A JAVA VIRTUAL MACHINE FOR THE ARM PROCESSOR

A report submitted to the University of Manchester
for the purpose of a Master of Science
in the Faculty of Science and Engineering

THE UNIVERSITY
of MANCHESTER

September 2004

By
Ming Chen
Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

Virtual machines, in particular the Java Virtual Machine (JVM), offer the ability for applications to be developed in a platform-independent manner and gain ever increasing popularity. The Jikes Research Virtual Machine (RVM) is a JVM that written in Java. This thesis presents a port of the Jikes RVM to the important embedded architecture, ARM.

Substantial work has been done porting the initial baseline compiler of the Jikes RVM. The work is presented, along with considerations for the optimising and adaptive compilation systems.

Not only is ARM an important embedded architecture, but also it is starting to extend a function as a Chip-Multi-Processor (CMP) architecture. This work also presents consideration of the Jikes RVM running in a ARM MPCore environment.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

The Author

Having graduated from the South China University of Technology with a BEng in Automation, I then worked in the R&D sectors of the Telecommunication industry. After four years working, I began the MSc in Advanced Computer Science with ICT Management course at the University of Manchester. The last part of this course is the project described in this dissertation. This work was undertaken as part of the Jamaica Project in the Advanced Processor Technologies Group.

Acknowledgements

Thank you to my supervisor Ian Watson, the research fellows, Andrew Dinn and Dr. Ian Rogers. Their 'real-time' support was critical for this project, and their professional spirit encouraged me to overcome many difficult situations.

Thanks to the other colleges in the JAMAICA and APT groups, Steve Furber, Chris Kirkham, Matthew Horsnell, Zhao JiSheng, their time and help is always invaluable.

Thanks to the experts who helped me through the Jikes RVM-researchers mailing list, Eliot Moss, David P Grove, Perry Cheng and Chris Hoffmann.

Thanks to all my other friends, my family and those who have helped me in the Acomb house. Spepial thanks to my best friend Fu,Xiaoyan for give me a lot of valuable feedback on my thesis.

Chapter 1

Introduction

The Jikes Research Virtual Machine (RVM) is a research oriented Java Virtual Machine (section 2.1.1) developed by the IBM T.J.Watson Research Centre. The flexible architecture design makes it suitable for Java Virtual Machine research. In the JAMAICA research group, the Jikes RVM platform is being leveraged to research into dynamic compilation techniques for parallel architectures. Most of the Jikes RVM is machine independent. There are two challenges in this project: one is to identify the machine dependent design; the other is to debug the ported system.

1.1 Purpose

This thesis describes the design and implementation of an ARM port of the Jikes RVM. The ARM backend of the Jikes RVM, built by this project, can be used in the following two potential research areas, which will be discussed as the further research work in section 6.2.2.

- Low-power oriented dynamic compiling optimisation for the ARM architecture.
- Compiling support of parallel computing for the ARM MPCore chip-multi-processor.

1.2 Scope

Considering the time limitation, this project has focused on a prototype implementation. This prototype system is based on the baseline compiler subsystem

and is developed it to a level where small Java programs such as 'HelloWorld' can be run.

1.3 Overview

There are six core chapters in this thesis as follows:

- Chapter 2, Background

  Chapter 2 will give an overview of the technical background of this project. Firstly, it will briefly introduce the architecture of the Jikes RVM, followed by a brief overview of the Java and Java Virtual Machine. After the analysis of the Jikes RVM, it will examine the architecture independent features of the main components of the Jikes RVM. Secondly, it will discuss the working mechanism of the baseline compiler subsystem. Finally, it will introduce the target hardware and operating system.

- Chapter 3, Design and Implementation

  Chapter 3 will explain the critical decisions made on this project. There are ten subtopics in this chapter.

  1. Methodology
     - Port from the PowerPC implementation
     - A Minimal Approach

     It introduces the methodologies and philosophies applied to this project.
  2. VM_Convention

     It shows the special virtual machine convention for the ARM processor.
  3. Register Map

     Because the ARM processor has just fifteen general purpose registers, it is necessary to reallocate the registers to match the requirement of the Jikes RVM.
  4. Register Zero

     The register zero has special meaning in the PowerPC's instruction. In the ARM, it is necessary to deal with register zero differently.

5. Synchronisation

   The ARM has a different synchronisation mechanism for locking than the PowerPC.

6. Application Binary Interface

7. Endian and Word Order

8. Limited length of offset and immediate operand

   In the ARM, the length of offset of the instructions is smaller than PowerPC's. There are some design decisions related to it.

