Proxy Compilation for Java via a Code Migration Technique

Jian Li

Software Systems Group
School of Informatics
University of Sussex

Submitted for the degree of Doctor of Philosophy.

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:

Abstract

There is an increasing trend that intermediate representations (IRs) are used to deliver programs in more and more languages, such as Java. Although Java can provide many advantages, including a wider portability and better optimisation opportunities on execution, it introduces extra overhead by requiring an IR translation for the program execution. For maximum execution performance, an optimising compiler is placed in the runtime to selectively optimise code regions regarded as "hotspots". This common approach has been effectively deployed in many implementation of programming languages. However, the computational resources demanded by this approach made it less efficient, or even difficult to deploy directly in a resource-constrained environment. One implementation approach is to use a remote compilation technique to support compilation during the execution. The work presented in this dissertation supports the thesis that execution performance can be improved by the use of efficient optimising compilation by using a proxy dynamic optimising compiler.

After surveying various approaches to the design and implementation of remote compilation, a proxy compilation system called Apus is defined. To demonstrate the effectiveness of using a dynamic optimising compiler as a proxy compiler, a complete proxy compilation system is written based on a research-oriented Java Virtual Machine (JVM). The proxy compilation system is discussed in detail, showing how to deliver remote binaries and manage a cache of binaries by using a code migration approach. The proxy compilation client shows how the proxy compilation service is integrated with the selective optimisation system to maximise execution performance. The results of empirical measurements of the system are given, showing the efficiency of code optimisation from either the proxy compilation service and a local binary cache.

The conclusion of this work is that Java execution performance can be improved by efficient optimising compilation with a proxy compilation service by using a code migration technique.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Des Watson for the continued support of my D.Phil study and research,. Throughout the entire research program, he provided encouragement, sound advice and a lots of good ideas. I would not imagine finishing my research without the invalable guidance from him.

Besides my supervisor, I would like to thank the rest of my thesis committee: Dr. Ian Wakeman and Dr. Dan Chalmers for their encouragement, insightful comments, and hard questions.

I would also like to extend my thanks to other members of the Software Systems Group, in particular: Dr. Jon Robinson, Stephen Naicken, Anirban Basu, James Stanier, Ryan Worsley, Simon Fleming, Yasir Malkani, Roya Feizy, Lachhman Dhomeja; and to members of the Theory Group, in particular: Dr. Berhard Reus; and also friends in University of Sussex: Dr. Tom Akehurst and Karen Schaller, for their kind assistance with thesis proof-reading and giving advice.

I wish to thank my extended family, particularly my parents Li Zhiqiang and Tan Xiaomei, for providing a loving environment and encouragement for me for the entire time. To them I dedicate this thesis.

Last but not least, many thanks to the (erstwhile Department and now) School of Informatics with their support and all the computing facilities and resources that I have been provided with.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The eternal mystery of the world is its comprehensibility.

Albert Einstein (1879–1955)

This chapter gives a broad overview of the research area corresponding to the work described in this dissertation, explaining the reason why improvements to the state of the art are necessary. The approach used to undertake the research is also given. Finally, the last section of this chapter provides a outline of the structure of the dissertation.

1.1 Overview

The work presented in this dissertation is concerned with the topic "Proxy Compilation for Java via a Code Migration Technique". It introduces a novel approach to implement a dynamic programming language, particularly on resource-constrained embedded systems.

While many mainstream programming languages use executable machine code as the compilation output, the popular Java programming language promotes a return to research on the execution intermediate representation (IR). In contrast to exe-

cutable machine code, an IR can provide better portability, and carry more semantic information to provide further opportunities for optimization. For example, Java offers no additional maintenance cost in executing applications on multiple platforms. Platform independent features are useful for deploying applications in heterogeneous environments without extra maintenance, especially in embedded systems, where a variety of hardware and system specifications are available.

Despite the benefits offered by the use of IR in languages such as Java, the IR introduces an extra layer of translation from IR to machine code in the execution. Much research has been carried out to fully exploit the optimization potential in the IR translation process for a better execution speed. However, the cost of optimizing compilation of the IR in a resource-constrained environment is often considered to outweigh its benefits. In the background chapter, Section 2.1 studies the execution model for the IR in Java in more detail.

The work presented in this dissertation studies the cost and benefits of an approach to migrate the expensive code optimization process from the host machine to a networked server, in order to reduce the optimizing compilation cost on the host machine.

1.2 Motivation

The original inspiration for this research project comes from work by Newsome and Watson [66]: the MoJo research project. In their work, MoJo is an implementation of the Java virtual machine targeted for resource-constrained environments by using ahead-of-time compilation. One goal of the implementation design was to provide inexpensive support for dynamic class loading via a proxy compilation technique. However, the implementation of MoJo requires the compilation to constantly handle all dynamic loading activities, which is a problem that led to the research presented in this dissertation.

There are several benefits of using remote compilation techniques for execution. First, it provides a dynamic optimizing compilation service to language execution, which overcomes the limitation on static compilation for languages that support dynamic loading. Second, it distributes computation-intensive optimizing compilation to networked servers, allowing lower demands on the thin client. Third, the

distributed compilation model can be deployed as a service in a scalable network computing envirnoment. For example, there is an increasing trend that Cloud Computing is used to provide scalable and virtualize computing resource as a service while hiding the details of underlying technology infrastructures. A dynamically scalable compilation service can be provided efficiently and widely over the Internet by the Cloud.

A survey of the literature related to the implementation of remote compilation shows that the remote compilation service approach has already been used in language implementations, including Java. However, these approaches mostly use a static compiler to provide a remote compilation service as a replacement for the local optimizing compiler, despite the fact that much progress has been made with dynamic optimizing compiler to improve compilation performance and optimization, which are identified as important issues. Furthermore, projects, such as MoJo, were only implemented as a proof of concept prototype such that only a limited set of Java features are supported. As a consequence, a broader evaluation of such systems is difficult.

Given such a situation, the goal of this research is clear: to demonstrate that an infrastructure to provide proxy compilation using a dynamic optimizing compiler can be efficiently applied on a full-scale Java Virtual Machine (JVM) implementation. This proxy compilation infrastructure would further provide potential opportunities to explore speculative optimizations and various characteristics of proxy compilation for Java in later research.

To achieve this goal, we have to:

- design and implement a proxy compilation infrastructure based on the use of a dynamic optimizing compiler,
- design and implement a client to utilize the proxy compilation service based on an existing JVM implementation,
- test and evaluate the proxy compilation system to determine the efficiency of the implementation.

Using this approach it is possible to examine experimentally to what extent proxy compilation is able to support the traditional fully functional JVM without causing implementation difficulties. By modifying a well-established JVM to deploy effective remote compilation, it offers the opportunity to explore the proxy compilation

scheme.

1.3 Terminology

In order to reduce the confusion arising from different terminology, this section gives definitions of the terminology used throughout this dissertation. Different researchers use a variety of terminology to define the same concept in different contexts. It is the same here in programming languages.

- *Compilation* is a general term used in the context of this thesis to address the concept of the process of translation from IR into machine code, but no technique is specified. In this dissertation, image loading and proxy compilation are among these techniques fitting into the category of this terminology, but are not exclusive.
- *Granularity* implies compilation granularity in this thesis. In compilation, it means the size of the compilation unit, e.g. methods, classes or entire program. "Fine-grained" means the compilation units are relatively low-level and small in term of code size, e.g. a basic-block.
- *Function and Method*: *a function* is used to refer to a subroutine to perform a specific task by lower-level programming languages. In contrast, a *method* is an object oriented concept of a member of a class referring to a subroutine.
- *Interpretation* is a general term implying indirect execution in contrast to the execution of native code, which is called "direct execution". In the context of this thesis, interpretation implies the use of an interpreter program that reads intermediate representation (IR) instructions and simulates their execution. There is no direct translation into target machine code. The term *interpretation* does not specifically imply that the translation procedure is taken instruction by instruction.
- *Image* is a file format used to store a collection of compiled routines that can be used in different execution instances. It shares various similarities to object files that all contain code and data integrated as a program, but an image does not necessarily map to a source file.
- *Linker* is a program routine that takes one or more objects generated by compilers and assembles them together into a single executable program.

1.4 Dissertation Structure

Chapter 2 presents background information related to programming language execution, particularly focusing on IR execution. It also presents a detailed survey of existing approaches to improve performance of IR execution while balancing the cost of dynamic translation.

Chapter 3 gives an overview description of the Apus[1] proxy compilation system. Many important issues related to applying a proxy compilation service are discussed. In Chapter 4, the design of a code migration framework that is used to provide code relocation for binaries generated by a dynamic optimizing compiler is presented, and issues of implementation are discussed. The proxy compilation protocol design adopted follows in Chapter 5, and the implementation issues of the compilation server are discussed later in the chapter.

A detailed experimental study of the Apus proxy compilation system is presented and summarized in Chapter 6. Last but not least, Chapter 7 draws a overall conclusion of this study on applying code migration techniques to provide proxy compilation to existing JVM implementations. Further possible extensions of this work and alternative design decisions are also discussed in the chapter.

[1] Our proxy compilation system is named after Apus which is a faint constellation in the southern sky. Its name means "no feet" in Greek, and it represents a bird of paradise (which were once believed to lack feet). It is similar to our solution, which removes dynamic compilation out of the host machine, making the host machine faster.

CHAPTER 2

Background

We know very little, and yet it is astonishing that we know so much, and still more astonishing that so little knowledge can give us so much power.

BERTRAND RUSSELL (1872–1970)

2.1 Program Execution

A high level programming language can be executed on a hardware architecture using primarily three different approaches. The target platform could choose to interpret the language directly or it could be compiled into the native instructions for the target platform before execution. Otherwise it could also be compiled into an intermediate representation which is then interpreted by the target platform. According to Rau [83], the level of program representation can be divided into three broad categories:

- *High-Level Representation,* (HLR) is one written in a high level language, such as Fortran, C, perl etc.
- *Directly Interpretable Representation,* (DIR) is an intermediate representation that is not bound to any specified architecture and carries semantic informa-

tion, such as JVM bytecode, LLVM bitcode.

- *Directly Executable Representation,* (DER) are native machine instructions format bound to a specified architecture, for example x86, SPARC, ARM.

HLR holds the original information on the program that later would be interpreted [83]. It is used mainly to improve the readability and writalibility for programmers but it is inefficient for execution on machines, since optimisation from HLR to executable DIR crosses a wide gap. However, the direct interpretation of a HLR provides better flexibility for users by avoiding compilation before execution. It is therefore used in a number of languages which are designed for small programs, such as scripting languages. Figure 2.1 illustrates the execution model for HLR.

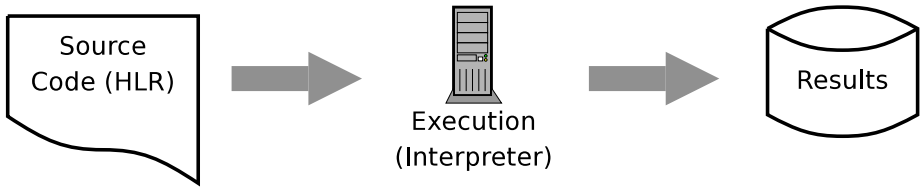


Figure 2.1: Execution Model for a HLR

Most compilers directly produce DER code. Execution and production of DER code is demonstrated in Figure 2.2. It is considered to be more efficient than other representations because there is no interpreting and runtime overhead and further inter-procedural optimisation [64] can be applied. However, there is a lack of portability of the DER code, since it is compiled for the specified hardware architecture. Furthermore, due to the fact that a DER contains little semantic information of the program, there is little space for the optimisation based the runtime profiling.

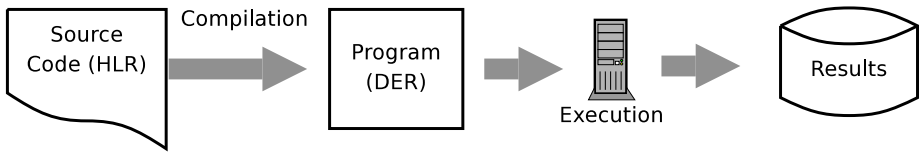


Figure 2.2: Execution Model for a DER

DIR lies in the middle of a HLR and DER as illustrated in Figure 2.3. It provides a degree of portability and does not impose a significant overhead of dynamic compilation at runtime compared to a HLR. A DIR is commonly produced by compilers which serve similarly as the front-end to build an representation of programs

including the intermediate representation, symbol tables and other data structures for associated information, such as debugging information. The DIR therefore carries more semantic information which allows further opportunities for optimisation during runtime. It has gained more attention in recent years in many projects [50, 58, 101]. Although DIR requires an extra interpretation during execution, Hoevel [41] also derived this approach under the condition of dynamic optimisation. DIR is "ideally" superior to HLR and DER in term of space and speed.

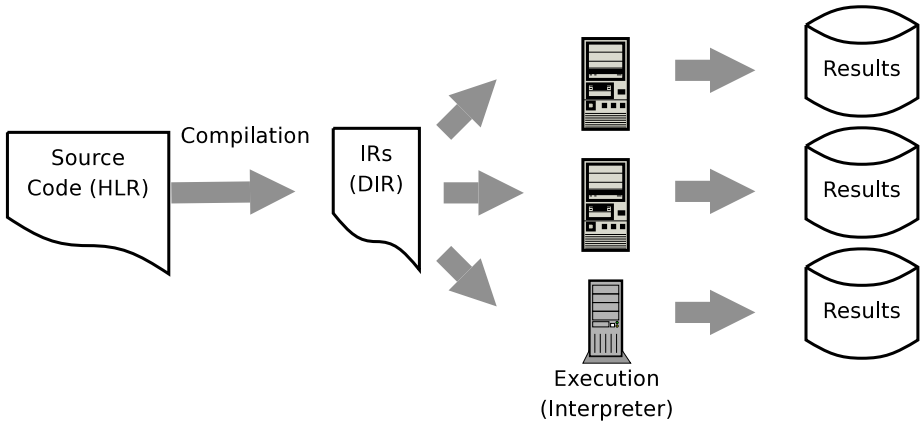


Figure 2.3: Execution Model for a DIR

2.2 The Java Programming Language

Java is a general-purpose dynamic programming language, which is widely used (4.5 billion devices [63]) ranging from high-powered servers with 64-bit processors to 8-bit embedded systems with limited memory. It is the first commercially successful programming language that uses an intermediate representation as the binary format with a virtual machine execution model. As it is widely successful, many of its characteristics, including native support for automatic memory management, dynamic loading and exception handling etc., became a milestone of programming language design and influenced other modern languages, such as C# [40], Python [33] and D [60]. In contrast to traditional programming languages where one-time translation is used to produce executable machine instructions, Java programs are first compiled into an intermediate representation format, referred to as bytecode for the Java Vir-

tual Machine (JVM), which provides an universal machine independent execution environment and runtime service.

As we have discussed above, indirect execution through Java bytecode creates extra overhead in performance and resource consumption on the host machine, compared to programming languages using one-time translation. Wirjawan et al. [110] classify the costs of bytecode execution into four main categories:

- *Interpretation overhead.* Bytecode needs to be fetched and translated during the actual execution of bytecode. The fetch and decode step does not contribute to effective CPU utilization.
- *Stack overhead.* JVM is a stack based virtual machine while many widely used machine architectures are register-oriented, although the number of available registers may vary. The execution of stack-oriented bytecode can be inefficient because of the extra steps of push and pop operation in typical register-oriented hardware.
- *Redundant operations.* The stack-oriented model requires data be reloaded into the stack for each access, while the register-based architecture would avoid this.
- *VM-specific overhead.* Some overheads are associated to individual architectures. For example, the JVM requires stack entries to be 4 bytes wide. Further runtime services that JVM provides also contribute to the factor of inefficiency, such as automatic memory management, the complex object model and runtime boundary checking.

2.3 Execution Model for Java

Java applications are compiled into *class files* [58], which are equivalent to the executable output from a native compiler, containing bytecode instructions, symbols and debugging information, etc. The Java program's execution normally consists of two steps: firstly, a Java compiler translates the programs into class files, and then the class files can be executed accordingly. By using the class file to encapsulate the machine independent bytecode instructions, a Java program can be distributed to heterogeneous platforms without recompilation. Furthermore, since there is more semantic information remaining in the compiled bytecode instructions than in regular native binaries, further optimisation for the specified platform can be carried out in

later executions.

Except for the hardware implementation of JVM with native bytecode instruction support, the execution of Java class files involves translating bytecode instructions to native instructions. The translation process of bytecode typically is either executed on the fly by using an interpreter or dynamic compiler, or statically compiled before the execution by Ahead-of-Time (AOT) compilation according to the specified user scenario.

Java interpreters are generally regarded as a portable and simple implementation for executing bytecode. Due to the fact that it usually demands relatively low hardware requirements, interpretation remains as a popular choice for JVM implementation for embedded systems [87, 104, 26]. In order to achieve reasonable performance by reducing interpretation costs, many state-of-the-art JVM implementations such as HotSpot [73] and Jikes RVM, changed from Jalapeño, [2] apply dynamic translation and optimisation, which is commonly known as a Just-in-Time (JIT) compilation, on bytecode immediately prior to the execution of that code. It is essential that the cost of the compilation process should not outweigh the benefits it delivers. As a result, JIT compilers generally cooperate with a lower cost execution technique to balance the overall translation cost, and only selected code regions regarded as "hotspots" are treated specially by the JIT compiler. Arnold et al. [6] concluded that in their Jikes RVM implementation, by using a fast translation technique combined with selective optimisation based on online profiling, JVM can achieve the best performance among the single execution models.

Arnold et al. [7] summarized that three basic components are commonly required for an effective selective compilation system: a) a profiling mechanism to identify candidates; b) a decision making component for choosing the optimisation strategies; c) a dynamic optimisation compiler to perform code optimisation on selected regions.

Despite the fact that selective compilation techniques were largely used in many VM implementations [73, 2] to provide a reasonable performance for Java programs, selective compilation techniques were even extended outside the Java programming language domains [42, 17]. However, it is not commonly used in the resource-constrained domain. The complexity of the selective compilation system and dynamic optimisation compiler imposes a heavy burden on computing resource for dynamic optimisation. Even though further research proposed many approaches to reduce the complexity of the optimisation process [27, 16] to allow dynamic optimisa-

tion to be used in a more resource constrained environment, compromises on binary efficiency have to be made.

As an alternative to dynamic optimisation, one can compile the class files for the Java program statically before the execution begins, such as in [13,91,103]. This approach is referred to as Ahead-of-Time (AOT) compilation, which does not require extra translation overhead on the execution, so well planned optimisation sequences can be considered [19]. One of the main concerns of the AOT compilation approach is how to handle class files that are dynamically loaded, as dynamic class loading is one of the key features of Java, making it different from other traditional programming languages.

2.4 Java on Embedded Systems

An embedded system is a general term used to describe a computer system designed for one or more specific functions. Commonly they range from high-end 32-bit powerful mobile or hand-held computer devices to low-end simple 8-bit nodes in sensor networks. As a result of embedded systems being dedicated to specific tasks, they all tend to have significantly limited resources compared to desktop and server computers. These limitations are usually in the area of power consumption, available size of ROM and RAM, processor speed, network accessibility and even physical size. Furthermore, there is high diversity of embedded systems in term of hardware architecture and operating system installed, even inside a single application, such as applications for mobile, handheld computers or sensor networks. This diversity is a compelling reason to use a virtual machine as an abstraction for deploying applications.

Wirjawan et al. [110] conclude there are three major benefits of using the abstraction of a VM on embedded systems. First, VMs allow applications to be developed on a universal platform, rather than tailoring the application to fit in to individual type of devices. Second, VMs provide a better separation between application and system, that is, reduce the cost of maintenance when new devices join in or new changes are applied to the system. Third, VMs provide a common execution framework among a range of devices. The intermediate representation of VMs becomes the basis for application distribution without involving issues of incompatibility.

One of the primary concerns of using JVM and bytecode rather than native machine code is the overhead introduced by the extra abstraction layer of indirect execution. Furthermore, runtime services provided by VMs are also responsible for extra cost in the execution. Thus, the application executes more slowly, and consumes more power and memory than their counterparts. For example, a JVM specified for embedded systems (referred to as VMStar), proposed by Koshy et al. [47] reported that their `iadd` bytecode ran approximately 10 times slower than the native implementation. What is needed for embedded systems is an execution environment that can provide benefits to VMs as platform independent application development and deployment without requiring unacceptable overhead.

2.5 Related Work on Language Implementation

Much research and progress has been made on the execution of intermediate representation, DIR. One of the most important programming languages of this type is clearly Java which is the basic of much of this research. Research into the issues of compilation efficiency and generated code efficiency continue.

In the following subsections, we present related work in the area of interpreters, Just-in-Time (JIT) compilers and adaptive compilation since our implementation is based on and is extended from this previous work. We also review work related to bytecode annotation, proxy compilation and distributed systems based Java, especially for embedded systems.

2.5.1 Interpreters

Interpretation was used in many implementation of high-level programming languages in the early days, such as Lisp [61], APL [86], Smalltalk-80 [25] and Pascal-P [70]. To this day, interpreters remain one of the important methods of dynamic translation used in many implementation of languages such as Java [62], Perl [109], Python [33], MATLAB [102]. Various projects have improved the performance of interpretation from the early basic switch dispatch solution, which uses a large set of switch statements and labels for each of the instructions in the virtual machine instruction set.

Threaded Code introduced by Bell and James [12] is regarded as one of the milestones in the improvement of interpreters. Bell and James suggested that the Threaded Code technique can simplify the logic of interpreting instructions by making a direct jump to the next instruction implementation from the end of the current instruction implementation.

Piumarta and Riccardi [76] took Bell's *Threaded Code* a step further. In order to reduce further the complexity of the interpreting logic in *Threaded Code* which requires a direct jump on every virtual instruction, they suggested dynamically combining blocks of instructions together in a new "superinstruction", and modifying the code to use the new instructions.

A number of recent systems have been developed based on the dynamic combination of "superinstructions". Ertl and Gregg [29] explored various interpreter generation heuristics in order to improve the branch prediction accuracy. Ertl and Gregg [30] extended Piumarta and Riccardi's technique to dynamically translate non-relocatable code that may throw exceptions. Gagnon and Hendren [34] extended Piumarta and Riccardi's work to provide dynamic class loading and multithreading.

Furthermore, there are significant advances that have been made to improve the performance of interpreters, but an interpreter cannot match the performance of the optimising compiler. However, interpreters remain an attractive choice as a fast execution engine for the selective compilation system.

2.5.2 Dynamic Compilation

In order to improve the performance further beyond interpreters, Rau [83] suggested using the *dynamic translator* technique to translate the intermediate representation to native code on the fly reduces average time spent on interpreting, per instruction executed. This technique is commonly known as JIT compilation which compiles and possibly optimises a sequence of code blocks to native code before they are executed. The earliest published work about JIT compilation dates back to McCarthy's work on the LISP system [61]. He suggested compiling functions into machine language and this process is fast enough so that the compilation output did not need to be stored.

To address the issue of slow optimisation in dynamic code generation, progress has been made. For example, Hölzle [42] adopted a fast near-linear register allocation

technique in the third-generation SELF compiler as the replacement for the graph colouring register allocation which was used in static compilation. A number of projects also investigated an alternative fast register allocation technique in place of graph colouring [113, 79, 14, 90, 1]. Chen and Olukotun's MicroJIT [16] reduces the dynamic optimisation process into three major passes over the code, increasing the compilation speed and also reducing the memory footprint. As an alternative to performing general optimisations on-the-fly, off-loading optimisation for general patterns into static compilation can significantly minimise the cost of runtime code generation by up to 6–10 cycles per instruction [53,27]. However, it introduces limits on available optimisations and restricts specialised applications.

In order to reduce the interpretation cost and achieve comparable translation speed to interpreters, the adaptive optimisation system in Jikes RVM, formally known as Jalapeño JVM, developed by Arnold et al. [6] implemented a basic JIT compiler, referred to as the baseline compiler in their system, mimicing the stack machine of the JVM specification just as the interpreter. That is, the baseline compiler is only responsible for directly translating bytecode instructions using a simple register allocation technique. As the baseline compiler focuses on translation speed rather than binary efficiency, it is therefore too slow to be used as a standalone execution engine in the JVM implementation.

2.5.3 Selective Compilation

While dynamic optimisation is successful in continually improving the efficiency of the translated code, it also introduces new problems, as we addressed above, into the design of the execution engine of virtual machines. To address these problems, a virtual machine can safely exploit the well-known fact that most programs spend the majority of their time in a small portion of the code [46]. The problem to improve the overall performance of the program can then turn into the problem of finding and focusing on better optimisation for the frequently-executed code sequences. The virtual machine then can use a less expensive strategy to dynamically translate the rest of the code.

Detlefs and Agesen [24] published a study investigating the trade off between different translation approaches. They found that compilation time does not affect the overall performance for the long running applications. A combination of interpreter

or fast compiler and an expensive JIT compiler on the longest running methods produces the best results, based on using an "oracle" study to determine the most important methods. Some VM implementations adopt a mixed interpreter and compiler strategy [73, 113, 96] including the unpublished work of Kaffe [99]. Others take a compiler-only strategy to combine a fast non-optimising compiler and a JIT optimising compiler, such as Jikes RVM multiple levels of optimisation to distribute the compilation time in a finer granularity [6].

To identify frequently running methods ("HotSpots"), SELF-93 implementations [43] use the invocation counter as a low-overhead, coarse-grained sampling mechanism to keep track of the number of method invocations. The Java Hotspot Server VM reported in [73] took a similar strategy to SELF-93. However, as Hansen [38] pointed out in 1974, determining the counter-based threshold that drives recompilations heuristically has its limitations; it is hard to change an optimisation count that affects only a portion of the performance curve.

Jikes RVM [6] reduces the cost of profiling by periodically sampling the call stack of the executing program. This mechanism is able to reduce the overhead over the counter-base mechanism on each method invocation. Furthermore, a cost-benefit model, where the recompilation decision is determined by the estimated additional performance benefits over the cost of recompilation, is used to select multiple level of optimisation for recompilation.

2.6 Feedback Direct Optimisation

Online profiling techniques provide a higher quality of information on understanding how programs are executed in the VM, and they provide a better optimisation opportunity for dynamic compilation rather than static compilation. In 2000, Smith [93] gave a brief review of motivation and history of such dynamic optimisation techniques, namely *feedback-directed optimisation* (FDO). He also highlighted three factors as the motivation for FDO:

- FDO can overcome the limitation on static compilation by exploiting the information which cannot be available during static compilation.
- FDO gives freedom to the software vendor to revert or change the optimisation decision without jeopardising the outcome of the programs.

- Runtime binding can give more flexible and easy changes to software systems.

Inlining which replaces a call site with the content of the function and has proved to be one of the most efficient optimisation techniques, especially for object-oriented programs. Many studies [42, 6, 22, 97] have examined a more aggressive inlining policy based on the online profiling information without imposing a heavy burden on both compilation time and code space. Arnold et al. [7] reviewed several individual studies revealing that fully automatic online profile-directed inlining for the Java programming language would improve performance by approximately 10%–17%.

Arnold et al. [8] exploited the profiling data to improve the accuracy of branch prediction, which can guide the code generator to lay out code blocks in a contiguous address space in order to reduce the rate of cache misses. It reports a modest performance improvement. Other similar work on optimising code layout by using online profiling data has been reported [15, 95].

Applying aggressive optimisations using the profile information introduces the issue of whether the optimised code sequence can only be valid in a specified situation. Many virtual machines use multiple version of optimised code with aggressive inlining to handle dynamic dispatch [32, 73]. To reduce the overhead of runtime speculation for applying multi-versioning, Arnold et al. [9] worked on fast runtime tests to identify regions of code within which speculative optimisations can be performed.

2.7 Annotation

Research has demonstrated that there is a trade-off between the optimisation time and the efficiency of the optimised code in dynamic compilation. Most research, however, has been confined to adapting fast alternative analysis algorithms or carefully selecting optimisation code regions, and has not demonstrated that a dynamic compilation system could cope reasonably with the increased size and complexity of a system based on interpreters. One of the general approaches is called *stage compilation*, where compilations of a single program are divided into two separate stages: static and dynamic compilation. Prior to the application execution, a static compiler compiles "templates", the essential optimised building blocks, which are assembled together, and also places runtime values into specified places (linking) using a dynamic compiler.

