8.2.1 Test configuration

The tests include a baseline configuration with no aggregation occurring, and configurations containing from 2–5 aggregated calls. For each configuration, the vector size is varied from 1 to 1024 doubles, doubling the vector size at every step. Both Ethernet and DSL connections have been tested.

The results before and after applying the framework on the benchmark program are shown. Results for a 'hand-optimised' version of the tests (where manually aggregated methods are provided on the server) are provided for comparison purposes.

The server used with the Ethernet connection was a 650MHz Pentium-III PC, while the server used with the DSL connection was a dual-processor 700MHz Pentium-III PC.

4.8.2.2 Results

As can be seen in the results in Figures 4.18–4.22, the optimisations generally result in an overall speedup whenever any aggregation occurs. The exceptions occur when an Ethernet connection is used, with two aggregated calls and argument size of less than 400 bytes. This is due to overhead.

In the baseline case with no aggregation occurring, a slowdown will occur due to the same overhead being occurred but without any compensating speedup from call aggregation. This is easily observable in the Ethernet test, but is not evident in the DSL test due to the overhead being orders of magnitude smaller compared to the communication times.

If the results of the hand-optimised versions are compared against those for the automatically optimised versions on the Ethernet network, there is a discrepancy of about 0.5 ms per call, which is mainly due to interpretive overhead from the Veneer virtual JVM and the call-delaying/plan-building mechanism. However, this overhead remains constant, and is therefore all but invisible when operating across the Internet via DSL, since it has much greater latencies and is subject to variations that could easily eclipse the 0.5 ms overhead.

4.8.3 Vector arithmetic — server forwarding

The vector arithmetic server is also used to test server forwarding. The following remote calls are executed on the client:

```
temp = r1.add(v1, v2);
result = r2.add(temp, v3);
```

In this configuration, `temp` is forwarded directly from `r1` to `r2` without ever reaching the client. `v3` is sent from the client to `r1`, and then forwarded to `r2`. `result` is conveyed back to the client via `r1`.

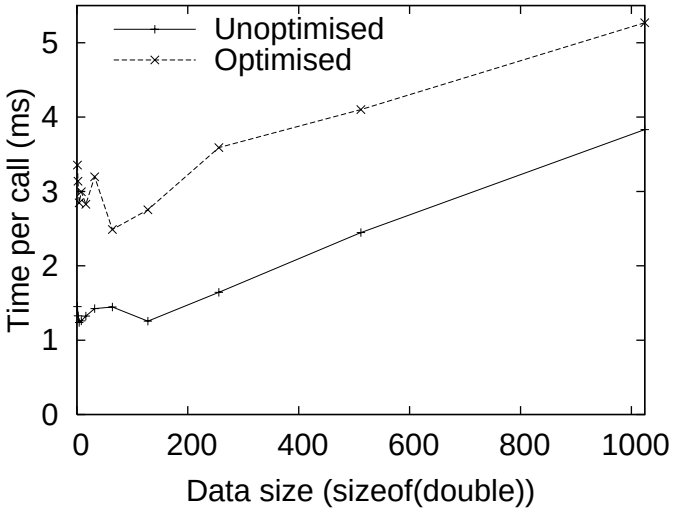
4.8.3.1 Test configuration

The tests consist of varying the size of the vector, changing the type of network connecting the server and the client, disabling plan caching, and forcing server-side variable copying to occur (even though it is not needed in this case since the arguments are not reused).

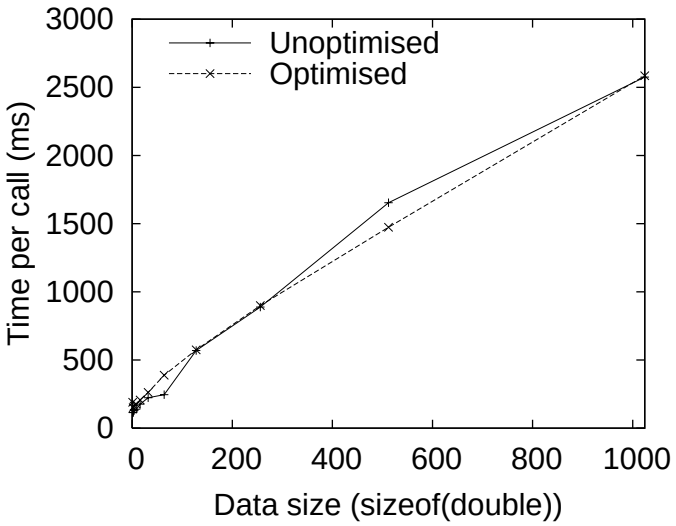
The pair of servers used when testing with an Ethernet client-server connection were a 650MHz Pentium-III machine and a 700MHz Pentium-III machine, while the servers used when testing the DSL client-server connection were a 500MHz Pentium-III machine and a dual-processor 700MHz Pentium-III machine. The faster machine was targeted as the forwarding server in both cases. The connection between servers is always a Fast Ethernet connection.

4.8.3.2 Results

As can be seen in Figure 4.23, server forwarding provides a large performance boost when the client and servers are connected over the Internet using a DSL line, but causes significant slow-down when the hosts are connected via Fast Ethernet.

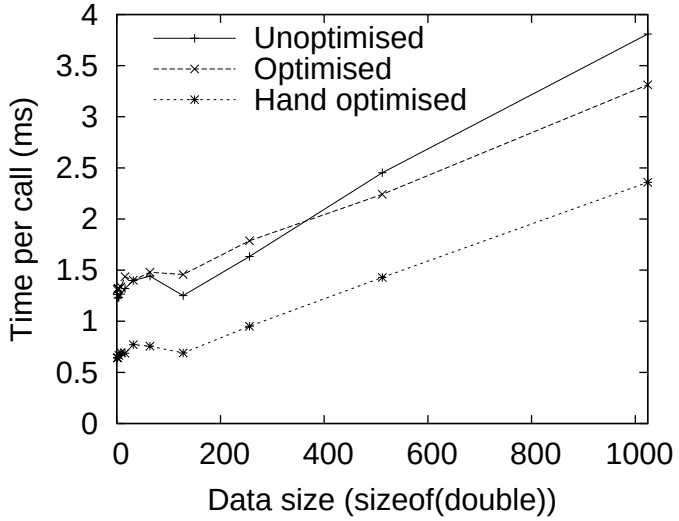


(a) Ethernet

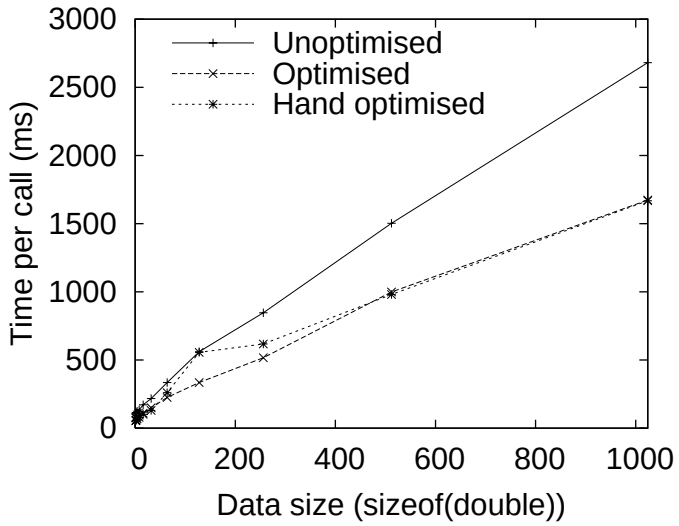


(b) ADSL

Figure 4.18: Results for the vector arithmetic example with no call aggregation

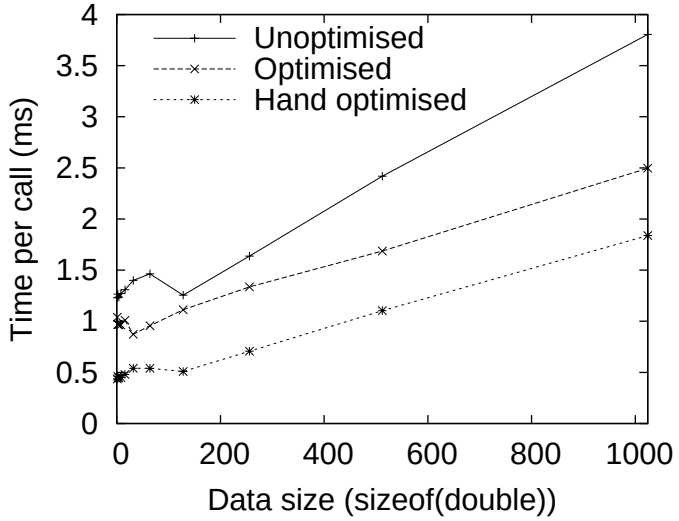


(a) Ethernet

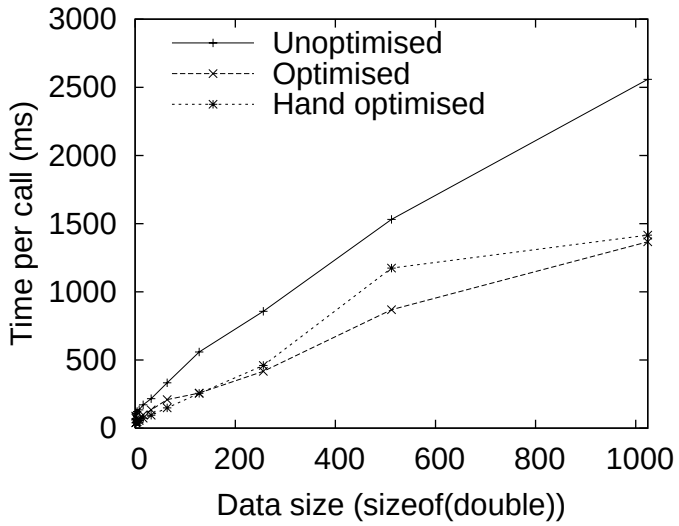


(b) ADSL

Figure 4.19: Results for the vector arithmetic example with two calls aggregated

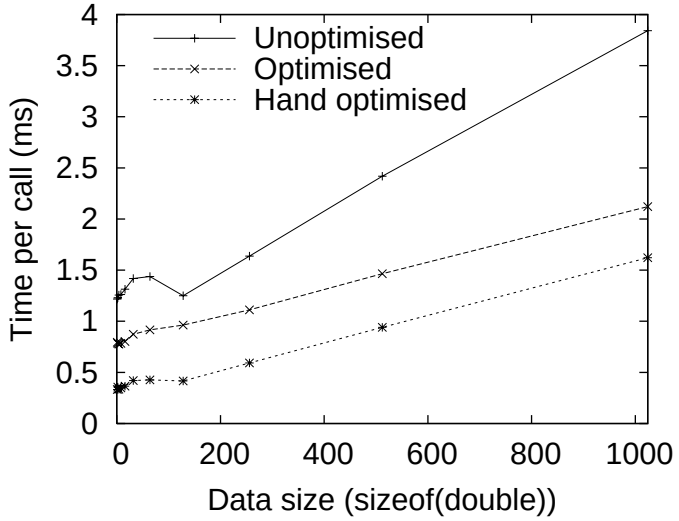


(a) Ethernet

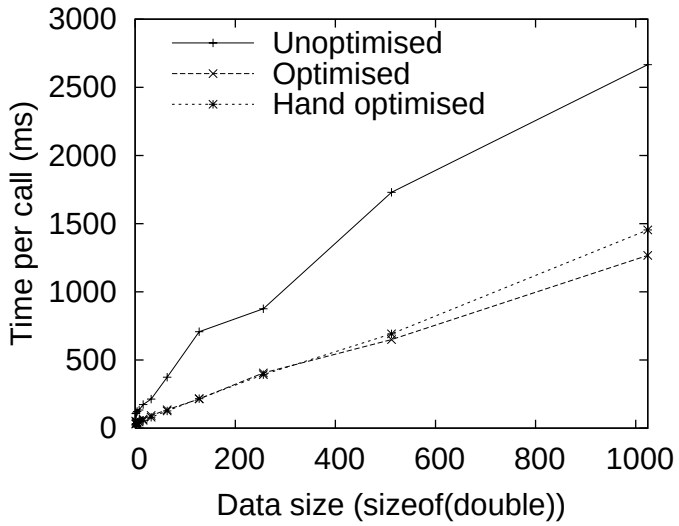


(b) ADSL

Figure 4.20: Results for the vector arithmetic example with three calls aggregated

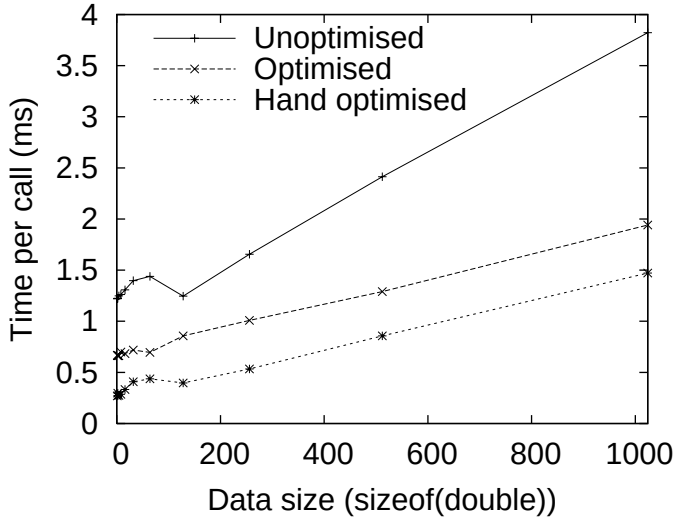


(a) Ethernet

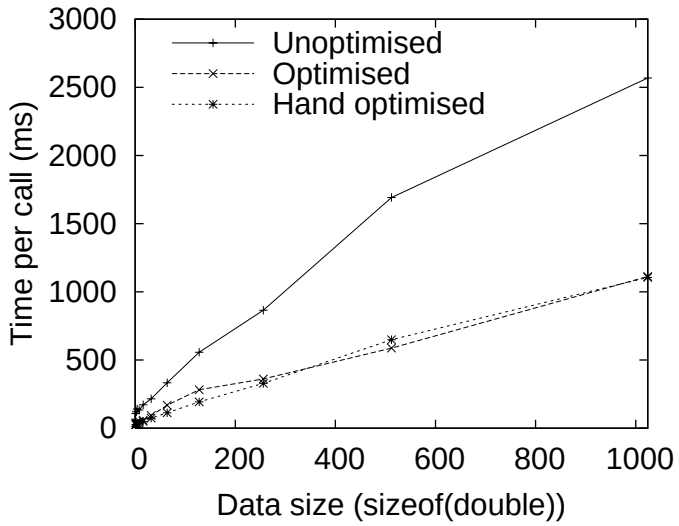


(b) ADSL

Figure 4.21: Results for the vector arithmetic example with four calls aggregated



(a) Ethernet



(b) ADSL

Figure 4.22: Results for the vector arithmetic example with five calls aggregated

Server forwarding enables the two servers (which are connected via a fast Ethernet connection) to communicate directly with one another rather than via the client. In the case where a DSL connection is used by the client to connect to the servers, there is a large increase in performance since the intermediate variable *temp* is no longer being passed back and forth along the slow ADSL connection, and is instead passed over the fast server-to-server Ethernet connection. Since *temp* is never passed back to the client, there is also an overall saving in the amount of data transferred.

However, when all the hosts are connected via Ethernet, there is little gain since the low latencies mean that server-to-server communications via the client are only marginally slower than a direct connection. Since the amount of runtime overhead remains unchanged, the overall result is a slow-down.

The overhead is greater than that of call-aggregation due to the preprocessing required to compute the overall communication pattern, and the fact that the plans are 'piggy-backed' on top of one another, resulting in more plan data being transferred overall. The variables $v3$ and *result*, which are not used by $r1$, must also be forwarded between the client and $r2$ via $r1$.

The effect of plan caching and value copying on the server side may also be seen in the graphs. Forcing value copying results in a constant slowdown of about 2 milliseconds that is independent of the network type since it is purely processing overhead. This is highly significant with the Ethernet connection, but is negligible with the DSL connection due to the magnitude of the times involved.

Disabling plan caching results in a substantial performance hit with both network types since the remote plans and the associated metadata must be sent in full on every iteration, increasing the quantity of data that must be sent with both network types.

4.8.4 Multi-User Domain

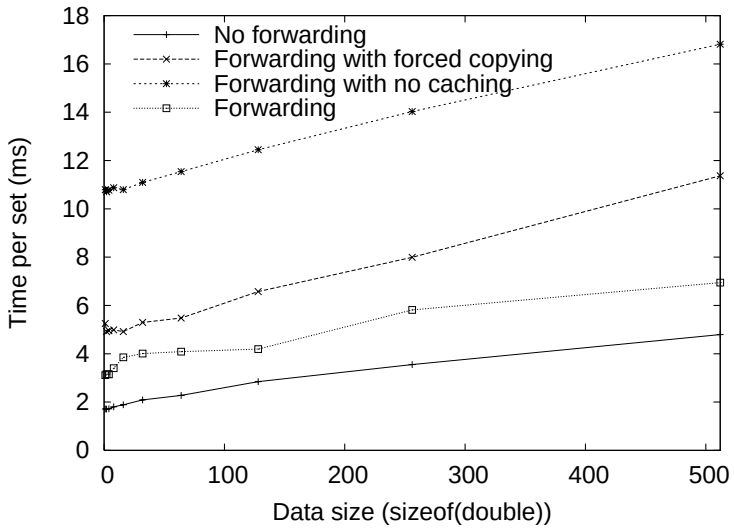
The Multi-User Domain (MUD) example [32] is a more realistic example that contains call aggregation possibilities. The main candidate for optimisation occurs in the `look` method of the `MudClient` class (shown in Figure 4.24), which retrieves a description of the room and its contents.

This benchmark has 7 aggregated calls with a modest payload — around 100 bytes of textual information in total. There is also an instance of forwarding occurring between the calls to the `getServer` and `getMudName`.

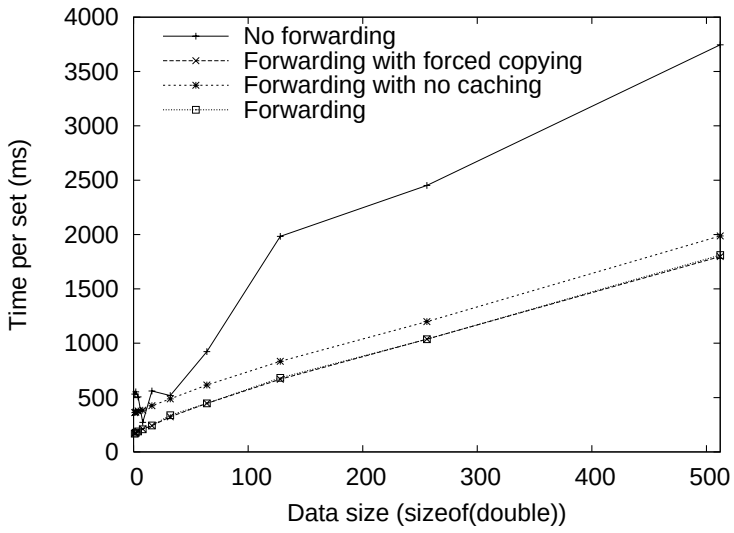
4.8.4.1 Test configuration

A test harness has been written to call this routine repeatedly, recording the average time per call. Caching and server-side parameter copying have been enabled. Note that the parameters are not actually copied in practice due to the structure of the code, but checking that it is unnecessary still requires some runtime overhead.