9. System Trap

   It will present the different implementations of system trap in ARM, Intel and PowerPC, and then it shows the solution chosen on this prototype implementation.

10. Floating Point

    It will examine the floating point implementation of this ARM port.

- Chapter 4, Result Analysis

  This chapter evaluates this prototype ARM port of the Jikes RVM.

  1. Achievements of this porting project
  2. Limitation of implementation
  3. Validation and testing
     - Testing the Assembler
     - Testing the Compiler

- Chapter 5, Engineering Experiences

  This chapter discuss the engineering challenges and solutions in this project.

  - Milestones of this porting project
  - Exception Driven Modification
  - Debugging the Jikes RVM

    This concerns guidelines about how to use gdb to debug Jikes RVM.
  - Cross Compiling

- Chapter 6, Conclusions and Further work

    1. Conclusion

       This will introduce the current progress of this project and add conclusion for this stage.

    2. Further work

       There are some suggestions for further work on this project.

Chapter 2

Background

Java is one of the most popular high-level programming languages of this networking age. Its platform independence, security and mobility features make it suitable for network-oriented computing.

Java's executable code, byte code, which is an intermediate code, can not be directly executed by the processor. It must be executed by a Java Virtual Machine (JVM). Compared with native executable code, executing byte code by a running JVM requires additional overhead, such as 'Just-in-Time' compilation , which may affect the performance, however, Java's features, like strong typing and dynamic execution, make it possible to increase its performance with compiler optimisation. A JVM can use run-time profile data to optimise the Java program dynamically. For some benchmarks, running on some JVM implementations, Java can achieve a similar performance and better than C.

Currently with the development of chip multi-processor architectures, JVMs might provide benefits from dynamic optimisation for parallel computing.

In this chapter, we will introduce the relationship between the Java Virtual Machine and the Jikes RVM, the architecture and machine dependent features of the Jikes RVM, as well as the development environment of this project.

2.1 JVM and Jikes RVM

The Jikes RVM is one of the state-of-the-art Java Virtual Machines.

2.1.1 Java and the Java Virtual Machine (JVM)

The Java Virtual Machine is the key component of the Java architecture. It is the heart of the Java network oriented features[Ven97]. The JVM provides a runtime environment to support these features.

The JAVA Programming language

Java's suitability for networked environments is inherent in its architecture, which enables secure, robust, platform-independent programs to be delivered across networks and run on a great variety of computers and devices. Platform independence, security, and network-mobility–these three facets of Java's architecture work together to make Java suitable for the emerging networked computing environment [Ven97].

JAVA Architecture

Java's architecture arises out of four distinct but interrelated technologies:

- The Java programming language
- The Java class file format
- The Java Application Programming Interface
- The Java Virtual Machine

The writing and running of a Java program starts from Java source code written in the Java programming language. Then the source code is compiled to the Java class file format, which contains Java Bytecode. If needed, operating system resources can be accessed by the Java Programming Interface at the source code level. Finally, the Java class file would be executed on the Java Virtual Machine.

Java Virtual Machine Architecture

A JVM's responsibility is to execute Java Bytecodes on the target hardware platform. In general, a Java Virtual Machine is platform dependent. There are different JVM implementation for different operating system and CPU architecture.

2.1.2 Jikes RVM

The Jikes RVM is a new generation Java Virtual Machine, developed by IBM J.Watson Research Centre[IBM04]. The Jikes RVM's well-designed architecture makes it flexible with the different memory management modules and different optimising algorithms, which makes it an ideal research platform for compiling technology. It is also written in Java.

2.1.3 Structure of the Jikes RVM

The Jikes RVM has four key components: core runtime, compilers, memory managers and an adaptive optimisation system [IBM04].

- Core runtime is responsible for managing all the underlying data structures required to execute applications and interfacing with libraries. An example of runtime objects are those responsible for threading.
- Compilers are responsible for generating executable code from byte codes. The executable code is held as Java arrays. Special VM_Maigc calls enable the Jikes RVM to incorporate with this code into its running image.
- Memory managers are responsible for the allocation and collection of objects during the execution of an application.
- Adaptive optimisation system is responsible for profiling an executing an application and judiciously using the optimising compiler to improve its performance.

2.1.4 Platform Independent Features of the Jikes RVM

In the four major components of the Jikes RVM, only the compiler subsystems are platform dependent. The other three components are platform independent.