In the early designs of stage compilation, code fragments appropriate for compilation into templates are hand picked by programmers. The 'C (Tick C) system designed by Engler et al. [28] and other research [77,78], suggested that the efficiency of dynamic compilation can be improved by manually compiling code fragments into a portable, optimised intermediate representation to be used at runtime. Alternative attempts at using static compiled templates in C [18,67] and ML [54,53] follow a *declarative* approach where user annotations trigger analysis and transformation of the programs into a template which can be fast assembled in the runtime compilation. However, special requirements for heavy user involvement either in the compiling process or coding to specify the code fragments for static compilation into templates creates a limitation on using the stage compilation technique which requires a substantial rewrite to fit the limitations of the system or miss the optimisation opportunities. Although later implementations in Dyc by Grant et al. [37] reduce the reliance on the programmer to specify code fragments for optimisation, code annotation on static variables is still required.

Azevedo et al. [10,11] took a different approach to applying stage compilation on Java, where they proposed their Java Annotation-Aware Just-in-Time (AJIT) System to attack the inefficiency of dynamic compilation resulting from the issue that the underlying stack model and many bytecode operators including sub-operations are the primary reason. Their AJIT compiler annotate the bytecode with machine independent analysis information without having to perform dynamic analysis. Their results show the off-line optimisation annotation can generate binary code as efficient as the Kaffe [99] JIT system, however, the cost of their annotation-aware code generator at runtime was not discussed in either of the publications. It is believed that simple optimisation and code generation from annotation, such as register allocation and instruction selection are still required [11]. In addition, since the bytecode annotation is statically generated as a separate operation, the limitation for stage compilation as mentioned above is still not solved.

While Azevedo's work addresses the platform independent annotation, Serrano et al. [88] proposed using optimised native binaries generated by a quasi-static compiler instead of optimised intermediate representations. This work wraps the optimised binaries, which would be statically or dynamically compiled by using the dynamic compiler in the JVM, into a persistent format only requiring simple modifications to be ready for execution, in order to reduce the overhead of compilation during pro-

gram execution. The authors' primary interest is improvement in compilation speed, however, the availability of the optimised binary remains as a critical issue like many other stage compilation solutions, although the requirement for code annotation is removed. This article proposed that optimised binaries can be generated by static optimisation, that is, bytecode without pre-execution optimised binaries are subjected to slow interpretation in execution. The article also proposed an alternative approach, in which optimised binaries are generated by dynamic compilation, that is, it required three compilers (interpreter, JIT compiler, quasi-static compiler) to be activated in the runtime, however, because of the limitation of computing resource in embedded environments, it already violated the original dual purposes of stage compilation, to achieve both compilation time efficiency and space efficiency.

2.8 Remote Compilation

There is relevant work in this area of using server-based compilation as an alternative to dynamic compilation in Java, as well as for static programming languages such as C and Fortran. Following the initial publication [56, 57] by the current author on the possible solutions to the problem of providing dynamic compilation for resource-constrained environments, during which period the work presented in this thesis was being completed, further independent results [110, 52] were published using similar approaches to the problem of providing dynamic compilation for embedded system and desktop.

Voss and Eigenmann [105] implemented a remote compilation system to perform dynamic compilation for the static programming languages C and Fortran. Portions of the code which are identified as "HotSpots" are selected for optimisation by a local or networked compiler with the knowledge of the current execution profiles. It is similar to the work presented in this thesis but with a completely different design goal. Their design is tailored for a distributed server or multiprocessor machines, where they rely on a multi-threaded target execution environment: using NFS and RPC. Our design goal is that dynamic compilation in the client should be able to offload to a compilation service without relying on a tight coupling between client and server. In addition, Java specific issues are not considered in this work. In fact, the program sections, referred to as intervals, are selected by the user or offline compiler, which becomes an issue for the Java programming language when Java

classes are loaded dynamically as incoming classes must be rewritten on the fly. In addition to these differences, we present much more detail on the approach to provide dynamic loading and linking. Delsart et al. [23] present a commercial remote compilation system called JCOD, designed to perform native compilation for Java. The compiler server in JCOD compiles VM service and VM-independent native code for an embedded system. The authors' primary interest is focusing on improving execution performance while minimising the increase of code size and memory footprint. As a matter of fact, their compilation server only performs a optimisations that specifically design to reduce overall code size.

Newsome and Watson [66, 65] describe a proxy compilation scheme called MoJo in which the compiler server compiles Java classes into C source code and then the GNU gcc compiler is used to produce object files. The design goal of the authors is aiming to provide Ahead-of-Time compilation with dynamic class loading support, targeted to resource-constrained environments. MoJo handles only a subset of Java features and does not support "HotSpots" method profiling, but instead compiles the entire class file before execution. As a consequence, client execution is halted until the optimised binary is installed from the client. In this sense, MoJo works more like an Ahead-of-Time compiler that batches the compilation for its client.

Work by Palm et al. [74] investigated the power consumption issue when dynamically loaded Java classes are compiled by using a tethered server rather than compiled on the embedded system. They concluded that energy consumption for their configuration on embedded systems can be reduced significantly, especially for longer running methods, by moving to a server. The proposed solution for the problem is similar to our thesis, however, the authors have not implemented their ideas experimentally.

Two pieces of individual research [110, 52] were published prior to this thesis, showing that remote compilation techniques used as an alternative to local dynamic compilation are still an active research topic. Wirjawan et al [110] exploit the possibility of applying remote compilation techniques in the Wireless Sensor Network (WSN) domain. The author proposed that the Java application is deployed with a tailored VM subset from the base station, later responsible for making the compilation and sending compiled binaries into individual nodes. Again, similar to MoJo, they are also using the static compiler (GNU gcc) as the code generator to produce the final machine code. Our approach focuses on a more universal solution of pro-

viding a compilation service with bytecode handling, security and code coupling. Teodorescu et al. [100] adopted a more aggressive approach for customising JVM for reducing the memory footprint in a WSN domain. The device runs on minimal kernels that download only limited parts of the run-time system and optimised application binaries on demand. In this case, all bytecode translations are performed on the server.

In 2007, Lee et al. [52] implemented their remote compilation system for the desktop environment, after their initial publication on embedded systems [75]. The design philosophy of their system is similar to ours, to off-load most of the heavy load of compilation to a more powerful server. However, in their approach, the optimised binaries from the compiler server are ready to be used, without requiring additional linking, reducing the linking cost while in execution. As a result, the client's execution state has to be sent to the server during the execution, and the optimised binaries cannot be cached in local storage since the execution state has been hard coded into the optimised binaries. In contrast, our design uses a relocatable format for organising optimised binaries. In addition, the compiler server in our work is intended to be an independent service over the Internet while Lee designed it as a proxy for the client to access application source from the Internet, and there may potentially be limited access from applications where client authentication is required.

2.9 Distributed Virtual Machines

The basic principle of remote compilation which migrates compilation tasks into a networked server can be seen as a specific instance of a distributed system in which components located at networked computers communicate and coordinate their actions through messages [21].

Sirer et al. [92] proposed a distributed virtual machine (DVM) for network computers. DVM explores the distributed service architecture to factor out the services of VM runtime, such as verification, security enforcement and compilation, to a centralised heterogeneous cluster behind the firewall. However, the research interests of the authors' publication are more focused on manageability and stability of the system instead of performance and memory consumption. The motivation behind

their work is different from us. They are concerned with manageability of security on various clients in a trust network, instead of being concerned with compilation performance and memory consumption.

In order to reduce power consumption for embedded systems, particularly in the WSN domain, recent research [71, 84, 85] focuses on migrating parts of application execution over wireless networks. Rudenko et al. [85] developed a remote processing framework to provide system support for directing and handling process migration. Kremer et al. [49, 48] improve the stability and reliability of process migration over inconsistent networks. The server periodically synchronises checkpoints (as determined by the compiler) for clients to allow the client monitoring the program record appropriate information, and continues the process when a server or network failure is encountered.

JESSICA 2 developed by Zhu et al. [115] investigated how to parallelize program execution on clusters. By building a single system image (SSI) to hide all distributed aspects of distributed computing, they expected to balance the workload over the cluster without intervention from the user. Similar work also can be found in [114, 4, 5]. However, their research is more focused on providing high-performance parallel execution environments for specifically parallelizable programs instead of general optimised code for program execution.

2.10 Summary

This chapter began by looking at the directly interpretable representation (DIR), as one of the three different program representations which has been used in program execution - as defined by Rau [83]. As well as the program representation itself, we then reviewed the characteristics of the Java programming language. Many of its features, such as platform independence, native support for dynamic class loading, and automatic memory management, became milestones for modern programming language design, radically distinguishing Java from many traditional programming languages. However, using a platform independent intermediate representation, bytecode, implies that extra compilation processes on the bytecode prior to executions are required.

We identified the two conflicting goals of dynamic compilation in Java execution,

code efficiency and optimisation cost, which includes the optimisation time and memory footprint of the optimisation compiler, during the Java program execution. Their goals are especially important for embedded systems, in which only limited resources are available.

As we described our research challenges with Java execution, we then conducted a wide review of many approaches used to solve similar problems for Java execution. In most designs, there is a trade-off between the conflicting issues, for example, fast online optimised compilation produces less efficient binaries.

Is it possible to provide an alternative dynamic code optimisation approach to maintain the efficiency of Java execution, while does not impose substantial optimisation burden on the host machine? With the research goals in mind, some observations can be made:

- An advanced technique for implementing a dynamic optimising compiler using the adaptive optimisation system can limit the cost of the IR translation during execution, while retaining the execution performance advantage via feedback directed speculative optimisations. However, this approach requires two translation engines, so a complicated optimising compiler is inevitable.
- Alternative approaches to overcome the cost of dynamic optimisation while having efficient binaries, are static compilation and binary annotation ahead of execution. However, as one of the important features of Java is native support for dynamic class loading, it is not feasible to compile the entire program ahead of execution.
- A remote compilation service is used to replace the local optimising compiler to reduce the IR optimisation cost in the program execution. The possibility of using a dynamic compiler as the remote compilation engine has not been given much attention in the literature, although more recent research projects have employed it.

Consequently, the aim of the work presented in this thesis is to demonstrate that a proxy compilation infrastructure that utilises a dynamic optimising compiler can effectively offload the dynamic bytecode translation cost in heterogeneous networked clients. The proxy compilation infrastructure has the potential to provide a sound foundation for extended research on proxy compilation based on a dynamic optimising compiler, such as speculative optimisation in a proxy compilation environment.

To explore the related issues, the Apus proxy compilation system is designed and implemented. The remaining chapters in this thesis discuss issues involving the design and implementation of the Apus proxy compilation system, and finally evaluate our system.

CHAPTER 3

Design Overview of the Apus Proxy Compilation System

The joy of discovery is certainly the liveliest that the mind of man can ever feel.

CLAUDE BERNARD (1813–1878)

Among many key factors dynamically affecting the overall performance of IR interpretation is the translation quality during execution. An important decision when building an interpreter between IR and native code is making the right balance between producing highly effective code and low translation overhead.

The previous chapter reviewed a range of approaches for dynamic compilation. It showed that they have various strengths, but at the same time, there are limitations. In almost all traditional designs of dynamic compilers and interpreters, a trade-off must be made between two or more conflicting goals, e.g. size against speed, or sophisticated optimisation against online compilation cost. With this in mind, this chapter proposes a caching mechanism for small granularity compilation to demonstrate that those key conflicting factors can be balanced.

This chapter presents some key design decisions made for the design of the overall architecture of our novel approach: the Apus proxy compilation system for the Java

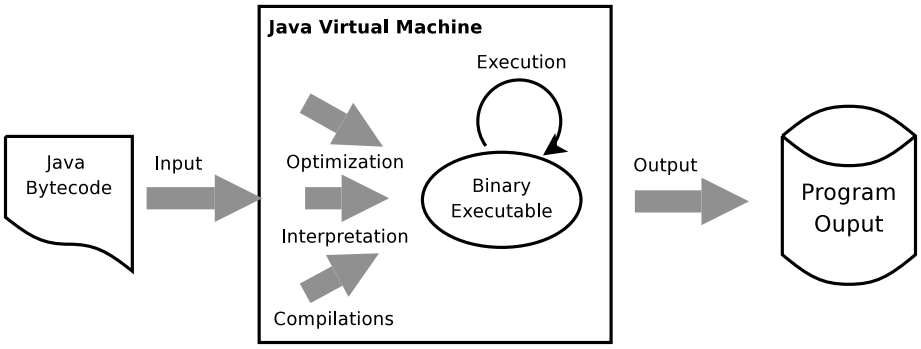


Figure 3.1: Intermediate Representation Execution Model

Virtual Machine. Firstly, the design motivation and requirements of the Apus system are presented. Some key design decisions along with the system architecture are discussed. Then the overall architecture is given and explained in detail. Last but not least, an introduction to the design of the Apus compilation server and clients is presented.

3.1 Basic Principles

As discussed in Section 2.1, Java bytecode belongs to the category of intermediate representations which share many properties that distinguish them from native machine code, as well as some common drawbacks, particularly execution efficiency. Section 2.3 discussed the execution efficiency of Java using three main approaches. Figure 3.1 demonstrates the reason why dynamic translation contributed to the inefficiency of Java execution using a mathematical model. The overall execution speed of a Java program can be simplified from Figure 3.1 as:

$$t_{overall} = t_{execution} + \Sigma t_{compilation} \tag{3.1}$$

The overall execution time $t_{overall}$ is the sum of the execution time of the binary code, $t_{execution}$ and the total cost of compilation $\Sigma t_{compilation}$. A sophisticated compilation system may consist of a set of compilers with different properties that are driven by policies that benefit the program execution performance. For example, in the adaptive optimisation system, the time for compilation is therefore a sum of the

time spent in compilation by all the available compilers and interpreters on the system during the execution. However, what this model doesn't reveal is the relation between $t_{execution}$ and $t_{compilation}$ which is determined by the choice of compilation strategy. Highly efficient code cannot be generated from a time efficient compiler, e.g. a system using a single pass with direct instruction mapping without applying in-depth analysis. However, as the survey in Section 2.5.3 reveals, systematically building up a compilation strategy from a set of available compilers can make a balance between $t_{execution}$ and $t_{compilation}$ that eventually leads to a minimised overall execution time $t_{overall}$.

It is known from Section 2.5.2 that an optimising compiler can produce efficient code to boost execution performance, at the cost of performing computationally heavy optimisation before and during the execution. In order to minimise the $t_{compilatioon}$ in the model while producing highly efficient binaries from bytecode, a proxy compilation system can attack those two criteria at the same time. As the survey in Section 2.8 reveals, migrating the compilation process to a networked server for translation from Java bytecode to native machine instructions can benefit the JVM in both execution performance and memory footprint, while overcoming the dynamic class loading issue introduced by Ahead-of-Time compilation.

3.2 Design Requirement

A Java virtual machine would be able to provide a reliable execution environment for Java programs, whilst improving the execution performance via the proxy compilation service. The performance described in this context should not only be limited to the execution speed, but should also include other criteria such as memory consumption and execution pause caused by dynamic compilation.

The proxy compilation system consists of two parts. The proxy compilation client is responsible for providing the execution environment for the Java programs. The client is a recipient of the compilation service, which is located on a networked compilation server. This server is responsible for translating the requested Java bytecode into an efficient binary for the client.

From what we describe above as the background of the proxy compilation system, we can conclude that the design requirement in principle for the proxy compilation

system can be represented as follows:

- *Correctness* - The binary produced by the proxy compilation system should be executed correctly, that is, the result of the Java execution should not be altered by the involvement of the proxy compilation system.
- *Reliability* - The availability of a proxy compilation service should not affect the availability of the execution environment. That is, an alternative approach to translate Java bytecode should be available on the proxy compilation client.
- *Efficiency* - Efficiency is required in two separate areas of the system. Primarily, the proxy compiler should produce an efficient binary from the bytecode. Secondly, there should be an efficient means by which the binary is made available to the client.
- *Accessibility* - There are two requirements on the accessibility of the system. Firstly, the proxy compilation system should be easily accessible by the client. Secondly, the source file should be efficiently accessible by the compilation server as well.

3.3 System Overview

Our current design and implementation of the Apus proxy compilation system is based on the guidelines from our design requirements in Section 3.2. Figure 3.2 shows an overview of the architectural design of the system. Broadly speaking, the goal for the Apus proxy compilation system is to reduce the compilation cost of Java execution while retaining efficiency.

The Apus proxy compilation system consists of multiple clients and a compilation server. The client is primarily intended to be a resource-constrained device that are likely to have some limitations compared to the compilation server which has significant computation power, large memory capacity and efficient storage access. Therefore, migrating the optimisation compilation process to the compilation server would provide more benefits over having the optimising compilation on the host machine competing for the limited computation resource.

We are making an assumption that the compilation client can obtain Java programs in a very flexible way. That is, the source files, the classfiles in Java programs, can be delivered to the compilation client without the knowledge of the compilation

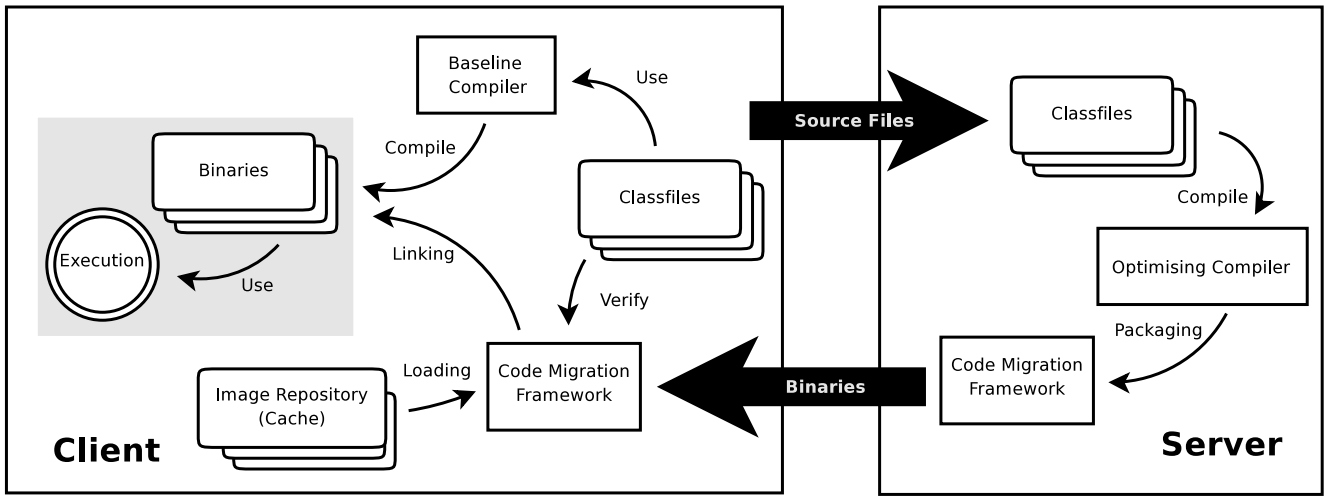


Figure 3.2: Apus Proxy Compilation System

server. In addition, the client should also be able to modify the program without the knowledge of the server. As a consequence, the assumption requires the client to provide source files to the server directly, if there is no alternative source available from the network. Binary validation on the compiled binary to ensure that it corresponds to the source files is required. Although such assumptions complicate the design of the proxy compilation model, the design requirement described extends the accessibility of the service by reducing the data dependence between the client and the server. Further discussion of the binary validation is in Section 4.7.5.

One of the important issues concerning relocating binaries from the compilation server to the client is how to minimize this cost to the client. In our implementation, the client is fully responsible for relocation and verification. Locating the linking process in the client would benefit the client in two areas. Firstly, it reduces the communication cost by avoiding the synchronisation of current execution states with the server. This would reduce overall time taken for a compilation request. In addition, for devices depending on battery power, lower network communication leads to less energy consumption. Secondly, a relocatable format allows binaries to be stored in the cache of the client's local storage. Therefore, in further execution, networked proxy compilation can be completely avoided if the corresponding binary is available locally. Test results in Section 6.6.2 prove that the linking process does not significantly affect performance of the proxy compilation process in the client.

There are three major procedures in a proxy compilation request. Firstly, the client sends out a compilation request to the compilation server to describe the requested target. This requested target is a Java method, selected by the client. Further discussion on how to select a method for proxy compilation is described below in Section 3.5. If the correct source files are not directly accessible by the compilation server, for example, if they are not in the cache of the server, all of the source files related to the requested target need to be sent to the server by the client. Secondly, the compilation server acquires the requested target and related source files. The server compiles the target with optimisation into a relocatable binary and sends it back to the client. Thirdly, the compiled binary sent from the server is modified according to the state in the execution environment ready to be executed. This linking process, which includes code verification and relocation is handled by the code migration framework.

In order to handle all the necessary proxy compilation communication while keep-

ing a reasonably small network overhead, a simple proxy compilation protocol (PCP) is designed and implemented. Details of the PCP are presented in Chapter 5.

To provide the proof of concept of proxy compilation and concentrate on the implementation of the system, the current Apus proxy compilation system is built on the Jikes RVM [3], which is a research Java virtual machine implementation with a well-established adaptive optimisation system.

3.4 The Choice of Compiler

One important decision in providing a proxy compilation service for JVM is the choice of the compiler used to translate Java bytecode to native machine instructions for the clients. Section 2.3 briefly discusses the three popular approaches for the translation of bytecode.

According to the design requirement and system overview laid out in previous sections, the proxy compiler is responsible for partially compiling the Java bytecode as the client's requests. In addition, the efficiency of the compiled binary and optimisation processes is essential for the service. As a result, only optimising compilers are being considered for the proxy compiler server.

Offsetting optimisation to a network server can reduce the bytecode optimisation pressure on the execution performance in the client, however, research carried out by Newsome [65] pointed out that the time when the binary is available for the execution is critical for the performance, since the efficient proxy compilation process can reduce the time on the client executing the non-optimised code. The proxy compiler therefore should be capable of delivering the efficient binary in a limited time frame. Much work in dynamic optimising compilers has been done to balance the compilation efficiency and code efficiency. As a result, a dynamic optimising compiler is used as the proxy compiler which is located in a networked server.

In the survey in Section 2.8, a majority of proxy compilation implementation [105, 66, 65, 110] use a static compiler, such as GNU gcc, as the back-end for code generation. A static compiler have advantages on code generation, as it can provide extended code optimisation procedures and support a wide range of target architectures. However, this advantage can be offset by exploiting speculative optimisation, such as aggressive inlining, on the bytecode by using dynamic compiler to generate

best optimised code according to the execution profile. Furthermore, since the AOT compiler is designed for offline compilations, it is therefore less time sensitive in the compilation process compared to dynamic compiler. In addition, implementation of proxy compilation for Java, such as [66, 65, 110] slows down this process even further by requiring an extra compilation process to translate Java bytecode into an intermediate representation that can feed into the compiler back-end.

There are more potentials to be exploited on dynamic optimising compilers for using on proxy compilation scenario. It is therefore easy to conclude that, as we have discussed the research motivation in Section 1.2, the purpose of the Apus proxy compilation system is to provide a infrastructure that allow further research would focus on using dynamic optimising compilers to provide proxy compilation for Java virtual machine.

3.5 Selected Optimisation

One of the important aspects for the client is how to take advantage of the Apus proxy compilation system to get the best benefits for Java execution. In order to provide a reliable execution environment for Java programs as we described in the design requirement, the client applies a hybrid execution model mixing an inexpensive baseline compiler and the proxy compilation service.

Although fully relying on a proxy compilation service would provide more efficient code for the Java programs overall, the execution environment is also dependent on the availability of the proxy compilation service and the network. For example, in the situation where the network connection is disrupted during Java execution, the client would be unable to have the native code for the input bytecode. This leads to a complete failure of the system that we need to avoid. It can be argued that having multiple compilers would increase the memory demand of the system, however, such an extra memory footprint can be offset by less code space required, since only a region of the program is subject to optimisations such as inlining and loop unrolling.

The client applies the selective optimisation approach on the proxy compilation service. Only selected methods, which are regarded as having a critical impact on the performance, are compiled by the proxy compilation service. A fast inexpensive baseline compiler is chosen as the default approach for bytecode translation, to deliver

the native code at high speed to reduce the execution pause caused by compilation.

In order to predict methods that can improve the performance, an adaptive optimisation system is responsible for the selection applying to proxy compilation in the client. Although it can be argued that it would be less demanding on the client if the compilation server is responsible for the method selection process, having the server choose does not work well as there is not enough information, particularly live runtime information, made available to the server.

3.6 Proxy Compilation Policy

In the previous chapter, we have discussed how the proxy compilation is integrated into the runtime environment in the client. Another important decision in applying a proxy compilation system is when the selected method is applied to the proxy compilation service for code optimisation.

The client applies the optimisation policy driven by the cost-benefit model proposed by Arnold et al. [6]. In their model, the compilation controller periodically samples the calling stack to identify methods where the application spends most of its time, then the compilation controller assumes that the sampled method will continue to be executed for as long as it has been executing so far. The sampling information feeds into a pre-defined cost-benefit model to predict how much time it can save by optimising the targeted method. If the improved running time as predicted plus the optimisation time of the method is less than the execution time of the current binary of the targeted method, then the controller decides to recompile the targeted method.

In the client, three different compilers are taking account of the selections of compilations. As mentioned in previous sections, the baseline compiler is used as the default compiler for all the bytecodes. The proxy compilation service, and loading binaries from a local cache are chosen for bytecode recompilation. The cost-benefit prediction model is based on aggregates of offline profile data to determine the optimisation strategy for the profiling methods. The cost of the compilation is expressed in terms of the number of bytecodes compiled per unit time. By using the modified cost-benefit model on the client, the compilation controller can then estimate when to use the binary caches if available, or proxy compilation to improve the execution

performance.

3.7 Summary

This chapter gives an overview of the architecture of the Apus proxy compilation system, and explains the basic principles of the system. Many important decision made in the design of the system are discussed. As shown, there is a necessity for choosing a dynamic optimising compiler as the translation engine for the proxy compilation server is discussed. In the client, selective compilation details given in this chapter show how the client integrates a proxy compilation service into the execution environment.

CHAPTER 4

Design and Implementation of Apus Code Migration Framework

We are generally the better persuaded by the reasons we discover ourselves than by those given to us by others.

BLAISE PASCAL (1623–1662)

The previous chapter outlines the overall architecture of the Apus proxy compilation system. One of the key problems is how to efficiently utilise the binary, which is produced by the dynamic optimising compiler in the compilation server, on the client, as the runtime status of the client is not accessible by the server.

In this chapter, we present the design and implementation of a code migration framework used to handle code relocation in our proxy compilation system to address issues described as above.

4.1 Introduction

As we discussed in Section 3.3, a migration process to verify and modify the binary for the current execution environment is required. This code migration process serves

two main purposes. Firstly, a process is used to verify the coupling relation between the binary and the corresponding source file. Once the source file, classfile, is changed after the code has been generated, a verification should identify the difference and stop the binary being used in the execution.

Secondly, while variables related to the execution environment are hard coded into binaries, a linking process is responsible for finding the location of those references to those variables and update the values accordingly, so the binaries can correctly reflect the execution environment. For example, given a static variable `v`, the optimised binary then references `v` by assigning the address `&v` to the operand of instructions to access the value of variable `v`. In order to retrieve the correct value of `v` in an execution environment, the process has to modify all references to `&v` in the binary of this execution environment.

In order to improve the loading efficiency and network efficiency as we discussed in Section 3.3, the linking process should be performed in the client system, as well as the verification process. As a result, the code migration is divided into two parts. In the compilation server, along with the optimising compilation, information to direct the linking process should be added to make the binary relocatable. The client then can use this information to migrate the binary into the current execution environment. Furthermore, a binary migration from the local storage as a cache also needs to be considered.

4.2 Design requirements

The purpose of the code migration framework design is to be a proof of concept for the idea of migrating of dynamic compilation results. Full support of the complete Java Virtual Machine specification on the code migration framework is not required as would be for a piece of commercial software. Therefore, the code migration framework is designed to contain just the key functionality required. Although the code migration framework is not intended as a production feature for the JVM, for commercial software deployment, it contains enough support for JVM specification so as to be realistic.