The server used with the Ethernet connection was a 650MHz Pentium-III PC with 256Mb of RAM, while the server used with the DSL connection was a dual-processor 700MHz Pentium-III PC.



(a) Ethernet



(b) DSL

Figure 4.23: Graph of results for the forwarding optimisation of the vector arithmetic example — the impact of turning caching off and forcing server-side variable duplication to occur has been shown on these graphs. Forwarding leads to a general slowdown when using an Ethernet connection, but a decent level of speedup is achieved with the DSL connection.

```
String mudname = p.getServer().getMudName();
String placename = p.getPlaceName();
String description = p.getDescription();
Vector things = p.getThings();
Vector names = p.getNames();
Vector exits = p.getExits();
```

Figure 4.24: Code for the `look` method of the MUD example

Time taken to execute `look` (ms)	Without optimisation	With optimisation	Speedup
Ethernet	5.4	5.8	0.93
DSL	759.6	164.9	4.61

Table 4.1: Results for the aggregation optimisation applied to the MUD example

Factor	Ethernet	DSL
Remote methods	0.62%	0.06%
Uncached RMI communication	0.78%	0.35%
Cached RMI communication	60.51%	97.92%
Client-side overhead	20.60%	0.91%
Server-side overhead	15.21%	0.61%
Argument copying overhead	2.29%	0.15%

Table 4.2: Percentage breakdown of the time spent executing 1000 iterations of the `look` method in the MUD example

4.8.4.2 Results

As can be seen in Table 4.1, the MUD example shows a slight slowdown when operating with an Ethernet network, but a large speedup with operating over the Internet.

The MUD application has been profiled to show a breakdown of the time taken to execute the `look` method in Table 4.2. As can be seen, the majority of the time in both cases is spent in client-server communication. However, on the Ethernet network, the additional overheads on the client and server side are responsible for about a third of the overall time, while the proportion of time due to overheads is insignificant by comparison when using DSL (since the overhead remains constant while the communication times have increased). If the overheads are minimised, then it should be possible to achieve a speedup of up to approximately 40% when operating on an Ethernet network.

4.9 Security

Distributed programs running under the DESORMI optimisations are both performing more work behind-the-scenes and are more intimately tied together than for standard RMI. This leads to a potentially more fragile environment with extra vulnerabilities. In this section, scenarios are considered in which normal operation is disrupted, either naturally or maliciously, and ways in which they might be resolved. Note that only security issues which occur with the DESORMI optimisations and not in standard RMI are considered here, since the goal is not to secure the RMI programming model.

4.9.1 Aggregation vulnerabilities

During call aggregation, a simple program is essentially uploaded from the client onto the server and executed by the server. Whenever this type of situation occurs, the question arises as to whether it is safe to execute this script. There are several approaches to tackle this problem.

- Signing — The code to be executed on the remote host is signed using a digital signature, which is used by the host that executes the program to confirm that the program does indeed originate from the claimed source and that it has not been tampered with.

  This approach by itself ensures that the program reaches the destination from the source intact, but it does not make any guarantees whatsoever regarding how safe it is to execute this program. In essence, the signer is saying, "Trust me, it is safe to run this program". However, even disregarding the issue of malicious programs such as trojans, spyware and viruses, even benevolent signers can make mistakes — they may have dangerous bugs in their programs, or may accidentally pass on a virus.

- Verification — The code for the downloaded program is examined by a program known as a verifier, which attempts to confirm that the program has no dangerous or illegal effects before the program is executed.

  In general, in order for this approach to work, the program must be supplied in a form that is either very simple or was designed to be amenable to verification. For example, in Java bytecode, there is the restriction that the height of the operand stack must be a constant at every instruction for all possible executions of a program. This can be confirmed by performing a dataflow analysis on the program.

  However, there are many hurdles to verification, some of which are very common indeed. For example, pointers and references pose problems for verifiers — it is often impossible to tell whether they will be null at a particular point in the code. Java handles this by inserting run-time checks into the compiled code.

- Sand-boxing — Sand-boxing is the practice of running programs in a restricted environment, so that a malicious program will be restricted in the amount and type of damage they can do. Java applets are particularly well-known for this (although this can now be applied to all Java applications in general). Also, services running on Unix-like systems are often executed in a `chroot` jail, so that an attacker who manages to obtain a command line will not be able to access other directories of the system.

In its present form, the plans can only make calls to local methods on the current host, or call remote methods on another host. The result is an impromptu sandbox of sorts, since the 'downloaded program' is severely limited in what it can do. Nevertheless, there are some vulnerabilities that may be exploited.

4.9.1.1 Object ID spoofing

In both the original and optimised cases, every remote object residing on a server is associated with an ID, which is part of the state within client-side stubs. When a remote call is made, this ID is also sent in order for the server to determine which object should be invoked.

This leads to a potential vulnerability where a client may access objects that it does not have a stub for by tampering with the object ID. In standard RMI, this means that one is effectively granting global access to an object as soon as it is exported, although encrypted tunnels and authentication at the application level may be used to restrict access.

The situation is similar when DESORMI is used, but the object IDs are allocated at the construction time of the remote object rather than at export time. This means that one

may be able to gain access to objects that have not been exported yet if explicit exportation is used.

This can easily be corrected by removing the callbacks to the Veneer runtime from constructors that do not derive from `UnicastRemoteObject` (which automatically exports objects at construction time). Instead, the Veneer policy can be modified to look for calls to the `export` method of `UnicastRemoteObject` in application code instead, and insert the registration callback just before these calls.

4.9.1.2 Denial-of-service attacks

A denial-of-service (DoS) attack is an attack where a server is bombarded with a constant stream of requests such that so many of the server resources (usually CPU time and network bandwidth) are tied up in fulfilling these requests that it leaves little remaining for legitimate requests, effectively dragging the server to a standstill. A distributed denial-of-service (DDoS) attack is a DOS attack involving multiple hosts bombarding the same server simultaneously, which is more effective since it is much easier to thoroughly saturate the available bandwidth of the server.

Standard RMI has no provisions against DoS attacks. One can mount a DDoS attack by getting many clients to sit in an infinite loop and make RMI calls as fast as possible. The effectiveness of an attack is determined by the number of calls that can be made per unit time — the higher the number, the more effective the attack. The maximum call rate is limited by the resources available to the server, since as the resources become exhausted, no more calls can be serviced, at which point the goal of the attacker has been achieved.

The problem is likely to become worse under aggregated RMI, since aggregating calls enables one to effectively make more RMI calls in less time while using fewer network resources. This means that an attacker will find it easier to reach the CPU saturation level using less network bandwidth (i.e. using fewer clients and/or with slower connections).

If plans are allowed to contain loop structures (see Section 6.2.14.1), then it will become even easier to saturate the server CPU, since the plans will now be able to operate independently of the client that produced them. This means that just one client may be able to bring a server to a halt by repeatedly sending plans that sit in an tight infinite loop, making calls to the server object as fast as possible.

4.9.2 Call interception

When the server forwarding optimisation is enabled, one server takes on the responsibility of handling a remote call to another server on behalf of the client. Since this responsibility was not present before, it can lead to new man-in-the-middle type vulnerabilities.

4.9.2.1 Argument interception

Consider this naively written program, which illustrates a scenario where a client buys an item from a vendor and instructs the bank to pay the vendor:

```
receipt = vendor.buy(item);
bank.pay(receipt, accountDetails);
```

Without forwarding, `accountDetails` is never seen by the vendor. However, with forwarding in place, the vendor is responsible for forwarding `accountDetails` in addition to the `receipt` object generated by the vendor itself. This means that the contents of `accountDetails` becomes visible to the machine that the vendor object resides on. A malicious vendor can therefore steal the account details of the customer, or alter the account details before passing it to the bank.

It may also seem possible for the vendor to charge more than the agreed amount by tampering with the contents of the receipt. However, the current code simply accepts the receipt blindly, and so would fall for this regardless of whether forwarding was present or not. If there was code to inspect the receipt before passing it onto the bank, then the vendor would no longer be responsible for forwarding the request to the bank because the inspection would result in the forcing of the remote call.

The best way around this is to make use of cryptography [81]. The client should negotiate with all servers that it will communicate with directly to obtain a session key using one of the many key-exchange protocols. All data that does not originate from the forwarding server should be encrypted using this session key. In addition to the raw data, the encrypted data should also include:

- A secure hash of the data, in order to detect possible tampering with the encrypted data by the forwarding server.
- A time-stamp, to prevent the forwarding server from mounting replay attacks by recording and later replaying a portion of encrypted data.

However, this may mean that the forwarding server must carry more data, since if any of the data in the encrypted set is also used by the forwarding server, a separate unencrypted copy must be supplied to it.

4.9.2.2 Repudiation

When forwarding is performed, the client expects the forwarding server to pass on the requested call onto the destination server exactly as requested, and no more or less. In the simple example given, the client has no way of telling whether the vendor will forward the request for payment just once — it may not forward it at all, do it multiple times, or forward it but deny the result to the client.

The simplest way of avoiding this problem is to have the target server send a direct acknowledgement to the original client that it has been called. This can be done as soon as the remote method has been called, and so can run in parallel with the processing of the remote method body. This idea is revisited in Section 6.2.7 with regard to the performance issues of server forwarding.

4.10 Alternative Platforms

There has been some success in adapting the DESORMI optimisations to other RMI-related technologies, which are covered briefly in this section.

4.10.1 RMI/IIOP

The optimisation framework has been successfully modified to cope with RMI under the IIOP protocol. The main modification is to change the runtime check for remote stubs to recognise RMI/IIOP stubs as well, since stubs in RMI/IIOP inherit from `javax.rmi.CORBA.Stub` rather than of `java.rmi.server.RemoteStub`. There was no need to change the server-side code, since RMI/IIOP servers also inherit from `java.rmi.Remote`.

RMI/IIOP clients receive their stubs by normal means (usually via COS naming services). Calls to these stubs are intercepted, resolved and eventually sent to the server by the underlying Veneer framework in the same way as calls via standard RMI/JRMP (see Section 4.2.3). Veneer always uses RMI/JRMP to communicate with the remote proxy regardless of the protocol used by the stubs themselves, effectively transforming RMI/IIOP communications into RMI/JRMP.

4.10.2 Enterprise JavaBeans

A basic form of the optimisations under the JBoss server [85] has been implemented. The use of interceptors to implement functionality in JBoss leads to several problems. In order to obtain the correct behaviour for a given remote call, all the interceptors in the relevant chain must be called, on both the client and the server side. This means that the current scheme of identifying the target of a remote call and calling it on the server side via Veneer will not work, since this would miss all the interceptors. In any case, Veneer is currently unable to run the JBoss server due to the extensive use of custom class loaders by JBoss.

The following scheme is used instead. On the server side, a Java Management Extension (JMX) [74] bean is deployed onto the JBoss server, which listens to requests from Veneer instances. When a client receives a JBoss dynamic proxy, it attempts to discover its originating server via the usual protocol, and establishes a connection to the custom JMX bean running on. Calls on the dynamic proxies are intercepted in the usual manner.

When the delayed calls are forced, the dynamic proxy is sent, along with the remote plan, to the JMX bean. The bean then invokes the methods via the supplied dynamic proxy to perform the calls. This is actually fairly efficient, since JBoss provides a special invoker for local calls that bypasses the RMI communication.

4.10.2.1 Possible improvements

There are many improvements that can be made on this scheme. One simple improvement is to cache dynamic proxies on the JMX bean and later refer to it using a short ID rather than being transported every time.

The transactional and security features of JBoss will not work properly with this scheme since the client-side interceptors are being executed within JBoss, and will therefore pick up the wrong contexts. A better solution might be to replace the invoker interceptor with one that places the `Invocation` object (see Section B.3.3.2) into the `State` object sent with the remote plans. The JMX bean can then send the `Invocation` object to the server-side interceptor stream. However, if the client-side interceptors perform actions as the result is returning, then these will be missed.

Another limitation is that remote calls made by beans running on JBoss are not affected by the DESORMI optimisations, since they do not run under Veneer. It is hoped that

Veneer will eventually be developed to the point where it can cope with JBoss and apply the optimisation to the beans loaded by JBoss at runtime.

4.11 Conclusion

Three RMI optimisations have been successfully implemented on top of the Veneer virtual JVM, working on the principle of restructuring distributed programs at runtime for an optimal communication pattern. The evaluation shows that these optimisations can be effective, but the cost of the Veneer virtual machine is a concern when working with fast networks. Further development of Veneer should minimise this.

Various problems have been discovered that result from the automatic restructuring. Workarounds for some problems have already been implemented, while potential solutions for the remaining problems have been investigated.

Chapter 5

Correctness of call aggregation

In order for a program optimisation to be useful, it must be correct. There are various notions of what 'correctness' means with regard to program optimisations. A good starting point is the following definition, which is given in [8]:

> A transformation is *legal* if, for all semantically correct executions *of the original program*, the original and the transformed programs perform equivalent operations for identical executions. All permutations of semi-commutative operations are considered equivalent.

In other words, if the same input is provided to both the original and optimised versions of a program, and if non-deterministic functions like `time` or `random` return the same results at equivalent points within the program, then the outputs of the two programs should be identical. Small variations due to operations that should be commutative in theory but are not in practice (such as floating-point arithmetic) are permissible.

This chapter presents a preliminary attempt at developing a formal model on which to prove that the DESORMI code transformations preserve the semantics of the program according to this definition. The behaviour of call aggregation is explained in terms of this model, showing that aggregated calls have equivalent behaviour to unaggregated calls under most circumstances. It is also explained why differing behaviour may occur, and how these may be compensated for.

5.1 General approach

Owing to the complexity of the Java language and runtime environment, a full semantic model for Java programs has not been attempted. Instead, global *traces* of the entire system of communicating clients and servers are considered instead.

When a program is executed, it executes a sequence of instructions. This sequence is known as a *trace*, which is denote by $\mathcal{T}$. $\mathcal{T}$ is made up of instructions $i_0, i_1 \ldots i_n$. Each instruction can read and write to a global state Σ. The effect of running the instructions in a trace $\mathcal{T}$ on a state Σ is denoted by $\mathcal{T}(\Sigma)$.

Now consider a program transformation Δ which transforms the trace of a program from $\mathcal{T}$ to $\Delta(\mathcal{T})$. Given an identical starting state Σ_0, two traces can be deemed to be equivalent if:

$$\mathcal{T}(\Sigma_0)[out] = (\Delta(\mathcal{T}))\,(\Sigma_0)[out]$$

where the subscript operator $[out]$ selects the subset of the state that represents the output of the program.

5.2 Concepts of the logical framework

In this section, the concepts that constitute the logical framework are introduced. These are used to reason about the equivalence of programs before and after transformation.

5.2.1 Global state

The global state of the system is denoted as Σ. Σ is a vector that contains many *variables*. A variable is represented by its position within Σ, and its value. The value of a variable at position x is denoted as $\Sigma[x]$. The values of variable sets may also be referred to in this manner.

State is unstructured in this framework, so that every element of a compound data type is explicitly represented by a separate variable. For example, an array with five elements would be represented by six variables in the framework — one variable for the reference to the entire array, and one for each element of the array.

5.2.2 Traces

A trace $\mathcal{T}$ is composed of a series of instructions, such that the effect of executing a trace is to apply the initial state to the composition of the component instructions. For example:

$$\begin{aligned}\mathcal{T} = [f, g, h] \Longrightarrow \Sigma_{after} &= \mathcal{T}(\Sigma_{before}) \\ &= f; g; h(\Sigma_{before}) \\ &= h \cdot g \cdot f(\Sigma_{before}) \\ &= h(g(f(\Sigma_{before})))\end{aligned}$$

A semicolon is used to denote reverse composition for convenience. An operator $\prec$ is used to denote the ordering of instructions in a trace. In the previous example, the ordering $f \prec g \prec h$ holds. The immediate predecessor and successor of an instruction i are denoted $pred(i)$ and $succ(i)$ respectively.

5.2.3 Instructions

The basic unit of computation is an instruction, which is a function of type $\Sigma \rightarrow \Sigma$. The global state before the execution of an instruction i is marked by a subscript (for example, Σ_i), although this is sometimes omitted when it is obvious what the state applies to.

There are only two types of instruction. A local instruction represents a block of operations of an arbitrary size. Execution of an instruction is modelled by applying the instruction to the state, such that:

$$\Sigma_{succ(i)} = i(\Sigma_i)$$

A remote instruction is a special type of instruction that does not perform any computation in itself, but rather triggers the execution of other instructions. Remote functions are denoted by an overscore — for example, $\overline{f}$. These will be covered later in Section 5.2.4.

Instructions have data definition and use properties, which for an instruction i are denoted by $def(i)$ and $use(i)$. These properties denote sets of indices into the global state, such that the subset of the global state that represents the values of variables read by an instruction i can be denoted by $\Sigma[use(i)]$. Similarly, the values written by the instruction can be denoted $\Sigma[def(i)]$.

Two invariants hold with regard to the *def* and *use* properties:

$$i(\Sigma) - i(\Sigma)[def(i)] = \Sigma - \Sigma[def(i)] \tag{5.1}$$

$$i(\Sigma[use(i)])[def(i)] = i(\Sigma)[def(i)] \tag{5.2}$$

Equation 5.1 states that only the subset of the global state in def_i will be modified by i, while Equation 5.2 states that only the subset of the state in use_i is necessary to compute the function.

Instructions are referentially transparent — given the same values for variables in the use set, the subsequent values of variables in the defined set will be the same. This is expressed with the following:

$$\forall i \in \mathcal{T} \cdot \forall \Sigma_1, \Sigma_2 \cdot \Sigma_1[use(i)] = \Sigma_2[use(i)] \Longrightarrow i(\Sigma_1)[def(i)] = i(\Sigma_2)[def(i)] \tag{5.3}$$

A function *reaches* is defined that is true if a variable v, starting from the point after instruction i, can reach a subsequent instruction j without being modified:

$$reaches_v(i, j) \Longleftrightarrow \forall k \in \mathcal{T} \cdot i \prec k \prec j \Rightarrow v \notin def(k) \tag{5.4}$$

Provided that no other intervening instruction defines the variable v, the value of v between its definition and any subsequent instruction will be identical.

$$reaches_v(i, j) \Longrightarrow \Sigma_{succ(i)}[v] = \Sigma_j[v] \tag{5.5}$$

The definition and use properties can also be defined for any list of instructions $\mathcal{L}$, where:

$$\begin{aligned} def(\mathcal{L}) &= \bigcup_{i \in \mathcal{L}} def(i) \\ def(\mathcal{L}) &= \bigcup_{i \in \mathcal{L}} def(i) \end{aligned}$$

5.2.4 Remote calls

A remote instruction $\overline{f}$ is associated with at least one other instruction that represents the events that are initiated by a remote call. These instructions, known as *remote actions*, are indicated by underlining. For example, if $\overline{f}$ leads to the instructions g and h being executed, then those instructions are denoted by $\underline{g;h}$. The association between a remote instruction and the instructions that result from applying it is represented by the $\underline{\langle\rangle}$ operator, so for the current example:

$$\underline{\langle \overline{f} \rangle} = \underline{g;h}$$

5.2.5 Remote actions

Instructions that result from executing remote instructions do not operate directly on the subset of the state accessed by the remote function, but on copies generated by the marshalling process.