2.2 Boot Image

As a Java Virtual Machine written in Java, Jikes RVM uses a boot image, which is a snap-shot of a working Jikes RVM, to boot itself. There is a short C bootstrap program, called the boot image runner, that reads the boot image from a file,

loads it into memory, and starts its running. Another important program is the boot image writer. This is a Java program that runs on an existing JVM (in this project, we use the Java HotSpot[1] Virtual Machine 1.4.2). In the boot image file, a minimal set of java classes are compiled into machine codes. The Jikes RVM compiler itself would be compiled into the boot image as well.

VM.boot()

VM.boot() is the first java method executed in the boot image file. It is a suitable place to add some bytecode test code, as discribed later in the testing section 4.3.

2.3 Compiler Subsystem

The function of the compiler subsystems of the Jikes RVM is to generate machine code from Java bytecode. In the compiler subsystem, there are two kinds of compiler: the baseline compiler and the optimising compiler. The difference between them is in the compilation time and the quality of the generated code.

The mechanisms of the baseline compiler are simple and straightforward. It directly translates bytecode into machine code. Therefore, the baseline compiler can generate machine code very quickly, but the performance of the machine code generated by it is relatively poor.

The optimising compiler has complex optimisation routines that create high performance machine code, however this comes at a higher cost of a longer compile time.

The Jikes RVM has an adaptive optimisation system to trade off the compiling time with the performance gain, based on a runtime profile. The basic idea is to use the optimising compiler to compile a small set of performance critical Java methods, and use the baseline compiler to compile all the other performance non-critical Java methods, which is referred to the 90/10 rule(90% of execution time occurs in 10% of the code[HP96]),

This project focuses on the implementation of the baseline compiler.

[1]According to Sun Microsystems, the HotSpot virtual machine, Sun's Java virtual machine (JVM), promises to make Java "as fast as C++." Specifically, Sun says that a platform-independent Java program delivered as bytecode in class files will run on HotSpot at speeds on par with an equivalent C++ program compiled to a native executable [Ven98].

2.3.1 Baseline compiler working mechanism

Most of the baseline compiler is machine-independent. The main class is VM_BaselineCompiler. Figure 2.1 shows the function of the VM_BaselineCompiler is to invoke VM_Compiler to emit the machine code for each bytecode. The body