The design requirements are listed as follows:

- Relocation information is provided within the compiled binaries. While the

binary and relocation information are located together, it reduce the engineering complexity to coupling both information in the linking, as a result, it can improve the code migration performance overall.

- The code migration framework should be as independent from the VM as possible. It should be seen as a separate module. From the clients' perspective, the code migration framework provides a similar service to a compiler, as one of the compilation services. The code migration framework simply loads binaries instead of compiling them during execution.
- Relocatable binaries should only be used as an annotation, which means the failure of loading relocatable binaries should not affect any outcome of the execution, except performance.
- Relocatable binaries should be capable of performing self verification against the change of the source IR from which it is translated. There are two main problems with the verification. It is not only a security issue but also an issue of building up a dependent relationship between binaries and source files. That is, the code migration framework should be aware of the modification of source files during execution without loading obsolete relocatable binary. On the other hand, the verification is also responsible for preventing malicious code injection into binaries which would potentially break the intention of using bytecode instead of native machine code.
- An efficient solution for organising relocatable binaries in local storage is essential to the overall performance of the system. It should provide a reasonably efficient relocatable binaries reading and writing procedure that does not contradict the primary purpose of using relocatable binaries, performance.
- The relocatable binary format must consider the balance between handling efficiency and space efficiency. Space and performance are a pair of contradictory factors. The relocatable binary format should be efficient enough to convert between compilation instances and flat images without jeopardising our motivation. Since the code migration framework is working closely with the Apus proxy compilation system, the relocatable binary format must be relatively compact to use over networks.

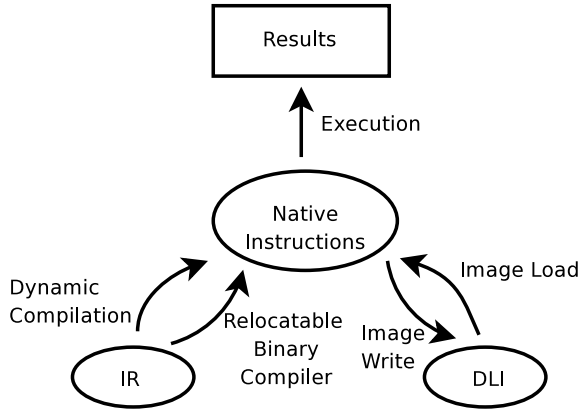


Figure 4.1: Apus Image Life Cycle

4.3 Apus Code Migration Framework Overview

This section gives an overview of the logical components of the Apus code migration framework architecture, avoiding the details of performing the actual verifying and linking procedures. It documents some of the important design decisions made, with reference to the design issues covered by the background material in previous chapters.

4.3.1 Architecture

A code migration framework is designed and implemented as shown in this chapter to solve the problem of reusing optimising compiled binaries from the proxy compilation server. In this framework, a relocatable binary format, the Apus image is used for the purpose of the proxy compilation service and binary cache management. It organise binaries and all the related information about code migration to allow binaries to be modified accordingly in different execution environment.

A simplified concept of the code migration framework is shown in Figure 4.1, which outlines various aspects of the data used in different stages during execution, together with the processes applied at each stage. During execution, bytecodes are either interpreted or compiled into native code then to be executed by the processor. In the code migration framework, relocation information is generated alone with a compilation process. Later, binaries can be stored in a persistent format (Apus image) and relocation is possible in different execution environments without requiring

a duplicated compilation.

It can be argued that the code migration framework is designed for a proxy compilation environment. However, since as it is addressed in the design requirement, Apus images should not require further modification to be stored and reused from local storage. As a consequence, the same concept can be applied to the proxy compilation service. The Apus code migration framework is then presented as a separated logical component in a complete JVM system, as no proxy compilation service is being mentioned. Chapter 5 discusses issues involving providing a networked proxy compilation service based on the Apus code migration framework.

The Apus code migration framework can be broken down into more detail, showing how each component interacts with the existing execution environment. Figure 4.2 shows the constituent components of the code migration framework. The figure descrbies a series of procedure for the execution of Java bytecode by using various appoarches, including using Apus images. Our implementation of Apus code migration framework is again based on the research JVM prototype, Jikes RVM. Thus, some of the components, for example, the Adaptive Optimisation System (AOS) [6] and Baseline compiler are a part of Jikes RVM implementation.

There are essentially three parts that are particularly interesting in our system. The AOS is responsible for dynamically selecting Java methods for recompilation, as well as when to use our image loader for Apus images. The selective optimising compilation policy has been discussed in Section 3.5.

Figure 4.2 give us a clear overview of how the Apus image is generated from relocatable optimised binaries. Firstly, the image producer is a modified optimising compiler that can produce relocation information during compilation. Secondly, the compilation output of the image producer is a executable binary that can be installed and used by the Runtime. Thirdly, this binary with relocation information can be collected with other related information for code migration and later written into the Apus image by the Image Writer if necessary.

The process to reverse the Apus image into an executable binary is called image loading, which is managed by the Image Loader component. The Image Loader shares the common interface with the other dynamic compilers, which allow it can be managed by the AOS to follow the proxy compilation policy.

In the rest of the section, we will introduce and discuss the challenges and design

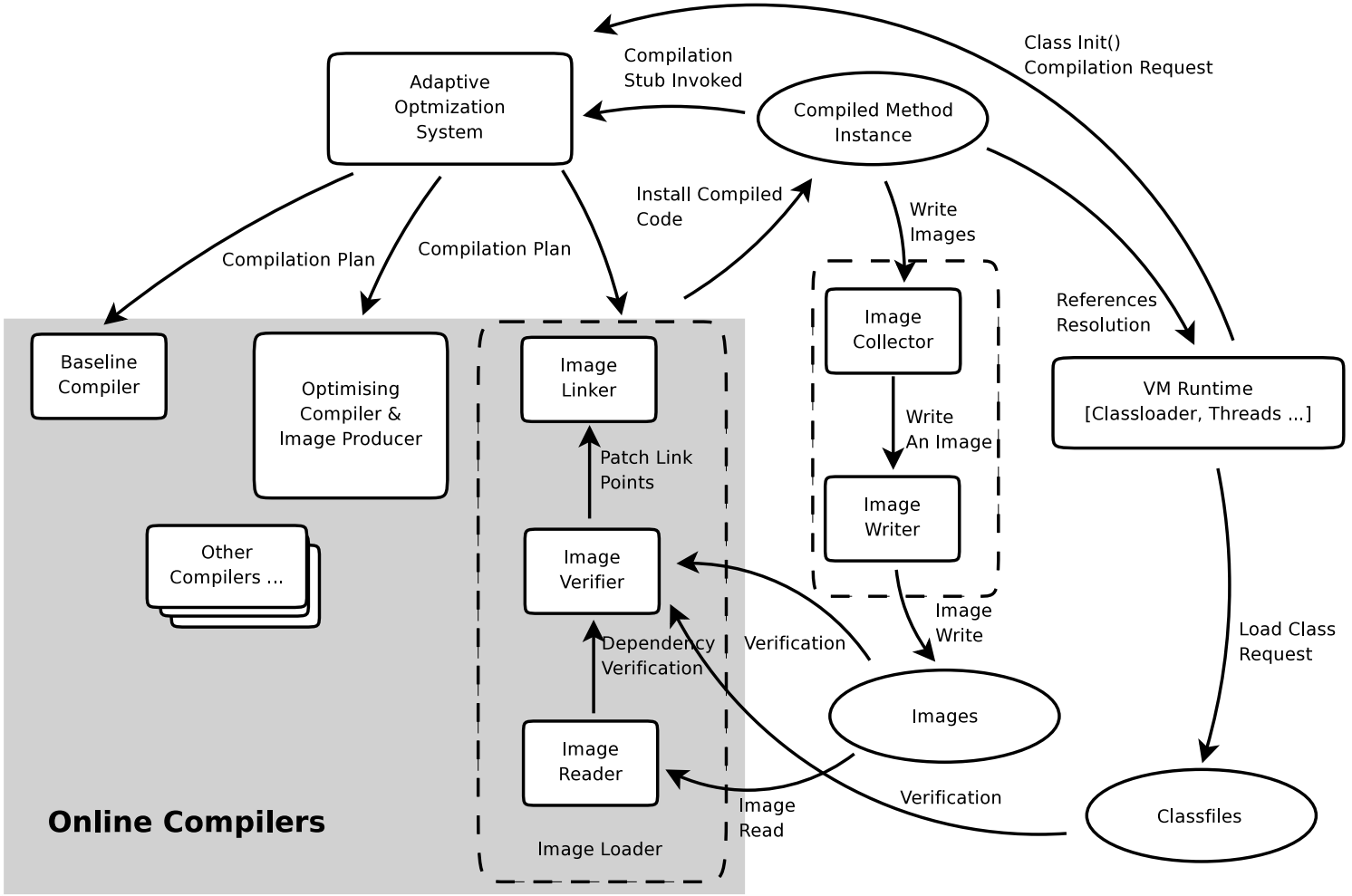


Figure 4.2: Code Migration Framework Architecture Breakdown

of each component in the code migration framework. We will adopt a top-down approach discussing the rationale behind the choice of VM and other aspects of the components in the code migration framework.

4.4 Image Loading Procedure

As we understand from previous descriptions, an image loading process is used to turn a relocatable binary in an Apus image into an executable binary in VM runtime. More details about the Apus image can be found in Section 4.7. An Apus image loading is triggered by a general compilation request by passing a method as the argument to the image loader unit. The image loading causes the specified method to be loaded, linked to the JVM runtime and installed. It is the compiler controller that determines when the image loader should load the Apus image of the specified method.

We should notice that the image loader does not specify the source of the Apus image. It is down to later implementations on particular user scenarios to direct the input to the image loader unit. As we described in Section 4.3, the potential input for the image loader is either coming from the proxy compilation service, or the local storage to hold the image cache. If Apus images are found, the image loader unit is responsible for the loading process, which includes image verification, dependency relation verification and code relocation.

Image verification described in Section 4.7.5 is the first step immediately after loading the binary. It checks that the Apus image representation is in the correct format, with proper relocation information as well. If a problem is detected during image verification, an error is thrown.

Dependency verification described in Section 4.8 is the process of checking all of the bytecodes, including the corresponding method and inlined method that have been translated and encapsulated, are consistent with the translated binary. If a problem is detected during dependency verification, an error is thrown.

Code relocation involves modifying data for the binary of a method according to the state of runtime. A link point table and symbol table are used to describe how and where to relocate the binary. Each entry of the link point table describes a relocation spot in the binary. A symbol representing the reference to a runtime

object in the binary is stored in the symbol table. Retrieving the object from the symbolic reference involves symbol resolution. If an unresolvable problem is detected during binary relocation, an error is thrown. More details about binary relocation and symbols can be found in sections 4.5 and 4.6 respectively.

Finally, after the successful completion of loading, verification and linking of the Apus image, the binary of compiled method is installed as the binary of the corresponding method and is ready to be executed.

4.5 Relocation

The term *relocation* is used here to refer to the process of adjusting data referred to in program instructions and associated runtime data corresponding to the current execution VM runtime. Code relocation is one of the central processes in code migration. This process modifies the relocatable binaries into executable form after the Apus images have been loaded and verified.

Code relocation is not a new area in program linking. Many sophisticated software relocation techniques have been successfully applied commerically, such as a.out [55], ELF [68] and COFF [35] relocation. However, since our design is based on Jikes RVM implementation, where the relocation involves a data structure that is implementation-oriented. We believe adapting an existing relocation format for a particular requirement is less valuable in term of space efficiency and time efficiency during the code migration. For example, a simple relocation format, a.out, contains a "exec header" segment, which it is used to store parameters used by the kernel to load binary files [82]. Furthermore, existing relocation formats do not provide a solution for our problem in described Section 4.6, when an external symbol table, a constant pool from a classfile, is referred to from our optimized binary.

The following subsections provide a list of information for the relocation binaries that has to modified for the current runtime. The data structure that we used to manage this relocation information is discussed.

4.5.1 Relocation Information

Before the relocation process, the image loader lays out the various parts of a Apus image into VM runtime. Data and references that are related to the current runtime needs to be located and modified accordingly.

In Java bytecode, a constant pool [58] is used to manage all the references used in the bytecode of the class. It allows the runtime to look up a referred variables by performing reference resolution on entries of the constant pool. However, to speed up constant pool lookup during the execution, direct reference to variables, such as direct memory addresses of static objects are placed directly into the machine instructions. Furthermore, instructions may expand to involve runtime service methods in which the location also varies across JVM instances. A list of runtime data that has been used in the binaries that require modification in code relocation is identified and listed as following:

- *Offset for static members of classes, methods and fields.* They need to be restored accordingly on the runtime state during the code relocation. Under the Jikes RVM implementation, pointers of static methods and fields are stored in the JTOC (Jikes Table Of Contents) array [2]. The offset of the static methods and fields are determined by the order of the class loading process. A different execution instance would lead to a different order of class loading, as the static methods and fields of the loaded classes would end up in different locations of the JTOC array.
- *Offset of class objects.* For the same reason as above, the offset of the class object needs relocation. In the Jikes RVM implementation, the address of class objects are determined by the order of the class loading process during Java execution. Since lazy class loading is used in Java execution, the order of class objects to be created is a different order under different executions.
- *ID of classes and their members.* This is one of the representations aside of direct memory address used by JVM to reference to an object. These IDs are generated depending on the execution sequence when they are initialised in the runtime. Therefore, for the same reason as dynamic loading results into a different class loading sequence in different executions, IDs for classes and their members require relocation in binary loading.
- *Runtime data.* Runtime data consists of references to methods and fields that

belonging to the JVM runtime. The Jikes RVM is written in Java, so the runtime information is organised in the same way as data from a Java program. It can be argued that relocation is not necessary, because the core runtime information is pre-compiled into a booting image that every runtime data should fix in its location in different executions. However, we provide runtime data relocation, improving the image compatibility as the booting image of Jikes RVM may change in different versions of clients.

- *Offset of string and floating point constants.* Constant relocation focuses on floating point constant `CONSTANT_Float` [58] and string constant `CONSTANT_String` [58]. Based on the Jikes implementation, the value of floating point constants and the pointers of string constant are stored in the global data structure, JTOC, along with the class loading process. Instructions access such constants by referring to a direct offset in the JTOC table. However, the offset of those constants in the JTOC depends on the next available slot in the JTOC during class loading. Therefore, this JTOC offset of the string and floating point reference should be changed accordingly.

The relocation information we have mentioned above is not only limited to the reference in the class of the binaries requiring relocation. As there are references to the runtime data as well as inlined methods, references that are not listed in the constant pool of the target class need to be considered.

4.5.2 Link Point Layout

A *Link Point* defines a set of data that is used for holding a single piece of information for a relocation. As will be discussed in a later section, there are a number of different relocation types in the relocatable binaries. For each relocation slot, exactly one Link Point is created. The Link Point holds the semantic information of the value of the relocation slot, which will potentially be invalid under different execution environments.

The Link Point is created along with the compilation when each operand contains its own semantic information. However, when these operands are integrated as a part of the binaries after code generation, the operands can only be known as a constant value in the instruction. The semantic information of those operands is lost in the process of code generation. This implies that those values used in the binaries

cannot be retrieved again. The purpose of the Link Point is to record the semantic information of a location of binaries that allows the value on the specified location to be modified accordingly from the current state of runtime.

```
000003|      CALL           [470024A4]
```

Figure 4.3: An example of an instruction requires relocation

For example, given an x86 instruction as shown in Figure 4.3, the reference of the callee in this instruction is the memory address of the method, `[470024A4]`. However, `[470024A4]` is only valid in a particular execution where `[470024A4]` is pointed to by the entry address of the target method. In a different execution environment, without correcting the operands of the instruction `CALL`, it becomes invalidated.

Figure 4.4 shows the format of a Link Point. The semantic information of the relocation point can be generalised into three types. The types of information are listed below:

- *Symbol* serves the same functionality in library files as is defined in [55]. Since objects are defined as basic elements in Java, the term symbol refers to the definition of an object reference and is used to retrieve this reference. The symbol layout and symbol management details are given in Section 4.6.
- *Location* indicates where the relocation slot is in the binary. There are two fields of information to address a specified relocation address, the location type specified as a relocation data region in the linking input, and an offset address which point to the offset value inside the data region. For x86 instruction

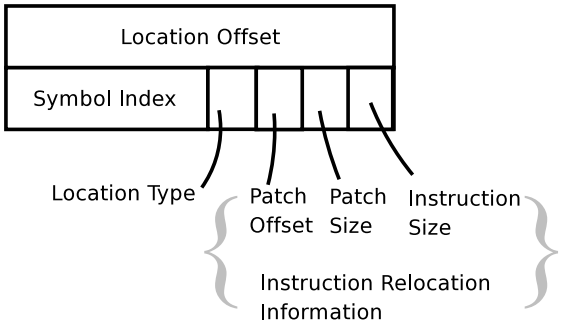


Figure 4.4: Link Point Layout

relocation, additional information about this instruction is given to avoid the complexity of instruction decoding while replacing the instruction operands. Details of instruction relocation are discussed in Section 4.6.

```
000003|      CALL           [470024A4]
        %% [METHODREF:< BootstrapCL, LA.foot; | OFFSET | INSTRUCTION(58)]
```

Figure 4.5: An example of an instruction with a Link Point attached

With a Link Point attached to the instruction in the previous example in Figure 4.3, Figure 4.5 demonstrates how the Link Point describes the semantic information of the instruction `CALL`, with symbol and location information recorded.

The Link Points are created during the compilation, and become a part of the relocatable binary as long as the binary of the compiled method is valid in runtime. It is a part of the Apus image to relocate the binary of compiled methods in different execution instances. In order to improve the relocation performance by reducing symbol resolution, a symbol can be shared between Link Points. The format for a Link Point therefore is a combination of an index referring to its symbol table entry and the location information.

4.5.3 Relocation Types

Relocation information as categorised in Section 4.5.1 is used in various situations in the compiled binaries. The following section discusses the possible types of relocation required in these situations.

The code relocation process is responsible for modifying binaries accordingly on the relocation information provided. The code modification is handled by the Linker in the code migration framework. For different relocation types, the modification process should react differently to the binaries. For example, the patch process for the instructions is different from the patch process on the associated runtime information of the binary.

4.5.3.1 x86 Instruction Relocation

Because of the complexity of the instruction format, relocating operands in instructions is somewhat trickier than relocating addresses and other values in associated runtime information of binaries. Despite the complexity of instruction encoding on the x86 architecture, from the linker's perspective, the x86 format is easy to handle, since the instruction architecture follows a universal format throughout all x86 instructions. The x86 series [45] instructions consist of optional instruction prefixes, primary opcode bytes, an addressing-form consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (optional), and an immediate data field (optional).

Instruction relocation is performed while interpreting entries of the relocation information table in Apus images. As Figure 4.4 shows, except for the instruction size which is used for debugging purposes, combining patch offset and patch size simplifies the relocation for the linker without knowing the instructions. The relocation information is generated along with the compilation of relocatable binaries.

In general, the instruction relocation operation is performed on a data slot in an instruction of the binary. All the necessary information required is described in a link point slot, as we have discussed in Section 4.5.2. The location information of the Link Point, which is the offset of the operand to be modified in the binary, is calculated in the compilation process along with the relocation information. As a consequence, the instruction relocation operation can just modify a data slot in value and relocation are described in a Link Point. That is, the code relocation process does not need to understand the structure of the instruction that needs to be modified.

There are two reason to simplify the instruction relocation steps by moving the instruction decoding from the linker to the compilation of relocatable binaries. Firstly, the Apus image may be used multiple times and it only takes one compilation cycle to generate an Apus image. Secondly, Apus images are not necessary generated during program execution, and compilation time is not a critical criterion any more.

The patch size is a safeguard for instruction relocation used as a measure taken during linking to prevent relocation data overflow from the relocation slot and even overlaping to the next instruction in sequence. A relocation overflow is the situation where the size of the data from the Link Point is larger than the available space in

```
000003|      JLE                  0                  | 7E00
000005|      PUSH              52[esi]                | FF7634
```

Figure 4.6: An example of a machine instruction sequence

the relocation slot. For example, in Figure 4.6 showing an x86 instruction sequence, the instruction `JLE` [45] only takes an 8-bit immediate (imm) data field, and writing a 16-bit value will certainly overflow the 8-bit imm data field and will also override the following instruction `PUSH` as well. If an overflow in the relocation slot is detected, the Apus image loading procedure should be aborted and `VM_ImageLinkingException` thrown. As a consequence, the optimized code for this method will not be available as the image loading fails. However, as we discussed in Section 3.5, the result of program execution should not be affected when the image loading fails.

Only having patch size to protect against relocation overflow is not enough. The machine code generator of the optimising compiler always tries to choose the most suitable instructions for the corresponding MIR (Machine code Intermediate Representation), and therefore a 8-bit instruction is chosen when the operand of the corresponding MIR is 8-bit, which could potentially cause a relocation overflow if the relocation data is larger than 8-bit. In order to prevent the situation when an instruction would cause a relocation overflow, the machine code generated is instructed to choose a 32-bit instruction for Link Points regardless of the operand size of the input MIR instruction. It can be argued that choosing a fixed size instruction would waste additional CPU cycles and increase binary size, however, this overhead is insignificant when compared with introducing the possibility of failure in the Apus image loading.

Instruction expansion is an operation to resize an instruction in the binary array. Although expanding an 8-bit instruction into a 32-bit instruction is theoretically plausible, in reality, instruction expansion requires shifting all of the binary after the expanded instruction for an extra space for the relocation slot. In additional, relative addressing used in calls and jumps are affected, and the machine code map used to convert the bytecode offset from the binary offset requires recalculation as well. In order to avoid such situation in the relocation process, choosing a suitable instruction with correct operand size for relocation is important during compilation.

Figure 4.10 demonstrates the compiled binary machine code with relocation in-

```
public void example() {
  multianewarray m[][] = new multianewarray[3][4];
}
```

Figure 4.7: Java program using a `multianewarray` instruction

```
public void example();
  Code:
   0:        iconst_3
   1:        iconst_4
   2:        multianewarray        #10,  2; //class "[[Lmultianewarray;"
   6:        astore_1
   7:        return
```

Figure 4.8: Bytecode instruction of the compiled program in Figure 4.7

```
********* START OF IR DUMP  Initial HIR   FOR < SystemAppCL, Lmultianewarray; >.example ()V
-13         LABEL0   Frequency:  0.0
-2      EG  ir_prologue             l0a(Lmultianewarray;,x,d) =
-2          guard_move              t1v(GUARD) = <TRUEGUARD>
2           guard_move              t3v(GUARD) = <TRUEGUARD>
2       EG  newarray                t2a([I,p) = [I, 2
2           int_astore              4, t2a([I,p), 1, <mem loc: array < BootstrapCL, I >[]>, <TRUEGUARD>
2           int_astore              3, t2a([I,p), 0, <mem loc: array < BootstrapCL, I >[]>, <TRUEGUARD>
2           guard_move              t5v(GUARD) = <TRUEGUARD>
2       EG  newobjmultiarray        l6a([[Lmultianewarray;,p) = [[Lmultianewarray;, t2a([I,p)
-3          return                  <unused>
-1          bbend                   BB0 (ENTRY)
*********   END OF IR DUMP  Initial HIR   FOR < SystemAppCL, Lmultianewarray; >.example ()V
```

Figure 4.9: The HIR translation of `multianewarray`

formation attached as comments from the program shown in Figure 4.7. Figures 4.8 and 4.9 show the bytecode of program in Figure 4.7 and the High-level Intermediate Representation (HIR) of the compiled bytecodes respectively. They are used to help understand the binary code shown in Figure 4.10 by showing the implementation of instruction `multianewarray` before the expansion of the runtime service calls.

4.5.3.2 Exception Relocation

The exception table is one of the associated piece of runtime information which contains the exception information, defined in a structure `Exceptions_attribute` as

```
000000|      CMP                  esp          40[esi] | 3B6628
000003|      JLE                    0                  | 7E00
000005|      PUSH              52[esi]                 | FF7634
000008|      MOV               52[esi]            esp  | 896634
00000B|      PUSH               18466                 | 6822480000
              %% [NONE | NONE | CMID | INSTRUCTION(12)]
000010|      ADD                  esp             -4  | 83C4FC
000013|      CMP              32[esi]              0  | 837E2000
000017|      JNE                    0                  | 7500
000019|      MOV                  eax              2  | B802000000
00001E|      MOV                  edx              2  | BA02000000
000023|      ADD                  esp             -8  | 83C4F8
000026|      PUSH                  12                 | 6A0C
000028|      PUSH          1191707124                 | 68F4010847
              %% [TYPEREF:< BootstrapCL, [I > | NONE | TIBOFFSET | INSTRUCTION(41)]
00002D|      PUSH                   1                 | 6A01
              %% [TYPEREF:< BootstrapCL, [I > | EPINDEX:10 | ALLOCATOR | INSTRUCTION(46)]
00002F|      PUSH                   4                 | 6A04
000031|      PUSH                  12                 | 6A0C
000033|      PUSH               19557                 | 68654C0000
              %% [NONE | NONE | SITE | INSTRUCTION(52)]
000038|      CALL           [47001E54]                 | FF15541E0047
              %% [METHODREF:< BootstrapCL, Lcom/ibm/JikesRVM/memoryManagers/mmInterface/MM_Interface;,
              %% allocateArray, (III[Ljava/lang/Object;IIII)Ljava/lang/Object; > |
              %% CONSTANTPOOL_INLINE:311(4) | OFFSET | INSTRUCTION(58)]
00003E|      MOV                  edx            eax  | 89C2
000040|      MOV                4[edx]             4  | C7420404000000
000047|      MOV                 [edx]             3  | C70203000000
00004D|      MOV                  eax          33621  | B855830000
              %% [METHODREF:< SystemAppCL, Lmultianewarray;, example, ()V >
              %% | STRING_APPLICATION_LOADER | ID | INSTRUCTION(78)]
000052|      ADD                  esp             -8  | 83C4F8
000055|      PUSH               -2265                 | 6827F7FFFF
              %% [TYPEREF:< SystemAppCL, [[Lmultianewarray; > | CONSTANTPOOL:10 | REFID | INSTRUCTION(86)]
00005A|      CALL           [47002D90]                 | FF15902D0047
              %% [METHODREF:< BootstrapCL, Lcom/ibm/JikesRVM/opt/VM_OptLinker;, newArrayArray,
              %% (I[II)Ljava/lang/Object; > | EPINDEX:155 | OFFSET | INSTRUCTION(92)]
000060|      CMP              32[esi]              0  | 837E2000
000064|      JNE                    0                  | 7500
000066|      ADD                  esp              8  | 83C408
000069|      POP              52[esi]                 | 8F4634
00006C|      RET                    4                  | C20400
00006F| <<< 000003
00006F|      INT                   67                  | CD43
000071|      JMP                    5                  | EB92
000073| <<< 000017
000073|      CALL           [470024A4]                 | FF15A4240047
              %% [METHODREF:< BootstrapCL, Lcom/ibm/JikesRVM/opt/VM_OptSaveVolatile;,
              %% OPT_yieldpointFromPrologue, ()V > | EPINDEX:149 | OFFSET | INSTRUCTION(117)]
```

```
000079|      JMP                  25                  | EB9E
00007B| <<< 000064
00007B|      CALL           [470024A8]                 | FF15A8240047
               %% [METHODREF:< BootstrapCL, Lcom/ibm/JikesRVM/opt/VM_OptSaveVolatile;,
               %% OPT_yieldpointFromEpilogue, ()V > | EPINDEX:151 | OFFSET | INSTRUCTION(125)]
000081|      JMP                  102                 | EBE3
*********   END OF:  Final machine code   FOR < SystemAppCL, Lmultianewarray; >.example ()V
```

Figure 4.10: The binary machine code translation of `multianewarray`

described by the `class` file format [58]. There should be at most only one exception table in the binary of a method. Each entry in the exception table represents an exception defined in the method. An exception entry in the exception table contains the binary offset to a `try` block, a starting offset of the `catch` block and the ID of the exception type.

The ID of the exception type is only available when the type instance exists, which also means that this presented class should already be loaded and linked. If the type reference in the corresponding exception table entry is not resolved in the exception table, the relocation procedure is responsible for resolving any referenced address by the symbol in the Link Point. However, considering the situation where the class loading on the exception type failed during the relocation process, the code migration on the related method of this binary has failed, and an `ImageLoadingException` is thrown.