5.2.5.1 Marshalling

Marshalling is modelled by two pseudo-instructions *copyto* and *copyfrom*. Note that these instructions are *not* inverses of the other.

Applying $copyto_{\overline{f}}$ to the state Σ has the effect of copying the portion of the state referenced by $use(\overline{f})$ to a subset of $use\left(\underline{\langle \overline{f} \rangle}\right)$. The copying effect of the *copyto* instruction is represented by a set of relations $\overset{\overline{f}}{\rightharpoonup}$. The following equations hold for $copyto_{\overline{f}}$.

$$\begin{aligned}
def\left(copyto_{\overline{f}}\right) \cap use\left(copyto_{\overline{f}}\right) &= \emptyset \\
use\left(copyto_{\overline{f}}\right) &= use(\overline{f}) \\
def\left(copyto_{\overline{f}}\right) &\subseteq use\left(\underline{\langle \overline{f} \rangle}\right) \\
x \overset{\overline{f}}{\rightharpoonup} y &\implies x \in use\left(copyto_{\overline{f}}\right) \wedge y \in def\left(copyto_{\overline{f}}\right) \\
&\quad \wedge \Sigma[x] = \Sigma[y] \\
\forall i \in \mathcal{T} \cdot def\left(copyto_{\overline{f}}\right) \cap use(i) \neq \emptyset &\implies i = \underline{\langle \overline{f} \rangle}
\end{aligned}$$

The meanings of the first four equations are straightforward. The last equation states that the variables that *copyto* copies into are unique, and can only be referenced by the associated remote action. This simulates the effect of binding the formal parameters of a call to the actual parameters.

Similarly, $copyfrom_{\overline{f}}$ copies a subset of the state referenced by $def(\underline{\langle \overline{f} \rangle})$ to $def(\overline{f})$ using the relation $\overset{\overline{f}}{\rightharpoonup}$, with the following equations holding:

$$
\begin{aligned}
def\left(copyfrom_{\overline{f}}\right) \cap use\left(copyfrom_{\overline{f}}\right) &= \emptyset \\
use\left(copyfrom_{\overline{f}}\right) &\subseteq def\left(\underline{\langle \overline{f} \rangle}\right) \\
def\left(copyfrom_{\overline{f}}\right) &= def(\overline{f}) \\
x \overset{\overline{f}}{\rightharpoonup} y &\implies x \in use\left(copyfrom_{\overline{f}}\right) \wedge y \in def\left(copyfrom_{\overline{f}}\right) \\
&\quad \wedge \Sigma[x] = \Sigma[y] \\
\forall i \in \mathcal{T} \cdot use\left(copyfrom_{\overline{f}}\right) \cap def(i) \neq \emptyset &\implies i = \underline{\langle \overline{f} \rangle}
\end{aligned}
$$

The relations $\rightharpoonup$ and $\rightarrow$ for the marshalling operations in a trace $\mathcal{T}$ are contained in a set denoted as $\rho(\mathcal{T})$.

5.2.5.2 Remote transformation

A transformation can now be introduced to model the effects resulting from a remote call. A transformation δ_r is defined for a trace that replaces each remote call with the remote effect using the following mapping:

$$\overline{f} \overset{\delta_r}{\longmapsto} copyto_{\overline{f}}; \underline{\langle \overline{f} \rangle}; copyfrom_{\overline{f}} \tag{5.6}$$

For convenience, shortened notations are introduced for the two marshalling instructions, so that Equation 5.6 can be written more simply as:

$$\overline{f} \overset{\delta_r}{\longmapsto} \bullet \underline{\langle \overline{f} \rangle} \circ \tag{5.7}$$

The relationship between the two in terms of their def-use set is:

$$
\begin{aligned}
use\left(\overline{f}\right) &\subseteq use\left(\bullet \underline{\langle \overline{f} \rangle} \circ\right) \\
def\left(\overline{f}\right) &\subseteq def\left(\bullet \underline{\langle \overline{f} \rangle} \circ\right)
\end{aligned}
$$

The effect of applying δ_r to a trace containing remote instructions is to add the immediate effects of the remote actions associated with those instructions into the trace, so that one can reason about the definitions and uses of the callee as well as the caller. Simply put, each application of δ_r reveals more of the 'global picture'.

The initial trace that contains only the instructions executed on a single client is referred to as $\mathcal{T}^0$. Every time δ_r is applied to $\mathcal{T}^n$, the result is denoted as $\mathcal{T}^{n+1}$. This repeated application may be necessary since remote actions themselves may be remote instructions. There are some relationships that can be defined between any $\mathcal{T}^n$ and $\mathcal{T}^{n+1}$:

$$
\begin{aligned}
def(\mathcal{T}^{n+1}) &\supseteq def(\mathcal{T}^n) \\
use(\mathcal{T}^{n+1}) &\supseteq use(\mathcal{T}^n)
\end{aligned}
$$

In any terminating program, there will reach a point at which $\mathcal{T}^{n+1} = \mathcal{T}^n$, where there

are no remote instructions remaining in the trace. If there is no such point, then there must exist a sequence of remote instructions that form an infinite loop. The trace where every remote instruction has been replaced is referred to as $\mathcal{T}^{\infty}$, regardless of whether there is a termination point or not.

5.2.6 Input and output

Input and output operations are abstracted as accesses to the global state. An input operation is defined as a read from the variable set *input*, while an output operation is a write to a variable in the set *output*. Two predicates $isinputop(i)$ and $isoutputop(i)$ can be defined which are true only if an instruction i performs input and output respectively by:

$$isinputop(i) \iff use(i) \cap input \neq \emptyset$$

$$isoutputop(i) \iff def(i) \cap output \neq \emptyset$$

5.2.7 Exceptions

Exceptions are also modelled as a form of dataflow. The throwing block T defines some exception variable e, which is subsequently used by the exception handler H. This effectively creates a relationship between T and H of the following form:

$$T \prec H \wedge e \in def(T) \cap use(H)$$

5.3 Correctness of clustered call aggregation

Consider a cluster of remote calls $\mathcal{R}^i$ which consists of a number of remote instructions $\overline{r_1}, \overline{r_2} ... \overline{r_k}$ in sequence. Under standard RMI, the effect of executing the cluster are as follows:

$$\begin{aligned} \mathcal{R}^n_{orig}(\Sigma) &= \overline{r_1}; \overline{r_2}; ...; \overline{r_k}(\Sigma) \\ \mathcal{R}^{n+1}_{orig}(\Sigma) = \delta_r\left(\mathcal{R}^n_{orig}(\Sigma)\right) &= \bullet\underline{\langle\overline{r_1}\rangle}\circ; \bullet\underline{\langle\overline{r_2}\rangle}\circ; ...; \bullet\underline{\langle\overline{r_k}\rangle} \circ (\Sigma) \end{aligned}$$

In its basic form, the effects of executing aggregated calls can be expressed as follows:

$$\begin{aligned} \mathcal{R}^n_{aggr}(\Sigma) = d_{aggr}\left(\mathcal{R}^n_{orig}(\Sigma)\right) &= \overline{(r_1; r_2; ...; r_k)}(\Sigma) \\ \mathcal{R}^{n+1}_{aggr}(\Sigma) = \delta_r\left(\mathcal{R}^n_{aggr}(\Sigma)\right) &= \bullet\underline{\langle\overline{r_1; r_2; ...; r_k}\rangle} \circ (\Sigma) \end{aligned}$$

To reason about the relationship between the original and aggregated forms, the remote action corresponding to the aggregate remote call is expressed in terms of the original remote actions as follows:

$$\underline{\langle\overline{r_1; r_2; ...; r_k}\rangle} = \underline{\triangleleft\langle\overline{r_1}\rangle\triangleright; \triangleleft\langle\overline{r_2}\rangle\triangleright; \ldots; \triangleleft\langle\overline{r_k}\rangle\triangleright}$$

5.3.1 Adapter operators

The $\triangleleft$ and $\triangleright$ operators denote adapter operators. In the unaggregated scenario, a remote action receives part of its use set from a private location in the global state that was written to by the preceding *copyto* operation. After the remote action is executed, the result is written into another private location, where it is picked up by the following *copyfrom* instruction. The private locations used by each remote action are disjoint.

In order to express the semantics of the aggregated case in terms of the remote actions executed in the unaggregated version, the private locations of the remote actions must be filled correctly prior to executing the action itself. However, the initial *copyto* operation at the beginning of the cluster is insufficient to perform this task, since some of the values used by an action may have been generated by a previous action that places the values in its own private space.

To solve this problem, another private space S is introduced, which is accessible by the copying and adaptor instructions only. The *copyto* instruction at the beginning of the cluster copies values from the client space to S, and the operator $\triangleleft$ copies the values needed by the next remote action from S to the private space of the action. The operator $\triangleright$ performs the complementary function of copying values defined in the private space of the remote action back to S, and the final *copyfrom* instruction copies from S back to the client space.

Note that this corresponds to the passing of arguments by copy. The consequences of copy-by-reference are dealt with later in Section 5.4.2.

Similarly to the *copyto* and *copyfrom* instructions, the $\triangleleft$ and $\triangleright$ operators have the operators $\frown$ and $\smile$ respectively associated with them, which define the relationship between the variables used by the remote action and the variables in private space S. These relations are also part of the set $\rho(\mathcal{R})$. For the expression $\triangleleft\underline{\langle\overline{r_i}\rangle}\triangleright$, the following equations hold:

$$x \overset{\overline{r_i}}{\frown} y \implies x \in S \wedge y \in use\left(\underline{\langle\overline{r_i}\rangle}\right) \wedge \Sigma[x] = \Sigma[y] \tag{5.8}$$

$$x \overset{\overline{r_i}}{\smile} y \implies x \in def\left(\underline{\langle\overline{r_i}\rangle}\right) \wedge y \in S \wedge \Sigma[x] = \Sigma[y] \tag{5.9}$$

The relationship between the relations in $\rho(\mathcal{R}_{orig})$ and $\rho(\mathcal{R}_{aggr})$ is as follows:

$$\left(x \overset{i}{\rightharpoonup} y\right) \in \rho(\mathcal{R}_{orig}) \iff \exists t \in S \cdot \left\{x \rightharpoonup t, t \overset{i}{\frown} y\right\} \subseteq \rho(\mathcal{R}_{aggr}) \tag{5.10}$$

$$\left(x \overset{i}{\rightharpoondown} y\right) \in \rho(\mathcal{R}_{orig}) \iff \exists t \in S \cdot \left\{x \overset{i}{\smile} t, t \rightharpoondown y\right\} \subseteq \rho(\mathcal{R}_{aggr}) \tag{5.11}$$

$$\{x \rightharpoonup s, x \rightharpoonup t\} \subseteq \rho(\mathcal{R}_{aggr}) \vee \{s \rightharpoondown x, t \rightharpoondown x\} \subseteq \rho(\mathcal{R}_{aggr}) \implies s = t \tag{5.12}$$

Equation 5.12 essentially states that there is a one-to-one mapping between variables in the client space and those in S. By combining Equations 5.10 and 5.12, the following can be derived:

$$\left\{x \overset{i}{\rightharpoonup} y, x \overset{j}{\rightharpoonup} z\right\} \subseteq \rho(\mathcal{R}_{orig}) \iff \exists t \in S \cdot \left\{x \rightharpoonup t, t \overset{i}{\frown} y, t \overset{j}{\frown} z\right\} \subseteq \rho(\mathcal{R}_{aggr}) \tag{5.13}$$

Similarly, by combining Equations 5.11 and 5.12:

$$\left\{x \xrightarrow{i} z, y \xrightarrow{j} z\right\} \subseteq \rho(\mathcal{R}_{orig}) \Longleftrightarrow \exists t \in S \cdot \left\{x \overset{i}{\smile} t, y \overset{j}{\smile} t, t \rightharpoonup z\right\} \subseteq \rho(\mathcal{R}_{aggr}) \tag{5.14}$$

The effect on the def-use sets of the remote cluster are as follows:

$$\begin{gathered} def(\mathcal{R}_{aggr}) \subseteq def(\mathcal{R}_{orig}) \\ \forall x \in (def(\mathcal{R}_{aggr}) - def(\mathcal{R}_{orig})) \cdot x \in S \\ use(\mathcal{R}_{aggr}) \subseteq use(\mathcal{R}_{orig}) \\ \forall x \in (use(\mathcal{R}_{aggr}) - use(\mathcal{R}_{orig})) \cdot x \in S \end{gathered} \tag{5.15}$$

5.3.2 The isolation property of deserialised variables

When only remote calls exist within a code sequence, the following property holds for all calls in the cluster:

$$\forall x \cdot y \xrightarrow{\overline{f}} x \quad \in \rho(\mathcal{R}) \Longrightarrow \quad \forall \overline{g} \in \mathcal{R} \cdot \overline{f} \prec \overline{g} \Rightarrow x \notin use\left(\underline{\langle \overline{g} \rangle}^{\infty}\right) \tag{5.16}$$

This property states that if some value of the variable x was set as a result of a *copyback* operation, then x cannot be used directly by any following remote action, regardless of the number of times δ_r is applied. This models the isolation property discussed in Section 4.6.1 that ensures that the variable that receives the result of a deserialisation contains the only reference to the copied object.

Note that it is still possible for a remote action $\underline{\langle \overline{g} \rangle}$ to get access to the *value* of x (but not to the variable itself) indirectly via a *copyto* operation that copies x into a variable that is accessible from the remote action. It is also possible that the remote action that generated x stored a copy which is accessible to $\underline{\langle \overline{g} \rangle}$.

5.3.3 Approach to proof

In order to show the transformation to be correct, it must be proven that $\mathcal{R}_{orig}^{n+1}$ and $\mathcal{R}_{aggr}^{n+1}$ are equivalent such that:

$$\mathcal{R}_{orig}^{n+1}\left[def\left(\mathcal{R}_{orig}^{n+1}\right)\right] = \mathcal{R}_{aggr}^{n+1}\left[def\left(\mathcal{R}_{orig}^{n+1}\right)\right]$$

The following approach is taken:

1. Show that the values used from the global state by any remote action in $\mathcal{R}_{orig}$ are the same as the corresponding remote action in $\mathcal{R}_{aggr}$. If this is true, then by the referential transparency property (Equation 5.3), the values defined by the remote action will be identical in both cases.

2. Given that the individual calls generate the same values as before, show that the state at the end of $\mathcal{R}_{aggr}$ is identical to the state at the end of $\mathcal{R}_{orig}$ minus the variables in S.

5.3.4 Showing that values used by remote actions are correct

Consider an arbitrary remote action $\bullet\underline{\langle\overline{r_i}\rangle}\circ \in \mathcal{R}_{orig}$, and the equivalent remote action $\triangleleft\underline{\langle\overline{r_i}\rangle}\triangleright \in \mathcal{R}_{aggr}$, which are the responses to the remote instruction $\overline{r_i}$. Since $\underline{\langle\overline{r_i}\rangle}$ refers to the same set of remote actions in both cases, they share the same set of variable defines and uses.

Now consider the use set of $\underline{\langle\overline{r_i}\rangle}$. For a given variable x in this set, there are three possible conditions under which the value of x may have been defined:

1. An instruction preceding the remote cluster set the value of x
2. A previous remote action generated the value of x and copied it into client space using a *copyfrom* operation
3. A previous remote action directly set the value of x without an intermediate *copyfrom* operation

There are also two methods by which the value of x may be received by $\underline{\langle\overline{r_i}\rangle}$:

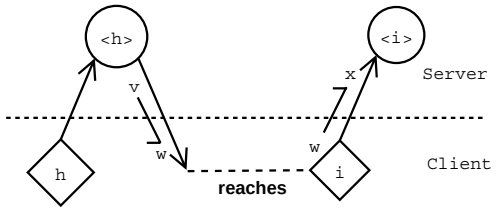
1. A *copyto* operation copies the value of x into the variable set used by $\underline{\langle\overline{r_i}\rangle}$
2. $\underline{\langle\overline{r_i}\rangle}$ directly uses x without an intervening *copyto* operation

The use set of $\underline{\langle\overline{r_i}\rangle}$ is divided into mutually exclusive subsets, with the classification of a variable depending on the combination of circumstances in which the variable is defined and used. The subsets are as follows:

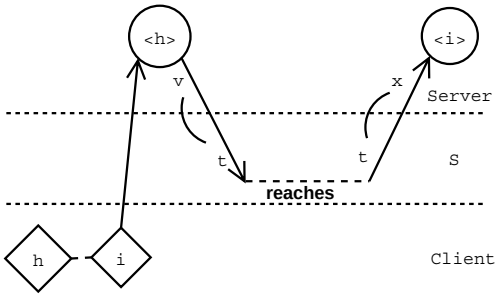
1. $use_{internal}\left(\underline{\langle\overline{r_i}\rangle}\right)$ — consists of variables that are copied by the *copyto* operation immediately preceding the remote action, with values originating from previous remote actions in the cluster that were copied by *copyfrom* operations.
2. $use_{external}\left(\underline{\langle\overline{r_i}\rangle}\right)$ — consists of variables that are copied by the *copyto* operation immediately preceding the remote action, with values originating from instructions preceding the remote cluster.
3. $use_{direct}\left(\underline{\langle\overline{r_i}\rangle}\right)$ — consists of variables that are directly accessed by the remote action without an intervening *copyto* operation, with values that may originate from direct assignment by previous remote actions or from instructions preceding the remote cluster.
4. $use_{indirect}\left(\underline{\langle\overline{r_i}\rangle}\right)$ — consists of variables that are copied by the *copyto* operation immediately preceding the remote action, with values originating from previous remote actions in the cluster via direct assignment.

The remaining potential case, where variables are directly accessed by the remote action with values originating from previous remote actions that were copied by *copyfrom* operations, is an invalid case since it invalidates the isolation property stated in Section 5.3.2.

Each subset is now covered separately, to ensure that the values in each set are identical in both the original and aggregated cases.



(a) Original



(b) Aggregated

Figure 5.1: Dataflow of internal values in a remote cluster

5.3.4.1 Internal values

Consider a variable $x \in use_{internal}\left(\underline{\langle \overline{r_i} \rangle}\right)$. In order for it to be in this set, x must be *internal* to $\mathcal{R}_{orig}$. This property is defined for a variable x used by a remote action with the following definition:

$$\begin{aligned} internal_{orig}(\overline{i}, x) \iff\ & x \in use\left(\underline{\langle \overline{i} \rangle}\right) \wedge \exists \underline{\langle \overline{h} \rangle} \in \mathcal{R}_{orig} \cdot \underline{\langle \overline{h} \rangle} \prec \underline{\langle \overline{i} \rangle} \\ & \wedge \exists v \in def\left(\underline{\langle \overline{h} \rangle}\right) \cdot \exists w \in use\left(\overline{i}\right) \cdot \left(v \xrightarrow{\overline{h}} w, w \xrightarrow{\overline{i}} x\right) \subseteq \rho(\mathcal{R}_{orig}) \\ & \wedge reaches_w\left(\bullet \underline{\langle \overline{h} \rangle} \circ, \bullet \underline{\langle \overline{i} \rangle} \circ\right) \end{aligned}$$

The path taken by the value of x is shown in Figure 5.1(a). The original source of the value for x is therefore v, which is defined by $\underline{\langle \overline{h} \rangle}$. It is copied into the client space variable w, which is later copied to x. From the definition of *copyto* and *copyfrom*, $\Sigma[v] = \Sigma[w]$ and $\Sigma[w] = \Sigma[x]$. The *reaches* condition furthermore ensures that $\Sigma[v] = \Sigma[w] = \Sigma[x]$.