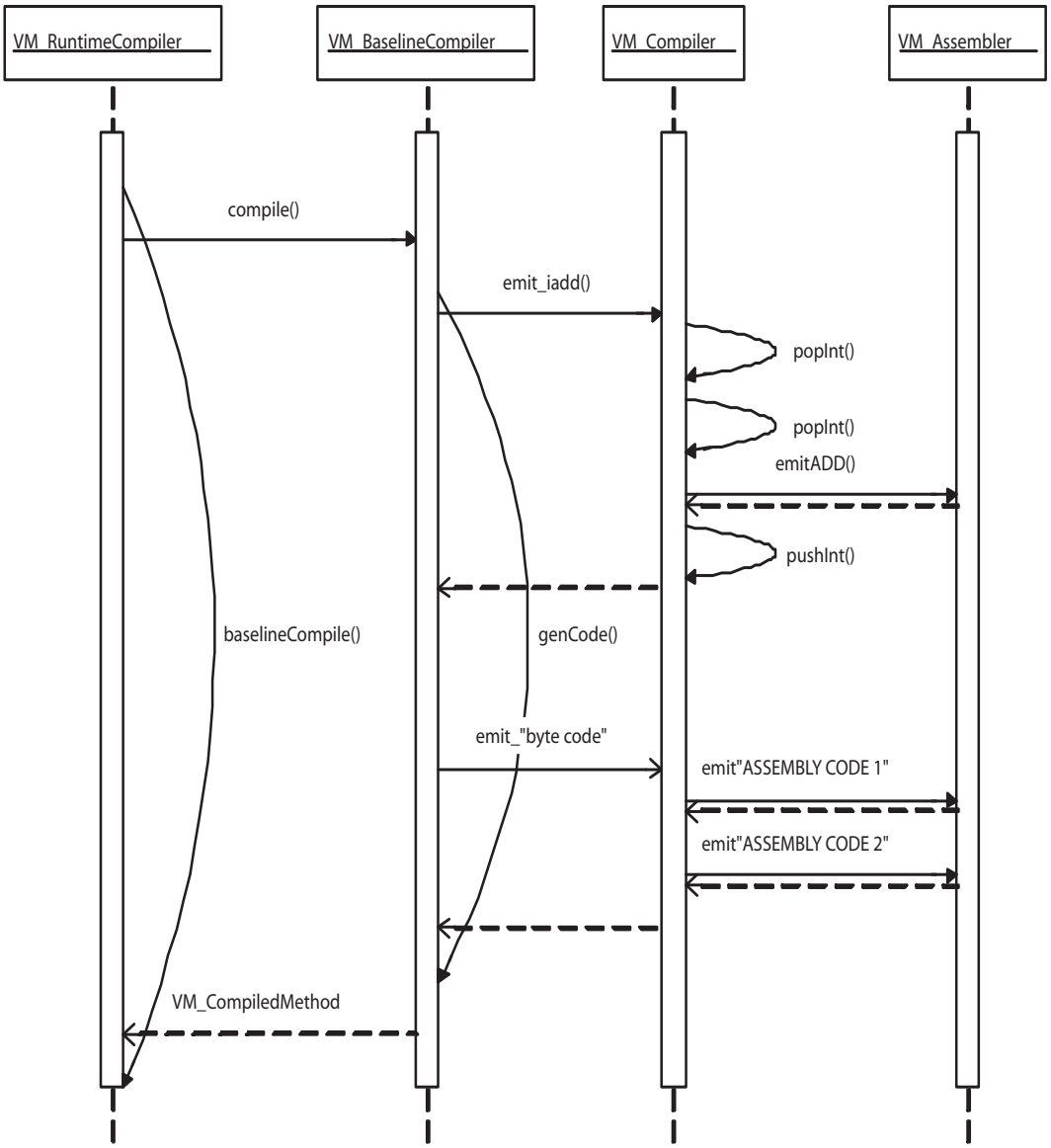

Figure 2.1: Call path of Baseline Compiler

of the compile method in the baseline compiler is a big *switch* table. For each bytecode, it invokes different emit_'bytecode'() in the VM_Compiler. For the

VM_BaselineCompiler, there is no difference among the different processors, because the bytecode is the same. Therefore the VM_BaselineCompiler is machine-independent.

Further as shown in Figure 2.2, the VM_Compiler and VM_Assembler are machine dependent. For the same bytecode, the VM_BaselineCompiler should be the same, whilst the VM_Compiler will generate a different assembly code sequence according to different processors. Obviously the VM_Assembler should be machine dependent, because its function is to generate binary machine code for each of the different processors.

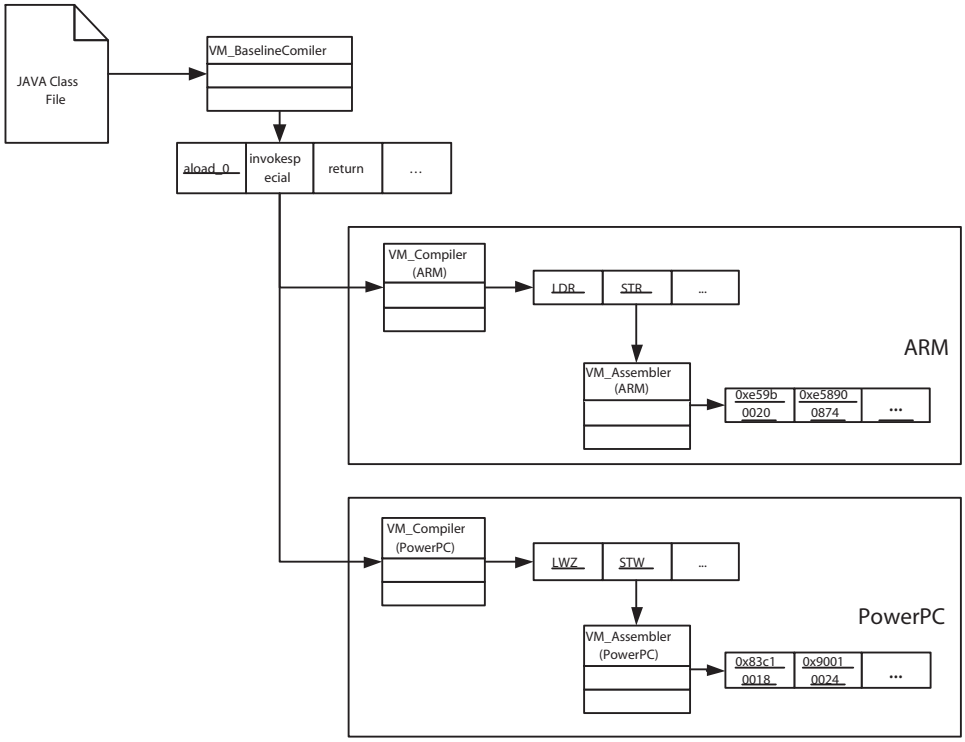

Figure 2.2: Baseline Compiler Work Mechanism

Baseline compilation consists of two main steps: GC map computation [2] and code generation. Code generation is straightforward, consisting of a single pass through the byte codes of the method being compiled.