The decision to resolve a exception type in the code migration process seems somewhat in contradiction to the class loading process described in the JVM specification [58]. It requires that when resolution is performed, any errors detected during resolution must be thrown at a point in the program where actions are taken by the program. However, the class loading error triggered by the exception table relocation should not interrupt the execution of the user program, but only interrupt the current progress of the Apus image loading.

4.5.3.3 Inline Table Relocation

The inline table is a part of the associated runtime information that encoded an inlined calling tree of the binary. The inlined calling tree is used to do reverse look up on calling stack from the current executing binary offset. The method ID is

used in the inlined calling tree to reference methods that are inlined. As we know from Section 4.5.1, the method ID is assigned by runtime while methods are loaded. Different execution environments assign different method IDs to the same method as the class loading sequence is different. As a consequence, relocation to the inlined calling tree is necessary. In order to modify the method ID in this inlined calling tree accordingly, decoding the inline table is necessary.

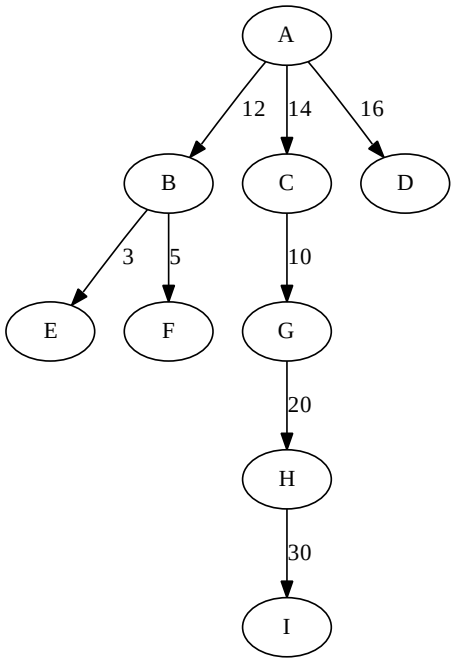


Figure 4.11: Example Inline Hierarchy

Figure 4.11 demonstrates the hierarchy of inlined methods in method `A`. Letters in the Figure represent the method reference of the inlined calling tree. The number along with the connection of nodes represents the bytecode offset of the call site.

According to Jikes RVM implementation, the definition of the inline table is described as following.

Three slots in the array are used for holding a inlined method. The information on each slot is listed respectively:

```
<(offset to the parent node) | bytecode offset of call site | callee method ID>
```

The `offset` slot that used to point to the array offset of the parent node is optional, and it should be only used on the first child node. The root of the tree starts with -1

as offset to the parent node. As an exception, there is no call site offset to the root node, as we already know it is 0. Nodes are listed as level-order as shown in the tree.

For example, the inlined calling tree described in Figure 4.11 is encoded as array as follows:

```
-1, A, -2, 12, B, 14, C, 16, D, -6, 3, E, 5, F, -9, 10, G, -2, 20 H -2 30 I
```

The location description of the relocation slot in the Link Point is the offset of the array offset of the inline table. The relocation approach on the inline table is similar to exception table relocation. An aggressive reference resolution approach is applied, in that methods referred to in the inline table are resolved and loaded during the inline table relocation. An `ImageLoadingException` is thrown and the image loading process stop, if any class loading related to the inline table relocation failed.

4.6 Symbols

This section discusses symbol and symbol management. They plays an important part in the linking process. A symbol is a representation of relocation information. The purpose of a symbol is that the linker can assemble relocation information regardless of any particular execution environment.

4.6.1 Representation

Relocation information is turned into symbols during the process of generation of relocatable binaries and written as a part of the Apus image. The simplest representation of a symbol can describe the relocation information in a string. However, the symbol resolution in string representation could introduce overhead to the code relocation process, as well as increasing storage overhead in the Apus image.

In order to minimize the cost of symbol resolution and storage overhead of symbols, the following section discusses some techniques that have been used in symbol representation to achieve such a goal. We break the relocation information into two categories by their characteristics.

4.6.1.1 References

Relocation information that refers to types, classes or their members is categorised as a reference representation. The relocation information can be, for example, the ID of a class instance, or the memory offset of a method instance. From this perspective, a reference representation then can be broken into a reference to a runtime object, namely `repType`, and a property of this runtime object, namely `linkType`. The property of a runtime object indicates the relocation information from the referenced object, such as ID or memory address.

As we know that, the constant pool of class files contains references used by the bytecode. It is the reference used in the bytecode requiring relocation in code migration. A references represented in the constant pool can be directly accessed by the constant pool entry index. Additionally, the reference resolution process can be avoided by keeping the references in the constant pool. It is therefore easy to concluded that space efficiency and speed can be achieved effectively by utilising the reference that exists in the constant pool of the corresponding class.

The following example shows in the relocation of a bytecode how a symbol representation can utilise the reference in the constant pool.

```
invokestatic    #27; //Method static_function:()V
```

The compiled machine instruction from the demonstrated bytecode instruction without inline optimisation applied is shown as below.

```
CALL          [47017FB0]                | FF15B07F0147
    %% [METHODREF:< SystemAppCL, Linvokestatic;, static_function, ()V > |
    %% CONSTANTPOOL:27 | OFFSET | INSTRUCTION(36)]
```

As we can see from the example, the representation of the Link Point method reference `SystemAppCL, Linvokestatic;, static_function, ()V` on the example is represented by entry index `27` in the constant pool.

Inlining optimisation can result in a situation where the reference in the relocation information does not belong to the constant pool of the class that requires relocation, in that a direct constant pool entry index to the class is not available. In this case, an additional reference to the class of the callee is used in the reference representation. The callee information of the inlined method can be accessed from the inline calling tree as we discussed in Section 4.5.3.3.

Constant pool do not store any runtime data. Due to the fact that only a handful of runtime data is used by optimising compilation for code generation purpose, an entry point look-up table is introduced for mapping the table index to references to VM runtime data.

A string based representation has not been chosen as the primary way to identify references. As we discussed, this is due to the inherent complexity involved in interpreting a string reference. Additionally, the increase in storage space necessary to store the former advocates considering an alternative method of representation. However, due to the limitation on using references from constant pool, (for example, the inlined calling tree does not hold all of the inlined callees). String representations for references are still used as a backup solution when the references in the constant pool that are accessible are not available.

4.6.1.2 Constants

The second category of the symbol is constant items. More specifically, they are string constants and floating point constants. The reason for relocation of constants is discussed in Section 4.5.1. In the common case, the constant symbols are represented in a similar way to reference representations. An entry index for the accessible constant pool is used. However, if the constant is the result of a constant propagation optimisation, in that this value is not held by any of the accessible constant pool (the situation that the propagated constant value is coincidentally equivalent to one of the values in the constant pool is considered here), the representation of the symbol therefore must be the constant itself.

4.6.2 Symbol Resolution

Symbol resolution is one of the critical steps in the linking procedure. Without symbol resolution, references to the VM objects are only a string, or a integer which is used to represent what the object is but not the object itself. Symbol resolution completes the process to convert those symbols into actual references to live objects in the VM.

Symbol resolution in our code relocation process is straightforward. Symbols are loaded during the Apus image loading process, and then instances of symbols are

initialised based on the data given by loaded symbols. Symbol resolution uses a lazy resolution technique that only conducts symbol resolving where it is triggered by requests from Link Points during relocation. The symbol representation is first resolved into a reference. The relocation value in the referred object is loaded according to the link type and stored into the symbol instance to be used by Link Points.

The symbol resolution process is divided into three states, which are recorded in the symbol instance. The first state *Loaded* indicates the symbol is loaded only. The *Resolved* status is only set once the symbol representation is resolved into a reference to objects. After the relocation value represented by the symbol is stored in the symbol instance, the symbol is marked as *Ready* to be used by Link Points. The relocation value is only available when the referred objects exists in the runtime. If the attempt to resolve a symbol reference to an object fails, it only indicates that the target object is not yet ready and the status of the symbol instance stays at *Resolved*.

For example, symbol `[METHODREF | CONSTANTPOOL:27 | OFFSET]` represents the method instance offset referred by constant pool entry 27. After the symbol representation is resolved during the second state, the symbol knows the method reference is a method reference:

```
< SystemAppCL, Linvokestatic;, static_function, ()V >
```

In the final step, unless class `invokestatic` is loaded, the symbol instance should stay at the state *Resolved*. Otherwise, the symbol is resolved and the value is loaded. The state of the symbol instance would marked as `READY`.

4.6.3 Symbol Table Management

The symbol table is kept as an array of table entries. Each symbol occupies an entry. A symbol is a unique reference to relocation information within an Apus image, in that there should be no other symbol pointing to the same relocation information. There is only one symbol table in the Apus image that is used by the Link Point table. The relation between Link Points and symbols is shown in Figure 4.12.

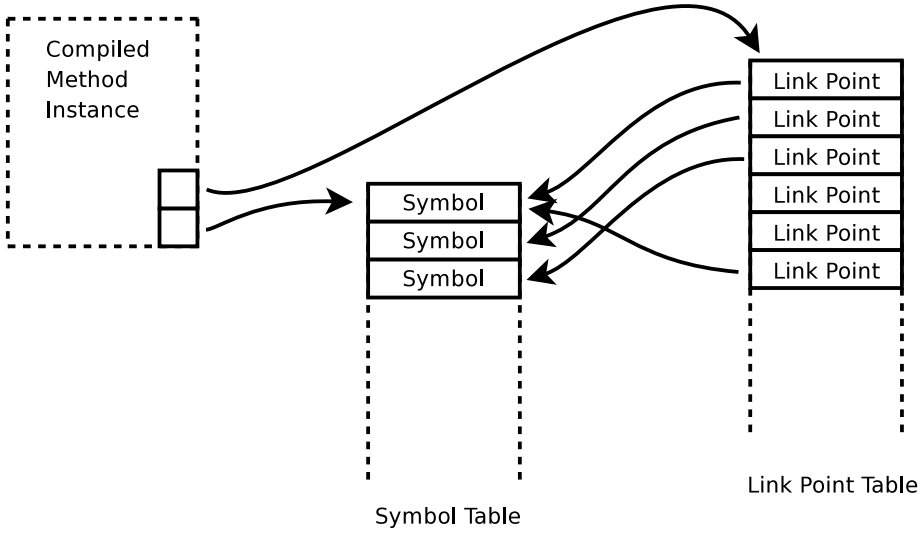


Figure 4.12: Relation between Symbol Table and Link Point Table

4.6.4 Reference Resolution

Symbol resolution involves resolving the symbolic reference into a VM runtime reference. The reference resolution is responsible for resolving those references into runtime objects such as an object of fields, methods, classes and interfaces. Values specified by `linkType` are retrieved from the referenced object. The value later is used to modify the relocation slot specified in the Link Point. An important requirement on the design of reference resolution, according to the JVM specification [59, p. 43–46], is that any errors detected during reference resolution must be thrown at the point when the program first actively uses classes or interfaces involved in the error.

The relocation process, based on its implementation, chooses to actively resolve references used by Link Points. That is, references which are not used by the Java program are resolved in the code relocation process regardless of when they will be used. Once the relocated binary is ready, any code that refers to this class or interface does not require further changes to adapt to the new execution instance. Consequently, code inserted by the compiler to validate reference resolution state and resolve references can also be removed during compilation.

If a reference resolution error is detected while loading the Apus image then an `ImageLoadingException` is thrown and the loading process is aborted. At this point, the compiler will decide how to handle the exception thrown by the image loader.

The latter process can potentially result in an increased amount of time taken to load the Apus image. However, this design has the trade off to avoid the complexity of handling other associated runtime information in the binary that is introduced by lazy resolution. The performance issue of the active reference resolution is discussed further in Section 6.6.2 where it is considered in the context of experimentation and measurement.

4.7 Apus Images

The Apus image is a container format designed to hold relocatable binaries. It is intended to serve as an intermediate data structure for migrating binaries for the proxy compilation service and for relocatable binary caching in the Apus proxy compilation system. The Apus proxy compilation system does not specified the Apus images as a file format, to be stored in a file system, since Apus image are being used across the network in the proxy compilation protocol.

In the following subsections, details of image generation, layout and management are discussed.

4.7.1 Image Layout

An Apus image contains a relocatable binary for only one method. This is due to consideration for the proxy compilation service.

It can be argued that putting relocatable binaries for multiple methods in a single Apus image would reduce the overhead of relocation information by sharing overlapped symbols. However, due to the fact that the compilation granularity for the proxy compilation server is a Java method, limiting an Apus image to only hold a relocatable binary for a method can reduce the space overhead that is used to identify methods in an image. Furthermore, considering a proxy compilation scenario, where the relocatable binaries should reply back to the client immediately after the compilation has finished, there is no obvious advantage for holding multiple methods in a single Apus image.

An Apus image file consists of a stream of 8-bit bytes. Multibyte data items are stored in big-endian order, where the high bytes come first. This format al-

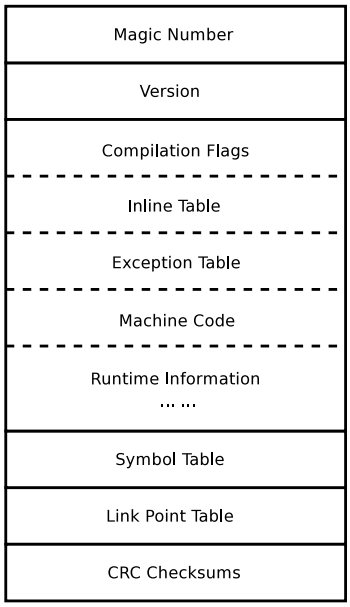


Figure 4.13: Apus Image Layout

lows the code migration framework to use J2SE interfaces `java.io.DataInput` and `java.io.DataOutput` to access the images. It also avoid the compatibility issues when delivering images across platforms.

The layout of an Apus image is shown in Figure 4.13. The first data block at the beginning of the stream is a 32-bit `Magic Number` that used to identify that the format is a legitimate Apus image format. `Version` is another 32-bit data block showing the version information of the code migration framework. It is encoded with the version number of the JVM that produced the images and the revision number of the Apus image format. The relocatable binary is stored after the version information. A compilation flag is a data field that holds information about the optimisation level of this binary. `Inline Table` and `Exception Table` are associated runtime information as we discussed in previous section about their relocation. `Runtime Information` contains data of associate runtime information that does not need to be relocated. `Symbol Table` and `Link Point Table` are stored at the end of the image. `CRC Checksums` are used to guarantee the integrity of the Apus images from unintended modification. The verification and security of the Apus image is discussed in Section 4.7.5

The code migration framework is designed to be for experimental purposes only.

In order to reduce implementation complexity, there is no need to provide downward compatibility with Apus images in previous implementations. The code migration framework can only work with compatible version of Apus images.

4.7.2 Creating Apus Images

The Apus images are created by an image writer, which is a part of the code migration framework. In the Java execution environment, the binary of a method is a object, like any other in the runtime. It is the image writer's responsibility to collect the associated information, and package the binary into a Apus image. Although there should be no restriction on when the Apus image should be created, according to the user scenario, there are two situations that we can consider. In the proxy compilation scenario, an Apus image is created right after the compilation is finished, and gets ready to ship back to the client. On the other hand, in the binary caching scenario, there is no urge for the execution environment to create Apus images from runtime objects, and store them in the local storage. It is understood that the relocatable binary caching process should not produce extra overhead on the execution. Apus images therefore are written to local cache after the exit of the Java program but before the exit of the JVM.

Based on the implementation, the Apus image is a binary stream handled by `java.io.DataOutput` and `java.io.DataInput`. Such an implementation can allow images to be easily adapted in the proxy compilation implementation as well as output to local file system as binary caching.

4.7.3 Apus Image Naming Scheme

In writing a relocatable binary into a local file system cache, a naming mechanism is needed to associate Apus image files with the corresponding methods. JVM uses a method signature to uniquely identify a method within a class. The signature of a method consists of the name of the method and the number and type of formal parameters of the method. Classes are identified by the fully qualified name and the class loader name. Therefore, a method can be identified uniquely within the Java name space by combining the identification of the declaring class and method signature.

The code migration framework uses a simple naming scheme for Apus image files representing a translated binary of methods in a Java class. In principle, the filename strings of Apus images s_{image} can be defined as follows:

$$s_{image} = s_{classloader} \cdot s_{package} \cdot s_{identifier} \cdot s_{signature} \cdot s_{suffix} \tag{4.1}$$

However, in order to reduce the length of filename strings, the class loader name on the classes loaded by the default class loader is omitted from the fully qualified image filename of methods.

The reason to use s_{image} explicitly is that the Apus image name is for debugging purposes, which allows users to find the corresponding method from the Apus image filename, although this decision introduces a challenge as a result of the maximum length of filename on the file system. We believe this challenge does not affect our study of code re-usability on dynamic compilation and any filename overflow on Apus images can be monitored carefully.

Although the challenge introduced by the naming system is not the first priority of our study, a more reliable naming system should provide an unique identity for each Apus image from the corresponding method and avoiding the limitation on the underlying file system. For example, a 128-bit MD5 hash code from a fully qualified unique name string from the method is used as the filename and the full qualified name string is stored inside the Apus image for verification purposes to avoid the danger of hash collision.

Given that the image name s_{image} includes the fully qualified package name, Apus image files are written into the primary image repository path specified by the `IMAGE_PRIMARY` environment variable. If `IMAGE_PRIMARY` is not set, the default image repository `./` (the current directory) is used by the code migration framework.

4.7.4 Searching Images

Searching for images is the process of finding the Apus image file for the corresponding method instance, during the image loading procedure. After an image is created, the linker has to be able to search it. The Apus image search happens during the first stage of image loading, after a loading attempt is initialised from the image loader.

The image loader searches for the corresponding Apus image for the input method instance by matching the image filename in the pre-loaded *image repository*. If there is more than one Apus images corresponding to the requested input, the first image found in the repository is used. If there is no matched Apus image found, than the loading process is abandoned and a failure is reported.

The image loader is designed to support multiple paths for image repositories. Image repository paths are defined by environment variables. The primary image repository `IMAGE_PRIMARY` is used as the primary path for searching Apus images, and also the destination for the newly produced Apus image files. Secondary image repository paths are only used for searching and loading Apus images. They are declared by the environment variable `IMAGE_REPOSITORIES` with ":" as a separator. Having multiple directories for the image repository allows Apus images to be spread over multiple locations. A possible usage for supporting multiple paths of image repository would be to allow using images that are pre-compiled from Java libraries and keep them in a separated location which can be shared by different executions of Java programs.

4.7.5 Verification and Security

Once Apus images are produced, measures have to be taken to prevent the risk of Apus images being unintentionally corrupted. An Apus image would be unintentionally corrupted due to, for example, the corruption of the file system or the alteration of data during transmission of files or as a result of a write error. Efficiency of Apus image loading is regarded as an important issue in code re-usability against dynamic compilation. Apus image integrity is protected by an efficient 32-bit checksum produced by CRC-32 and the checksum is verified in the image loader, as the last step of reading Apus image files.

There are also concerns over intentional modification of Apus images, in order to execute malicious code that can introduce potential harm to the host system, such as a computer virus. However, if the file system hosting the Apus image repository is compromised over the attack, there is no reason to add efforts to protect Apus images from malicious alteration. We can also assume that the host JVM may also be compromised in such a situation already. As a result, we can argue that implementing security measures, such as a digital signature against malicious attack

on Apus images is not necessary.

4.7.6 Performance Issues

One of the primary performance issues related to the use of binary caching is the time spent in scanning image files in the registered repository paths over and over again on each image loading attempt. From the discussion in Section 4.7.2, we can conclude that Apus image files are used by the different JVM instances apart from the one that generated them. As a result, it can be safely assumed that there should be no update in repository directories during the execution of Java programs or the updates can be ignored. The multiple image file scanning on repository directories therefore becomes redundant and the extra IO operation on directories can be avoided by caching the repository directories during VM booting. The image loader first builds a *image repository* on existing Apus image files based on the repository directories during VM booting, and image searching is therefore used to find Apus image files as requested.

Another performance issue is the time spent on reading and writing the input file, the Apus image. Although Java object serialisation [98] provides an integrated solution for objects to save state into a sequence of bytes, however, this process turns out to be slow and becomes a bottleneck of the image writing process. As described in design requirement, efficiency in writing and reading Apus image is important to the system. As a result, the image writer applies a specified simple parsing for Apus images.

4.8 Dependency Verification

Method dependency defines a constraint relationship between the relocatable binary and the associated method. The main purpose of the method dependency is to ensure the relocatable binary precisely reflects the original source file, the classfile. For example, given a compiled binary `B` from method `M`, a method dependency relation is defined as $M \rightarrow B$. Binary `B` stays valid only if the associated method `M` does not change since `B` has been generated.

If inline optimisation is used throughout the compilation, the relocatable binary

not only contains code generated from the input method, but also contains code generated from other methods inlined into the root method. The method dependency relation to determine compiled a method binary should therefore include inlined methods.

Classfile Message Digests (CMD) generated from the corresponding classfiles are used to verify the integrity of contained methods for the method dependency relationship to relocatable binaries. The classfile timestamp is chosen as the CMD representation for a classfile in our current implementation. Given that the security of the local file system is protected by the operating system as mentioned in Section 4.7.5, we can conclude that the timestamps for files are also equivalently reliable. The classfile timestamp has the limitation that it is only available to the Java class binary stored on the local file system. However, a further improvement on CMD can be made by hashing the class binary stream using 128-bit MD5 or quick CRC-32.

Given that classes cannot be changed after they are loaded into the VM runtime, the building of method dependency between the Apus image and methods can therefore be delayed until it is required. The CMD is generated from the corresponding classfiles before the Apus image writing, in order to avoid competing computing resource from the Java program execution. The CMD from the associated method and inlined methods of the Apus image are inserted into a database holding the class digests, along with the identity of the corresponding class, which is the combination of a fully qualified package name and the class identifier. This database is exported into a `classdigest.db` file and stored in the same directory as the Apus image files.

There is only one database used to store all the dependent CMD for the representative Apus images in the code migration framework. It is loaded during the initialisation of the code migration framework, which is employed as a part of the VM booting procedure. The image loader verifies the method dependency of the Apus image by comparing the CMD from associated and inlined methods and the corresponding CMD in the database. Once any CMD pair of the method dependency fails to match, the dependency verification fails and leads to the breakdown of the ongoing Apus image loading process. The Apus image is then deleted to avoid any further unsuccessful loading attempt.

The reason for centralisation of the database is to reduce redundant CMD information that would occur if they are distributively stored inside Apus images. The database do not store any method dependency relation for Apus images. It only

serves for the purpose of holding a digest of classfiles at the time when the Apus images are built. Hence, such information can be used with dependency relations described in Apus images to perform the dependency verification during image loading.

4.9 Producing Relocatable Binaries

In this section, we cover in more detail how the relocation information is produced along with the optimising compilation.

The relocatable binaries are a collection of machine dependent binaries with associated relocation and runtime information produced by the Apus optimising compiler. The Apus optimising compiler is a dynamic optimising compiler with additional operations to produce relocatable information for the binary. The reason to use a dynamic optimising compiler for code generation is discussed in Section 3.4.

4.9.1 Implementation outline of the Apus optimising compiler

The Apus optimising compiler is implemented based on the optimising compiler in Jikes RVM [2]. It takes advantage of the code optimisation ability of the compiler to produce a better quality binary for Apus images. The procedure of the compilation is shown in Figure 4.14. It can be primarily divided into three sections according to the IR (Intermediate Representation) level. Bytecode is transformed from HIR (High-level Intermediate Representation) into LIR (Low-level Intermediate Representation) and MIR (Machine-level Intermediate Representation) and finally into machine code. Optimisation is performed at each level of IR. The lines between major compilation phases are check points for validating Link Points handled by the compilation phase, LPVP (*Link Point Verification Phase*). More details of the LPVP phase can be found in Section 4.9.2.

Before machine code is generated from MIR, another LPVP phase is used to update invalid Link Point locations after the optimisation has been applied on MIR. Link points from the exception table and inline table from the compiled method are collected. Symbols are extracted from all the labelled Link Points in the compiled

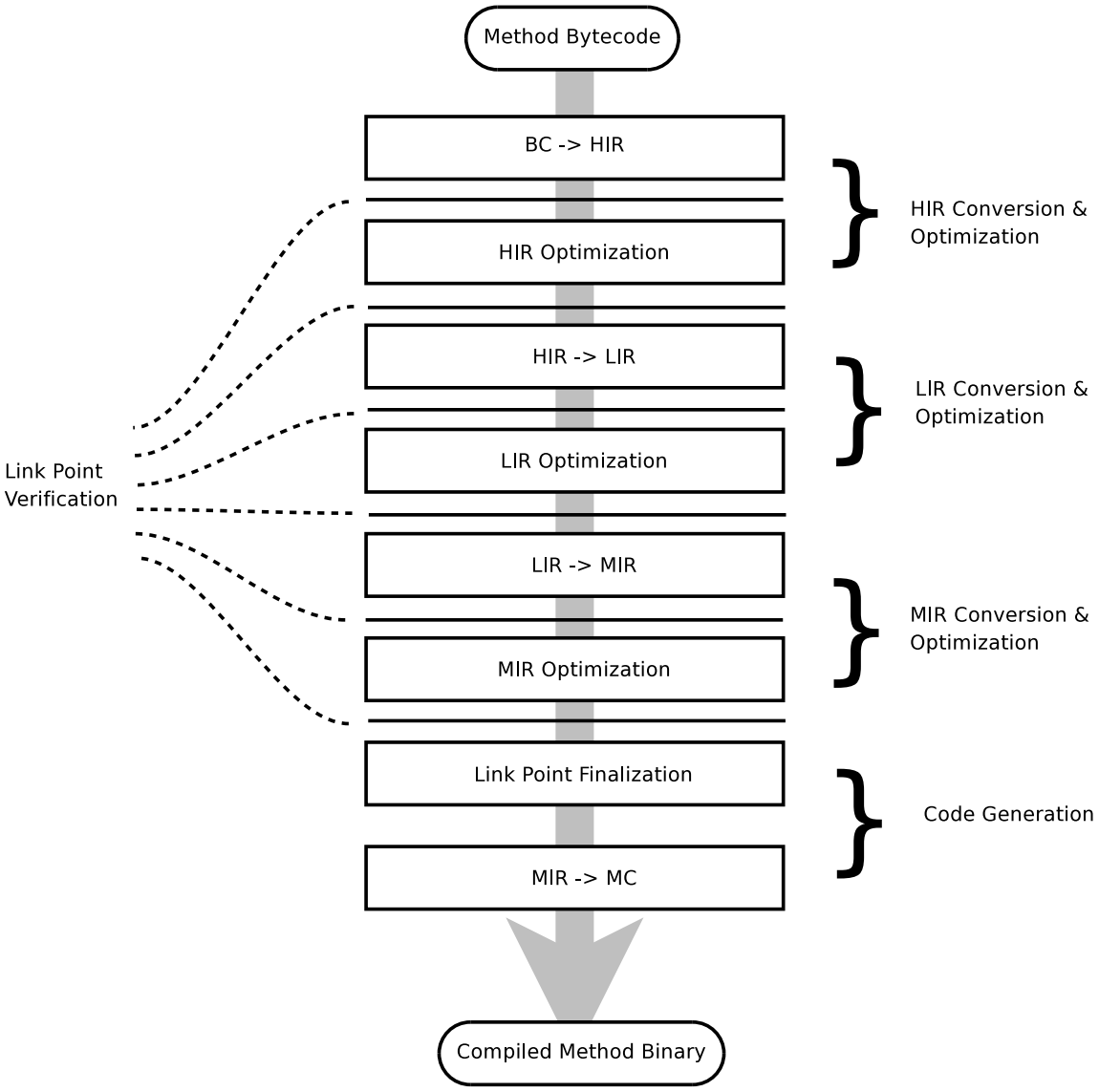


Figure 4.14: Simplified compilation procedure of the Apus optimising compiler

method, and then they are inserted into the symbol table and Link Point table respectively.