Extending this further to the aggregated case by applying Equations 5.10 and 5.11, the following can be derived:

$$\begin{aligned} internal_{aggr}(\overline{i}, x) \iff\ & x \in use\left(\underline{\langle \overline{i} \rangle}\right) \wedge \exists \underline{\langle \overline{h} \rangle} \in \mathcal{R}_{aggr} \cdot \underline{\langle \overline{h} \rangle} \prec \underline{\langle \overline{i} \rangle} \\ & \wedge \exists v \in def\left(\underline{\langle \overline{h} \rangle}\right) \cdot \exists t \in S \cdot \left(v \overset{\overline{h}}{\smile} t \wedge t \overset{\overline{i}}{\frown} x\right) \subseteq \rho(\mathcal{R}_{aggr}) \\ & \wedge reaches_t(\triangleleft \underline{\langle \overline{h} \rangle} \triangleright, \triangleleft \underline{\langle \overline{i} \rangle} \triangleright) \end{aligned}$$

The aggregated scenario is shown in Figure 5.1(b).

The *reaches* condition must now be proven to hold. This is done using a proof by contradiction, by considering the implications if the *reaches* term in $internal_{aggr}$ was actually false.

Firstly, consider Equation 5.11. Whenever a $\smile$ relation is generated, a $\rightharpoonup$ relation must be generated at the same time. This means that when $v \overset{\overline{h}}{\smile} w$ was generated in $internal_{aggr}$, the following relation must also have been generated at the same time:

$$\exists z \cdot w \overset{\overline{h}}{\rightharpoonup} z \in \rho(\mathcal{R}_{aggr})$$

Since w is in the private space S, only the $\triangleright$ adaptor instructions can possibly overwrite w. The following must then be true for some instruction k:

$$\triangleleft \underline{\langle \overline{h} \rangle} \triangleright \prec k \prec \triangleleft \underline{\langle \overline{i} \rangle} \triangleright \wedge \exists y \cdot y \overset{k}{\smile} w \in \rho(\mathcal{R}_{aggr})$$

By applying Equation 5.14 to the relations that exist in $\rho(\mathcal{R}_{aggr})$ given that the *reaches* condition is false, the following must then hold:

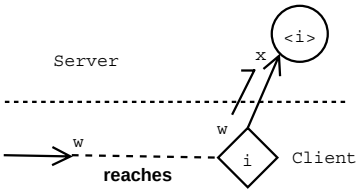
$$\left(v \overset{\overline{h}}{\rightharpoonup} z, y \overset{k}{\rightharpoonup} z\right) \subseteq \rho(\mathcal{R}_{orig})$$

This would mean that in the original, the value returned by $\underline{\langle \overline{h} \rangle}$ and stored in z will have been overwritten by the value returned by k. However, since the last term in the definition of $internal_{orig}$ explicitly states that the value should at least reach $\underline{\langle i \rangle}$, this cannot be the case. Therefore, if the *reaches* condition is true in $internal_{orig}$, then it must also be true for $internal_{aggr}$.

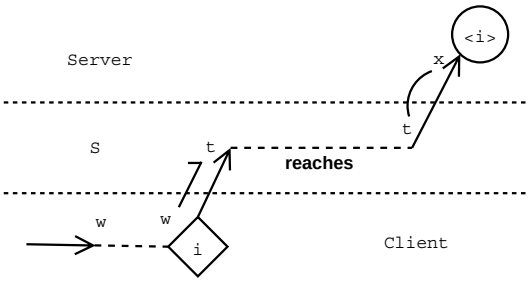
v is immediately copied by the adaptor into the variable w. By Equation 5.9, $\Sigma[v] = \Sigma[w]$. It is now known that w reaches $\triangleleft \underline{\langle \overline{i} \rangle} \triangleright$ without being overwritten. Combining this with Equation 5.8, the expression $\Sigma[v] = \Sigma[w] = \Sigma[x]$ can be derived. This proves that $\underline{\langle \overline{i} \rangle}$ receives the same values from the set of internal variables in both the original and aggregated cases.

5.3.4.2 External values

Consider a variable $x \in use_{external}\left(\underline{\langle \overline{r_i} \rangle}\right)$. In order to be in this set, x must be *external* to $\mathcal{R}_{orig}$. The definition of an external variable is complementary to that of an internal variable, and is defined for a variable x used by a remote action as:



(a) Original



(b) Aggregated

Figure 5.2: Dataflow of external values in a remote cluster

$$external(\overline{i}, x) \iff x \in use\left(\underline{\langle \overline{i} \rangle}\right) \wedge \exists w \in use\left(\overline{i}\right) \cdot w \overset{\overline{i}}{\rightharpoonup} x \\ \wedge reaches_w\left(pred(\mathcal{R}_{orig}), \underline{\langle \overline{i} \rangle}\right)$$

This case is illustrated in Figure 5.2(a). In this case, $\Sigma(w) = \Sigma(x)$.

Extending this further to the aggregated case:

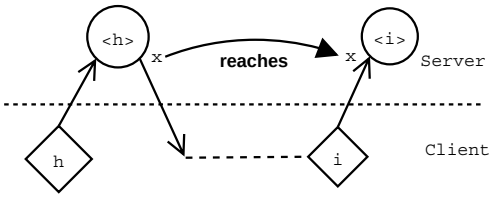
$$external(\overline{i}, x) \iff x \in use\left(\underline{\langle \overline{i} \rangle}\right) \wedge \exists w \in use\left(\overline{i}\right) \cdot \exists t \in S \cdot w \rightharpoonup t \wedge t \overset{\overline{i}}{\frown} x \\ \wedge reaches_w\left(pred(\mathcal{R}_{aggr}), \underline{\triangleleft \langle \overline{i} \rangle \triangleright}\right)$$

This is shown in Figure 5.2(b). To show that the *reaches* condition is true, again consider the situation assuming that the *reaches* condition was false. There must be some instruction k where:

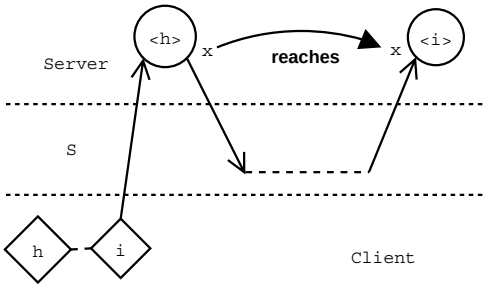
$$k \prec \underline{\triangleleft \langle \overline{i} \rangle \triangleright} \wedge \exists y \cdot y \overset{k}{\smile} t \in \rho(\mathcal{R}_{aggr})$$

However, the introduction of the extra $\smile$ relation has effectively transformed $external(\overline{i}, x)$ into $internal(\overline{i}, x)$, since x now originates from k. Since it cannot be the case that an external variable will map back into an internal variable in the unaggregated case, the *reaches* condition must be true instead.

Given that the *reaches* condition is true, then by the definition of the $\frown$ operator, $\Sigma(w) = \Sigma(t) = \Sigma(x)$, hence the value received by $\underline{\langle \overline{i} \rangle}$ remains the same in both cases.



(a) Original



(b) Aggregated

Figure 5.3: Dataflow of direct values in a remote cluster

5.3.4.3 Direct values

A variable $x \in use_{direct}\left(\underline{\langle \overline{r_i} \rangle}\right)$ must satisfy the following property in the unaggregated version:

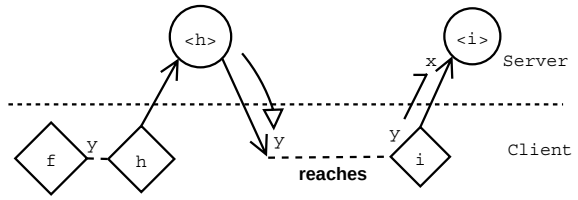
$$direct(\overline{i}, x) \iff \exists h \cdot x \in def\,(h) \cap use\left(\underline{\langle \overline{i} \rangle}\right) \wedge reaches_x\left(h, \underline{\langle \overline{i} \rangle}\right)$$

This is illustrated in Figure 5.3(a). This means that x must be directly defined by some instruction h that is either a previous remote action, or by an instruction outside of the remote cluster. Since there are is no marshalling involved, this expression does not change under aggregation (see Figure 5.3(b)).

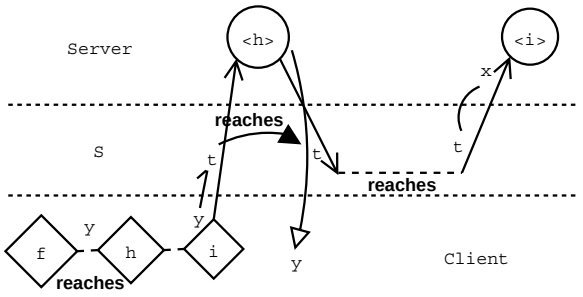
From Equation 5.15, any new variable definitions that are made during the transition to aggregated calls are to variables in the private space S, which a remote action cannot access directly. From the definition of *reaches* (Equation 5.4), it can be determined that the property $reaches_x(h, \overline{i})$ must still hold after the transformation, since $x \notin S$, therefore any new definitions that appear in $\mathcal{R}_{aggr}$ cannot affect x. By Equation 5.5, $\Sigma_h(x) = \Sigma_{\overline{i}}(x)$ in both cases.

5.3.4.4 Indirect values

An indirect variable $x \in use_{indirect}\left(\underline{\langle \overline{r_i} \rangle}\right)$ is defined by the following condition:



(a) Original



(b) Aggregated

Figure 5.4: Dataflow of indirect values in a remote cluster

$$indirect(\overline{i}, x) \iff x \in use\left(\underline{\langle \overline{i} \rangle}\right) \wedge \exists \overline{h} \cdot y \in def\left(\underline{\langle \overline{h} \rangle}^{\infty}\right) \wedge \left(y \overset{\overline{i}}{\rightharpoonup} x\right) \in \rho(\mathcal{R}_{orig}) \\ \wedge reaches_y\left(\underline{\langle \overline{h} \rangle}, \bullet\underline{\langle \overline{i} \rangle}\circ\right)$$

This situation is shown in Figure 5.4(a). The remote action $\underline{\langle \overline{h} \rangle}$ directly defines a variable y on the client without using a *copyfrom* instruction, and the value of y is later copied into x by a *copyto* instruction used by $\underline{\langle \overline{i} \rangle}$.

Unlike the previous cases, this case is *not* handled correctly. Consider what happens if the value of x was originally set by some instruction f that occurs before the remote cluster. In the original case:

$$\exists f \cdot f \prec \overline{h} \wedge y \in def(f) \wedge \left(y \overset{\overline{i}}{\rightharpoonup} x\right) \in \rho(\mathcal{R}_{orig}) \wedge \neg reaches_y\left(f, \bullet\underline{\langle \overline{i} \rangle}\circ\right)$$

The value of y defined by f does *not* reach $\bullet\underline{\langle \overline{i} \rangle}\circ$ because it is directly overwritten by $\underline{\langle \overline{h} \rangle}$ which lies between f and $\overline{i}$. However, in the aggregated form, this is transformed to:

$$\exists f \cdot f \prec \overline{h} \wedge y \in def(f) \wedge \exists t \in S \cdot \left(y \rightharpoonup t, t \overset{\overline{i}}{\rightharpoonup} x\right) \subseteq \rho(\mathcal{R}_{aggr}) \\ \wedge reaches_y(f, \mathcal{R}_{aggr}) \wedge reaches_t\left(copyto, \triangleleft\underline{\langle \overline{i} \rangle}\triangleright\right)$$

This is illustrated in Figure 5.4(b). In this case, the value of x received by $\underline{\langle \overline{i} \rangle}$ comes

from f rather than from $\overline{h}$, despite $\overline{h}$ coming after f. This is because there is no relation of the form $s \xrightarrow{\overline{h}} y$ in the *copyfrom* operation of $\overline{h}$, hence the application of Equation 5.11 will not result in the generation of an equivalent $s \overset{\overline{h}}{\smile} t$ relation in the aggregated version that will overwrite the old value of t.

This problem showed up during the practical implementation of the RMI optimisation as a problem caused by callbacks (see Section 4.6.2).

5.3.5 Showing that values that reach the end of the cluster are correct

There are two ways in which variables may be defined by remote actions in a remote cluster — by direct assignment or by a *copyfrom* instruction.

5.3.5.1 Direct values

It is easy to show that directly defined variables that reach the end of the cluster in the original case will also reach the end in the aggregated case using the same argument as that in Section 5.3.4.3 — i.e. the only extra definitions in the aggregated case are to variables in S, and since S does not exist in the original case, there cannot be any new definitions that interfere with a variable that reaches the end in the original case from reaching it in the aggregated case.

5.3.5.2 Copied values

By the end of $\mathcal{R}_{orig}$, the k *copyfrom* values will have defined various values in client space, some of which may have overwritten previous ones. Consider a variable v that is defined at the end of $\mathcal{R}_{orig}$. This should satisfy the following expression:

$$\exists u \in def\left(\underline{\langle \overline{i} \rangle}\right) \cdot u \xrightarrow{\overline{i}} v \wedge reaches_v\left(\bullet\underline{\langle \overline{i} \rangle}\circ, succ(\mathcal{R}_{orig})\right)$$

From the definition of the $\rightharpoonup$ (Equation 5.9), $\Sigma(u) = \Sigma(v)$. Moving to the aggregated version, the variable v would need to satisfy the following expression instead:

$$\exists u \in def\left(\underline{\langle \overline{i} \rangle}\right) \cdot u \overset{\overline{i}}{\smile} t \wedge t \rightharpoonup v \wedge reaches_t\left(\triangleleft\underline{\langle \overline{i} \rangle}\triangleright, copyfrom\right)$$

Again, the *reaches* condition is proven to hold by contradiction. Assume that the reaches condition was false — i.e. that there exists an instruction $\langle \overline{h} \rangle$ such that:

$$\triangleleft\underline{\langle \overline{i} \rangle}\triangleright \prec \triangleleft\underline{\langle \overline{h} \rangle}\triangleright \prec copyfrom \wedge \exists s \in def\left(\langle \overline{h} \rangle\right) \cdot s \overset{\overline{h}}{\smile} t$$

If this condition is true, then using Equation 5.14:

$$\left(u \xrightarrow{\overline{i}} v, s \xrightarrow{\overline{h}} v\right) \in \rho(\mathcal{R}_{orig})$$

This would mean that the *reaches* condition for the original should not hold, since v would have been overwritten during the execution of $\overline{h}$. Since it does hold in the original, one can conclude that the *reaches* condition is true for the aggregate case. Therefore, using the definitions of $\smile$ and $\rightharpoonup$:

$$\Sigma(u) = \Sigma(t) = \Sigma(v)$$

5.3.6 Conclusion

It is now shown that for any n, $\mathcal{R}^{n+1}_{aggr}$ defines the same values as $\mathcal{R}^{n+1}_{orig}$ as long as no indirect variables occur.

It is now possible to show that remote aggregation is correct under certain conditions by a form of induction. Starting from $\mathcal{R}^0_{orig}$ where there are no remote actions, it can now be shown that $\mathcal{R}^1_{aggr}$ is valid provided there are no indirect variables in $\mathcal{R}^1_{orig}$. By applying the same line of reasoning to every remote cluster that occurs in $\mathcal{R}^1_{aggr}$, it can shown that $\mathcal{R}^2_{aggr}$ is correct, again provided that no indirect variables exist at any point. This process can be continued indefinitely to show that the entire computation $\mathcal{R}^\infty_{aggr}$ is correct.

Note that this is valid assuming that only remote clusters (i.e. sequences of adjacent remote calls that do not contain any local code) are aggregated, and that no indirect variables occur at any stage.

5.4 Problems with aggregation of clusters

This section focuses on the problems that may occur with aggregating clusters. The issues of callbacks (which manifest themselves as indirect variables in this theoretical framework) and pass-by-reference semantics are covered.

5.4.1 Callbacks

A problem arises when the use set of a remote instruction directly coincides with the definition set of a previous remote action, without any intervening *copyfrom* instruction. This problem has been encountered before during implementation, as covered in Section 4.6.2.

The solution is to add an extra *update* operation at the end of every remote action that defines an indirect value in the aggregated case. The effect of this *update* operation is defined by the following rule:

$$update_{\overline{h}} \in \mathcal{R}_{aggr} \implies \forall y \in def\left(\underline{\langle \overline{h} \rangle}\right) \cdot \left(\exists \overline{i} \in \mathcal{R}_{orig} \cdot (y \xrightarrow{\overline{i}} x) \in \rho(\mathcal{R}_{orig}) \wedge (y \rightharpoonup t) \in \rho(\mathcal{R}_{aggr})\right)$$
$$\Rightarrow (y \xrightarrow{\overline{h}} t) \in \rho(\mathcal{R}_{aggr})$$

This rule states that when an *update* operation occurs after $\underline{\langle \overline{h} \rangle}$, for every variable x directly defined by $\underline{\langle \overline{h} \rangle}$ that is later copied by a *copyto* instruction, the value of x will be copied into the variable t in the private space S. The variable t is the same variable that x is copied to by the *copyto* instruction in the aggregated case. This effectively makes indirect variables behave like internal variables when aggregated.

5.4.2 Reference semantics problem

Arguments are passed to methods by reference for local method calls, which results in problems as shown in Section 4.6.6. If pass-by-reference semantics for argument passing is

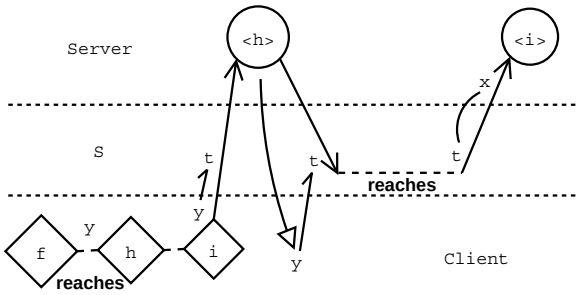


Figure 5.5: Effect of the *update* operation on indirect variables

employed, then the following rule will always hold for all adaptor operators in the aggregated case:

$$\forall y \in def\left(\underline{\langle \overline{r} \rangle}\right) \cdot x \overset{\overline{r}}{\frown} y \implies y \overset{\overline{r}}{\smile} x \tag{5.17}$$

This rule means that all changes made to values copied from space S by a remote action are guaranteed to be written back at the end of the remote action. This may cause some of the assumptions made in Section 5.3 to be invalidated.

In particular, the main approach taken to prove that *reaches* properties hold in the aggregated case is to 'back-map' every $\smile$ relation back to the equivalent $\rightharpoonup$ relation in the unaggregated scenario using Equation 5.11. However, this will not work with reference semantics, since $\smile$ relations may be introduced that do not have a corresponding $\rightharpoonup$ relation in the unaggregated form.

Consider trying to prove $reaches_v(i, j)$ for some $v \in S$. In addition to considering the $\smile$ relations due to Equation 5.11 for instructions between i and j, those that are due to Equation 5.17 must also be dealt with. Suppose an instruction k exists, where:

$$\exists k \cdot i \prec k \prec j \wedge \exists w \cdot v \overset{k}{\frown} w \in \rho(\mathcal{R}_{aggr}) \wedge w \in def(k) \tag{5.18}$$

From Equation 5.17, the relation $w \overset{k}{\smile} v$ occurs as a result of the definition of w by k.