[2]The baseline compiler computes GC maps by abstractly interpreting the byte codes to determine which expression stack slots and local variables contain reference at the start of each bytecode[IBM04].

2.4 Target Hardware Environment

2.4.1 ARM, MPCore and Intel Xscale

ARM stands for Advanced RISC Machine. The ARM architecture incorporates a number of features from the Berkeley RISC design, but a number of other features were rejected [Fur00]. Those that were used are:

- A load-store architecture
- Fixed-length 32-bit instructions
- 3-address instruction formats

The significant features of the ARM processors, such as high performance, low power consumption and low system cost, have made it the most popular processor in the embedded system market.

The ARM MPCore is an ARM multi-chip-processor, which has four processors on-chip. To support low power optimisation research in a parallel computing environment, the JAMAICA research group decided to build a Jikes RVM backend on the ARM MPCore. Because currently there is not an operating system on the ARM MPCore, the porting has been started from an ARM single chip system.

The Intel Xscale core is an ARM V5TE compliant microprocessor. The Intel Xscale core implements the integer instruction set architecture of ARM V5, but does not provide hardware support for floating-point instructions. It also provides the Thumb[3] instruction set[Int04].

2.4.2 ARM Workstation

This project uses an IYONIX PC ARM workstation, which uses the Xscale processor/600M. The model of workstation is IYONIX PC X200

[3] The Thumb instruction set consists of 16-bit instructions that act as a compact shorthand for a subset of the 32-bit instructions of the standard ARM. Every Thumb instruction could instead be executed via the equivalent 32-bit ARM instruction. However, not all ARM instructions are available in the Thumb subset; for example, there is no way to access status or coprocessor registers. Also, some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of Thumb instructions[ARM00].

2.4.3 Linux

Linux is the most popular free open source operating system. In this project, Debian Linux has been used. The Linux kernel used in this project is version 2.4.22.

2.5 Summary

- Java Virtual Machines in general are platform dependent.
- The design of Jikes RVM makes it mostly platform independent.
- The VM_Compiler and VM_Assembler are the key components to be ported in this project.

Chapter 3

Design and Implementation

This chapter describes some critical design decisions made when porting the Jikes RVM to the ARM processors. Section 3.1 focuses on the engineering principles and philosophies applied to this project, and the other eight sections discuss special technical problems faced and the solutions adopted.

3.1 Methodology

There are two important decisions made in terms of the methodologies applied to this project: firstly, to port the Jikes RVM from the PowerPC implementation rather than the Intel IA32 one; secondly, two special methodologies have been applied, such as 'exception driven modification' and module testing.

3.1.1 Port from the PowerPC implementation

There are two existing implementations of the Jikes RVM: Intel IA32 and PowerPC.

As discussed in section 2.1.4, the porting of the Jikes RVM is focused on the baseline compiler. The core components of the baseline compiler are VM_Compiler and VM_Assembler. VM_Compiler emits assembly code from the Java bytecode; then VM_Assembler is responsible for emitting binary machine code from the assembly code. Whether porting from the PowerPC or the Intel IA32, there is no difference for the VM_Assembler, which must be totally re-designed, but it would make a difference for porting the VM_Compiler.

The Intel IA32 is a CISC architecture, whilst both PowerPC and ARM have

some common RISC features, such as load and store architecture, fixed-length 32-bit instructions and 3-address instruction formats. For this reason, most of the design of the VM_Compiler for the PowerPC can be reused by an ARM version, while porting from the Intel IA32 would require major changes.

In a load-and-store architecture, only registers can be operands of data processing instructions. It is necessary to load data from memory first, and then process the data, finally store the result back to memory. In a CISC architecture, like the Intel IA32, there is no such restrictions. It can use memory addresses as operands directly. In the Intel IA32, just one instruction can replace three instructions on the ARM and PowerPC. For example, in order to generate machine code for bytecode *iinc* (Increment a local variable by a constant), there are three steps in the PowerPC and ARM:

1. Load the value from memory to register.
2. Increase the value in register by constant.
3. Store the value from register to memory.