4.9.2 Link Point Tracking

Link Point tracking is a problem created by code optimisation. The problem is therefore only limited to Link Points in instructions. During IR optimisation, instructions may be mutated and sequences of instructions may be changed. As a result, labelled Link Points on the instruction are no longer valid if the labelled operand or instruction does not exist anymore. A Link Point would be invalid in two nonexclusive situations made by code optimisation. Due to the limitations of the implementation on the optimising compiler, Link Points would not fully follow instruction mutation as described above, and would lead to the loss of relocation information. Therefore, the following solution is applied to the problem.

If the semantic description of the labelled operand changes by optimisation, the associated Link Point becomes invalid. The invalid Link Points should be removed in the compilation. For example, applying a constant propagation operation on a floating point instruction can lead to a new floating point value by performing the instruction at compilation time. As a result, the two original floating point values are removed from the code, as well as the Link Point associated with them.

In the case that a labelled location of a Link Point becomes invalid because of instruction mutation, the check point for validating the Link Point is deployed between IR conversion and the optimisation phase to ensure the correctness of location information of the link point. Location information is composed of the label operand and the associated instruction. LPVP is used to recover the valid one of those two elements of location information. If any Link Point loses all the location information and becomes untraceable, as identified by any of the LPVP phases, the compilation output will not be valid for relocation purposes. The LPVP phase is setup as a guard to ensure the binary is valid for relocation.

4.10 Summary

This chapter introduces the design and implementation of the code migration framework to provide re-usability of binaries produced by dynamic compilation. It describes the concepts and architecture that accommodates various user scenarios and details how to construct a mechanism to provide code relocation and a format to store compiled binary and relocation information. The Apus image production and procedure loading details in this chapter show how to ensure the binary produced by dynamic compilation can be used under different execution instances systematically.

CHAPTER 5

Proxy Compilation Protocol (PCP) Design

Research is to see what everybody else has seen, and to think what nobody else has thought.

ALBERT SZENT-GYÖRGI (1893–1986)

Proxy compilation requires a mechanism to migrate binaries that are generated by the proxy compiler service to the user agent in client hosts. This problem was discussed in the previous chapter and a code migration framework was introduced to handle formatted binaries to deal with issues relevant to binary relocation, such as binary integrity and dependency verification to the corresponding bytecode.

Proxy compilation also requires a mutually agreed communication protocol between the proxy compilation server and the user agent. In this chapter, we will start with the design requirements for the proxy compilation protocol. Then the protocol specification is given with an example of a PCP session. Implementation of the proxy compilation protocol over the code migration framework is described, finally followed by a further improvement on the protocol.

5.1 PCP Design Requirements

The protocol itself should not attempt to understand the binary format, that may or may not be Apus images from the compilation server (although the protocol is implemented based on code migration framework), nor should the protocol should specify the compiler behind the proxy compilation server host. The purpose of the protocol is intended to permit a JVM user agent to dynamically access a compiler on a server host in a useful fashion.

As a result, the protocol should specify how to deal with the communication for proxy compilation for JVM user agents. As the purpose of migrating is online code optimisation cost, operating the protocol itself should be light-weight without out-weighing the code optimisation. Additionally, the initial intention of the proxy compilation system is to provide code optimisation for a resource-constrained environment, although it is the fact that the implementation of the system is based on Jikes RVM, which is only currently available for the desktop environment.

From what it is described above as the background of PCP, we can easily conclude that the design requirement in principle for PCP can be described as follows:

- *Functionality* - The protocol should specifically have to provide a proxy compilation service only. It also should be able to provide what is needed to manipulate a compilation with priority constraints.
- *Independent* - The protocol should not depend on any specific hardware, JVM, compiler or binary format.
- *Efficient* - The efficiency of the protocol is important, including both a time and be cost efficient. Firstly, the protocol should minimise a compilation cycle to be able to complete more compilation in a limited period. Secondly the protocol should be able to operate in a limited resource environment. Lastly it is important to reduce the data transmission cost from and to the user agent in the client host.
- *Extensibility* - The protocol should be such that any necessary changes can be made consistently and easily.
- *Consistent* - No requirements should conflict with each other.

5.2 Background

In this section, a number of existing protocol architectures are reviewed, based on the protocol design requirement described in the above section.

The Java programming language provide a basic communication mechanism: sockets [39]. It provides an endpoint of a bidirectional communication flow across an Internet Protocol-based (IP) computer network. Sockets provide a fast and efficient communication approach for network applications. However, the use of sockets requires the client and server to deploy some application-level protocol to encode and decode messages for exchange. The following subsection reviews some protocol description approaches and related protocols that can be used above sockets for network applications.

5.2.1 Backus-Naur Form

As a basic protocol description, Backus-Naur Form (BNF) can be used to formally describe context-free grammars, including Internet protocols. An extension of BNF, Augmented Backus-Naur Form (ABNF) [72] is especially focused on expressing formal system for bi-directioned communication protocols. It was adapted by many popular Internet protocols including HTTP and SMTP.

Newsome [65] successfully implemented a full specification of proxy compilation protocol used by MoJo to dynamically load Java classes at runtime. The context-free grammar for various messages used by the protocol is specified in BNF notation. While the protocol is intended to show efficiency in a resource-constrained environment, however, the protocol set up a strict request-reply system to prevent further compilation requests from the client until the server has completely supplied the request. That is, having parallel and priority requests requires multiple sessions from the same server. Furthermore, it puts a challenge on reducing the turn around time for a request in a strict request-reply situation. Another issue of the protocol in Newsome's design is that in the proxy compilation system in his thesis, the proxy compiler is the only compiler available in the runtime, therefore the class can be loaded only when the class has compiled completely. Consequently, it can be understood that compilation granularity is based on classes and execution only can progress further if the compilation requests are completed.

5.2.2 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) [94] is an ISO standard notation often used by telecommunication and computer networking, for the purpose of removing ambiguities in communication by providing a set of rules that describes a machine independent coding technique.

ASN.1, however can potentially achieve higher efficiency than ABNF over Packed Encoding Rules (PER) [112] that provides a more compact encoding. It is less attractive, due to only a few commercial tools [81, 69] supporting PER encoding in Java. Furthermore, since the JVM already provides a programing model for machine-independent communication, ASN.1 become redundant on the communication between JVMs. Moreover, parsing ASN.1 messages is not trivial due to the complexity of the notation rules.

5.2.3 Distributed Object Model

An alternative to this notation is a distributed object model for Java introduced by Wollrath et al. [111] using Remote Method Invocation (RMI). The RMI system allow a Java program to cause a procedure of an object, which it is located in a different virtual machine, to be executed without the programmer explicitly coding the details of the remote interactions. As an alternative to RMI, Voss and Eigenmann [106] have successfully implemented a remote optimising compiler based on Remote Procedure Call (RPC). And Sirer et al. [92] designed a distributed Java virtual machine including a distributed JIT compiler by using remote invocation. However, details of the protocol in their system are not discussed in the paper.

Firstly, RMI, however, does not co-operate well with our proxy compilation system where efficiency of protocol handling on the client host is essential. Firstly, RMI relies on a remote invocation mechanism where the reply is strictly after the completion of the request from the client. This feature potentially introduce unnecessary inefficiency while the client tries to send out multiple requests before receiving replies from the server.

Secondly, in order to apply proxy compilation over a RMI architecture, a *client stub*, *remote reference layers* and *transport* are needed to package and ship off the argument and return values of the remote calls. In such a system which relies on

proxy compilation and fast compilation, the cost of maintaining the protocol over RMI may be too high.

The Yhc web service [89] took another solution by applying web services [36] on an Haskell [44] online compiler that allows the submission of Haskell source code for compilation into Javascript in order to use the Javascript back-end without installation.

Regarding the purpose of providing a distributed object model for Java, web service is not distinguished functionally from RMI, but the difference lies in a more language-neutral, environment neutral programming model. The service mainly falls into two categories. The "Big Web Service" use the Simple Object Access Protocol (SOAP) [107] standard over Extensible Markup Language (XML) [108] to exchange domain-specified data. In contrast, Representational State Transfer (REST) [31] simplified interfaces to exchange such information over HTTP without an additional message layer such as SOAP.

However, although web service extends the distributed object model for Java over a language-neutral standard, the heavy overhead on processing a web service protocol is still as cumbersome as RMI for a small, language-aware protocol.

5.3 Proxy Compilation Protocol Specification

In this section, a specification of the protocol used in the proxy compilation system is given.

5.3.1 Basic Operation

Initially, the server host starts the PCP service by starting listening on port 969[1]. If the client host wants to connect to the PCP server, it starts a connection to the PCP service port. Once the connection is established, the server host sends out a greeting message to the client. And then the client host and server host of PCP can exchange messages until the connection is closed or aborted.

The PCP is composed of a series of commands. Each command consists of a case-insensitive keyword, possibly followed by one or more arguments. Apart from

[1]The port number is a random choice that does not collide with any well-known services.

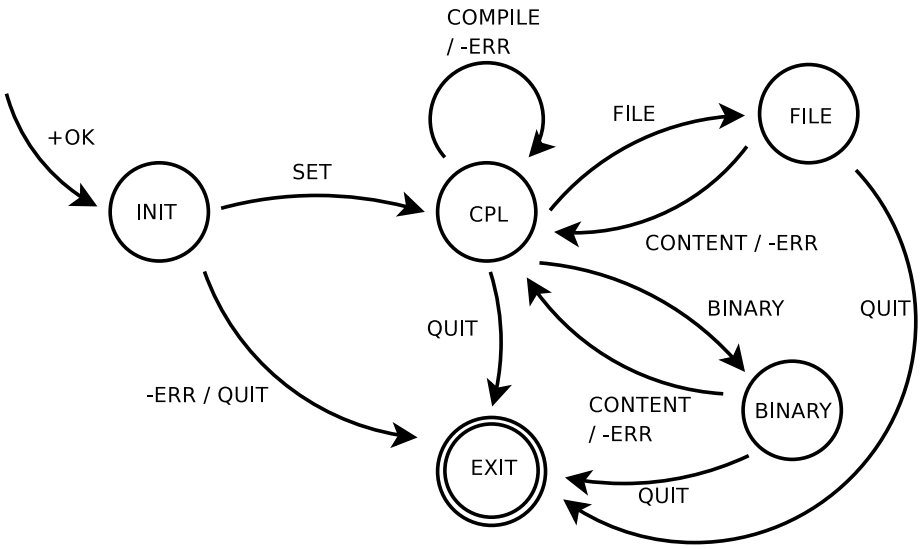


Figure 5.1: State Transition Diagram of the Proxy Compilation Protocol

commands with binary value arguments, all commands are single line, and terminated by a CRLF pair. Keywords and text arguments consist of printable ASCII characters. Keywords and arguments are each separated by a single SPACE character. Each command can take one binary argument. The binary argument consists of the argument length and the binary stream. Commands in a PCP session are bi-directional.

The response of a command must be a confirmation message, which consists of the same structure as a text command, a keyword and arguments. The response keyword is used to indicate the status of the request command. A text argument ending with `CRLF` is followed by a keyword. There are two possible responses. A positive response (`+OK`) indicate a success reply from the command, and a negative response (`-ERR`) indicate a failure.

Figure 5.1 shows the state transition diagram of the Proxy Compilation Protocol. A PCP session progresses through a number of states. Once the connection is established between the client host and the server, the server sends out a positive confirmation message as greeting. Then the session enters the `INIT` status, where the client host is able to setup the global configuration for the PCP session on the compiler server. Once the compilation initialisation is finalised, the PCP session enters the `COMPILATION(CPL)` status. In this state, the client host would be able to submit compilation requests and receive compiled binaries from the compiler server

until the client host issues a `QUIT` command. The `FILE` status is used to manage file exchange between the client host and server host, in the situation where the client host needs to submit a source file to the server. The `BINARY` state is used to receives compiled binary from the server. Once files or compiled binaries are received by the peer, the PCP session is changed back to `COMPILED` status. If the server host receives a `QUIT` command, it terminates the connection and recycles the allocated resources used in the session.

If an unrecognised or unimplemented command is received by either client or server, a negative response must be issued. A negative response also must be issued if commands are issued under invalid states. However, there is no indication in the recipient confirmation to distinguish a negative response in the cases of the sender being unable to comply with the command and not recognising the command.

The PCP server would have a keep-alive timeout limit, however, such a timer would need to have at least 10 minutes duration to determine if connection broken. If there is activity from the client during that interval, the keep-alive timer should be reset to the beginning. If the keep-alive timer times out, the server should send out a `QUIT` command and should finish the PCP session.

5.3.2 The INIT Status

Once the client host establishes a connection to the server host, the PCP server sends out a positive message to the client. For example, the greeting message from the PCP server can be:

```
OK  hello, dave!
```

After the greeting message is sent out, the PCP session begins with the `INIT` state. In this state, the client can use command `SET` to setup global properties for the compilation and send out application meta data in the PCP server. The hardware architecture and operating system in the client host are sent to the PCP server through the `INIT` state, as the server needs to select the appropriate compiler for the PCP session.

The PCP server then responds with a confirmation according to the properties set by the client. If the server is able to support all the requested properties from the client, it responds with a positive confirmation (`+OK`) and the PCP session is

progressed into the COMPILATION state. If the server is unable to comply with any of the requested properties, a negative confirmation must be issued and the current PCP session is terminated by the server.

In order to simplify parsing in the server, the argument of the command SET has to follow a certain format. A single property consists of a property name, immediately followed by the symbol equals "=" and the property value. If there is more than one property in the SET argument, a SPACE is used to separate the properties.

5.3.3 The COMPILATION State

Once the server has successfully supported the specification and acknowledged the setting from the client, the PCP session is now in the COMPILATION state. In the PCP session life cycle, the COMPILATION is the centre of the session. The client may now issue PCP compilation commands repeatedly. After each command from the client, the server sends a response. Eventually, the client issues a QUIT command to terminate the PCP session.

In the COMPILATION state, there are three primary commands to use to complete a compilation request from the client. The client issues the COMPILE command to begin a compilation request. This command can only be used in the COMPILATION state, otherwise, the server should respond with a negative response (-ERR) and take action according to the current state. If the command is accepted by the PCP server, the server replies with a positive response (+OK) and continue with an identifier in the message section as the unique task ID for the compilation request.

The COMPILE command takes at least three arguments, however, further advanced implementations may include other information, as parsed from the message. The arguments are the keyword of the command, followed by the implementation dependent method identity. An integer then follows to address the number of source files the target method depends on. However, this argument does not indicate the number of source files required for the target method in the program, but the number of source files the server should acquire before the compilation takes place. The last argument for the command is optional and is used to indicate the compilation priority.

In order to send a dependent source file for the target method, the FILE command is first used to inform the PCP server of the source file information. A FILE command

consists of the task ID corresponding to the source file, a string representing class identity, which is implementation dependent, and date and file size are sent as a sequence. The PCP server responds with a positive command (`+OK`) to accept the source file, followed by a source file digest from the PCP server if it exists. If a positive response is received from the `FILE` command, the PCP session changes to the `FILE` state.

After the compiled binary is ready from the PCP server to send back to the client, the PCP server issues the `BINARY` command with the job ID as the single argument of the command. If a positive response is received, the PCP session enters the `BINARY` state. However, action for a negative response for `BINARY` is not specified.

Once the `COMPILE` command is issued, the client can expect to receive the corresponding `BINARY` command for the job at any time in the `COMPILATION` state, after all the dependent source files have been sent.

5.3.4 The FILE State

Once the PCP server receives the `FILE` command, the PCP session is now in the `FILE` state. The server should then be ready to receive source files described in the `FILE` commands. Two possible command may come from the client in this state. The `CONTENT` command carries the content of the specified file and the `SKIP` command is used to indicate that the specified source file in the PCP server is valid and the current file transferring is skipped.

If the source file is transferred to the PCP server successfully, the PCP server should therefore reduce the number of source files required by the request compilation task. And the PCP session consequently goes back to the `COMPILATION` state, after a positive response for the corresponding command. For example, the dialogue for transferring a source file to the PCP server is shown in Figure 5.2. Messages sent from the client host are prefixed with `C:`, conversely, `S:` represents messages sent by the PCP server.

5.3.5 The BINARY State

The `BINARY` state server has a similar responsibility to the `FILE` state server to manage transferring binary data between server and client, but here the binary data

```
C: FILE 3021 A 1243008542000 457
S: +OK
C: CONTENT <file A.class>
S: +OK
```

Figure 5.2: An example of transfering a source file to PCP server

flows from the PCP server to the client host. Once the `BINARY` command is received by client, the PCP session enters the `BINARY` state, and a `CONTENT` command is expected to transfer the compiled binary. If the binary is successfully received by the client, the request compilation task is now complete and it should then be removed from both sides. The PCP session is then put back to the `COMPILATION` state.

However, a failed compilation attempt in the PCP server should report the failure by using `BINARY` command, followed by a negative response (`-ERR`), with specified error message to indicate the requested compilation task failed, informing the client about the failed compilation task. The compilation task should also be removed from both sides after the failure is reported to the client.

5.4 ABNF Grammar

In the previous section, the details of the state transition protocol is given. Figure 5.3 describes the context-free grammar of the protocol of proxy compilation. The grammar specifies the formal structure of messages used in the communication between server and client for the proxy compilation service.

Note that the `class-name` and `method-name` are strings which are used to carry the unique identities of the specified class and method respectively.

5.5 Implementation Details

The implementation details and a number of specified issues about the PCP implementation are addressed in this section.

In order to evaluate the design of the protocol and evaluate the effect of applying proxy compilation for the Java programming language by using the code migration

```
commands = set / compile / file / binary / content /
           skip / quit / response

response = ("+OK" / "-ERR") [SP message]

quit = "QUIT" [SP message]

set = "SET" properties CRLF

compile = "COMPILE" method-name SP dependent-files
          [SP priority] CRLF

file = "FILE" task-id SP class-name SP file-date SP
       file-size CRLF

binary = "BINARY" task-id CRLF

content = "CONTENT" length data CRLF

properties = 1(property-name "=" property-value)
             *(SP property-name "=" property-value)

property-name = *VCHAR

property-value = *VCHAR

message = *CHAR

class-name = *VCHAR

method-name = *VCHAR

file-size = 8HEXDIGI

file-date = 8HEXDIGI

task-id = 4HEXDIGI

priority = 2HEXDIGI

length = 8HEXDIGI

data = *OCTET
```

Figure 5.3: ABNF Grammar for the Proxy Compilation Protocol

framework, a PCP client and server pair have been implemented. Both client and server of PCP are implemented in the Java programming language, in order to integrate with the adaptive compilation system of the JVM in the client. The PCP client is implemented as one of the available compilers for the adaptive compilation system.

5.5.1 Cross Compilation Consideration

The PCP client and server are based on the same JVM under identical software and hardware platforms. The cross platform proxy compilation is not supported in the current implementation of the proxy compilation system. Although the decision to solely support a single architecture is made reluctantly, it can easily be argued that the same proxy compilation procedure can be applied no matter what underlying architecture is used, as a result, the impact of proxy compilation based on the code migration framework can be evaluated regardless of the hardware architecture. In addition, implementing a fully functional JVM with the code migration framework for other systems was impractical given the time and resources available for this research.

5.5.2 Concurrency Consideration

One of the important aspects of the PCP specification is to reduce the time for a complete compilation cycle by breaking the compilation procedure into a sequence of state transitions. To use the command `COMPILE` to submit a compilation request does not require the client to wait until the compilation result is sent back from the server.

This mechanism allows the client to maintain multiple proxy compilation requests in a single PCP session. As a result, the turnaround time for a proxy compilation request can be minimised by synchronising the bytecode compilation and message handling for the protocol in the PCP server.

In the prototype implementation of the server, a dispatcher thread is assigned to manage the PCP session to the client for dispatching the compilation request and delivering the compiled binary to the client, and another slave thread is used for the compilation.

5.5.3 Acknowledged Source File

In principle, the client should avoid exchanging the source files[2] to reduce transmission cost.

The PCP specification does not require a dependent relationship between the compilation target and the corresponding source file. The client agent can verify the application with the PCP server by using the `SET` command during the `INIT` state. If the application is verified by the server, the source file exchange between client and server can be completely avoided.

The information used to verify the application by the PCP server is a unique application ID which can be embedded into the "METAINF/MANIFEST.MF" of the application package by the application vendor. Although the unique application ID would not provide a sound relationship between the application and the source, it can be argued that the dependency information would prevent the binary from a different source file to be executed by the client agent. Furthermore, if the application is changed unaware, the expected result would be faulty and there is no reason whatsoever to need proxy compilation service.

The unique application ID in the prototype implementation is Java 128-bit UUID [51]. However, due to its lack of support in the GNU classpath 0.92 used by the prototype implementation, omit the standard algorithm to generate the UUID and generate it from four random integers.

5.5.4 Source File Handling

The efficiency of the proxy compilation can be increased by minimising the source files re-transmission between client and server, if the application cannot be identified by the PCP server as in the last subsection. The client issued source files are stored in the source repository in the PCP server. Prior to sending out the content of the source file, a `FILE` command is used to send out the detailed information on the specified file. If the detailed information is matched with the file in the source repository in the PCP server, a digest of this source file is sent back in the message section of the positive response to the previous `FILE` command. If the file digest

[2]The source file referred by proxy compilation system is the .class file contains application bytecode.

provided by the PCP server matches the file in the client, then the client can issue a `SKIP` to allow the PCP server to pick up the specified source file in the local repository.

In order to reduce the overhead of verification, the prototype implementation uses the same approach to produce a digest of the source file for dependency information on the Apus image (Section 4.8), the last modified date of the file is used as the digest for verification.

5.5.5 Binary Caching

The binary of the application produced by the PCP server is the Apus image that contains relocation information to allow the binary to be used in different execution instances. Thus, the client agent stores the binary from the PCP server in a local binary cache in the image repository and it can be used afterwards.

Since the current prototype implementation only supports a single architecture, there is therefore no need to have separate image repositories for the PCP server. Since the compilation has been cached into the Apus image in the image repository in the PCP server, the turnaround time for individual requests for proxy compilation can be reduced.

5.5.6 Priority

In the PCP prototype implementation, each request for proxy compilation comes with a priority value. A queue is implemented in both the client and server to handle incoming requests with priority. Each node in the queue represents a request. The first element of the queue should always have the minimum priority value and should be processed first.

5.5.7 Error Handling

A very simple error handling mechanism is applied in the prototype implementation. As it is addressed in the PCP specification, if there is a negative response received in the `INIT` state, the connection is terminated. If there is a negative response in

the other state, the PCP should send out a `BINARY` and a (`-ERR`) respectively to terminate the corresponding task.

Once the client VM enters the exit phase, the client agent issues a `QUIT` command to the PCP server and threads relative to the PCP client agent are marked as daemons.

5.6 Example PCP Session

To illustrate a complete proxy compilation session on the implemented prototype, an example of dialogues between client and server is given in Figure 5.4. As the example demonstrates, the client compiles methods `void foo1()` and `void foo2()` of class `example.Foo` by exercising the proxy compilation service.

Messages sent from the client host are prefixed with `C:`, conversely, `S:` represents messages sent by the PCP server.

5.7 Security Considerations

The PCP does not provide any form of security support to guard against malicious actions of both client and server. From the discussion in [21], the main security threads on code migration by PCP can be categorized into:

> Tampering - the unauthorised alteration of information. This would potentially endanger the binary to be modified with malicious code against the client host.
> Vandalism - the unauthorised access to the proper operations on the system.

An extension to use cryptographic protocols that provide security for communications over sockets, such as Transport Layer Security (TLS) and Secure Sockets Layer (SSL), can allow the proxy compilation system to elimiate threads described above. Furthermore, regardless which specific security policy is used, it would be interesting to measure the impact of the additional procedures over the system on time and footprint on the cilent.

```
S:  <wait for connection on TCP port 969>
C:  <open connection>
S:        +OK hello, dave!
C:        SET vm=2.4.6 clp=gnuclasspath-0.92 arch=x86-32
S:        +OK
C:        COMPILE Lexample/Foo;foo1()V 1 0
S:        +OK 1
C:        FILE 1 Lexample/Foo 1243633850000 449
S:        +OK
C:        CONTENT <example/Foo.class file content>
S:        +OK
C:        COMPILE Lexample/Foo;foo2()V 0 0
S:        +OK 2
S:        BINARY 1
C:        +OK
S:        CONTENT <binary for Foo::foo1()>
C:        +OK
S:        BINARY 2
C:        +OK
S:        CONTENT <binary for Foo::foo2()>
C:        +OK
C:        QUIT
S:        +OK client quit
C:  <close connection>
S:  <wait for next connection>
```

Figure 5.4: An example of a Proxy Compilation Protocol session

5.8 Summary

This chapter presents the motivation, design requirement and specification of the Proxy Compilation Protocol to be used within the Apus proxy compilation system to provide code migration with fine granularity for the Java programming language. A number of issues regarding the improvement of efficiency and reliability of PCP are detailed. The evaluation together with analysis on the data collected from experiments on the PCP are shown in Section 6.6.3.

CHAPTER 6

Evaluation and Results

The strongest arguments prove nothing so long as the conclusions are not verified by experience. Experimental science is the queen of sciences and the goal of all speculation.

ROGER BACON (1214–1294)

In Chapters 4 and 5, the design and implementation of the code migration framework and the PCP to provide a proxy compilation service were discussed. In order to verify the design, the performance of the implementation relative to the existing Java Runtime Environment must be demonstrated.

This chapter details how the prototype implementation of the Apus proxy compilation system was evaluated in typical user scenarios, using different suites of benchmarks, and provides analysis of the results obtained. Section 6.1 describes the test objective for the experiment. Sections 6.2 – 6.5 briefly introduce the characteristics of the selected benchmarks, measurement techniques and the testing environment for the following experiments. Testing results and analysis of the Apus proxy compilation system are shown in Section 6.6.

6.1 Test Objective

The fundamental purpose of writing the Apus proxy compilation system for Java is to show that, the design and implementation scheme outlined in the previous chapters has been successfully applied to produce a complete working prototype, which can verify our assumption of improving compilation efficiency by applying a dynamic compiler as a proxy compilation service via code re-usability. However, the purpose of our examination process is then to establish the efficiency of this implementation in a series of application scenarios, more specifically, in terms of execution speed and memory footprint. The definition of an efficient implementation is naturally subjective, so a list of qualities is given to quantitatively analyse the efficiency of the implementation of a system taking the advantage of code re-usability. The extent to which the implementation of the Apus proxy compilation system is efficient should be considered with respect to the following objectives:

- The use of the code migration framework as a replacement for dynamic compilation in regular Java program execution should not result in significant performance penalties when compared to the JVM without the code migration framework deployed.
- An Apus image generated from compilation should preserve the optimisation across execution instances, that is, most of the optimisation applied in the Apus images should bring performance benefits to the execution instance using them.
- The proxy compilation client should make a balanced decision over when to use different available compilers, including proxy compiler, cache and baseline compiler. The essence of this objective is that cost and benefits of using the proxy compilation service should be justified.
- When Apus images are accumulated as a result of multiple execution on the same Java program, the execution performance of this Java program should be improved compared with the previous execution.
- Use of proxy compilation should bring code optimisation into JVM without extra consumption of resource. That is, the use of networking to complete a compilation request should not outweigh the cost of performing equivalent dynamic compilation in a local optimising compiler in terms of performance penalties on execution speed and memory consumption.
- Generating Apus images while in execution should not introduce large per-

formance penalties in dynamic compilation, neither should it require large resource consumption overhead, such as memory. That is, producing Apus images during Java program execution should not significantly slow down the execution.

6.2 Choice of Test Programs

Fairness is one of the important aspects of selecting benchmarks. The Apus proxy compilation system is designed to compile with the JVM specification, and it is built on a JVM that has good support for the standard Java API. This implies that a widely available range of existing benchmarks for JVM are available for the Apus proxy compilation system.

Each benchmark described in Table 6.1 is compiled with the `javac` compiler provided by Sun's Java 2 Standard Edition v1.4.2, because the current version of Apus prototype implementation is only compatible with the Java 1.4 classfile format.

We evaluate our system using the SPECjvm98 [20] benchmark suites and the Scimark 2.0 [80] benchmark suites. Table 6.1 provides a short description of each benchmark, the number of classes, methods and bytecode instructions that comprise the benchmarks, and the size, in bytes, of its class files.