Mapping the $\frown$ relation back to the $\rightharpoonup$ relation in the unaggregated cluster using Equation 5.10 (which remains valid since no rules generating new $\frown$ relations are introduced by pass-by-reference semantics):

$$\exists k \cdot i \prec k \prec j \wedge \exists u \cdot u \overset{k}{\rightharpoonup} w \in \rho(\mathcal{R}_{orig}) \wedge w \in def(k) \tag{5.19}$$

If this expression is true, then $reaches_v(i, j)$ must be false in the aggregated case due to the write-back assignment to v caused by the definition of w. However, this is not necessarily the case for the unaggregated call since there is no rule forcing an equivalent $w \overset{k}{\rightharpoonup} u$ relation to exist. This means that the proofs in Section 5.3 are only valid using reference semantics if Equation 5.19 is always false wherever a *reaches* condition appears in an unaggregated context.

5.5 Aggregating across local code

The conditions necessary for delaying remote calls across local code are:

1. The local code must not depend on any values from previous remote calls
2. The local code must not overwrite any values used by previous remote calls except those passed as call parameters
3. The local code must not perform any I/O

In terms of the framework, for a block of local instructions $\mathcal{L}$ following a remote cluster $\mathcal{R}^n$, these may be expressed as:

$$def(\mathcal{R}^\infty) \cap use(\mathcal{L}) = \emptyset$$

$$def(\mathcal{L}) \cap (use(\mathcal{R}^\infty) - use(\mathcal{R}^n)) = \emptyset$$

$$\neg isinputop(\mathcal{L}) \wedge \neg isoutputop(\mathcal{L})$$

Under these conditions, it is trivial to show that $\mathcal{L}$ can execute unchanged since the first condition ensures that there are no dependencies between the relocated remote call and $\mathcal{L}$. Since it is prohibited to delay across local code that performs I/O by the third condition, all I/O operations are guaranteed to occur in the same order since any I/O performed by the remote calls will be forced to come before the I/O performed by the local code.

The issue of hidden anti-dependencies due to callbacks is dealt with by the second condition. However, direct anti-dependencies must be dealt with. Direct anti-dependencies occur when:

$$antidep(\mathcal{R}, \mathcal{L}) \Longleftrightarrow \mathcal{R} \prec \mathcal{L} \wedge use(\mathcal{R}) \cap def(\mathcal{L}) \neq \emptyset$$

These are corrected by a combination of precopying and variable renaming. Consider two remote clusters $\mathcal{R}_1$ and $\mathcal{R}_2$ separated by a local block $\mathcal{L}$. When aggregated, $\mathcal{R}_1$ is moved to after $\mathcal{L}$, and a *save* operator (denoted as $\oplus$) is left in its original place. A *save* operator is also placed for $\mathcal{R}_2$ even though it is not moved relative to $\mathcal{L}$. Before the corresponding remote action for $\mathcal{R}_i$ is executed, a *load* operator (denoted as $\ominus$) is used to recover the state saved by the corresponding *save* operator.

$$\begin{aligned}
\mathcal{R}^n_{orig} &= \overline{\mathcal{R}_1}; \mathcal{L}; \overline{\mathcal{R}_2} \\
\mathcal{R}^{n+1}_{orig} &= \bullet\underline{\langle\overline{\mathcal{R}_1}\rangle}\circ; \mathcal{L}; \bullet\underline{\langle\overline{\mathcal{R}_2}\rangle}\circ \\
\mathcal{R}^{n+1}_{aggr} &= \oplus\mathcal{L} \oplus \bullet\left(\ominus\underline{\langle\overline{\mathcal{R}_1}\rangle}; \ominus\underline{\langle\overline{\mathcal{R}_2}\rangle}\right)\circ
\end{aligned}$$

The effect of the *save* operator is to make a copy of the relevant subset of the used state as it existed before the execution of $\mathcal{L}$. This copied state is conveyed to the server using the *copyto* operator. Before the execution of each cluster:

$$\begin{aligned} use\,(save_i) &= use\,(copyto_i) \\ use\,(copyto_{\mathcal{R}_{aggr}}) &= \bigcup def\,(save_i) \end{aligned}$$

The behaviour of the *save* operator is defined by the $\hookrightarrow$ relations, while that of the *load* operator is defined by the $\mapsto$ relations.

$$\begin{aligned} x \hookrightarrow y &\implies \Sigma(x) = \Sigma(y) \\ x \mapsto y &\implies \Sigma(x) = \Sigma(y) \\ a \rightharpoonup b \in \rho_{orig}(\mathcal{R}_i) &\implies \left\{ a \overset{\mathcal{R}_i}{\hookrightarrow} c, c \rightharpoonup d, d \overset{\mathcal{R}_i}{\mapsto} b \right\} \subseteq \rho_{aggr}(\mathcal{R}_{aggr}) \end{aligned}$$

For an arbitrary cluster of calls $\mathcal{R}_i$, a variable a that was originally copied to a variable b by a *copyto* instruction is instead copied to a variable c while remaining in the client space. When the remote calls are finally executed, c is copied to a variable d in a private remote space. Finally, when $\mathcal{R}_i$ is about to be executed, d is copied into b to be used by $\mathcal{R}_i$.

Variables c and d can only be accessed only by one pair of $\hookrightarrow$ and $\mapsto$ operators, and so by the definition of the various relation types, it can be determined that $\Sigma(a) = \Sigma(c) = \Sigma(d) = \Sigma(b)$. Intuitively, one can see that the value of b that reaches $\mathcal{R}_i$ will be the value of a in its original position before the clusters were aggregated, regardless of whether a is modified by local code later.

5.6 Evaluation

The approach presented in this chapter is very limited since it is not a true semantic model. It deals only with data-flow between blocks of code, but not how the blocks themselves use and modify the data, and so cannot be used to predict what a program will actually do. It can only be used to reason about the equivalence of programs that undergo restructuring transformations, where blocks of code are rearranged but not modified in terms of functionality.

Nevertheless, the model is believed to be sufficient for showing the correctness of aggregation applied to remote clusters, since remote call aggregation does not change the functionality or the ordering of the called methods. Although the proof in Section 5.3 could be better formulated and strengthened much further, the basic logic is sound.

Section 5.5 is much less convincing. There are conditions that are trivial to define in this framework, but can be very difficult to determine in actual practice, implying a lack of explanatory power. One example is the possibility of a callback occurring in a remote call that is to be reordered with respect to local code that uses a value defined by the callback. Numerous approaches to detecting this have been covered in Section 4.6.2, and yet it only takes a short expression to express this in the logical framework, belying the difficulty involved.

Another problem is that there is no real notion of control-flow in the framework, since it dealing with traces. For example, the problem with executing code that should not have been

reached due to exceptions (see Section 4.6.4) does not occur in the framework because the erroneously executed code simply does not exist in the original trace. All that occurs in the framework when an exception is thrown is that the remote call that resulted in an exception is prohibited from being delayed across the exception handler due to the dependency on the thrown exception.

5.7 Related work

Considering the widespread degree to which optimising transformations in general are used by compilers, there has been a surprising scarcity of work regarding formal semantic proofs of their correctness, although the theoretical foundations of program analyses [70] used to drive the transformations are well established. One recent attempt is described in [47], which uses temporal logic to prove the correctness of classical optimisations such as dead code elimination and constant folding. Another approach [49] uses an automatic theorem prover to prove optimisations written as a set of rewrite rules on a domain-specific language called Cobalt.

There does not appear to be any previous work that formalises the semantics of RMI, although formalisations of various aspects of component models such as COM exist. One example is [78], which focuses mainly on the interface aspect of COM components.

5.8 Conclusion

In this chapter, a logical framework has been developed on which the effects of the aggregation of remote calls on the data-flow of a program can be modelled. By using a process of simple logical deduction, it can be shown conceptually what happens to remote calls when a RMI transformation is applied, and demonstrate that the results of such calls are equivalent to unaggregated calls under most circumstances. The conditions under which the optimisation leads to incorrect results are shown, and how one may compensate for these cases. The various issues that arise are linked back to practical issues that arose during the implementation process discussed in previous chapters.

Chapter 6

Conclusions

In this chapter, the contributions made by this thesis are summarised, and in light of the various problems and weaknesses that have been discovered during the investigation, suggestions are made for improvements on the current work and directions which future research on this topic may take.

6.1 Summary

This thesis presents a new approach to automatically optimising distributed applications written in Java. This approach is based on the concept of delayed evaluation, where remote calls are not executed immediately, but are delayed for as long as possible. By delaying multiple calls simultaneously, it is possible to build up knowledge of the wider context in which remote calls are made. This enables a number of optimisations to be made when the delayed calls are eventually forced to execute.

Call aggregation reduces the overall network latency incurred by sending all pending remote calls to the server using a single network transaction. It also conserves bandwidth by sharing data between calls and discarding unneeded results. Server forwarding uses knowledge of the data dependencies between calls and the network topology to exploit connections between servers, which can be considerably faster than client-server connections. Plan caching exploits implicit knowledge that clients and servers have of each other gained during previous interactions to reduce the communication overhead of the RMI optimisations.

These optimisations require runtime support in order to perform the delaying of remote calls and to efficiently compute the data dependencies between sections of code. This was achieved by building Veneer, which offers an interpreter-like model of execution that can easily be customised by modifying the interpretive code. Veneer performs well compared to other interpreters built on top of Java virtual machines since most of the virtual machine functions are delegated to the underlying virtual machine, with Veneer intervening only where necessary to achieve the desired effect.

A prototype implementation of the new RMI optimisations was built on top of Veneer, and evaluated using a selection of test applications. The experimental results show that the optimisations are effective in speeding up RMI calls, with the effectiveness increasing as the number of calls that are simultaneously delayed increases. However, the runtime overhead of Veneer is a significant factor that limits the potential speedup when operating on fast

networks.

A number of issues have been identified which may lead to differing program semantics after the optimisations are applied. They arise mainly due to the reordering of remote code with respect to code running on the client. These are resolved by strategically placing force points in the client, which result in any delayed remote calls immediately executing, thereby preserving the relative ordering of execution.

In order to verify the correctness of the RMI optimisations, a theoretical framework was formulated on which to reason about the effect of the RMI transformations on the data flow of distributed programs. By showing that the same data arrives to equivalent blocks in the unoptimised and optimised versions of the program, it is proven that the aggregation optimisations do indeed preserve the behaviour of the program under most circumstances. The exceptions and the workarounds can also be explained in terms of the framework.

6.2 Future work

In this section, some ideas are presented for directions which future research and development could take. These include topics such as further evaluation, optimising the existing prototype, implementation on different platforms such as .NET, improvements and extensions to the DESORMI approach, and integration with other schemes such as asynchronous remote calls.

6.2.1 Further evaluation

There are many other tests that could be performed on the DESORMI framework. The most obvious is to simply extend the suite of test examples, preferably including real-world examples used in industry.

The experiments performed in Section 4.8 have concentrated on evaluating the effect of the DESORMI optimisations with regard to client performance. However, relatively little attention was paid to the performance of the server. An obvious experiment to try would be to bombard an DESORMI-enabled server with requests from multiple clients to determine how well it scales under increasing loads.

The DESORMI optimisations should have a two-fold effect on the server:

- It should decrease network utilisation — the total number of messages should be reduced due to call aggregation, and the size of the messages should be reduced by the data optimisations and plan caching.
- It should increase CPU utilisation — the server needs to perform various extra operations such as:
  - Iterating through the remote plan
  - Checking to see if argument copying is necessary, and copying if it is
  - Storing and looking up plans in the cache

The outcome will almost certainly depend on the server configuration and the applications being run, but nevertheless the results of such experiments should prove interesting and will be more evidence regarding the viability of the approach detailed in this thesis.

6.2.2 Improved early detection of RMI calls

Confirming that potential remote call sites are actually remote at runtime is not an expensive operation, but it does incur a little runtime overhead. However, there are several ways in which this overhead may be further reduced if necessary.

6.2.2.1 Using type-inference

One method of reducing the spurious detection of potential remote calls is to perform type inferencing on the bytecode in an attempt to find out the possible runtime types of an invoked object [33]. If the inferencing establishes that all possible types for the object are classes that are not descended from `java.rmi.RemoteStub`, then the method call cannot remote.

However, this approach is not as useful as it might appear. Stubs are generally loaded at run-time from the server, and are never instantiated directly by the client. The result is that static type-inferencing cannot possibly detect if an object is in fact a stub, but only if it is definitely *not* a stub. Dynamic class-loading can also make the problem harder by extending the class hierarchy of the program at runtime.

6.2.2.2 Using data-flow analysis

Remote communication generally starts with a remote stub being fetched from a RMI registry. Since RMI registries always return references to remote stubs, it should be possible to trace the path taken by this stub through the program and mark all invocations on this stub as being definitely remote. However, since pointer analyses tend to indicate 'may point to' relationships rather than 'must point to' relationships between objects and references, the scope for improvement might be limited since a reference *must* point to the stub returned from the registry in order to be identified as definitely being remote.

6.2.2.3 Reducing runtime type-checking

The runtime test to check that a potential remote call site is actually remote does not necessarily need to be done at every potential remote call site. If a subsequent call site uses the same receiver, and the reference holding the receiver object is not changed in all possible control paths between the first test and the call site, then it is safe to assume that the remote status of the receiver has not changed since remote stubs are immutable.

6.2.3 Using high-level information

The DESORMI optimisations presented by this thesis are fully automatic and need no assistance from the programmer. This means that the optimisations must operate conservatively, favouring correctness over performance. This is particularly evident when dealing with local code that separates remote calls, where there are a multitude of conditions that will lead to remote execution being forced. However, there may be tremendous scope for improvement if the programmer was allowed to give hints to the DESORMI optimiser as to how to proceed, perhaps via program annotations. This allows the optimiser to make use of the high-level knowledge possessed by the programmer regarding program behaviour.

The simplest and most effective hint would be an annotation that marks local code as being safe to reorder remote calls across, since this would simplify the operation of the optimiser considerably. However, it may be difficult for the programmer to decide whether or not this property is true for a block of local code. In lieu of this, there are many other annotations that can help the optimiser to decide whether call aggregation is safe — for example, annotations that state that certain force conditions cannot occur, or that calls to certain methods in libraries will not perform any I/O or modify their arguments.

Note that current DESORMI optimiser already makes use of high-level information to a limited degree. For example, the fact that some classes are immutable is used in Section 4.6.6.3 to avoid copying arguments to remote calls. However, this knowledge is hard-coded into the optimiser, and is therefore inflexible.

6.2.4 Intermediate local method calls

At present, pending remote calls are always forced before any calls to other local methods are made. This is a conservative means of ensuring that any I/O performed by the delayed remote methods and the callee will appear to an outside observer to occur in the correct order.

This restriction might be relaxed in several ways. Exceptions can be made for frequently occurring special cases that are known to be free of I/O operations, such as calls to the string-building methods. However, this only helps when direct calls to these methods are made.

A more effective way might be to set up a Veneer policy that modifies all classes loaded by the custom class loader to include a notification to the Veneer runtime in methods that contain any direct calls to I/O methods. In this way, any call chain that leads to a method containing I/O will result in the notification method being called just before any I/O is performed, which gives the executor the opportunity to force any pending calls before returning from the notification method.

Yet another solution might be for the server to analyse the available remote methods and conservatively determine the subset that is not capable of performing I/O. This information can be communicated as metadata from the server Veneer instance to the client instance. If all delayed remote methods are known to be incapable of I/O at a point where local I/O may occur, then it is safe to delay the remote methods across the local I/O operation. Although the usefulness of this approach might be limited if the I/O detection is overly conservative, it should at least be no worse than the current scheme of forcing whenever a local I/O operation may occur.

6.2.5 Improved data reuse

In the current scheme, data used by delayed calls is preserialised when any local code is encountered that has the possibility of changing any of that data. From that point onward, all of the serialised data is considered 'old', and invalid for subsequent remote calls. This means that subsequent remote calls must use a second independent set of data that is collected from that point on. This conservative scheme is safe but inefficient, since it is likely that only a small subset of the data will have been changed, resulting in the same data being sent multiple times. One obvious solution is to use more precise data analysis

techniques to better discern which subset of the data is being altered, and invalidate only that subset instead of the entire set.

An alternative solution that has been considered is to seize control of the data serialisation process. The idea is to serialise data as normal until an operation occurs that may invalidate some of that data. At that point, the serialisation routine enters a 'safe' mode. In this mode, when a request is made to serialise the contents of a variable, the data is transformed into a sequence of bytes as usual, but instead of being appended onto the end of the byte stream, it is compared with the previous serialisation of that variable byte-for-byte. If they match, then the data for that variable has been left untouched, and a reference to the previous data can be appended instead of the entire byte sequence for that variable. This variable can also be marked as being 'clean', so that subsequent serialisations of the same variable do not have to go through the search again until another invalidation point is encountered.

The performance of this scheme depends on the efficiency of the search. If the time taken for the search is smaller than the time taken to transport the duplicated data in the original scheme, then the scheme will pay off.

6.2.6 Using performance metrics to control forwarding

Currently, the behaviour of the server forwarding algorithm is controlled by a set of heuristics that may not be valid in all cases. The assumption that server-server connections are always much faster than client-server connections is especially dubious. This assumption is usually true for present day enterprise systems, where companies typically rent servers in a well-connected Internet hosting service provider, and the clients connect using a wide spectrum of methods, from dial-up modems to DSL lines and beyond. However, this assumption is not necessarily true when applied to peer-to-peer systems, where servers and clients often connect using networks of equal speed.

An alternative to using heuristics is to measure the explicitly latency and bandwidth of the connections between servers, and adjust the forwarding scheme accordingly to select for the highest performing set of connections. This could be done using the cost model developed in Section 4.4.1.

It can be difficult to calculate the data size due to polymorphism and dynamic data structures. For example, if the method invoked on server A returned a vector, then the data size is in general unpredictable in advance since the number of elements and the type of the elements is unknown.

6.2.7 Direct data forwarding

Server forwarding can result in better performance due to the fact that it is generally faster for two parties A and B to communicate directly rather than via an intermediary C. However, the current server forwarding scheme only partially accomplishes this. For example, consider the test program in Section 4.8.3. For convenience, it is reproduced it here.

```
temp = r1.add(v1, v2);
result = r2.add(temp, v3);
```

Although server forwarding enables *temp* to be sent directly from server *r1* to *r2*, the variables *v3* and *result* are both forced to pass through server *r1*. A speedup is still achieved

when using DSL because the time required to forward *v3* and *result* over the Ethernet link connecting the servers is so small compared to the cost of transferring *temp* back and forth over the DSL link. A more efficient scheme might be for the client to send *v3* to server *r2* directly, and receive *result* directly back. This might be done as follows:

1. A session ID is generated for the current group of aggregated calls.
2. The client sends *v1*, *v2*, a reference to the remote proxy on *r2* and the session ID to *r1*, and simultaneously sends *v3* and the session ID to *r2*
3. *r1* uses *v1* and *v2* to compute *temp*. *r1* sends *temp* to *r2* tagged with the session ID and finishes.
4. *r2* waits until it receives a copy of *temp* tagged with the current session ID. It then uses the received value of *temp* in conjunction with *v3* to compute *result*, which it returns directly to the client.
5. The client may proceed when every remote call that results in a live value returns (in this case, the client must wait for *r2* to return, but not necessarily *r1*).