However, for the Intel IA32, one step is enough. It can directly use a memory address as a operand, thus just one *add* instruction is sufficient.

Therefore, in this project, the port is based on the PowerPC implementation of the baseline compiler, whilst the Intel IA32 implementation can provide a reference in some cases. In this way, we can reuse the existing design for the RISC architecture in the Jikes RVM.

3.1.2 A Minimal Approach

The Jikes RVM is a complex system. There are 12,993 lines of code (567 methods) in the Jikes RVM/PowerPC's baseline compiler.

The Jikes RVM is mostly written in Java. It cannot initially run on itself, so it uses another Java Virtual Machine to execute its Java bytecode, which generates a binary boot image file. Then it uses a small bootstrap program written in C, to load the boot image file into memory and execute it. This process generates all of its symbols internally and does not create a symbol table for a debugger, such as gdb, so it is impossible to trace the machine code compiled from each method. For this reason, it is difficult to use gdb to debug the binary machine code in the boot image.

Considering these two issues, it is easy to introduce bugs in such a complex system and hard to find them with limited debugging tools.

As a result, the philosophy of this project is to change as little as possible to decrease the possibility of generating bugs. Therefore the project plan is to make 'HelloWorld' work with minimum modifications at first, then focus on optimising it later. Chapter 5 has some further discussion about the details of applying these two methodologies in practice.

3.2 VM Convention

In this section, the register, stack, and calling conventions, which apply to the Jikes RVM on the ARM processor, will be described.

The stack frame layout of the ARM is the same as the PowerPC, but the calling conventions are modified slightly to fit with the difference in registers of the ARM.

3.2.1 Register Categories

In the PowerPC implementation, the general registers can be roughly categorised into four types[IBM04]:

- Scratch: Needed for the prologue[1]/ epilogue[2] of a method. They can be used by the compiler between method calls. As in the PowerPC, there are two scratch registers in the ARM.
- Dedicated: Reserved registers with known contents, such as JTOC (Jikes RVM Table of Contents), FP (Frame Pointer), PR (Processor Register).
- Volatile (caller save, or parameter): These can be used by compilers as temporaries but they are not preserved across calls. They are different from scratch registers in the way that they can be used to pass parameters and result(s) to and from methods.

[1]The responsibilities of method prologue are: 1. Execute a stack overflow check, and grow the thread stack if necessary;2. Save the caller's next instruction pointer; 3. Save any nonvolatile floating-point registers used by callee; 4. Save any nonvolatile general-purpose registers used by callee; 5. Store and update the frame pointer FP; 6. Store callee's compiled method ID; 7. Check to see if the Java thread must yield the VM_Processor.

[2]The responsibilities of method epilogue are: 1. Restore FP to point to caller's stack frame; 2. Restore any nonvolatile general-purpose registers used by callee; 3. Restore any nonvolatile floating-point registers used by callee.; 4. Branch to the return address in caller.

- Non-volatile (Callee save, or preserved): They may be used but must be saved on method entry and restored at method exit.

3.2.2 Stack Conventions

Stacks grow from high memory to low memory in the PowerPC. The layout of the stack frame is not changed for the ARM.

3.3 Register Map

On the ARM, there are just 15 general-purpose registers available, less than one half of the number in the PowerPC, which has 69 registers, 32 of them are General Purpose Registers (GPRs). It is necessary to examine how to allocate the limited number of registers to make the RVM work with minimum changes.

First of all, in order to be compatible with the register layout of a Linux system call, it is necessary to conform to the Procedure Call Standard for the ARM Architecture (AAPCS) [Ear03]. See Figure 3.2, the parameters are passed in the registers (r0-r3) and on the stack, so r0-r3 are used as OS_Parameter_Volatile registers and r4-r13 are used as OS_Nonvolatile registers.

On this PowerPC, shown in Figure 3.1, scratch registers are a part of OS_Nonvolatile registers. The Jikes RVM needs two scratch registers. If we allocate the scratch registers in the same way, there are just two volatile registers available to pass the parameters, which is not enough for the Jikes RVM. Some bytecode operations, such as resolved_newarray, are so complex that they need seven registers. On the ARM, there are 15 registers available. In these 15 registers, there are 4 registers (link register, stack pointer register, procedure-call scratch register and frame pointer) defined on the AAPCS, which are necessary for the operating system and virtual machine. At the same time, there are three register reserved by the virtual machine, which are JTOC, Processor Register and KLUDGE_TI_REG. So there are just eight registers left for the volatile, scratch and nonvolatile registers. On this ARM port, these three need to share the remaining eight registers. So there is no choice but to make the scratch register sector overlap the non-volatile register sector. This means that some scratch registers need to be stored in or loaded from the stack when entering and exiting a method. Figure 3.2 shows the whole register map of the ARM port.