SPECjvm98 is a benchmark suite containing a range of tests to simulate conventional applications. The simulated applications are described in Table 6.1. Since it covers a wide range of applications and was used in many other publications, it is therefore considered to represent fairly the overall performance of the system and the results can be fairly compared with other JVM implementation. Each individual benchmark in SPECjvm98 reports the running time in milliseconds of a single execution of the benchmark as the result.

With regard to the consideration of the effects of optimisation produced by dynamic compilation that optimise code based on the execution environment, tests consequently focus on the performance of optimised code loaded from Apus images.

In order to evaluate the effectiveness of optimisations provided by proxy compilation service we wish to focus on, performance of executing result code should isolated from external effects. Therefore, I/O and automatic memory management operations during the benchmark would potentially offset the effect of code optimisation. Sci-

Benchmarks	**Description**	**Number of Classes**	**Number of Methods**	**Number of Bytecodes**	**Size of Class-files(bytes)**
SPECjvm98 Benchmarks					
_201_compress	An implementation of modified Lempel-Ziv method (LZW)	12	44	1727	35678
_202_jess	Java Expert Shell System based on NASA's CLIPS expert shell system	150	684	18096	396536
_209_db	Performs multiple database functions on memory resident database	3	34	1514	10156
_213_javac	Java compiler from the JDK 1.0.2	171	1170	40962	569654
_222_mpegaudio	decompresses audio files conforming to the ISO MPEG Layer-3 audio specification	51	311	33562	120182
_227_mtrt	two threads each render the scene in the input file time-test model	25	176	6367	57000
_228_jack	A Java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS)	56	315	18515	130889
Scimark 2.0					
FFT	Fast Fourier transform	1	10	541	2718
SOR	Jacobi Successive over-relaxation	1	3	116	580
MonteCarlo	Monte Carlo integration	2	10	783	3522
SparseCompRow	Sparse matrix multiply	1	3	72	510
LU	Dense LU matrix factorization	1	11	499	2166

Table 6.1: The set of benchmarks used to evaluate the Apus proxy compilation system

Processor	Two Intel Xeon Quad Core 2.00 GHz
Processor L1 Cache	32KBytes
Processor L2 Cache	4096KBytes
Address Size	38 bits physical, 48 bits virtual
Physical Memory	8 GBytes DDR2
Storage	ext3 mounted on local SCSI disks
Operating System	CentOS
Linux Kernel	2.6.18 SMP
Based Jikes RVM Version	2.4.6
Classpath	GNU Classpath 0.92
Javac	Jikes Compiler 1.22
Bootstrap Compiler	Sun Java SDK 1.4.2
Connectivity	Broadcom Gigabit Ethernet

Table 6.2: Test environment details

mark 2.0 benchmark suites are specified for scientific and numeric computing tests that run on small test data. It is ideal for exercising the proxy compilation implementation on the computation-intense workload. It measures several computational kernels and reports results for each computation and a composite score in approximate Mflops/s.

6.3 Testing Environment

The client and server for the proxy compiler were built and run on two blade servers with identical specification. The hardware and software specification of those machines is shown in Table 6.2.

In each proxy compilation test, we assign one machine to provide the proxy compilation service and the other machine was used as the proxy compilation client in all tests. The proxy compilation server and client are wired directly to a Cisco gigabit Ethernet switch. Naturally, the client should normally have lower hardware specification than the proxy compilation server. A high specification proxy compilation client would have potential differences in the areas of overall running time and number of compilation requests.

```
JNIEXPORT void JNICALL Java_Memory_meminfo (JNIEnv *env, jclass this){
    char command[100];
    sprintf(command, "cat /proc/%d/status", getpid());
    system(command);
}
```

Figure 6.1: JNI code to print out the memory consumption information to standard output in C.

6.4 Measurement Techniques

The measurements for the tests used in the following section are primarily categorized into four types, compiled binary size, memory consumption, process time and benchmark scores. In this section, those measurements are explained and techniques of measurement are detailed, in order to help the understanding of the following results. The simplest of measurement is the scores provided by the benchmarks. It can be simply read from their standard output.

Measurement of program memory consumption relies on the information from the process information pseudo-filesystem provided by the Linux operating system. The process information pseudo-filesystem is used as an interface to access kernel data structures. It is commonly mounted at `/proc`. For a given process id (pid), file `/proc/pid/status` contains the total program size, data and stack segment size of the program together with other information. In all test programs, the memory consumption refers to the physical RAM, no swap space should be needed.

A Java Native Interface (JNI) call `meminfo()` is used to retrieve the current pid and print out the contents of the file of the corresponding process, since standard Java library does not provide direct access to the current process id. This JNI call `meminfo()` is inserted into the end of the `main` function in each benchmark suite mentioned previously in Section 6.2. The code for `meminfo()` is shown in Figure 6.1.

The process time measures the execution time of a certain region of code in a specified Java thread. It is obtained by measuring the accumulated interval of the specified Java thread. Therefore, it can truly reflect the elapsed active time spent by the Java thread on the specified code region, without being affected by other threads or other parallel processes.

For each test configuration, the benchmark was run three times. In order to reduce the likehood of the effectiveness of cache, the median value of those three runs is chosen, for every measurements described in this chapter. Minimised system load on both server and client host is ensured. However, there is minimised network traffic between the server and client host. Despite the fact that the deployed test environment is essentially identical in each test, the result of the tests still varied slightly, with at most 5% difference from each other, due to system caching, or other kernel activities which are difficult to control.

6.5 Building the System

The implementation of Apus proxy compilation system is based on the Jikes RVM version 2.4.6. It means that the logical configuration of the Jikes RVM is a significant factor in the results. The booting image of the Apus client is built on the `production` configuration with full optimising compiler and adaptive system built in. The binaries of the booting images are produced using the high optimisation level which has all possible optimisations Jikes RVM provided turned on, that is, it is the best built configuration for Jikes RVM to deliver best performance on the testings. This built is used in the tests shown in Sections 6.6.1 and 6.6.2.

The booting image of both the proxy compilation server and client should also be built under such configurations. However, the irresolute bug[1] found in the Jikes RVM socket stream implementation inserting random bytes in the socket communication between the client and server, leads to the breakdown of the PCP as a consequence. The proxy compilation tests have to use the JVM built in `prototype-opt` configuration. Under the `prototype-opt` configuration, the only different from the `production` configuration is that the booting image contains non-optimised code built by the baseline compiler. As a result, the VM runtime cannot deliver the most efficient support and requires further optimisation during execution. This build is used in tests in Sections 6.6.3 and 6.6.4.

[1]The identical error has been raised in the mailing list in previous version of the Jikes RVM. The description of the error can be found in http://osdir.com/ml/java.jikes.rvm.devel/2004-02/msg00065.html.

6.6 Results and Analysis

The following sections show the results of running benchmark programs in various configurations and with different problem sizes in some situations. In each test, a test strategy is given to describe how to conduct the test to evaluate the proxy compilation system according to the test objective. Accompanying each table and graph of results, highlights of some key observations about the data are given.

6.6.1 Optimisation Migration

To understand the effectiveness of applying the code migration framework integrated with an adaptive optimisation system as the replacement for the optimising compiler, a number of experiments were carried out as follows.

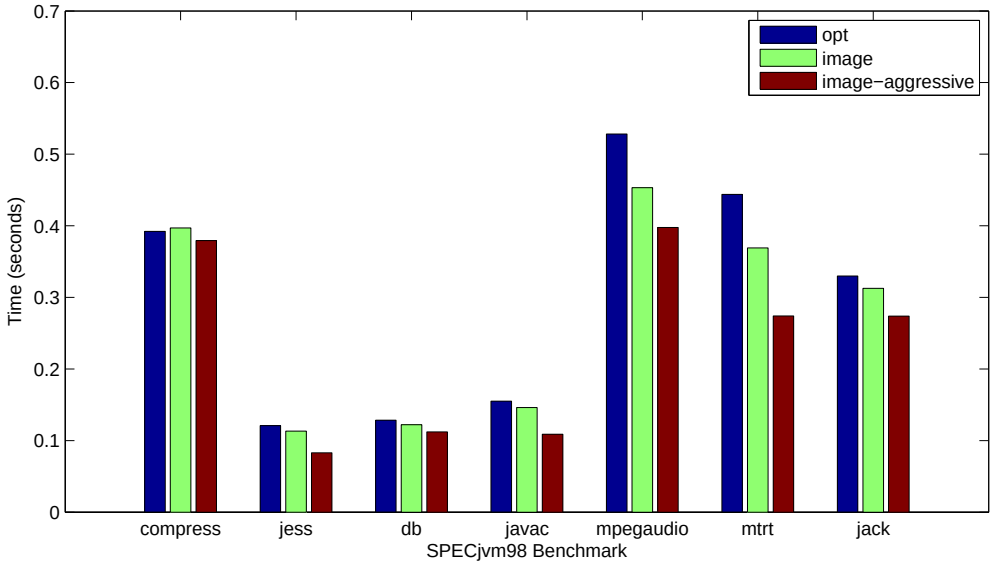
The results from this experiment, presented in Figure 6.2, depict the effects of the average execution time by using relocatable binaries which produce in a different execution instance on seven SPECjvm98 benchmarks. In each test, benchmarks are run 100 times using input size 10. The x-axis presents the name of each benchmark being tested. The y-axis presents the average execution time of a run in the benchmark for a certain period. All the benchmarks run on the default memory configuration, except `db` requires additional memory with maximum stack size set to 512MB.

Figures 6.2 divides the entire execution period into two parts. Figure 6.2(a) shows the *startup* regime which consist of the first 10 runs of the benchmark. In contrast, Figure 6.2(b) shows the *steady* regime which represents the period between the 10th run to the finish of the benchmark.

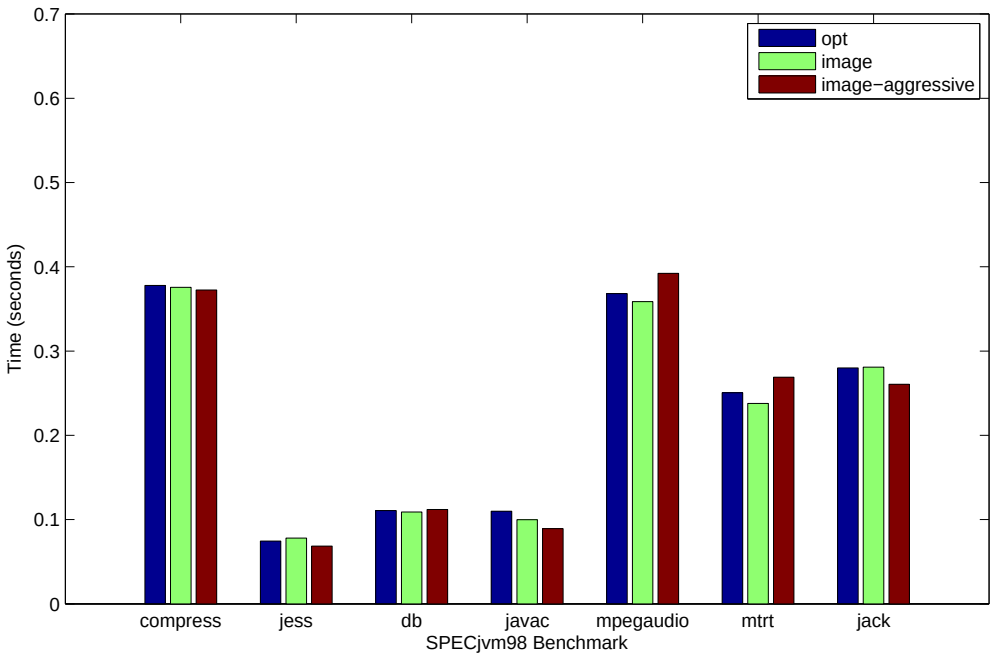
For each benchmark, three bars are shown: each bar represents a configuration on JVM. We are using the following configuration for the dynamic compilation in JVM:

- *opt* optimising compiler with adaptive optimisation system.
- *image* code migration framework and optimising compiler with adaptive optimisation system.
- *image-aggressive* Using Apus images to replace baseline compilations without the decision from adaptive optimisation system.

Throughout the tests, the first time to run the benchmark is used as the production



(a) Startup



(b) Steady

Figure 6.2: Comparison on the average execution time of SPECjvm98 (input size 10).

run to prepare the corresponding Apus images. Those Apus images later are used for the execution in other configurations (`image` and `image-aggressive`). As a consequence, the binaries used in the execution of `image` and `image-aggressive` are identical to the binaries used in `opt`.

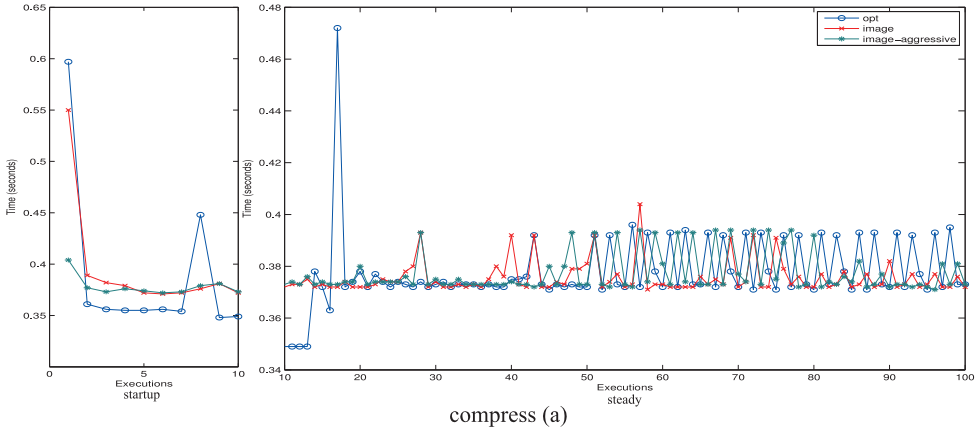
The result shows that in the startup regime, the `image-aggressive` configuration clearly delivers a better performance than the other two configurations. In the configuration of integrating the code migration framework into an adaptive optimisation system, the `image` configuration produces a competitive result. In benchmark `mpegaudio` and `mtrt`, `image` configuration can deliver 15% better performance than the counterpart, `opt` configurations. Since three configurations are using identical binaries on the benchmarks, it is expected that similar performance is delivered on the steady regime, as we can see in Figure 6.2(b).

It is clear from the results that the code migration framework can improve the benchmark execution speed on the startup regime compared to the adaptive optimisation configuration. At the same time, the dynamic compiled binary migration does not pose a performance penalty for applying the same binary on the similar execution instance.

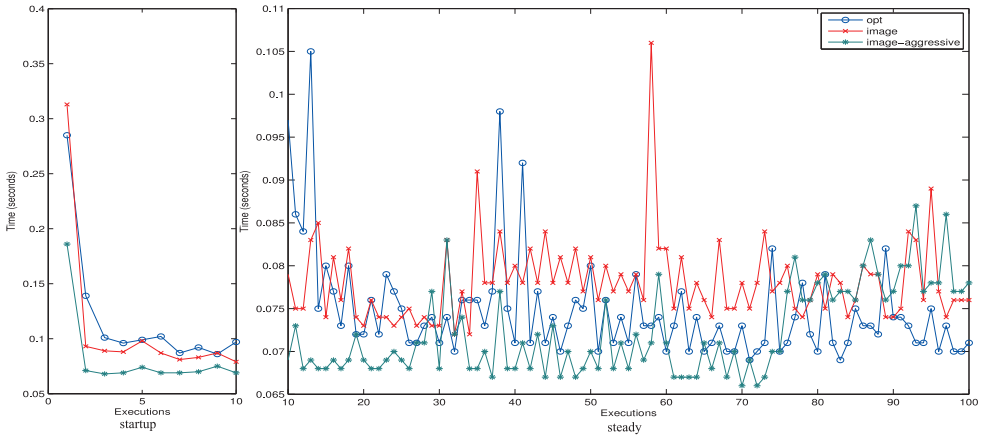
A study of the effectiveness behind the difference of average execution time in the startup and steady regime in the test results shown on Figure 6.2 is taken. Figure 6.3 shows the time taken for each execution (the time spent on a execution is plotted on the y-axis, each execution on the sequence is plotted on the x-axis).

The results depict that the effect of applying the code migration framework can be seen from the first execution of every benchmark. That is, in the first run of every benchmarks, despite the `image-aggressive` configuration loading every available image as a replacement for baseline compilations, it outperforms the adaptive optimisation configuration by up to 2.7 times (the first run of mpegaudio, see Figure 6.3-(e)).

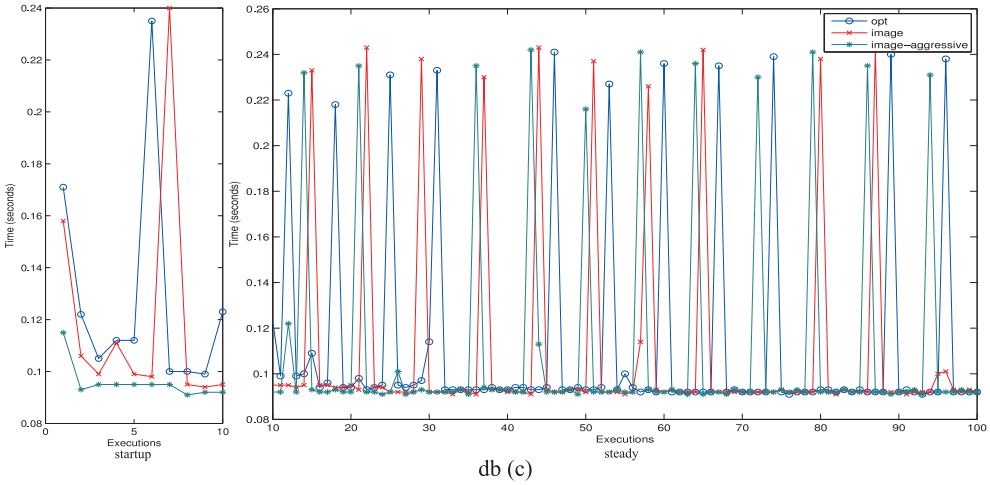
The `image` configuration shows a smooth performance improvement in execution as the adaptive optimisation takes effect. It shows a smoother transition between runs compared to the `opt` configuration, as the result of the low cost of delivering the optimisation binary in the place of the corresponding bytecode. In Figure 6.3-(c), three configuration individually shows a large noise in the execution time plot. After investigation into the log output, the time when GC collection process occurs



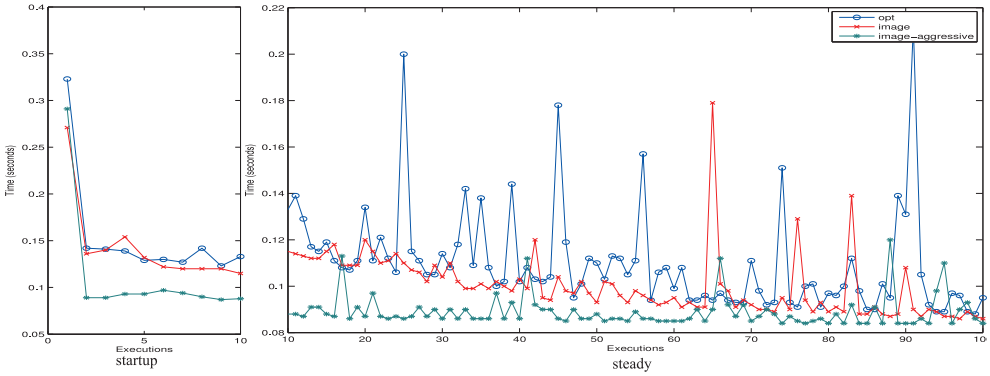
compress (a)



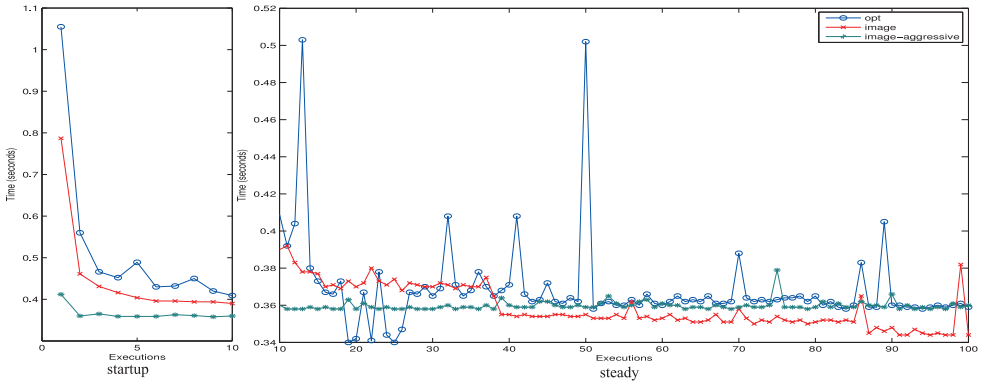
jess (b)



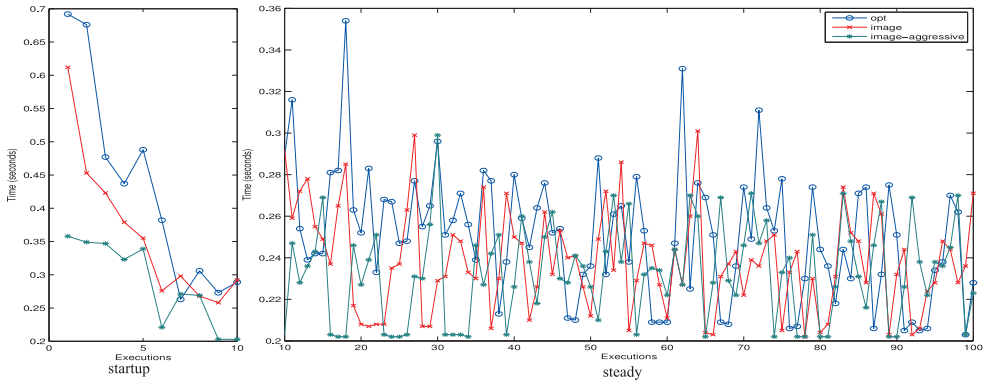
db (c)



javac (d)



mpegaudio (e)



mtrt (f)

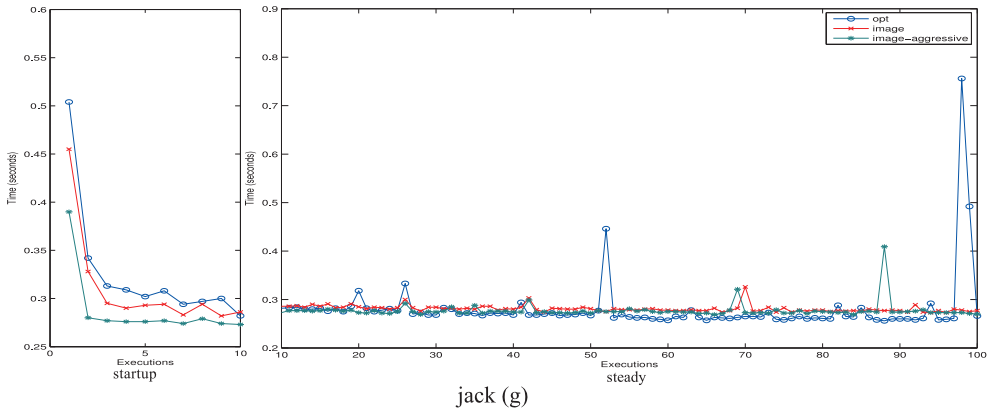


Figure 6.3: Details of execution time of different configurations on SPECjvm98 (input size 10).

matches the executions that take long than others. Therefore, we believe that GC process cause the large noise in Figure 6.3-(c).

However, despite the fact that the compilation speed of Apus images loading has been correctly setup in the adaptive optimisation system as the data, which will be shown in the next subsection, it can be seem that comparing to the `image-aggressive` configuration, the adaptive optimisation system is taking too conservative a decision of on when to apply image loadings on the method recompilation.

As a result of such a conservative decision on applying Apus image loading, the `image` configuration shows a similar pattern to the `opt` configuration on the improvement of performance alone in the execution of benchmarks.

In this subsection, two observations can be concluded from the analysis above:

- Experiments in the section have demonstrated that code generated from a dynamic optimising compiler can be sucessfully migrated to a different JVM runtime without significant performance degradation.
- The existing method profiling technique used to identify frequent executed methods cannot adapt the fast method compilation provided by code migration framework. That is, the benefits of the code migration framework to improve Java execution cannot be fully utilised with the AOS implementation in Jikes

RVM.

6.6.2 Using Static-Compiled Images

With the relocation information, the dynamic compiler in the JVM can also be deployed as an AOT (Ahead-Of-Time) compiler statically. Methods can be compiled using a specified optimisation level instead of relying on the online profiling system which makes the compilation decision based on the benefits (performance improvement) and costs (compilation time). Since compilation time is not an important parameter for making compilation decisions any more, every method in the application can have a pre-compiled Apus image ready before the execution begins.

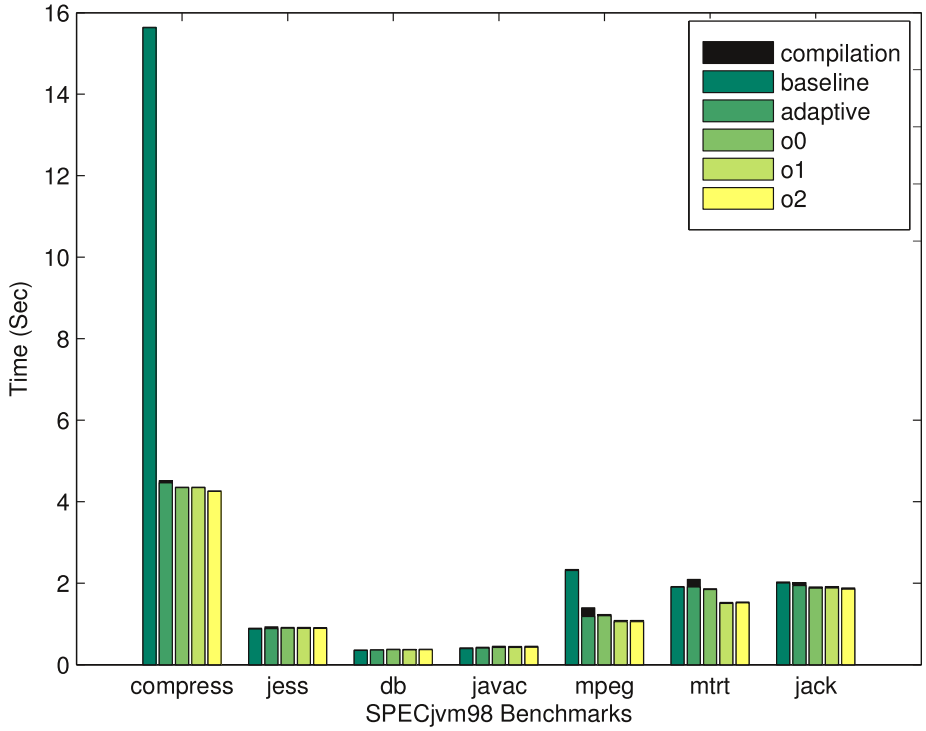
Under the tests in this subsection, testing programs were statically compiled into Apus images under different optimisation levels from the least optimised 0 to high optimised 2 results shown as `o0, o1, o2` respectively.

In Figure 6.4, we experiment with the execution time of SPECjvm98 benchmarks while applying statically compiled Apus images using various optimisation levels against adaptive optimisation configuration. For each benchmark, two bars are shown. The black bar on the top represents the time spent on compilation and image loading during execution. The coloured bars represent the execution time without the time of compilation and image loading. The two bars in total for each benchmark represents the overall execution time spent in 10 iterations of the benchmark and the test harness[2].