It might be possible to use one-way sends during the sends from the client to *r1* and from *r1* to *r2*, since the sender does not need anything from the receiver. However, it is probably better to use the standard RMI call-response communication pattern since the response can act as an acknowledgement that the send has completed successfully. If this acknowledgement is missing, then *r2* can wait indefinitely for a value that never arrives.

Another major advantage of direct forwarding is that the issues covered in Section 4.9.2 disappear since *r1* is no longer responsible for handling data intended for *r2* or the client.

6.2.8 Interprocedural delayed RMI

In the present scheme, delayed calls are always forced before the method that made the calls exits. There are two main reasons for this. Firstly, there is currently no guarantee that there is an intercepted method further up the call chain, meaning that some RMI calls may never be called if not forced at the end of the current method.

It is fairly easy to solve this problem by keeping track of the active executors in each thread. Every time an executor is entered, a reference to it is pushed into a thread-local stack, and every time it exits, it is popped off. By checking the executor stack of the current thread, it is easy to determine if there is an intercepted method further up the call chain.

However, a more important problem occurs if the caller of the current method is not an intercepted method. If it uses any data that was defined as a result of the delayed remote calls, then it will incorrectly use the old values that were present before the calls were executed, since it will be unaware of the need to force delayed calls.

A partial solution for this might be for an intercepted caller to scan program fragments for local calls at runtime. If it can be determined that the callee is an intercepted method, then a flag can be set that informs the executor of the callee that it is safe to leave unforced remote calls in the remote queue since the caller will be aware of the possibility of delayed calls remaining on the remote call queue on exit.

6.2.9 Improved alias handling

One common operation encountered is to store the results of calls in an array. For example:

```
for (int i = 0; i < 100; i++) {
    result[i] = r.f(i);
}
```

If `r` is a remote reference, then this code would lead to a force on every iteration, since the assignment of the result of the remote call to the array `result` is considered a use of the result. This prevents the aggregation of the 100 calls in the loop.

One way to combat this might be to make a special case for assignment statements, such that the assignment does not cause a force. Instead, the assigned-to variable is added to the set of variables defined by the remote call, so that subsequently reading from it will cause remote execution to be forced. One caveat with this scheme is that it must be possible to intercept all reads from `result`. In general, this means that `result` must not have escaped from the current method.

6.2.10 Inter-thread delayed RMI

Currently, one delayed call queue is maintained for every thread of execution on the client. It might be possible to reduce this to just one queue, with all threads sharing the same queue. When this queue is flushed, this will cause the delayed calls of multiple threads to be forced. This should be relatively safe, since the order in which calls are placed into the shared queue reflects the order in which they would have occurred in the absence of the DESORMI optimisations.

The effect on performance depends on a number of factors. A naive implementation that executes calls strictly in the order in which they occur may actually reduce performance, since remote call clusters that appear in one thread may be split up by unrelated remote calls being interleaved with the calls of the cluster, thereby reducing the opportunities for optimisation. A more intelligent implementation would take advantage of the flexibility provided by the Java memory model to optimally reorder the calls that occur in a thread relative to calls from other threads so that calls containing optimisation opportunities are situated next to each other to allow optimisation to occur.

The performance may also be reduced if a simple scheme is adopted whereby one thread forcing execution will cause the delayed calls from all other threads to be forced. For example, consider a program with two threads. Thread A has few optimisation opportunities and forces execution frequently, while thread B has many optimisation opportunities and rarely forces. If forcing thread A also forces thread B, then thread B will be forced as frequently as thread A, which may destroy optimisation opportunities in thread B.

A better solution would be to implement partial execution forcing, where a force initiated by thread A will only force the remote calls delayed by thread A plus the calls from thread B that may be profitably combined with calls from thread A. The remaining calls should remain on the queue.

6.2.11 Server plan code optimisation

When the server caches a plan, it builds up a 'canned' sequence of code that is customised for the client that sent the plan. At present, this facility is used as a form of data compression, to avoid sending plans that have been sent before in full form. However, on the server side, these code sequences provide a good point to apply compiler optimisations.

At present, all remote plans (whether cached or not) are interpreted by the server side executor, which substitutes the actual server object in place of the remote stub and executes the call using reflection. However, on the server side, the exact identities of the remote call receivers are known, and their bytecode implementations are accessible. It should therefore be possible to safely inline the bodies of the methods into the plan, optimise it using standard compiler techniques, and execute the result. This inlined plan can then be associated with the cached plan for subsequent executions.

The degree of speedup will depend on the nature of the inlined methods — there may be no noticeable speedup at all in some cases. Nevertheless, this should at least eliminate the overhead of reflective dispatch in all cases.

However, there are potential problems. The difference between the pass-by-value semantics of remote calls and the pass-by-reference semantics of local calls may necessitate extra code being inserted between calls to compensate for this, adding an another layer of complexity.

The cost of inlining and optimising the plans may also be a significant factor. If the call sequence is only executed a few times, then the speedup gained by executing optimised might not be worth the initial cost of optimisation. The decision to optimise should therefore be based on a cost model balancing the costs and benefits of optimisation. A simple way of achieving this might be to associate a counter with each plan that is incremented every time the cached plan is used, and only perform the optimisation after a certain trip-count has been passed.

6.2.12 Asynchronous delayed RMI

One obvious possibility for improving the performance of RMI is for the remote calls operate in parallel with the client instead of the client blocking whenever remote calls are executed. However, this involves executing remote calls as soon as possible for the best chance of operating in parallel, and so is fundamentally at odds with the concept of call aggregation which executes calls as late as possible. Parallel execution works best if the client can continue working while the remote call is processing, while aggregation works best if the client cannot.

A compromise can be made by having a 'committer' thread running alongside the main application threads. The task of this committer thread is to periodically flush the contents of the delayed call queues. Since the force occurs in the committer thread, the application threads will remain unblocked, so that the application and the flushed server methods can execute concurrently. If a force point has not yet been reached by the application, then it should be safe for the delayed calls to be executed in parallel to the application since by the definition of a force point, no code that is dependent on the delayed calls has been encountered.

The behaviour of the system at previous force points will also change. Consider a point in

the original scheme where execution must be forced. There will now be several possibilities as to what will occur at this point in the new scheme:

- The calls have already been flushed by the committer thread, and the results have been written back — in this case, the point is no longer a force point, since all dependencies have been resolved at that point. However, this may later lead to new force points where there were none before if there are still unforced calls on the queue, since these calls would previously have been flushed at this point.

- The calls have already been flushed by the committer thread, but the results have not been written back yet — in this case, the application thread must wait for the results to be written back into the local variables. This can be done by using the usual spinlock or wait-notify mechanisms. Once the write back has occurred, execution may continue. New force points may still occur in this scenario, because new calls may have been added to the stack since the committer thread was started.

- The calls have not yet been flushed by the committer thread, but the committer thread is still active — if it could be ensured that the calls currently in the queue are independent of the calls currently being forced by the committer thread, then in theory it should be possible to flush the queue immediately, which may lead to two remote plans from the same application executing concurrently on the server side. However, this is highly unlikely to work in practice, since the calls may have side-effects on the server side that later calls will depend on. In lieu of this, the application thread must wait until the committer thread has finished, and then manually flush the queue. There will be no new force points, since the queue will be empty at this point.

- The calls have not yet been flushed by the committer thread, and the committer thread is inactive — the application thread should immediately try to flush the queue. The `flush` method should be protected by a lock to prevent both threads from flushing at the same time. It does not matter if the committer thread awakens and acquires the lock just before the application thread does, since regardless of which thread 'wins', the entire queue will be flushed — the 'loser' merely attempts to flush an empty queue. In this case, no new force points are created.

The main variable in this scheme is how frequently calls are forced by the committer thread. At one extreme is the situation with the current framework, where calls are forced only when a dependency has arisen. The other extreme is for calls to be forced as soon as they are placed on the queue, which makes the call operate in parallel with the client. Asynchronous delayed RMI would make it possible to strike a balance between the two extremes, balancing the factors as shown in Figure 6.1.

Experimentation is needed to determine an optimal value, and the best trade-off will probably vary from application to application and from run to run. It might even be possible to add instruments to the program to automatically adjust this on-the-fly in response to usage patterns.

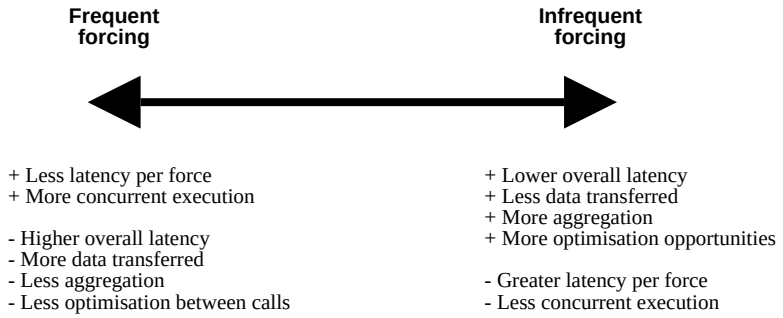


Figure 6.1: Balancing factors in asynchronous delayed RMI

6.2.13 Delaying local code

Remote calls are currently delayed as they are encountered during the course of program execution, and their execution is forced only when some local code is encountered that necessitates it. The occurrence of these force points effectively limit the number of calls that may be aggregated at any one time. One way in which this limit may be broken is by speculatively aggregating remote calls beyond the force point.

This might be done by making the executor begin delaying blocks of *local* code rather than remote calls when a force point is reached, adding any remote calls encountered to the current remote queue. When no more local code can be delayed due to data or control dependencies, the accumulated remote calls can be forced, followed by the delayed local code.

This approach carries a number of risks. Most of the caveats that apply to delaying remote calls also apply to delaying local code — for example, an exception might occur in local code that was reordered relative to a remote call, meaning that the remote call should not have been executed in the first place.

The dependence between data generated by local code and used by later remote calls must be carefully taken into account when delaying local code, since this can lead to incorrect data being supplied to the remote method if the relationship escapes detection. This is a much harder problem compared to detecting data generated by remote calls, since there are fewer restrictions on what data the local code may produce.

Another problem is the risk of I/O becoming reordered if the reason for a force was due to I/O in local code. Previously, any code that may lead to I/O resulted in a force, so that any I/O that might be performed by the remote methods would occur before the I/O of local code. However, if the remote methods moved to before the local I/O instruction also perform I/O, then the I/O will be performed in the wrong order. Remote I/O detection (see Section 6.2.4) may help with this situation.

6.2.14 Code motion between hosts

The basic concept behind fragmentation is to split a program into pieces that may be manipulated as first-order objects, which includes the ability to send those pieces to remote hosts to be executed remotely. Currently, only limited use is made of this facility by sending remote method calls to remote hosts. This could be extended to include sections of the

client code that do not involve remote calls, so that portions of the client can be executed on the server.

A similar situation occurs on the server side. Methods in remote objects are not fragmented unless they contain remote calls of their own, but there is no reason why they cannot be fragmented as well, and the code fragments sent to the clients or other servers to be executed.

6.2.14.1 Exporting loops

One application for moving code would be exporting entire loops from client to server. For example, a loop may contain a loop condition that is dependent upon values defined by the remote calls, thereby causing the delayed calls to be forced on every iteration of a loop. Even if there is no such dependency, the resulting remote plan generated by the loop would be directly proportional in size to the number of iterations taken in the loop, which seems to be needlessly inefficient considering that it is the same group of calls being repeated over and over again.

If the entire loop is exported across to the server, then the server can perform the loop test locally without needing to return the results of each iteration back to the client. Instead, the final result can be returned as the entire loop finishes, with intermediate results that are dead at loop exit being discarded. There is also the advantage that loops will be supplied in a much more compact form compared to the current one (where all loops are effectively completely unrolled), thereby saving network bandwidth when transferring the remote plan.

A problem with this approach is when local objects are referred to within the plan body. This data will also need to be moved to the server if the entire loop is exported. However, not all of this data may be serialisable, and even if it is, the resulting byte-stream, which represents the *closure* of the used data, may be very large. This is especially inefficient considering that it is unlikely that all of the transferred data will be used.

One solution might be to perform a loop fission transformation on the loop [8], splitting the loop into two independent loops. One loop would contain only remote calls, while the other would only contain the local code. If this transformation cannot be done, then the old scheme of forcing on every iteration can be used as backup.

6.2.14.2 Code fragments as mobile agents

If code motion of program fragments is taken to the extreme, then this would be a radical departure from the conventional distributed programming model with two static programs communicating with each other via strictly defined channels. Instead, programs consist of a 'sea' of program fragments that may migrate across network boundaries to find the most suitable host to be executed on. In some respects, this ties in with the concept of multi-agent systems [31], which perform an overall task using a group of autonomous agents that may migrate from host to host, interacting with their current host at each step.

6.2.15 Veneer performance optimisation

Despite the many optimisations performed on Veneer, it is still relatively slow if fragmented methods are being entered often. There are many possibilities for improvement, a few of which are listed here.

6.2.15.1 Method-specific state

Much of the extra overhead caused by Veneer is due to the need to load and store the intermediate state of methods. Although the overheads have been dramatically reduced by pooling state and using reusable value wrappers, there remains substantial overhead, even if the array containing the stored state is accessed directly rather than via accessor methods.

One way to reduce the overhead further might be to generate a new state class for every method instead of having a generic state class for all methods. The method state can be stored in fields rather than as elements in an array. This means that there is no need to explicitly fill the state with default values when initialised, and method-specific code blocks such as fragments can access the state by directly manipulating the fields of the state, eliminating the need for type-casts.

The accessor methods can be retained so that generic block types such as method calls can still refer to local variables by an index. These bodies of these methods will need to be generated for every state class though, since the types and names of the fields will vary from class to class.

6.2.15.2 Switchable fragmentation schemes

The act of breaking a method down into smaller blocks will always slow performance down due to overhead incurred when dispatching to the blocks and restoring/saving local context, and also because an optimising JIT compiler will no longer be able to optimise across the fragmented blocks of code. One way of relieving these problems is to note that it is not always necessary to intercept execution at various points.

For example, if it was discovered that a potential remote call site did not actually make remote calls at runtime, then there is no need to intercept execution at that point as long as the receiver of the call remains constant. In that case, two copies of the code can be made — one that is fragmented into three blocks with the breaks occurring before and after the potential remote call, and another that consists of a single block containing the same code. Before entering these blocks, the executor can decide on which variant of the code to execute. If the receiver of the call is known to be non-remote, then the single large block can be chosen for better performance.

6.2.15.3 Low-level support

It might be possible to get substantially better performance if portability was sacrificed, and support for fragmentation scheme was added directly into the underlying JVM. Some useful additions might be:

- A way of saving and restoring the current method state to and from a form that can be manipulated by the Veneer runtime, since the current method of doing this (by accessing an array element-by-element, type-casting at every step) is slow.
- A way of dispatching directly to the executor when an intercepted method is called. The current approach uses a stub that initialises the method state before dispatching to the executor. If the JVM dispatches directly to the executor, the extra level of indirection might be avoided, and if state saving/restoring is handled by the JVM, then the JVM would also be the logical place to initialise the state.

6.2.15.4 Specialisation

One of the reasons that slowdowns occur in Veneer is due to the multiple levels of indirection that exist:

- A method stub calls an executor, with the executor type depending on the runtime policy and the method plan
- The behaviour of an executor depends on the plan supplied to it, and the blocks within the plan

Some of these indirections might be removed by a process of specialising these runtime entities with respect to one another. For example:

- Specialising a stub on the runtime policy and method plan can result in a fixed executor selection, enabling the executor to be inlined
- Specialising an executor with respect to a plan and its blocks results in a fully compiled version of the plan as executed by that executor

In terms of actually implementing these specialisations, it seems unlikely that generalised partial-evaluation systems like Tempo [23, 83] would be able to cope with specialising Veneer, so another option is to move towards an explicit code-generation scheme.

For example, instead of the executor actually executing instructions between blocks, it can generate a compiled version of those instructions instead, which can be automatically inserted between the code blocks. However, this is a step back towards the static program rewriting approach (see Section 3.1.1), with the inherent complexities of that approach.

6.2.16 .NET Remoting

Remoting is the .NET equivalent of RMI which provides a distributed object model for the .NET framework. Although it shares much in common with RMI, there are also important differences. For example:

- Objects are bound to explicitly stated port numbers — in RMI, this is hidden from the developer since RMI can share ports between objects. This means that no bootstrapping phase with a remote registry is needed with remoting, but the port number does need to be known to the client.
- No remote interface is necessary — there is no need to define a separate interface to declare the methods that may be remotely accessed.
- No stub compiler is necessary — the communication infrastructure is part of the Common Language Runtime, with no need for external stubs and skeletons. Instead, stubs are generated at runtime on demand.
- Custom sinks — custom sinks are very similar to JBoss interceptors (see Section B.3.3.2), in that they permit a developer to process and perhaps modify the data that is sent to and received from the server before and after a remote call is made.

At the time of writing, a version of the call aggregation optimisation is already being developed for the Mono implementation of .NET. This implementation is called the Runtime Remoting Optimiser (RROpt) [22], and is based on modifications to the Mono interpreter, Mint.

Although RROpt operates at the level of the virtual machine, it suffers from high overheads because it does not perform any program analysis beforehand, so everything must be performed at runtime while the program is running. The ability to avoid returning dead return values from remote calls is missing due to the lack of data-flow analysis, and none of the other DESORMI optimisations (i.e. server forwarding and plan caching) have been implemented either.

6.2.17 Veneer using .NET

The .NET framework [77] provides several features that the Java platform lacks, which might have proven very useful when developing the Veneer virtual JVM.

6.2.17.1 Pass by reference

References to objects are passed by value to methods in Java. This makes it possible to perform operations on the referenced object that are visible outside of the method, but it makes it impossible for a callee to change the actual parameters of a method call. This has led to the use of 'packing' and 'unpacking' code in fragment methods, since directly assigning to a reference variable will only update the local copy, and not the one passed in by the caller[1].

The .NET framework provides two mechanisms around this. The first is the ability to take the address of variables which makes it possible to pass in a pointer to the variable containing the reference to a method, enabling the callee to change the reference. However, the use of pointers renders the code 'unsafe', since the verifier cannot no longer guarantee the type-safety of the program in the presence of pointer manipulation.

The second method is by using reference parameter types. .NET supports passing parameters by reference in a similar way to C++, such that changing the parameter within the method body will change the passed value in the caller too. Unlike the first method, this is type-safe.

6.2.17.2 Interprocedural control flow

Java is severely restricted in terms of the options available for directing control flow. This leads to problems when attempting to write an efficient interpreter, or interpreter-like programs.

Consider trying to write a direct-threaded interpreter, where every instruction implementation jumps directly to the implementation of the next instruction in line. Since this kind of jump is not permissible in either Java or .NET, the closest alternative available is a virtual call. However, there are only three ways in which execution can move from one method to another — by calling a method, by returning, or by throwing an uncaught exception. If every instruction implementation calls its successor, then the stack will rapidly

[1] Note that Java inner classes also use a similar mechanism when accessing their enclosing classes.

overflow. The solution currently used is to return the ID of the successor and have the caller call the successor, which incurs additional overhead.

However, MSIL [53] (Microsoft Intermediate Language — the .NET equivalent of Java bytecode) provides the `jmp` instruction and the `tail` prefix that might be used to move from instruction implementation to instruction implementation more efficiently.