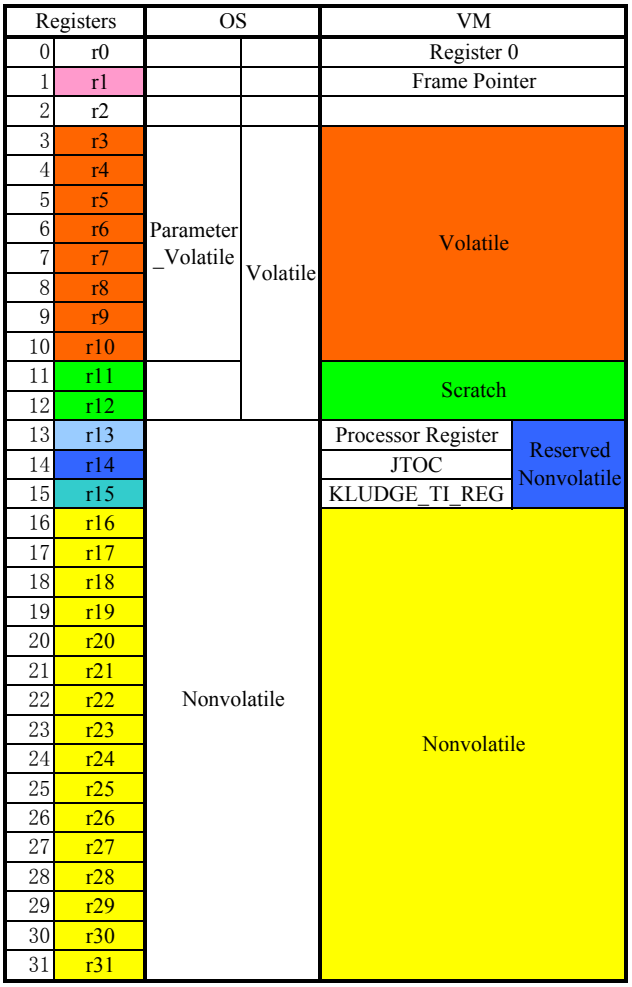

Figure 3.1: Register Map of the PowerPC Implementation

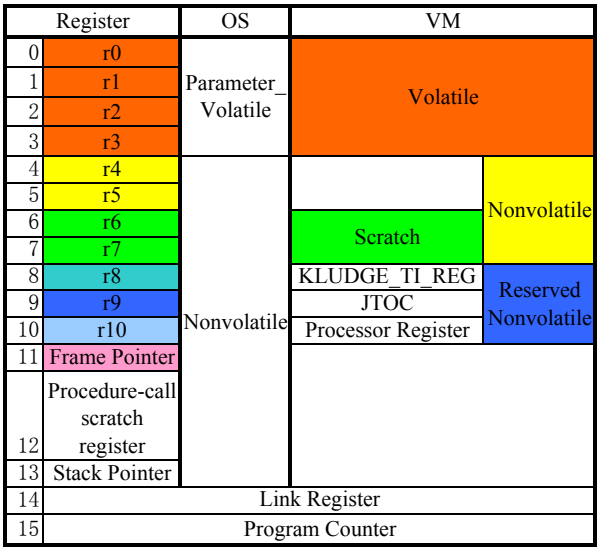

Figure 3.2: Register Map of the ARM port

3.4 Synchronisation

In the Jikes RVM, there are six different mechanisms to handle synchronisation issues. For normal library code and most VM code, monitorenter and monitorexit are sufficient. The other four lower-level primitives provide building blocks for implementing monitorenter and monitorexit. Some VM systems, such as thread scheduling and GC, resort to lower-level primitives for situations where normal Java object locking is inconvenient or illegal.