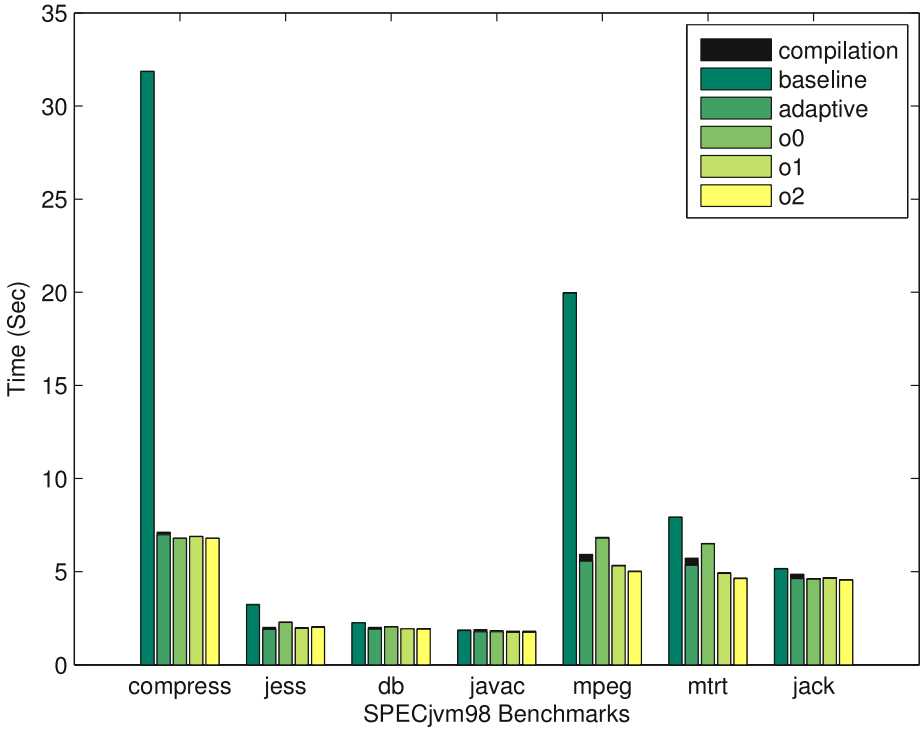
The static compiled Apus image at optimisation level 2 (`o2`) out-performed adaptive optimisation in overall execution time by between 26.4% (mtrt) and -3.8% (db) with the input size 1. The advantage of using static compiled Apus images drops to between 5.3% (compress) and -21.3% (jess) when the input size of the benchmarks increased to 100.

The execution by using an Apus image with higher optimisation level shows a shorter execution time on benchmarks (mpegaudio, mtrt, jack etc.) with different input sizes. With the input size greater than 1 unit, the execution by using Apus images with optimisation level 0 is slower than most of the executions with adaptive

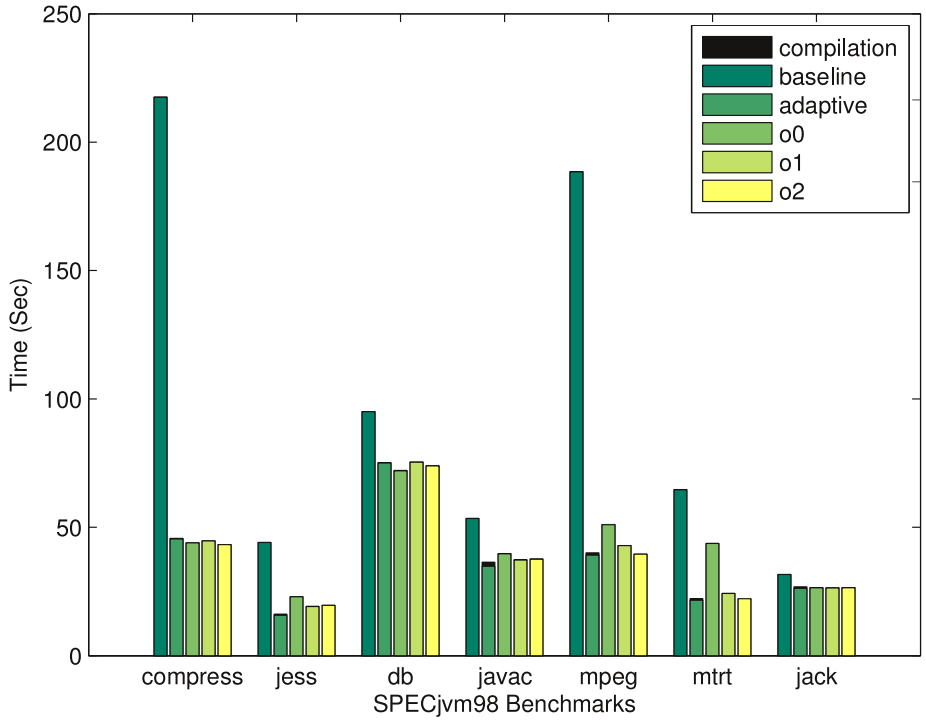
[2]The reason to include the test harness into the execution time is to show the compilation time against overall all execution time more accurately, since the compilation may be triggered during the execution of the test harness.



(a) SPECjvm98 Benchmarks (Size 1)



(b) SPECjvm98 Benchmarks (Size 10)



(c) SPECjvm98 Benchmarks (Size 100)

Figure 6.4: Performance comparison in SPECjvm98 using statically generated Apus image in the start-up status

optimisation. However, with highest optimisation levels that the compiler can provide, it does not guarantee the best performance throughout, such as jess, db, javac with input size 1.

In order to inspect the image loading performance, Table 6.3 shows the comparison of performance between three configurations on SPECjvm98 benchmarks with input size 10 for the same experiment as in figure 6.4. In each case, the following results are reported:

- `Total Time` represents the overall time consumed on the benchmark execution in milliseconds including all the scheduled runs. This is different from the average execution time used in previous results.
- `# Meths` represents the total number of methods being compiled in the tests.
- `Comp` represents the total CPU time used in the compilation in milliseconds.
- `BCB/ms` represents the compilation speed in term of bytecode bytes per milisecond.

Benchmark	Compiler	Total Time(ms)	# Meths	Comp(ms)	BCB/ms	MCKB
compress	baseline	31862	215	8.93	1143.93	132.5
	opt	7118	12	126.34	47.68	10.7
	image	6804	6	2.9	389.29	8
jess	baseline	3241	623	18.29	1138.02	277.9
	opt	2008	22	89.42	23.23	13.1
	image	2038	16	14.18	104.42	30.9
db	baseline	2263	209	8.93	1144.46	129.7
	opt	2004	7	81.93	17.82	10.4
	image	1931	6	5.59	151.47	9.4
javac	baseline	1868	912	34.06	1400.64	555.4
	opt	1894	25	81.24	31.97	14
	image	1803	30	26.89	186.76	88.9
mpegaudio	baseline	19969	387	20.31	1493.69	497.5
	opt	5921	54	339.52	41.4	41.1
	image	5032	44	13.72	464.38	54.8
mtrt	baseline	7932	339	12.57	1148.84	197.3
	opt	5719	41	361.43	22.75	47.1
	image	4654	29	16.87	235.29	89.9
jack	baseline	5170	448	19.99	1261.96	359.7
	opt	4866	26	215.39	26.47	25.1
	image	4574	26	22.42	201.55	61.3

Table 6.3: Compilation details of three different execution configurations on SPECjvm98 (input size 10) on start-up status. The Apus images are static compiled at optimisation level 2.

- `MCKB` represent the total size of generated machine code in kilobytes.

Since the baseline compiler is the default compiler in the execution, as we expect, it compiles far more methods than the others `opt` and `image` with best compilation speed of 1493 bytecode bytes per millisecond (BCB/ms) (mpegaudio), however, it clearly deliver the worst performance amongst the three. The image loading shows a consistently better performance on translation rate over optimising compiler throughout all the tests. In mpegaudio benchmarks, Apus image loading can translate 464 BCB/ms which is about 10 times greater than opt, reducing the compilation time for recompilation time from 339 to 13 ms, at the same time achieving a better performance on the overall execution time. Another interesting column to look at is the total number of methods (`#Meths`) handled by the code migration framework, which shows that the `#Meths` is less than using the optimising compiler configuration, however, MCKB is greater than the latter configuration.

Image loading details for the tests on SPECjvm98 benchmarks are given in table 6.4. Results are reported as:

- `Load` represents the total CPU time spent on the image loading process in milliseconds.
- `Link` represents the total CPU time spent on the image linking process in milliseconds.
- `veri` represents the total CPU time spent on the image verification process of images in milliseconds. This time partly overlaps the loading and linking processes.
- `#Sym` represents the overall number of loaded symbols.
- `#LPs` represents the overall number of loaded link points.
- `Resolving` represents the total CPU time spent on aggressive resolving of symbol references during the image linking process.
- `IMGB` represents the overall loaded size of Apus images in bytes.

Loading images from the file system accounts for more than 50% of time spent on the entire Apus image loading process. In the case of the test of javac on input size 100, the loading process accounts for 62% of the overall time. The verification process is included into both image loading and linking processes, since the declaring class of the target method has to be verified before loading the images and inlined methods are verified during the linking process. The verification process accounts

Benchmark	Input	# Meths	Comp(ms)	Load(ms)	Link(ms)	Veri(ms)	# Sym	# LPs	Resolving/ms	IMGB	BCKB
compress	1	9	3.2	2.34391	0.850009	1.25545	80	235	0	15451	1.4
	10	6	2.9	2.23895	0.65151	1.23468	67	222	0	13842	1.3
	100	9	3.12	2.27886	0.833182	1.22716	80	235	0	15479	1.5
jess	1	2	2.14	1.7522	0.386695	0.899762	23	24	0	1428	0.1
	10	16	14.18	8.49059	5.66792	6.74367	510	1844	0.401943	77212	1.8
	100	33	21.41	13.662	7.71706	7.18736	939	3106	1.03607	134919	3.1
db	1	2	1.51	0.869612	0.0349442	0.175175	4	4	0	358	0.1
	10	6	5.59	3.35408	2.22688	2.42075	217	520	0	25982	0.8
	100	8	6.59	3.9233	2.65057	2.80434	231	516	0	27046	0.8
javac	1	1	1.05	1.008	0.0355173	0.319391	8	10	0	390	0
	10	30	26.89	16.3974	10.4644	10.8716	1173	4849	0	214465	7.6
	100	186	141.18	85.862	55.1186	38.8856	5433	20010	14.1695	901953	37.2
mpegaudio	1	18	7.19	5.57828	1.60078	3.22742	205	479	0	47730	6.2
	10	44	13.72	10.1573	3.54593	5.19911	516	1424	0.47558	103636	9.6
	100	77	23.48	17.2693	6.16839	8.10062	1166	3101	0.848464	179026	12.5
mtrt	1	13	10.54	7.40352	3.12436	3.17965	314	3178	0	119546	4.5
	10	29	16.87	12.4179	4.41933	5.08994	531	4279	0	170082	6.2
	100	37	19.99	14.8319	5.10894	5.73735	650	4393	0	177716	6.9
jack	1	11	9.2	4.85774	4.32416	4.30214	322	931	0.413049	41085	1.7
	10	26	22.42	13.9779	8.41213	8.08452	874	4095	0.86884	162924	4.7
	100	57	35.96	24.7249	11.1617	11.2977	1387	6093	0.870782	275690	16.1

Table 6.4: Details of image loading in SPECjvm98. The Apus images in the tests are statically generated at optimisation level 2.

for about 30% of the overall time of the Apus image loading process, in the worst case where the input size is 1 on jack, the verification took as much as 46% of overall time of the Apus image loading.

Since the aggressive reference resolution strategy, as described in Section 4.6.4, was used during the linking process, it is interesting to find out how this strategy has an effect on the Apus image loading process. In 12 out of 21 cases, reference resolution does not happen during the linking process. And furthermore, in most of the cases involving reference resolution, it took about 10% of time in the link process. However, in one case, javac input size 100, the reference solution accounts for 25% of the time in link process, 10% of overall Apus image loading time. However, the number of symbols loaded in the test reached 5433 in the test.

In this subsection, experiments are taken to measure the effectiveness of Apus image generated from a static offline optimising compiler. Some observations are made:

- Configuration of using only Apus images can be more efficient than the adaptive configuration in the start up stage (26.4% - -3.8%), however, the advantage disappear when the input size of the benchmark increases (5.3% - -21.3%).
- The code migration technique shows a significant reduction on time to translating Java bytecode into optimised binaries, compared to a dynamic optimising compiler. In the best scenario, code migration shows 10 time (mpegaudio) quicker than optimising compiler.

6.6.3 Proxy Compilation

The experimental results in this section evaluate the effects of proxy compilation on average execution speed and memory consumption as an alternative approach to the optimising compiler.

Here is a list of dynamic compilation configurations and JVM that we test throughout this subsection:

- *baseline* The baseline compiler without adaptive optimisation system on Jikes RVM 2.4.6.
- *optimisation* The optimising compiler with adaptive optimisation system on Jikes RVM 2.4.6.

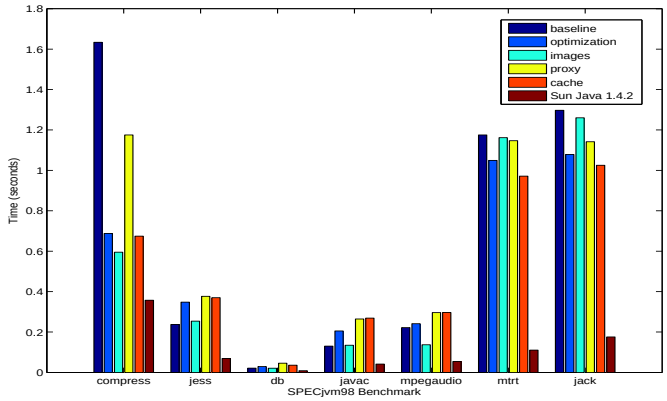
- *image* Apus image framework with adaptive optimisation system, and feeding with static-compiled Apus images.
- *proxy* Proxy compiler integrated with adaptive optimisation system.
- *cache* Proxy compiler integrated with adaptive optimisation system, this configuration also includes the cache of Apus image from the execution by proxy configuration.
- *Sun Java 1.4.2* Sun Java 1.4.2 VM.

Both the server and client of Apus proxy compilation system are built under the configuration of `prototype-opt` for the reason as it is described in Section 6.5. That is, the entire JVM binary is produced by the baseline compiler, as compared with the previous tests, where the JVM is built in the `production` configuration that uses the optimising compiler to optimise the binary of the runtime. In consequence, JVM built in `prototype-opt`, which does not have optimised binary, generally leads to longer execution times on the benchmark compared with the `production` configuration, as we can see in the following figures.

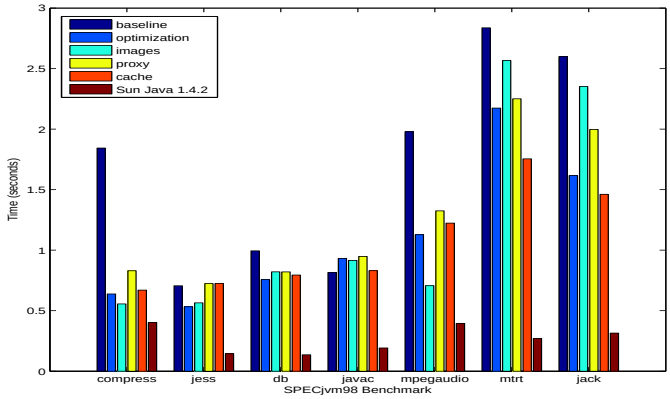
The first group of test results (listed in Figure 6.5) shows the effects of using different dynamic compiler configurations listed above when tested on SPECjvm98 benchmarks. The graph shows the average execution time in the start up state, therefore, the lower bar represents better performance.

The results show the performance of various JVM configuration on SPECjvm98 benchmark in the startup stage, which is the first ten iterations of the benchmark execution. It can be seen that Sun Java 1.4.2 clearly delivers a shorter execution time throughout the tests compared to others based on the Jikes RVM. However, the difference of execution time between the Jikes RVM (`opt`) and Sun Java (`Sun Java 1.4.2`) reduces when the benchmark input size was increased to 100.

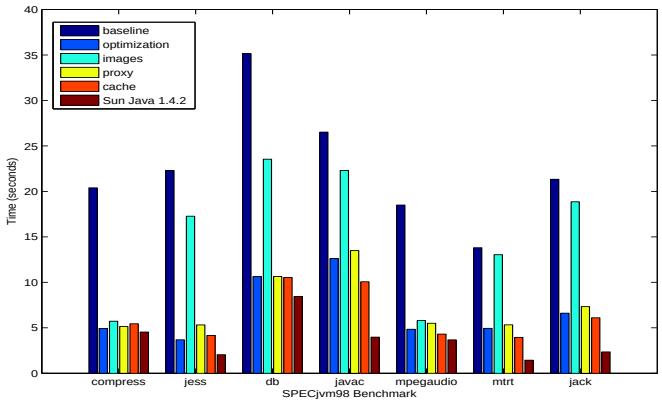
Using the same input size (100), the configuration that solely relying on statically generated Apus images (`images`) shows a longer execution time on benchmarks (`jess`, `db`, `javac`, `mtrt` and `jack`) than proxy and optimisation configuration. We believe that this difference is because the Jikes RVM runtime in `images` do not benefit from the code optimisation, as the runtime binaries are optimised. Therefore, those results suggest that in the startup stage, the unoptimised binary of Jikes RVM runtime has a large impact on the overall execution speed. This assumption is also confirmed with the previous observation, which the difference of execution time



(a) Execution Speed on SPECjvm98 Benchmarks (Size 1)



(b) Execution Speed on SPECjvm98 Benchmarks (Size 10)



(c) Execution Speed on SPECjvm98 Benchmarks (Size 100)

Figure 6.5: Average execution time comparison on proxy compilation and other configuration on SPECjvm98.

between Jikes RVM and Sun JVM reduce while increasing the input size.

The proxy configuration (`proxy`) shows it achieves a competitive execution speed, but generally slower in most of the cases than the optimising compiler configuration `opt`. The execution time slowing down by proxy configuration, compared to optimising compiler configuration, reduces from 21% to 11% on average while the input size of SPECjvm98 increases from 1 to 100. We believe this is due to a longer execution to able more methods to be optimised by the proxy compilation service. Further investigation in this assumption can be seen on later discussion on Table 6.5.

To reduce the code optimising over network, a local repository to cache the optimised binaries is used in `cache` configuration. Test results in Figure 6.5 shows a positive impact on the binary caching. The same as `proxy` configuration, `cache` configuration shows a consistent improvement on the execution time, compared to the `opt` configuration, from -8% increased to 7% on average, with a maximum improvement to 25% in `mtrt` benchmark.

Table 6.5 shows more details of compilations on various configuration than Figure 6.5(c), where optimised binaries make more distinguishable improvement from the baseline configuration. Beside measurements mentioned in the previous section, we introduced *"turn around"* compilation time to measure the time spent in execution during a completed compilation cycle. It measures the period of time from starting the compilation process to the time when the binary is produced successfully. The purpose of this measurement is to give information on how long it takes on average for the optimised binary to be available after the request was sent. It is related to the compilation time (Comp Time), especially for the situation where long latency is involved in the compilation process, such as network communication, only limited online compilation tasks can be completed in the limited time frame of the execution.

The `proxy` and —cache— configurations optimised a similar numbers of methods in the entire benchmark execution. In contract, the `opt` configuration optimised more methods than other configurations, due to the fact that the optimisation includes methods from the optimising compiler itself. Under the same circumstances, the `image` configuration have the least methods optimised during the tests, because the optimised binaries for the runtime is not available. For the same reason, overall, the `cache` configuration delivers a faster execution over `opt` configuration, and the `images` configuration delivers the longest execution on the tests where input size is

Benchmark	Compiler	Avg Exec(sec)	# Meths	Comp Time(ms)	Turn Around(ms)	BCB/ms
compress	opt	4.92	85	3066.4	6349.4	3.19
	images	5.72	10	11.15	11.142	78.05
	proxy	5.86	22	47.66	8242.4	29.4
	cache (proxy)	5.62	7	7.52	3045	60.56
	cache (image)		16	22.19	22.179	55.63
jess	opt	3.68	145	3259.6	9380.4	3.24
	images	17.27	60	91.13	113.59	48.73
	proxy	5.32	101	160.32	32766	35.42
	cache (proxy)	4.16	46	72.65	17302	33.2
	cache (image)		74	96.33	183.99	56.3
db	opt	10.63	124	3617.3	14150	2.98
	images	23.54	11	33.26	263.56	35.06
	proxy	10.64	51	107.53	17701	25.12
	cache (proxy)	10.53	19	26.86	8841.6	38.68
	cache (image)		28	48.84	71.095	44.08
javac	opt	12.61	645	12146	37756	4.41
	images	22.29	284	440.74	440.58	52.12
	proxy	13.49	317	485.21	135420	39.11
	cache (proxy)	10.07	69	81.61	34627	39.57
	cache (image)		229	273.49	357.33	72.21

Table 6.5: Details of Proxy Compilation on SPECjvm98 Benchmarks (size 100) *part i*

Benchmark	Compiler	Avg Exec(sec)	# Meths	Comp Time(ms)	Turn Around(ms)	BCB/ms
mpegaudio	opt	4.84	242	5951.5	14023	4.17
	images	5.8	81	69.95	139.25	105.92
	proxy	5.5	97	177.87	31237	63.84
	cache (proxy)	4.31	16	25.99	6132.9	46.14
	cache (image)		72	62.87	62.845	149.67
mtrt	opt	4.94	192	4801.1	20080	3.67
	images	13.03	66	82.45	281.39	40.67
	proxy	5.32	82	134.7	38813	33.96
	cache (proxy)	3.95	28	48.16	18500	27.87
	cache (image)		55	83.69	145.21	57.33
jack	opt	6.61	373	7368.3	20349	4.08
	images	18.86	91	175.46	218.11	76.97
	proxy	7.32	183	248.51	60517	43.05
	cache (proxy)	6.1	35	36.54	15454	32.64
	cache (image)		125	146.24	226.53	60.07

Table 6.5: Details of Proxy Compilation on SPECjvm98 Benchmarks (size 100) *part ii*

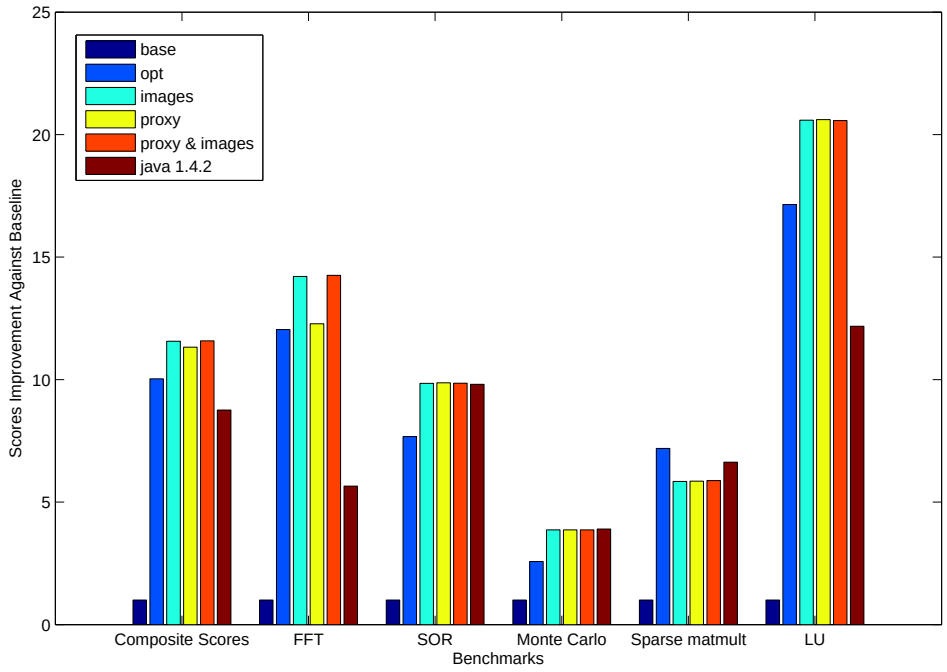


Figure 6.6: Performance improvement against baseline on Scimark 2.0

100.

From Table 6.5, we can observe that `images` configuration has the best code optimisation speed in term of BCB per ms, ranging from 35.06 to 78.05, compared to the `opt` configuration, which reach between 2.98 to 4.41. The optimisation performace of `proxy` configuration, with extra socket IO operation for proxy compilations, lies on between `images` and `opt` configuration, achieve between 25.12 to 63.84.

Although `proxy` configuration shows a relatively small overhead on compilation, it can have the biggest turnaround time per method on average (380.65 ms) among the tests, compared with `opt` configuration (67.89 ms) and `images` configuration (6.84 ms). That is, code is taking longer to be deployed. We believe this is caused by network latency. Comparing configurations between `proxy` against `cache`, while `cache` configuration using the available Apus images from local repository partly replace the proxy compilation, the results from Figure 6.5(c) shows a small performance improvement on `cache` by 7% on average.

Let us examine those configurations described above on small, computing-intense

benchmarks. Figure 6.6 shows that the the same group of compiler configurations tested on a computationally intensive benchmark, Scimark 2.0. Test result represent the benchmark score improvement over the baseline configuration. That is, a higher bar means a better performance. Details of the compilation in various configurations of the test can be found in Table 6.6.

The results show that the performance of using proxy compilation with cache has better performance than the optimising compiler configuration. The proxy compilation without cache configuration shows an approximately 10% decrease in performance on only one occasion. From Table 6.6, the image configuration reveals that only a small number of methods, which only belong to the benchmarks, almost one for each benchmark, affected the outcome. It therefore limits the online optimisation. Unlike `images` and `proxy`, in which case bytecode are directly compiled at the highest optimisation level, the optimising compiler applies a more conservative strategy, which reduces the efficiency of the optimised code overall. Combining the reasons mentioned above, it is therefore `images` and `cache` producing the best results overall, since the best optimised code can be deployed in the fastest time.

By graphing the comparison of the average execution time and showing the compilation cost of various configuration of JVM, one can see the effects of using proxy compilation and cache from the local Apus image repository. Figure 6.7 shows details of execution time of each run in the `_213_javac` benchmark. The y-axis represent the execution time. As proxy compilation bears a longer turn around time, it starts with longer execution time on the first run, and improves its execution speed at a slower pace, but reaches a competitive execution speed as optimisation configuration at the fifth run. With the benefits from the cache that are produced by previous execution, `cache` has the best starting run among all the configuration and saturates faster than `proxy` and `opt` as well.

Figure 6.8 presents the size of peak memory usage of the tests from SPECjvm98 and Scimark 2.0 benchmarks in different compiler configurations. The results in this figure are collected from the same tests throughout this section for the reason of coherence. The memory usage measurement technique was described in section 6.4.

The results show that the proxy compilation with cache configuration (cache) produces competitive results to the counterpart, optimisation configuration, except on test `compress` and `db` in which consume about 5 – 10% more memory than optimisation configuration. It also shows a marginally smaller memory consumption

Benchmark	Compiler	Score	# Meths	Comp Time(ms)	Turn Around(ms)	BCB/ms
scimark 2.0	opt	340.63	48	1536.2	3021.8	3.64
	images	392.86	6	43.16	12.305	146.32
	proxy	384.52	20	24.91	7501.5	57.09
	proxy(cache) proxy	393.32	8	9.16	2482	37.41
	proxy(cache) image		12	9.93	9.9274	107.95

Table 6.6: Compilation details of three different configurations on Scimark 2.0 Benchmarks.

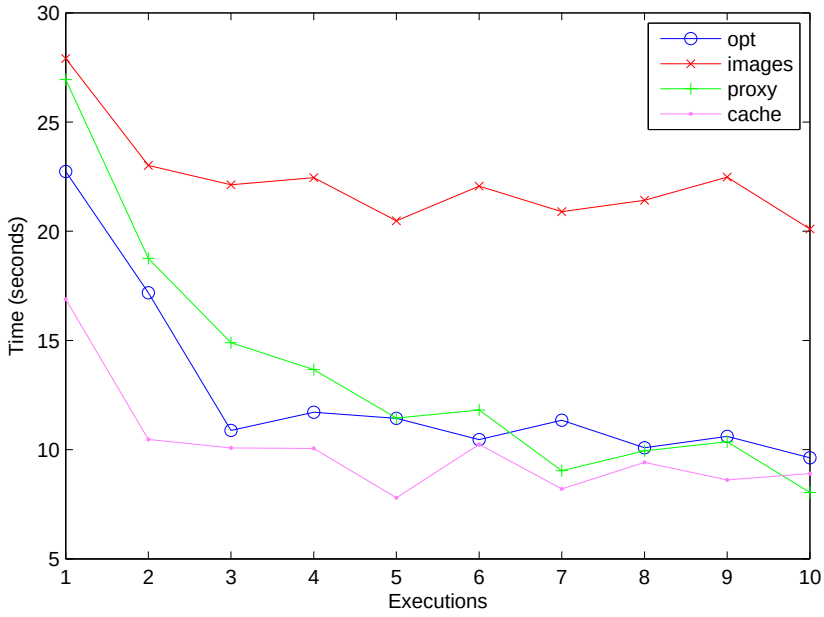


Figure 6.7: Details of every execution speed of SPECjvm98 benchmark _213_javac, input size 100.

than the proxy compilation without cache configuration (proxy), as the result of using the cache from a local Apus image repository. For the same reason, overall, static-compiled image configuration (images) show a better result on memory usage than `cache` and `proxy`.