`jmp` instantly transfers control from the current method to the specified method. The stack frame of the current method is destroyed and replaced with that of the destination method. The arguments of the new method are set to those of the old method.

The `tail` prefix is used in conjunction with one of the `call` family of instructions to form a tail call. The tail call destroys the current stack frame, replaces it with that of the destination, fills in the arguments, then transfers control to it.

The main differences between `jmp` instructions and `tail.call` instructions are:

- `tail.call`s can only occur before the method returns. `jmp`s can occur anywhere in a method.
- `jmp` is an unsafe instruction, and hence any code that contains it is unverifiable. `tail.call` instructions are safe however.
- For `jmp` instructions, the destination method must have the same type signature as the method that contains the `jmp` instruction. For `tail.call` instructions, the return types of the caller and callee must be the same.

Both of these instructions provide an efficient means of transferring control directly from one method to another without overflowing the stack. Unfortunately, neither of these instructions are directly accessible from the set of standard .NET languages (i.e. C#, J#, Visual Basic .NET and C++ with Managed Extensions), so an implementation would have to work directly with MSIL.

Note that there are some Java virtual machines that are capable of automatically optimising tail-recursive method calls away (e.g. the IBM JVM), but there are others that do not (e.g. the Sun HotSpot JVM). Since tail-call optimisation is not part of the standard Java virtual machine specification, it would be unwise to rely upon this non-standard feature for such an essential aspect of the virtual JVM.

6.3 Conclusion

This chapter concludes the thesis with a summary of the main points. Although the main goals set out in Chapter 1 have been accomplished, much work remains to be done. The concrete implementation of some of the semantic-preserving techniques needs to be completed, and further evaluation with full-scale applications in active use would be desirable. There are also many ways in which the work can be improved and extended beyond the scope covered by this thesis. A few of these improvements are discussed in this chapter as future work.

Appendix A

Working with Java bytecode

This appendix introduces various aspects of working with Java at the bytecode level. First, the internal structure of class files is presented. The process by which class files are loaded by the Java runtime to become the classes used by Java programs is then covered. Finally, tools are introduced that simplify the task of manipulating programs at this low level of abstraction.

A.1 Class files

A class file is the binary representation of a single compiled class in a program. It is composed of:

- A header
- A constant pool
- Class information
- Field definitions
- Method definitions
- Attributes

The simple Java program in Figure A.1 is referred to in the next sections as an example.

A.1.1 Constant pool

The constant pool acts as a repository for symbolic information in the class file. The other sections of the class file, and some entries within the constant pool itself, refer to constant pool entries via the index. The constant pool for the example code is shown in Figure A.2.

For example, consider entry 3, which refers to the `println` method called by `main`. The method exists within the class defined by entry 26, which refers to a class with the name in entry 38. The signature of the method is given by entry 27. In the signature definition, the name of the method is given by the string in entry 39, while the type signature is given by the string in entry 40.

```
public class Example {
    private static String name = "Example";

    public static void main(String[] args) {
        try {
            for (int i = 1; i <= 10; i++)
                System.out.println(i);
        } catch (Exception e) {
            System.out.println("Exception raised");
        }
    }
}
```

Figure A.1: A simple Java program

The multiple levels of indirection are there to remove redundancy in a similar manner to that found in relational databases. This can clearly be seen in the definitions of the references to `System.out` (entry 2) and `System.err` (entry 5), which share the same class and type signature entries. However, this is for the sake of space-efficiency rather than for correctness during updates, since class files are regenerated by compilers on updates anyway.

Some points to note are:

- Instance constructors have the name `<init>`
- Class constructors have the name `<clinit>`
- Components of a package name are separated by forward-slashes, as opposed to the dots used in the Java language
- Method type signatures are of the form:
  "(<*parameter types*>)<*return type*>"
- Types are represented in a terse code. For example, `V` represents the type `void`, and `I` represents the type `int`. References to classes are of the form `[L`<*class name*>`;`

A.1.2 Method definitions

Statements in method bodies are represented using Java bytecode, which is a compact representation of Java code — the Java equivalent of machine-code.

The execution model is simple. When a method is called, a stack-frame is created which contains an operand stack and a set of local variable slots. The operand stack is initially empty, and the local variable slots are filled with the value of `this` (if the method is an instance method), followed by the arguments supplied to the method. Instructions take their operands from the operand stack, and push their results back onto it. There are also load and store instructions that move data to and from the local variables.

Instructions are subject to various constraints. For example:

- The depth of the operand stack at each instruction must be constant for all possible executions of the method.

```
1)CONSTANT_Methodref(class_index = 11, name_and_type_index = 23)
2)CONSTANT_Fieldref(class_index = 24, name_and_type_index = 25)
3)CONSTANT_Methodref(class_index = 26, name_and_type_index = 27)
4)CONSTANT_Class(name_index = 28)
5)CONSTANT_Fieldref(class_index = 24, name_and_type_index = 29)
6)CONSTANT_String(string_index = 30)
7)CONSTANT_Methodref(class_index = 26, name_and_type_index = 31)
8)CONSTANT_String(string_index = 32)
9)CONSTANT_Fieldref(class_index = 10, name_and_type_index = 33)
10)CONSTANT_Class(name_index = 32)
11)CONSTANT_Class(name_index = 34)
12)CONSTANT_Utf8("name")
13)CONSTANT_Utf8("Ljava/lang/String;")
14)CONSTANT_Utf8("<init>")
15)CONSTANT_Utf8("()V")
16)CONSTANT_Utf8("Code")
17)CONSTANT_Utf8("LineNumberTable")
18)CONSTANT_Utf8("main")
19)CONSTANT_Utf8("([Ljava/lang/String;)V")
20)CONSTANT_Utf8("<clinit>")
21)CONSTANT_Utf8("SourceFile")
22)CONSTANT_Utf8("Example.java")
23)CONSTANT_NameAndType(name_index = 14, signature_index = 15)
24)CONSTANT_Class(name_index = 35)
25)CONSTANT_NameAndType(name_index = 36, signature_index = 37)
26)CONSTANT_Class(name_index = 38)
27)CONSTANT_NameAndType(name_index = 39, signature_index = 40)
28)CONSTANT_Utf8("java/lang/Exception")
29)CONSTANT_NameAndType(name_index = 41, signature_index = 37)
30)CONSTANT_Utf8("Exception raised")
31)CONSTANT_NameAndType(name_index = 39, signature_index = 42)
32)CONSTANT_Utf8("Example")
33)CONSTANT_NameAndType(name_index = 12, signature_index = 13)
34)CONSTANT_Utf8("java/lang/Object")
35)CONSTANT_Utf8("java/lang/System")
36)CONSTANT_Utf8("out")
37)CONSTANT_Utf8("Ljava/io/PrintStream;")
38)CONSTANT_Utf8("java/io/PrintStream")
39)CONSTANT_Utf8("println")
40)CONSTANT_Utf8("(I)V")
41)CONSTANT_Utf8("err")
42)CONSTANT_Utf8("(Ljava/lang/String;)V")
```

Figure A.2: The constant pool of the class file produced by compiling the Java program in Figure A.1

```
Method void main(java.lang.String[])
   0 iconst_1
   1 istore_1
   2 goto 15
   5 getstatic #2 <Field java.io.PrintStream out>
   8 iload_1
   9 invokevirtual #3 <Method void println(int)>
  12 iinc 1 1
  15 iload_1
  16 bipush 10
  18 if_icmple 5
  21 goto 36
  24 astore_1
  25 getstatic #5 <Field java.io.PrintStream err>
  28 ldc #6 <String "Exception raised">
  30 invokevirtual #7 <Method void println(java.lang.String)>
  33 goto 36
  36 return
Exception table:
   from   to  target type
    0    21    24   <Class java.lang.Exception>
```

Figure A.3: Disassembly of the class file produced by compiling the Java program in Figure A.1

- Local variable slots can be reused. This means than one can fill a local variable slot with data of type `a`, then replace it with data of type `b`. However, it is not valid to put data of type `a` into a slot and extract it as data of type `b`, even if the types are convertible.

Class files that fail these constraints will be caught by the verifier, and most Java Virtual Machines will refuse to run them by default. It is usually possible to override this behaviour if necessary.

A disassembly of the compiled `main` method of the example program using `javap` results in the output shown in Figure A.3. The numbers prefixed by a hash represent references to the constant pool.

Exceptions are handled by a table, which associates the location of an exception handler with a range of locations and the type of exception handled. If an exception of the specified type or a descendant of the type is thrown within the range of an exception handler, then control will be passed to the exception handler immediately. The operand stack is flushed, and the thrown exception pushed onto it.

A.2 Class loaders

Class loaders [52] are objects that are responsible for loading the classes used in a Java program. They are instances of classes that descend from `java.lang.ClassLoader`, which may be extended and instantiated like any other class.

The main task of a class loader is to return a `java.lang.Class` object, given the name of the class to be loaded. Class loaders can either defer the task to other class loaders, or they can load the bytecode for the named class and pass it to one of the predefined `defineClass`

methods defined in `ClassLoader` in order to load the bytecode into the JVM. Once a class is loaded, the class loader is called upon to recursively load all classes that are referenced from the class that has just been loaded. Class loaders should therefore keep a cache of loaded classes for efficiency.

The usual reason for creating new class loaders is to load bytecode from sources other than those specified when the JVM was started — e.g. from the Internet, or bytecode generated on-the-fly.

A.2.1 Class namespaces

In Java, a class is identified by two characteristics — its fully-qualified name, and the class loader used to define it. A class named C defined by class loader CL_1 is regarded as being different from a class named C defined by another class loader CL_2, even if CL_1 and CL_2 are structurally equal.

Note that the defining class loader of a class is not necessarily the same as the class loader used to load the class. This is because class loaders may delegate to other class loaders — the defining class loader is the class loader that ultimately calls the `defineClass` method on the loaded bytecode.

A.2.2 The delegation model

With the exception of the bootstrap class loader, all class loaders must have a parent class loader. The class loaders therefore form a tree that is rooted at the bootstrap class loader.

When a class loader is called upon to load a class, then under the delegation model, it should first pass the request onto its parent class loader. The class loader should only define the requested class itself if no ancestor class loader (since the parent class loader should in turn try its parent first) can load the class.

This provides the property that for any class loader CL and for any class C, then:

$$\begin{aligned} defines(CL, C) \quad \Rightarrow \quad & \\ & \forall cl \in (\{CL\} \cup \{a | ancestor\_of(a, CL)\} \cup \{d | ancestor\_of(CL, d)\}) \cdot \\ & defines(cl, C) \Rightarrow cl = CL \end{aligned} \tag{A.1}$$

where

$$\begin{aligned} ancestor\_of(a, CL) \quad \Rightarrow \quad & parent\_of(a, CL) \vee \\ & (parent\_of(p, CL) \wedge ancestor\_of(a, p)) \end{aligned} \tag{A.2}$$

In other words, if one were to follow the class loader hierarchy from any point back to the root class loader, there can be at most one class loader that defines any one given class.

The delegation model has numerous advantages. Consider an arbitrary class C. All occurrences of C in any class will be compatible with occurrences of C appearing in other classes defined by class loaders further up the class loader hierarchy. This is vital if data

containing instances of C are to be exchanged between classes defined at various levels of the class loader hierarchy.

Another problem that has been encountered is that it is impossible to load classes that contain calls to native code via JNI more than once [1] since the native code cannot tell the two class definitions apart. This makes it vital for classes containing native code to be shared via delegation.

A.2.3 The base class loader hierarchy

When the Sun JVM is first started, three class loaders are present. The boot-strap class loader is responsible for loading the class files for the standard Java library, which are stored in `rt.jar`. The boot-strap class loader is not accessible to Java programs, and is the parent class loader for all class loaders with the parent set to null.

The next class loader is the extensions class loader, which is descended from the boot-strap class loader. In the HotSpot JVM, this is implemented by `sun.misc.Launcher$ExtClassLoader`. This is responsible for loading the extension classes that reside in the `ext` sub-directory of the JRE.

The last class loader is the system class loader, which in turn is descended from the extensions class loader. This is implemented in the HotSpot JVM by `sun.misc.Launcher$AppClassLoader`. This is the class loader used by default to load application classes from the classpath supplied by the user.

A.3 Tools for working with bytecode

Working with raw bytecode is unwieldy due to the numerous elements involved in building a class file. Numerous tools have therefore been developed to ease the process of low-level programming for the Java platform.

A.3.1 Soot

Soot [88, 89] is a framework developed by the Sable Research Group for the static optimisation of Java class files. It takes compiled class files as input, and outputs optimised versions that are ready to be executed by a standard JVM.

In order to perform static optimisations, it is necessary to perform program analyses on the class files. Java bytecode is awkward to analyse in its original form because it is based on a stack architecture, with most instructions receiving implicit arguments from, and/or leaving return values on, the operand stack. It would be necessary to keep track of the possible contents of the operand stack at every instruction in order to analyse such code properly.

The developers of Soot have opted to convert the bytecode into intermediate representations that are more amenable to analysis rather than analysing the bytecode directly. The intermediate representations are Baf, Jimple and Grimp, and more recently, Shimple (an SSA form of Jimple).

Baf is a low-level representation that maps directly onto the bytecode, while Grimp is a form of Jimple that contains aggregated expressions. The main form that is used by this thesis is the Jimple representation.

A.3.1.1 Jimple

Jimple [90] is an intermediate representation of bytecode that explicitly specifies all of its arguments and return values. Variables stored on the operand stack and local variables are treated orthogonally in Jimple, although stack variables are assigned names that are prefixed with a $ symbol. Note that this is for the users benefit, and is not significant in the eventual transformation back to bytecode.

Jimple instructions are simple — that is, each instruction may only perform one operation, much like bytecode instructions. The main difference between them, apart from the explicitness of variable accesses, is that Jimple does not have to generate separate instructions to load variables and constants into other variables.

Jimple also separates the DU-UD webs [68] of variables, such that variables that belong to different webs are always renamed, even if they are physically the same variable within the bytecode. This provides many of the benefits associated with the Single Static Assignment (SSA) [25] form of variables. The main differences occur in loops and branches, since Jimple has no notion of using ϕ-functions to merge variables.

A.3.1.2 Other facilities

Soot also provides many other code-analysis facilities, such as:

- A framework for dataflow analysis
- Static type inference
- Pointer analysis — the Spark [50] pointer analysis framework was recently introduced in version 1.2.4 of Soot.

A.3.2 BCEL

The Byte Code Engineering Library, originally named the JavaClass library, is a set of Java classes for programmatically creating and manipulating Java bytecode. Unlike Soot, it works directly at the bytecode level rather than use intermediate representations. It provides a thin layer of abstraction over the bytecode level, handling administrative details like the addresses of instructions, the construction of the constant-pool, adjustment of the number of local variable slots and maximum stack height etc.

The main advantage of BCEL over Soot is its high speed and low memory overhead, since it provides a *much* thinner interface to the bytecode. This is also its greatest disadvantage, since it lacks the rich set of analysis tools provided by Soot.

Appendix B

Writing distributed programs in Java

This appendix covers the basics of programming with Java RMI, RMI-IIOP, and Enterprise JavaBeans. A closer look is taken at the JBoss server, which provides an unusual implementation of Enterprise JavaBeans.

B.1 Java RMI

Remote Method Invocation is the native Java API for creating distributed programs. Unlike CORBA, RMI was designed for use with the Java platform only. One major advantage of this is that this eliminates the need for abstraction mechanisms within the protocol to deal with architectural differences such as the endianess of the communicating hosts, making for a cleaner and simpler programming interface when compared to CORBA[1]. RMI makes extensive use of the principle of polymorphism from the Object-Oriented Programming paradigm to make the calling of methods on remote objects nearly transparent from a syntactic point of view.

Within programs using RMI, there need not be a distinct separation of roles in terms of server and client — it is quite possible for two communicating programs to simultaneously take on both roles with respect to each other. The terms 'server' and 'client' will henceforth be referring to the role played by a program during a specific remote transaction. The 'server' will refer to the party that contains the remotely-accessible object, whilst the 'client' will refer to the party that accesses that object from a potentially different host.

B.1.1 Programming with RMI

A programmer wishing to create an object with methods that may be accessed remotely must first declare a *remote interface*, which specifies the remotely accessible methods. Object fields and static methods cannot be accessed directly via RMI. By definition, a remote interface must inherit from the `Remote` (directly or indirectly) interface, and all of its methods must throw `RemoteException` or one of its super-classes in case a network error occurs.

[1] It could be argued that platform-abstraction *does* take place when using RMI, but it is situated at the level of the Java platform itself, rather than at the API level

The actual implementation of the remote object must then implement the remote interface just created, and also inherit from `RemoteObject` at some point. After the new class has been compiled, a stub compiler is then used to produce a stub class for the remote object. The generated stub inherits from the same remote interface as the object implementation.

In order for the new object to be remotely accessed, it must first be *exported*. This can be done explicitly by calling the `exportObject` methods of the `UnicastRemoteObject` or `Activatable` classes, or implicitly by inheriting from one of these classes, which export the object within the constructor.

When a client wishes to access a remote object, it must first obtain a stub that points to that object, and invoke calls directed to the remote object on the stub instead. Stubs acts as proxies, forwarding the method arguments onto the remote object along and relaying the return value back from the server. This process is known as *marshalling*. The protocol that the stub uses to communicate with the remote server is known as the Java Remote Method Protocol (JRMP).

Parameters and return values of RMI calls are generally copied across using the serialisation mechanism. The only exception is when one attempts to pass a remote object — in this case, the reference to the actual remote object is replaced by a stub that is associated with it.

Stub classes typically reside on the same host as the server. Clients obtain stubs to remote objects in two ways — as the return value of another remote call, or via a name registry. Name registries are simple remote servers that use a well-known stub and address. Servers can associate stubs with a name on the registry using the `bind` methods of the `java.rmi.Naming` class, while clients can fetch stubs by name using the `lookup` method.

RMI calls are synchronous — that is, when an RMI call is made by a client, it will wait until a result is returned by the server before continuing execution. However, it is possible to emulate the effect of asynchronous RMI calls by spawning off new threads to perform the RMI call. The new thread will be suspended during the call, but the parent thread may continue unimpeded.

B.1.2 Object serialisation

Object serialisation is the process by which Java objects are transformed into a stream of bytes so that they may be stored on permanent storage, transported from machine to machine etc. The byte stream may be reconstituted back into Java objects again at a later time by the deserialisation process.

B.1.2.1 Automatic object serialisation

The easiest way of making a class support serialisation is to make the class implement the `java.io.Serializable` interface. This interface is a marker interface, and contains no method declarations. The code contained in the `java.io.ObjectOutputStream` class will then be able to serialise the class by using runtime inspection.

When an object is serialised, serialisation takes place starting from the topmost class `Object`, and proceeds down the class hierarchy. For every class in the hierarchy, the runtime uses the reflection API to inspect the instance fields belonging to that class, and recursively serialises them one after another. Deserialisation occurs in a similar fashion — by creating

an instance of the serialised class then filling the fields in, starting from the topmost class in the class hierarchy.

B.1.2.2 Controlling object serialisation

The simplest form of control that may be gained over the serialisation process is by declaring fields as `transient`. Transient fields are not included in the automatic serialisation process, and will be skipped over.

More control over the serialisation process can be gained by implementing `writeObject` and `readObject` methods in the class to be serialised. These methods enable the implementer of the class to control how instances of that class are serialised/deserialised by using the stream that is supplied as an argument to these methods. If these methods are present in a class, then they are called instead of performing automatic serialisation. Automatic serialisation may still be performed by calling `defaultWriteObject` on the stream. Objects are still serialised and deserialised recursively starting from the top of the class hierarchy.

Even more control can be gained over the process by making the classes implement `the Externalizable` interface rather than the `Serializable` interface. The `Externalizable` interface declares two methods `writeExternal` and `readExternal` which must be implemented. These two methods perform a similar task to `writeObject` and `readObject` in explicitly stating what to read or write to the I/O streams. The main difference is that externalisable classes do not recursively traverse the class hierarchy. This means that the `writeExternal`/`readExternal` methods are responsible for serialising the contents of parent classes as well.

B.2 RMI-IIOP

Java Remote Method Invocation over the Internet Inter-ORB Protocol (Java RMI-IIOP) is a variation of standard Java RMI (or more accurately, Java RMI-JRMP) that uses the CORBA IIOP protocol to communicate between hosts rather than the Java-specific JRMP. This is especially useful when dealing with legacy code because it allows Java clients/servers written using the RMI programming model to interact with CORBA servers/clients that may be written using other languages without needing to write or compile any IDL.

Minor changes are required in order to transform a program from using RMI-JRMP to RMI-IIOP:

- Stubs must be generated with the `-iiop` flag
- Common Object Services Naming (COS Naming) services must be used in place of the RMI registry
- `javax.rmi.PortableRemoteObject` is used in place of `UnicastRemoteObject`.
- Remote interface type-casts must be replaced with the CORBA-compliant `narrow` method of the `PortableRemoteObject` class.

There are some minor limitations on what can be done with RMI-IIOP compared to RMI-JRMP due to the restrictions of the CORBA standard. These are detailed in the RMI-IIOP specification.

B.3 Enterprise JavaBeans

Enterprise JavaBeans (EJB) [65] forms a major part of the Java Enterprise Edition (J2EE) specification, and is the standard for distributed component programming in Java. The EJB programming model is similar in most respects to the RMI programming model, with stricter discipline imposed and the addition of 'value-added services'.

B.3.1 Beans

The remote server-side components in the EJB architecture are known as 'beans'. Beans are similar in many ways to RMI servers, in that they implement a set of methods that may be called remotely via a *remote interface*. Since version 2.0 of the EJB specification, Beans may also expose a set of methods via a *local interface*, which are methods that may be called directly by other beans residing within the same container for efficiency.

Beans differ from RMI servers in two main respects. Firstly, a bean class is separated from its remote and local interfaces, in that its class definition does not implement these interfaces. This is because clients do not communicate directly with the beans, but with the container that contains the beans. Bean classes must implement from the `EntityBean`, the `SessionBean` or the `MessageBean` interfaces, depending on their type. These interfaces declare callback methods, which are called by the container when a life-cycle change occurs (e.g. if the beans are about to be stored to a hard disk).

Secondly, each EJB supports at least two remote interfaces — the normal interface for invoking business methods on the EJB, which must extend the `EJBObject` interface, and a 'home' interface, which must extend the `EJBHome` interface. The home interface is a formalisation of the 'factory pattern' in that it permits remote clients to manipulate the life cycle of EJBs in the remote server. Clients may use the home interface to request the EJB container to create new instances of an EJB, or to find an existing EJB instance using some key.

There are three main types of bean — entity beans, session beans and message beans.

B.3.1.1 Entity Beans

Entity beans are used to access data stored on the server, which is typically stored in the form of a database. Entity beans are created or located via the home interface, using the primary key(s) as an identifier. An entity bean represents a record in the underlying database, such that calling `get` and `set` methods on the bean have the effect of retrieving and setting data in the database. There are two main types of entity bean:

- Entity beans with Container Managed Persistence (CMP) have the container manage the link with the underlying database. In the newer versions of EJB (version 2.0 and above), all one has to do is to define abstract `get<field>` and `set<field>` methods in the bean class, and define the data relationships in the XML descriptor for the bean. The container will automatically generate bodies for the abstract access methods. It is also possible to handle relationships between tables, such that one can access linked entries in related tables via a single entity bean.
- Entity beans with Bean Managed Persistence (BMP) leave the task of managing the connection to the underlying database to the creator of the entity bean.

B.3.1.2 Session Beans

The methods provided by session beans are usually called upon to perform server-side tasks, often on data provided by entity beans. Session beans are also sub-divided into two categories:

- Stateless session beans do not maintain a conversational state with the client. A single instance of a stateless session bean may service requests from multiple clients, and furthermore, a client accessing the bean multiple times might not even access the same instance on every call. In other words, all instances of a particular class of stateless session bean are interchangeable. There is nothing actually preventing one from inserting instance fields into a stateless session bean, but these will be of limited use since one cannot predict which bean instance will be called by which client.
- Stateful session beans maintain a conversational state with the client, such that each client is serviced by a dedicated instance of the bean (conceptually at least). This means that the effects of any remote calls on the bean will persist from call to call, and such calls are guaranteed to be from the same client.

B.3.1.3 Message Beans

Message beans behave like stateless session beans in most respects, but its methods are executed in response to messages sent via the Java Message Service (JMS) [67] rather than RMI calls. It may also send messages in response.

B.3.2 Containers

Clients do not interact with beans directly, but with the *container* that contains the beans. Containers are responsible for the added functionality provided by EJB over standard RMI. Containers manage the life-cycle of beans by creating, destroying, and *passivating* them (i.e. swapping them out onto persistent storage to free up memory), possibly in response to a call to the home interface of a bean.

Containers also handle calls to business methods, and have full control over how they are handled. For example:

- Calls to the accessor methods of entity beans with container-managed persistence will be handled directly by the container, which will fetch the data directly from the database. The call will not actually reach the bean instance.
- If a method of a stateless session bean is called, then the container tries to fetch a free instance of the bean from a pool of beans. If none are available, then the container instantiates a new one. The corresponding method on the bean is then called by the container, and the result is passed back to the client. When finished, the bean instance is returned to the bean pool.

B.3.3 JBoss

JBoss [85] is currently the most popular open-source implementation of the Java Enterprise Edition (J2EE) specification, and includes support for Enterprise JavaBeans. The JBoss

server is unusual in many ways. When a client requests a stub from JBoss, a dynamic proxy is returned instead of a standard RMI stub.

B.3.3.1 Dynamic proxies

Dynamic proxies have been a part of the standard Java library since Java 1.3. Dynamic proxy classes are generated at runtime in order to implement a Java interface. The constructor of all dynamic proxies accept an object that implements the `InvocationHandler` interface as an argument. The `InvocationHandler` interface is declared as follows:

```
public interface InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable;
}
```

When one of the methods implemented by the dynamic proxy class is called on an instance of the proxy, the proxy delegates the call to the `invoke` method of the `InvocationHandler` object supplied at construction time. The `invoke` method is supplied with a reference to the proxy (since multiple proxies can share the same `InvocationHandler`), a `Method` object identifying the method that was called, and an array of `Objects` representing the object. The `invoke` method may then make a decision regarding what to do next based on the supplied arguments.

B.3.3.2 Interceptors

All method calls made to JBoss dynamic proxies are passed through a chain of interceptors. An interceptor is an object that provides some aspect of the added functionality of EJBs over standard RMI. An interceptor accepts an instance of `Invocation` which contains the `Method` object identifying the called method, the call arguments, and a hash-map containing data generated by other interceptors. An interceptor may perform some action, possibly modifying the `Invocation` object in the process, and then either call the next interceptor in the chain, or return. Interceptors are able to perform actions when the next interceptor in the chain returns if necessary.

For example, consider Figure B.1, which shows the standard configuration for handling stateless session beans. When a method on the interface containing the business methods of a stateless session bean is invoked, it is passed first to `StatelessSessionInterceptor`. This interceptor checks the identity of the method called, and if it is a simple method such as `toString` or `equals`, then the interceptor handles these calls and returns immediately, without any remote communication occurring. If `StatelessSessionInterceptor` cannot handle the call, then the next interceptors are called. `SecurityInterceptor` and `TransactionInterceptor` associate the current security and transactional contexts with the call. `InvokerInterceptor` is responsible for sending the `Invocation` object across the network to the server via RMI.

There is a similar chain of interceptors on the server-side, which ultimately leads to the invocation of the implementation of the business method inside the session bean. The result of the call filters back down the chain of interceptors to be delivered to the client.

```
<jboss>
 ...
 <invoker-proxy-bindings>
  ...
  <invoker-proxy-binding>
   <name>stateless-rmi-invoker</name>
   <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
   <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
   <proxy-factory-config>
    <client-interceptors>
     <home>
      <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
     </home>
     <bean>
      <interceptor>
        org.jboss.proxy.ejb.StatelessSessionInterceptor
      </interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
     </bean>
    </client-interceptors>
   </proxy-factory-config>
  </invoker-proxy-binding>
  ...
 </invoker-proxy-bindings>
</jboss>
```

Figure B.1: A portion of the XML configuration file `standardjboss.xml` from JBoss 3.2.1 that describes the interceptors that should be applied for stateless session beans

There are numerous advantages to using interceptors. It makes JBoss highly modular and reconfigurable, and using dynamic proxies means that JBoss rarely needs to manually generate new classes at runtime, which rival J2EE servers such as JOnAS must do every time a new bean is deployed. However, JBoss does pays a price in terms of performance [17] due to the extensive use of reflection throughout the application server, although this is often overshadowed by network overheads.

Bibliography

[1] Java bug parade — Java 1.2 class loader fails to load multiple classloaders with native lib. Bug ID: 4225434. Available at http://developer.java.sun.com/developer/bugParade/bugs/4225434.html.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.

[3] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, June 1997.

[4] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. Implementing Jalapẽno in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324. ACM Press, November 1999.

[5] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[6] K. Arnold and J. Gosling. *The Java Programming Language — Second Edition*. Addison-Wesley, 1998.

[7] The O'Reilly Java Authors. *Java Enterprise Best Practices*. O'Reilly, 2002.

[8] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[10] Olav Beckmann and Paul H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In David O'Hallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *Lecture Notes in Computer Science*, pages 123–138. Springer-Verlag, 1998.

[11] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the ninth annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 341–354. ACM Press, 1994.

[12] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 183–200. ACM Press, October 1998.

[13] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ specification, 1998. At http://www.cis.unisa.edu.au/~pizza/gj/Documents/.

[14] Douglas J. Brear, Thibault Weise, Tim Wiffen, Kwok Cheung Yeung, Sarah A. M. Bennett, and Paul H. J. Kelly. Search strategies for Java bottleneck location by dynamic instrumentation. *IEE Proceedings — Software*, 150(04):235–241, August 2003.

[15] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

[16] Derek L. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[17] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246–261. ACM Press, November 2002.

[18] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies. Mojo: A dynamic optimization system. In *Third ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[19] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19. ACM Press, November 1999.

[20] R. Christ, S. L. Halter, K. Lynne, S. Meizer, S. J. Munroe, and M. Pasch. SanFrancisco performance: A case study in performance of large-scale Java applications. *IBM Systems Journal*, 39(1):4–20, 2000.

[21] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, October 1998.

[22] Sam Clegg. Reducing network overheads of .NET remoting through runtime call aggregation. Master's thesis, Imperial College of Science, Technology and Medicine, 2003.

[23] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation International Seminar*, volume 1110 of *Lecture Notes in Computer Science*, pages 54–72. Springer-Verlag, February 1996.

[24] Xerox Corporation. AspectJ programming guide. At http://www.eclipse.org/aspectj/.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark 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.

[26] Markus Dahm. Byte code engineering library manual. At http://jakarta.apache.org/bcel/manual.html.

[27] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer-Verlag, August 1995.

[28] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Magaging web server performance with AutoTune agents. *IBM Systems Journal*, 42(1):136–149, 2003.

[29] John Eberhard and Anand Tripathi. Efficient object caching for distributed Java RMI applications. In R. Guerraoui, editor, *Proceedings of Middleware 2001*, volume 2218 of *Lecture Notes in Computer Science*, pages 15–35. Springer-Verlag, November 2001.

[30] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56. ACM Press, June 1997.

[31] Jacques Ferber. *Multi-agent Systems: An Introduction to Distributed Artificial Intelligence.* Addison Wesley, 1999.

[32] David Flanagan. *Java Examples in a Nutshell.* O'Reilly UK, 2000.

[33] E. Gagnon and L. Hendren. Intra-procedural inference of static types for Java bytecode. Technical Report 1999-1, McGill University, March 1999.

[34] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode.* PhD thesis, McGill University, Montreal, December 2002.

[35] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133. ACM Press, January 1998.

[36] Aniruddha S. Gokhale and Douglas C. Schmidt. Principles for optimizing CORBA internet inter-ORB protocol performance. In *31th Hawaii International Conference on System Sciences, VOL. 7*, pages 376–385. IEEE Computer Society, January 1998.

[37] M. Golm and J. Kleinoder. metaXa and the future of reflection. In Jean-Charles Fabre and Shigeru Chiba, editors, *Proceedings of Workshop on Reflective Programming in C++ and Java. UTCCP Report 98-4, ISSN 1344-3135*, Center for Computational Physics, University of Tsukuba, Japan, October 1998.

[38] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification — Second Edition*. Addison-Wesley, 2000.

[39] Object Management Group. The Common Object Request Broker Architecture: Core specification. Version 3.0.2, December 2002. Available at http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[40] Manish Gupta, Jong-Deok Choi, and Michael Hind. Optimizing Java programs in the presence of exceptions. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 422–446. Springer-Verlag, June 2000.

[41] M. Horstmann and M. Kirtland. DCOM architecture, 1997. Available at http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp.

[42] Frequently asked questions about the Java HotSpot virtual machine. Available at http://java.sun.com/docs/hotspot/PerformanceFAQ.html.

[43] Java platform debugger architecture. At http://java.sun.com/j2se/1.4.2/docs/guide/jpda/.

[44] JProbe. Available at http://www.quest.com/jprobe/.

[45] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[46] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java remote method invocation (RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 19–36, 1998.

[47] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 283–294. ACM Press, January 2002.

[48] Rosanna Lee and Scott Seligman. *JNDI Tutorial and Reference Guide (The Java Series)*. Addison Wesley, 2000.

[49] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 220–231. ACM Press, June 2003.

[50] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In Görel Hedin, editor, *12th International Conference on Compiler Construction (CC 2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, April 2003.

[51] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification (The Java Series)*. Addison Wesley, 1999.

[52] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44. ACM Press, October 1998.

[53] Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press International, 2002.

[54] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification — Second Edition*. Addison-Wesley, Reading, MA, USA, 1999.

[55] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *International Workshop on Distributed Object Management*, pages 79–91. Morgan Kaufmann, August 1992.

[56] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 260–267. ACM Press, June 1988.

[57] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, pages 747–775, November 2001.

[58] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aike Plaat. An efficient implementation of Java's Remote Method Invocation. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 173–182. ACM Press, May 1999.

[59] Floyd Marinescu. *EJB Design Patterns*. John Wiley and Sons, 2002.

[60] Eduardo R. B. Marques. A study on the optimisation of Java RMI programs. Master's thesis, Imperial College of Science, Technology and Medicine, 1998.

[61] Scott McLean, James Naftel, and Kim Williams. *Microsoft .NET Remoting*. Microsoft Press International, September 2002.

[62] Gary Meehan and Mike Joy. Compiling lazy functional programs to Java bytecode. *Software — Practice and Experience*, 29(7):617–645, 1999.

[63] Sun Microsystems. RMI specification. Available at http://java.sun.com/products/jdk/rmi/.

[64] Sun Microsystems. The Java HotSpot performance engine architecture. White paper, April 1999. Available from http://java.sun.com/products/hotspot/whitepaper.html.

[65] Sun Microsystems. Enterprise JavaBeans specification 2.0 Final Release 2, August 2001. Available from http://java.sun.com/products/ejb/docs.html.

[66] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):38–46, 1995.

[67] Richard Monson-Haefel and David Chappell. *Java Message Service*. Addison Wesley, 2000.

[68] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[69] Christian Nester, Michael Phillippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 152–159. ACM Press, June 1999.

[70] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, December 1999.

[71] Alexandre Oliva and Luiz Eduardo Buzato. The implementation of Guaraná on Java. Technical Report IC-98-32, University of Campinas, September 1998.

[72] OpenJGraph. Available at http://openjgraph.sourceforge.net/.

[73] Optimizeit. Available at http://www.borland.com/optimizeit/.

[74] J. Steven Perry. *Java Management Extensions.* O'Reilly, 2002.

[75] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[76] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[77] Jeff Prosise. *Programming Microsoft .NET (core reference).* Microsoft Press International, 2002.

[78] Riccardo Pucella. Towards a formalization for COM part I: The primitive calculus. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–342. ACM Press, November 2002.

[79] Rajeev R. Raje, Joseph I. Williams, and Michael Boyles. Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[80] Alexandru Salcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 12–23. ACM Press, June 2001.

[81] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* John Wiley & Sons Inc., 1995.

[82] Ulrik Schultz. Partial evaluation for class-based object-oriented languages. In O. Danvy and A. Filinski, editors, *Program as Data Objects: International Conference on the Theory and Application of Cryptographic Techniques*, volume 2053 of *Lecture Notes in Computer Science*, pages 173–197. Springer-Verlag, May 2001.

[83] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390. Springer-Verlag, June 1999.

[84] Shamik Sharma, Anurag Acharya, and Joel Saltz. Deferred data-flow analysis. Technical Report TRCS98-38, University of California, Santa Barbara, 30 1998.

[85] Scott Stark, Marc Fleury, and The JBoss Group. *JBoss Administration and Development, Second Edition*. Sams Publishing, 2002.

[86] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 264–280. ACM Press, 2000.

[87] Antero Taivalsaari. Implementing a Java Virtual Machine in the Java programming language. Technical Report TR-98-64, Sun Microsystems Laboratories, March 1998.

[88] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135. ACM Press, November 1999.

[89] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In D. A. Watt, editor, *Proceedings of the 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer-Verlag, March 2000.

[90] Raja Vallee-Rai and Laurie Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical Report 1998-4, McGill University, July 1998.

[91] Ronald Veldema and Michael Philippsen. Compiler optimized remote method invocation. In *Proceedings of the 5th IEEE Conference on Cluster Computing*, December 2003.

[92] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 34 of *ACM SIGPLAN Notices*, pages 187–206. ACM Press, November 1999.

[93] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jaques Cohen, editors, *Proceedings of the International Symposium/Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer-Verlag, September 1992.

[94] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, June 1995.

[95] Kwok Cheung Yeung and Paul H. J. Kelly. Automated optimisation of distributed Java programs across network boundaries. In *10th Workshop on Compilers for Parallel Computers, CPC 2003, Amsterdam, The Netherlands*, January 2003.

[96] Kwok Cheung Yeung and Paul H. J. Kelly. Optimising Java RMI programs by communication restructuring. In Markus Endler and Douglas C. Schmidt, editors, *Proceedings of Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, June 2003.

[97] Kwok Cheung Yeung, Paul H. J. Kelly, and Sarah Bennett. Dynamic instrumentation for Java using a virtual JVM. In *Workshop on Performance Analysis and Distributed Computing*, 2002.

[98] Quinton Y. Zondervan. Increasing cross-domain call batching using promises and batched control structures. Master's thesis, Massachusetts Institute of Technology, 1995.