Although the proxy compilation configuration does not show a distinguishable advantage over optimisation configuration in terms of memory consumption in the tests, we believe there are a number of issues on the implementation of proxy compilation configuration that would affect the overall outcomes. Due to the limitation on the implementation which is used in the tests, a optimising compiler is integrated on the proxy compilation client. In addition, the proxy compilation client configuration also includes a code migration framework and PCP which is not included in the optimisation configuration. We discuss a number of possible improvements to reduce memory consumption for proxy compilation configuration in the summary in Section 6.7.

There are some conclusion can be made from this subsection:

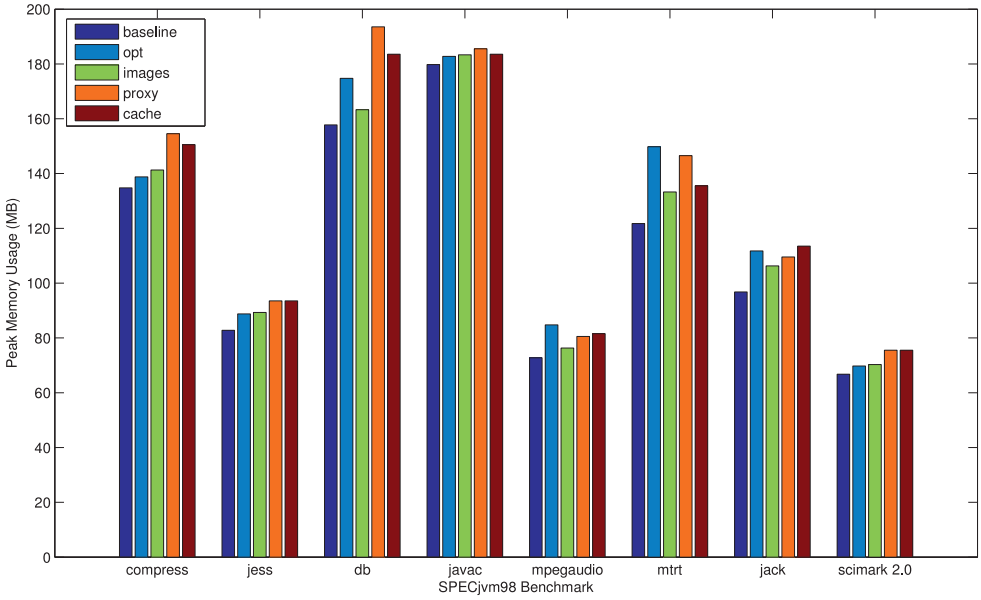


Figure 6.8: Peak memory usage on SPECjvm98 benchmarks (input size 10) and Scimark 2.0

- The results show that the execution time on the `proxy` configration slower on SPECjvm98 (input 100) than `opt` configration for 11% on average. We believe this is due to the reason of network latency. Proxy compiled method takes longer to deploy on average (380.65 ms) than optimising from local optimising compiler (67.89 ms), or from Apus images (6.84 ms). However, the different between proxy configuration and `opt` configuration is changed to 7% faster to by using a cache in the proxy client.
- The code optimisation speed provide by `proxy` configuration (25.12 - 39.11 BCB/ms) is significantly faster than `opt` configuration (2.98 - 4.41 BCB/ms). However, results also indicates that network latency increase the time to wait for the binary in a proxy compilation, and lead to a slow down in the benchmark execution, from 7% faster in `cache` configuration than `opt` configuration to 11% slower in `proxy` configuration.
- The memory consumption by the `proxy` configuration is greater than the `opt` configuration between 5% - 10%, we believe this is due to the fact that a optimising compiler is integrated with the client implementation.

Benchmark	Conf	Avg Exec(sec)	# Meths	Comp Time(ms)	BCB/ms
compress	opt	4.10	15	191.64	41.06
	images	4.18	18	201.35	31.79
jess	opt	1.53	65	474.09	18.48
	images	1.52	62	464.51	18.02
db	opt	7.36	15	155.12	15.36
	images	7.34	17	180.91	13.88
javac	opt	3.62	264	2026.42	24.41
	images	3.58	257	1803.79	22.13
mpegaudio	opt	3.68	112	1050.42	33.64
	images	3.52	116	938.82	34.60
mtrt	opt	1.89	66	605.87	22.67
	images	1.64	72	620.82	21.81
jack	opt	2.39	73	378.42	30.14
	images	2.41	66	486.72	25.87

Table 6.7: Performance of code migration framework on SPECjvm98 benchmarks.

6.6.4 Cost of Generating Relocation Information

This section evaluates the cost of producing Apus images described in Chapter 4 by comparing the performance of the Apus image compiler and the original optimising compiler of Jikes RVM. To allow the experiments to focus on the impact of relocatable code during the online optimisation, all the tested JVM configurations use the same optimisation settings throughout the tests.

Table 6.7 shows the performance of each benchmark. This experiment compared the performance on the relocatable code generator (images) to the original optimising compiler (opt) over SPECjvm98 benchmarks for a total 10 runs. In both configurations, both compilers are used as an online optimising compiler in the execution. The major difference is the relocatable code generator produces relocatable binary as well as the optimisation.

Overall, the results show that the `images` configuration does not impose a noticeable performance overhead on average execution time on the benchmarks in general. In the worst case of `images` configuration, the `db` benchmark reduces the compilation speed by a mean of less than 10%, and delivers a similar average execution time as well. Overall, producing a relocatable binary while in online compilation does not produce a distinct overhead on both execution time and compilation speed.

6.7 Summary of Results

The purpose of the testing and analysis described in this chapter is to provide quantitative data that can be used to evaluate the efficiency of the implementation of the Apus proxy compilation system. Based on the detailed results and analysis in this chapter, some general conclusions can be drawn:

- The code migration framework can increase the compilation speed in terms of bytecode per millisecond by 8 times on average compared to the optimising compiler, by using corresponding Apus images from the local image repository.
- Apus images from static-compiled binaries are as efficient as their counterpart, the binaries produced by the dynamic optimising compiler in the startup stage of the execution. The use of Apus images shows a better performance in the situation where the running time of benchmarks is short.
- Performing image linking in the client does not impose a significant performance burden in the binary loading process.
- Binaries which were dynamically generated can be relocated and used faultlessly in a different execution instance by using the code migration framework, and the fast loading migrated binaries can deliver a better performance on the startup phase of the application running and are as efficient as that produced by the original execution instance in the steady state.
- On average proxy compilation substantially outperforms optimisation compilation in every test in term of compilation speed, however, it consumes more CPU cycles than loading an Apus image from a local repository.
- Configuration of combining the proxy compilation with caches in a local repository, in which images are collected from previous executions is faster than the optimising compiler configuration in the startup stage of applications.
- Producing relocation information for the Apus image in the optimising compiler only marginally slows down the optimising compilation process, however, it does not impose any observable penalty to the average execution time of applications.
- The memory consumption of the Apus framework configuration is marginally smaller than optimisation configuration in most cases.

In conclusion, the above shows that using the code migration technique can be considered an efficient alternative technique to achieve code optimisation. The proxy

compilation with caching in a local repository is competitively efficient as an optimising compiler. However, the testing and results show that there are areas in which the proxy compiler implementation is not as efficient as it could be:

- The code migration framework configured with adaptive optimisation system only marginally outperforms the optimising compiler configuration and is visibly slower than the more aggressive approach of using Apus images. This problem is caused by the adaptive optimisation system not adjusting to identify hot methods fast enough in a large scale to replace the non-optimised binaries reflecting the use of lightweight Apus image loading compared to the dynamic optimisation.
- Although proxy compilation consumes fewer CPU cycles than optimising compiler on average, due to a less predictable network latency, the turnaround time for the compilation on the proxy compiler is longer than optimising compiler. As a consequence, as testing results show, this can cause a potential slow-down in the execution time, especially in the startup state. However, this disadvantage can be offset by providing a caching approach for the compiled binaries, which reduces the reliance on the proxy compilation.
- Various test results show that the use of the proxy compiler in the client leads to a higher memory footprint than optimisation configuration. This issue is partly caused by the proxy compiler implementation containing the full package of the optimising compiler and the sophisticated adaptive optimisation system which is responsible for more than the identification of hotspot methods, such as sampling execution and giving feedback to the optimising compiler.

CHAPTER 7

Conclusion

Some things need to be believed to be seen.

GUY KAWASAKI (1954–)

This chapter closes the dissertation by drawing a list of contributions based on the conclusions from the experimental results stated in Chapter 6. A number of possible extensions to our experiments, as well as some possible directions for further research are discussed. These suggest ways in which the research could be improved or extended. Finally, some comments summarise the research work covered in the entire dissertation.

7.1 Summary of contributions

The purpose of this work is to support the thesis that efficient Java execution can be achieved through an efficient optimising compilation, which is provided by a proxy compilation service.

The following specific contributions have been achieved:

- A proxy compilation system has been designed and implemented on an existing JVM implementation. It utilises a full implementation of a dynamic optimising

compiler in the remote compilation service. Further analysis and test results demonstrate the efficiency of this implementation.

- Comparisons show that applying a remote compilation server for dynamic bytecode optimisation using an existing JVM implementation can achieve comparable performance to the equivalent JVM implementation. Execution performance using a proxy compilation technique combining a cache from previous optimisation binaries shows a mean performance improvement of 25% compared to an adaptive optimization compiler with equivalent configuration.
- A code relocation framework is designed and implemented to organise relocation information and binaries in a format that can be relocated into different execution instances correctly and efficiently. This framework has been applied to handling code migration on optimised binaries coming from a remote compilation server and cached binaries from a local repository.
- Adopting the client-side binary relocation can allow remote compiled binaries to be cached in the client at small cost. Test results show that the impact on overall performance by the relocation cost is smaller than the duplicated transmission cost introduced by server-side relocation.
- Applying a remote compilation server for dynamic code optimisation can significantly reduce the cost on the host machine, while producing binaries with comparable efficiency to traditional optimising compilation.
- Using a dynamic optimising compiler to produce relocatable binaries does not impose a significant performance penalty on the compilation process nor the efficiency of the optimised binaries.

7.2 Possible Extension

The Apus proxy compilation system was specifically designed for the purpose of proving the concept of providing a proxy compilation service for existing JVM implementations, while maintaining a fully functional JVM implementation. To this end, the system does not intend to implement many other features specified for a proxy compilation scenario to increase the performance of Java program execution, such as an improved optimisation procedures, frequently executed code region prediction etc. These were omitted because they were not considered essential to the goal of this research in the available time. Further improvements of this kind could

be built or extended upon the current structures.

Despite the basic set of features and functionality that have been implemented in the Apus system, there are many possible extensions that are possible. Typically, these would improve the performance of the Apus system further, and some would support new features. In the following sections, a brief discussion is presented to address how some interesting features and extensions could be implemented using the existing infrastructure.

7.2.1 Selective Compilation

As the results presented in Chapter 6 suggested, low overhead of image loading and proxy compilation largely reduce the cost of bytecode optimisation in the execution. The Apus system utilised call stack sampling implemented by the Jikes RVM as the selective optimisation approach [6] to balance the cost and benefits from the bytecode optimisation. However, as the test results show in Section 6.6.1, execution performance shows a significant improvement on giving the highest priority to utilising the optimisation cache compared to using the existing code selection model to make the decision.

In order to take advantage of the low cost of bytecode optimisation provided by the Apus system, it would be interesting to explore a more aggressive code selection approach for code optimisation (in this context, the techniques used for code optimisation are image loading from the local repository, such as cache, and remote compilation).

Beyond the aggressive code selection, one possible extension is an on-stack replacement (OSR) [32] mechanism which enables an execution to be transferred into an optimised version, even while the method runs. Although OSR is regarded as an engineering challenge, and further relocation is necessary to the optimised code to support OSR, such a facility would allow optimised code to be activated sooner while the current non-optimised code version contains a long-running loop.

7.2.2 Profiling Driven Optimisation

Dynamic optimisation, many systems such as SELF-93 [42] and Jikes RVM [14] apply online profiling information to assist the decision making over optimisation

compilation during runtime. As briefly introduced in Section 2.6, such a technique would be able to overcome the limitation of static compilation to directly apply many aggressive optimisation procedures of dynamic compilation to improve execution efficiency.

However, profile driven optimisation comes with a limitation that the optimised code can only be applied in specific situations. In a proxy compilation environment, this optimisation technique would potentially increase the load on network transmission when multiple compilations of the same method are requested. In order to cache all the profile driven optimisation from the compilation server to reduce the overall transmission cost, the Apus code migration framework could provide support for multiple version of optimised binaries to be stored at the local repository. A thin guard proposed by Arnold and Ryder [9] could then be used to choose the appropriate version of optimisation for the specified method.

7.2.3 Multi-Platform Support

The main motivation for the Apus proxy compilation system to reduce the cost of dynamic optimisation is to improve the execution efficiency on embedded systems. However, this research prototype is built on a well established research-oriented JVM implementation (Jikes RVM), which is designed for the desktop environment with x86 and PowerPC architecture supported. It would be interesting to measure the impact of applying the Apus system on an embedded system.

Beside a Jikes RVM implementation for targeted embedded systems, a cross platform dynamic compiler is required for the proxy compilation server where the server and client are on heterogeneous platforms. In addition, to generate binaries for the specified platform, a further extension to the proxy compilation protocol to identify the client platform and the cache management on the compilation server is also required.

7.2.4 Extending Accessibility of Source Files

The opportunity exists for the client to restrict the transmission of source files to the compilation server even more. In a situation where the source files of the requested program are accessible online, the compilation server can directly download

the source file from the URI address information provided by the clients. A routine verification on the source files on the client are necessary. The client would then benefit from a faster proxy compilation session. The client may also benefit from higher efficiency binaries, since further profile driven optimisation can be applied to the target method while the compilation server has access to all the related source files.

7.3 Conclusion

Java has been marked as one of the milestones in the development of programming languages, supporting seamless compatibility on various platforms and supporting many powerful features. Dynamic optimisation allows sophisticated optimisation techniques to be applied in execution to overcome the performance issue of interpreting the bytecode. One of the tasks of compiler researchers is to reduce the cost of dynamic compilation while delivering the most efficient binaries.

The work described in this dissertation details the design and implementation of the infrastructure of a proxy compilation system for Java. This system addresses many common issues in utilising dynamic optimising compiler as a remote compilation service, and provides a foundation for further research on exploiting dynamic optimisation in a remote compilation scenario. Test results confirmed that the Apus proxy compilation system can deliver equivalent efficiency optimised binaries while improving the compilation speed, reducing the demand for computation and memory footprint on the client.

References

[1] A.R. Adl-Tabatabai, J. Bharadwaj, D.Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, 2003.

[2] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–236, 2000.

[3] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[4] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[5] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, volume 411, 1999.

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 47–65. ACM Press New York, NY, USA, 2000.

[7] M. Arnold, SJ Fink, D. Grove, M. Hind, IBM Thomas J Watson Rearch Center Sweeney, PF, and NY Hawthorne. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[8] M. Arnold, M. Hind, and B.G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 111–129. ACM New York, NY, USA, 2002.

[9] M. Arnold and B.G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. *Lecture Notes in Computer Science*, pages 498–524, 2002.

[10] A. Azevedo, A. Nicolau, and J. Hummel. Java annotation-aware just-in-time (AJIT) complilation system. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 142–151. ACM New York, NY, USA, 1999.

[11] A. Azevedo, A. Nicolau, and J. Hummel. An annotation-aware Java virtual machine implementation. *Concurrency: Practice and Experience*, 12(6):423–444, 2000.

[12] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[13] P. Bothner. Compiling Java with gcj. *Linux Journal*, 2003(105), 2003.

[14] M.G. Burke, J.D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 129–141. ACM New York, NY, USA, 1999.

[15] P.P. Chang, S.A. Mahlke, and W.M.W. Hwu. Using profile information to assist classic code optimizations,". *Software Practice and Experience*, 21(12):1301–1321, 1991.

[16] M. Chen and K. Olukotun. Targeting Dynamic Compilation for Embedded Environments. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium table of contents*, pages 151–164. USENIX Association Berkeley, CA, USA, 2002.

[17] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: a flexible high-performance managed runtime environment An earlier version of this paper was published online [1]. *Concurrency and Computation: Practice and Experience*, 17, 2005.

[18] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156. ACM New York, NY, USA, 1996.

[19] K.D. Cooper, P.J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, 1999.

[20] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks, 1998. http://www.spec.org/jvm98/index.html.

[21] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman, 2005.

[22] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. *ACM SIGPLAN Lisp Pointers*, 7(3):273–282, 1994.

[23] B. Delsart, V. Joloboff, and E. Paire. JCOD: A lightweight modular compilation technology for embedded Java. *Lecture Notes in Computer Science*, 2491:197–212, 2002.

[24] D. Detlefs and O. Agesen. The case for multiple compilers. In *OOPSLA.99 VM Workshop: Simplicity, Performance and Portability in Virtual Machine Design*, pages 180–194. ACM New York, NY, USA, 1999.

[25] L.P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302. ACM New York, NY, USA, 1984.

[26] P. Drews, D. Sommer, R. Chandler, and T. Smith. Managed runtime environments for next-generation mobile devices. *Intel Technology Journal*, 7(1):68–76, 2003.

[27] D.R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 160–170. ACM New York, NY, USA, 1996.

[28] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 131–144. ACM New York, NY, USA, 1996.

[29] M.A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 278–288. ACM New York, NY, USA, 2003.

[30] M.A. Ertl, C. Thalinger, and A. Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4(1):31–38, 2006.

[31] R.T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[32] S.J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 241–252. IEEE Computer Society Washington, DC, USA, 2003.

[33] Python Software Foundation. Python programming language. http://www.python.org.

[34] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. *Lecture Notes in Computer Science*, pages 170–184, 2003.

[35] G.R. Gircys. *Understanding and using COFF: UNIX common objects file format*. O'Reilly, 1988.

[36] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[37] B. Grant, M. Philipose, M. Mock, C. Chambers, and S.J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 293–304. ACM New York, NY, USA, 1999.

[38] G.J. Hansen. *Adaptive systems for the dynamic run-time optimization of programs.* PhD thesis, Carnegie Mellon University Pittsburgh, PA, USA, 1974.

[39] R. Harold and M. Loukides. *Java network programming*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000.

[40] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# programming language*. Addison-Wesley, 2006.

[41] L.W. Hoevel. "Ideal" Directly Executed Languages: An Analytical Argument for Emulation. *IEEE Transactions on Computers*, 100(23):759–767, 1974.

[42] U. Hölzle. *Adaptive optimization for SELF: Reconciling high performance with exploratory programming*. PhD thesis, Sun Microsystems, Inc. Mountain View, CA, USA, 1995.

[43] U. H
"olzle and D. Ungar. A third-generation SELF implementation: reconciling responsiveness with performance. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, page 243. ACM, 1994.

[44] P. Hudak, J. Hughes, S.P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 12–1. ACM New York, NY, USA, 2007.

[45] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*. Intel Corporation, 2006.

[46] D.E. Knuth. An Empirical Study of FORTRAN Programs. *Software–Practice and Experience*.

[47] J. Koshy and R. Pandey. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pages 243–254. ACM New York, NY, USA, 2005.

[48] U. Kremer, J. Hicks, and J. Rehg. A compilation framework for power and energy management on mobile computers. *Lecture Notes in Computer Science*, 2624:115–131, 2003.

[49] U. Kremer, J. Hicks, and J.M. Rehg. Compiler-directed remote task execution for power management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*. Citeseer, 2000.

[50] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society Washington, DC, USA, 2004.

[51] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace. *RFC4122, July*, 2005.

[52] H.B. Lee, A. Diwan, and J.E.B. Moss. Design, Implementation, and Evaluation of a Compilation Server. *ACM Transactions on Programming Languages and Systems-TOPLAS*, 29(4), 2007.

[53] P. Lee and M. Leone. Optimizing ML with run-time code generation. *ACM SIGPLAN Notices*, 31(5):137–148, 1996.

[54] M. Leone and P. Lee. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. ACM New York, NY, USA.

[55] J.R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[56] J. Li. Distributed Java Just-in-Time Compiler for Resource Constrained Environment. In *Proceeding of 17th Isle Of Throne Annual Graduate Workshop*. University of Sussex, Eastbound, UK, 2004.

[57] J. Li and D. Watson. Proxy Compilation, Even Better Than Ever? In *In Proceeding of 6th Annual Postgraduate Symposium on The Convergence of Telecommunications, Networking & Broadcasting*. Liverpool John Moores University, Liverpool, UK, 2006.

[58] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[59] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*, pages 43–46. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[60] Digital Mars. D programming language. http://d-programming-language.org.

[61] J. McCarthy. *Recursive functions of symbolic expressions and their computation by machine, Part I*. ACM New York, NY, USA, 1960.

[62] Sun Microsystem. Java programming language. http://www.java.com.

[63] Sun Microsystem. Learn About Java Technology, 2009. http://www.java.com/en/about.

[64] S.S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[65] M. Newsome. *Thesis: Proxy Compilation of Dynamically Loaded Java Classes for Embedded Systems*. University of Sussex, 2003.

[66] M. Newsome and D. Watson. Proxy compilation of dynamically loaded Java classes with MoJo. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*, pages 204–212. ACM New York, NY, USA, 2002.

[67] F. Noel, L. Hornof, C. Consel, and J.L. Lawall. Automatic, template-based runtime specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*. IEEE Computer Society Washington, DC, USA, 1998.

[68] M.L. Nohr. *Understanding ELF Object Files and Debugging Tools*. Number ISBN: 0-13-091109-7. Prentice Hall Computer Books, 1993.

[69] OSS Nokalva. ASN.1 Tools for Java, 2009. http://www.oss.com/products/asn1java/details.html.

[70] K.V. Nori, U. Ammann, K. Jensen, HH Nageli, and C. Jacobi. Pascal-P implementation notes. *Pascal–The Language and its Implementation*, pages 125–170.

[71] M. Othman and S. Hailes. Power conservation strategy for mobile computers using load sharing. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2(1):44–51, 1998.

[72] P. Overell and D. Crocker. RFC2234: Augmented BNF for Syntax Specifications: ABNF. *RFC Editor United States*, 1997.

[73] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*. USENIX Association Berkeley, CA, USA, 2001.

[74] J. Palm, H. Lee, A. Diwan, and J.E.B. Moss. When to use a compilation service? *ACM SIGPLAN Notices*, 37(7):194–203, 2002.

[75] J. Palm, H. Lee, A. Diwan, and J.E.B. Moss. When to use a compilation service? *ACM SIGPLAN Notices*, 37(7):194–203, 2002.

[76] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, pages 291–300. ACM New York, NY, USA, 1998.

[77] M. Poletto, D.R. Engler, and M.F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. *ACM SIGPLAN Notices*, 32(5):109–121, 1997.

[78] M. Poletto, W.C. Hsieh, D.R. Engler, and M.F. Kaashoek. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, 1999.

[79] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[80] Roldan Pozo and Bruce Miller. Scimark 2.0, 2004. http://math.nist.gov/scimark2.

[81] Marben Products. ASNSDK TCE-Java – Highly Powerful ASN.1 Tools for Java, 2009. http://www.marben-products.com/asn.1/tce_java.html.

[82] The FreeBSD Project. a.out(5) freebsd file formats manual, 1993. http://www.freebsd.org/cgi/man.cgi?query=a.out&sektion=5.

[83] B.R. Rau. Levels of representation of programs and the architecture of universal host machines. In *Proceedings of the 11th Annual Workshop on Microprogramming*, pages 67–79. IEEE Press Piscataway, NJ, USA, 1978.

[84] A. Rudenko, P. Reiher, G.J. Popek, and G.H. Kuenning. Saving portable computer battery power through remote process execution. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2(1):19–26, 1998.

[85] A. Rudenko, P. Reiher, G.J. Popek, and G.H. Kuenning. The remote processing framework for portable computer power saving. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 365–372. ACM New York, NY, USA, 1999.

[86] H.J. Saal and Z. Weiss. A software high performance APL interpreter. *ACM SIGAPL APL Quote Quad*, 9(4):74–81, 1979.

[87] T. Sayeed, A. Taivalsaari, and F. Yellin. Inside The K Virtual Machine, 2001.

[88] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 66–82. ACM New York, NY, USA, 2000.

[89] T. Shackell, N. Mitchell, A. Wilkinson, et al. Yhc: The York Haskell Compiler.

[90] N. Shaylor. A Just-In-Time compiler for memory constrained low-power devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*. USENIX Association, San Francisco, California, USA, 2002.

[91] N. Shaylor, D.N. Simon, and W.R. Bush. A Java virtual machine architecture for very small devices. *ACM SIGPLAN Notices*, 38(7):34–41, 2003.

[92] E.G. Sirer, R. Grimm, A.J. Gregory, and B.N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 202–216. ACM New York, NY, USA, 1999.

[93] M.D. Smith. Overcoming the challenges to feedback-directed optimization (Keynote Talk). *ACM SIGPLAN Notices*, 35(7):1–11, 2000.

[94] D. Steedman. *Abstract syntax notation one (ASN. 1): the tutorial and reference*. Technology Appraisals, 1990.

[95] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[96] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 180–195. ACM New York, NY, USA, 2001.

[97] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a java just-in-time compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*. ACM New York, NY, USA, 2002.

[98] Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. *Java Object Serialization Specification*, 1.4.4 edition, 2001.

[99] Transvirtual Technologies. Kaffe project homepage. http://www.kaffe.org.

[100] R. Teodorescu and R. Pandey. Using jit compilation and configurable runtime systems for efficient deployment of Java programs on ubiquitous devices. *Lecture Notes in Computer Science*, pages 76–95, 2001.

[101] T.L. Thai and H. Lam. *.NET framework essentials*. O'Reilly & Associates, Inc., 2002.

[102] Inc. The MathWorks. Matworks: Products. http://www.mathworks.com/products.

[103] A. Varma. *A retargetable optimizing Java-to-C compiler for embedded systems.* PhD thesis, University of Maryland, 2003.

[104] KS Venugopal, G. Manjunath, and V. Krishnan. sEc: A portable interpreter optimizing technique for embedded Java virtual machine. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM.02*, pages 127–138, 2002.

[105] M.J. Voss and R. Eigenmann. A framework for remote dynamic program optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 32–40. ACM New York, NY, USA, 2000.

[106] M.J. Voss and R. Eigenmann. A framework for remote dynamic program optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 32–40. ACM New York, NY, USA, 2000.

[107] W3C. Simple object access protocol (SOAP) version 1.2, 2007. http://www.w3.org/TR/soap/.

[108] W3C. Extensible Markup Language (XML), 2009. http://www.w3.org/XML/.

[109] L. Wall and M. Loukides. *Programming perl.* O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000.

[110] I. Wirjawan, J. Koshy, R. Pandey, and Y. Ramin. Balancing computation and code distribution costs: The case for hybrid execution in sensor networks. *University of California, Davis, Technical Report TR-CSE-2006-35*, 2006.

[111] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. June, 1996.

[112] ITU-T Recommandation X.691. *Information Technology–ASN. 1 encoding rules: Specification of Packed Encoding Rules (PER).* Telecommunication Standard Sector of ITU, 2002.

[113] B.S. Yang, S.M. Moon, S. Park, J. Lee, S.I. Lee, J. Park, YC Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: a Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of Parallel Architectures and Compilation Techniques*, pages 128–138. ACM New York, NY, USA, 1999.

[114] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11), 1997.

[115] W. Zhu, C.L. Wang, and F. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. page 381, 2002.