There are four different lower-lever synchronisation mechanisms in the Jikes RVM. All of the others are based on VM_Magic.prepare and VM_Magic.attempt. Therefore VM_Magic.prepare and VM_Magic.attempt are the lowest level synchronisation mechanisms of the Jikes RVM. They are platform dependent. As a result, in this project most of the attempts are put into the following two operations:

1. Magic prepare and attempt calls

2. Protect VM_Processor.vpStatus.

3.4.1 Magic prepare and attempt calls

Generally the prepare call fetches the contents of a memory location and begins a conditional critical section. The attempt call ends the conditional critical section. This attempt call will return true if and only if there were no intervening writes to the guarded memory location.

- PowerPC

  On the PowerPC, the compilers implement prepare and attempt using the *lwarx* and *stwcx* instructions.

  Lwarx and *stwcx* are used as a pair. *Lwarx* is used at the start of the code which needs synchronisation protection. It loads the value from memory like a normal load instruction and also creates a RESERVE for use by a store word conditional indexed (*stwcx*) instruction in the cache. When the RESERVE has been set, the processor enables hardware snooping for the block of memory addressed by the RESERVE address. If the processor detects that another processor writes to the block of memory it has reserved, it will clear the RESERVE bit. At the end of the protected code, the *stwcx* instruction will check whether the RESERVE is still available. If nothing touched the block of memory, while protected code is executing, it will store the changed value back to the reserved memory address [Mot01]. As discussed above, the combination of *lwarx* and *stwcx* fits the 'magic prepare' and 'magic attempt' mechanism. In the 'magic prepare' *lwarx* loads the value from the memory and reserves the memory. *Stwcx* will check and store back the changed value on the 'magic attempt'.

- IA-32

  On the Intel IA-32, there are no such instructions as *lwarx* and *stwcx*. It uses a more complex approach. In the 'magic prepare' part, it uses the normal load to load the value from an address. In the 'magic attempt', it uses CMPXCHG, which can compare and exchange atomically. It exchanges the value if the compare is successful. So it can be used to check whether the lock is available. If no other processor (program) tried to change the protected memory, it will update the memory with new value. Otherwise it will leave it unchanged and return fail.

- ARM

  On the ARM, there is a SWP instruction. It does a simple atomic swap operation without compare. Comparing with the other architectures, IA32 and PowerPC, the solution of ARM is similar to IA32. The solution is to use normal load on the 'magic prepare', then use the SWP instruction to implement a lock on the 'magic attempt'. Because the SWPs function is so simple, it does not provide a compare and conditional write facility, which must be done explicitly.

3.4.2 Protect VM_Processor.vpStatus

In the Jikes RVM, VM_Processor is created on a operating system pthread. The vpStatus is the field of VM_Processor which stores the status of the thread. There are three options: IN_JAVA, IN_NATIVE and BLOCKED_IN_NATIVE. Most of time the status should be IN_JAVA. This shows that the thread is executing Java code. When it is going to invoke a C function or a system call, the status should be changed to IN_NATIVE; when the code exits from C code, the status should be changed back to IN_JAVA. Sometimes a C function or system call cannot return immediately as they are waiting for a hardware response. At that time, the status of VM_Processor should be BLOCKED_IN_NATIVE. In the JVM, there are several Java threads which share one VM_Processor. The transition from IN_NATIVE to BLOCKED_IN_NATIVE is not done by the pthread which is running the native code. Indeed another glue thread writes the field whilst the executing pthread is blocked on an I/O call. These two threads are possibly going to change the status at the same time. Therefore the change of the status of VM_Processor should be atomic.

There are two places where the status of VM_Processor is changed in the Jikes RVM. Both are used to call a C function from Java. One is in VM_OutOfLineMachineCode, in which, it implements certain Magic calls; the other place is in the VM_JNICompiler, where it is used to plant code to invoke a JNI function.

On the ARM, there is only one instruction for the synchronisation. It is SWP. Its function is simple. It can atomically swap the value from

the memory and registers. If a semaphore function is needed, the ARM architecture suggests the following algorithm:

```
Do
   Mark the value of the address as 'I am looking' by SWAP.
   If someone else is looking at that address,
      then atomic update fails.
   Else if someone is using the address,
      then atomic update fails, restore the original value to the
address.
   Else no one else is looking at or using at the address,
      then mark the address as 'I am Using' by SWAP.
      Now it is safe to do some operations on the value get from the
address.
      Update the address as 'Free to use', atomic update is
successful.
   End if
While atomic update fails;
```

[ARM00]

Following this idea, this porting used the additional 'I am looking' status to guarantee the comparison operation itself to be atomic.

3.5 Application Binary Interface

On the PowerPC, long parameters are stored in the two general purpose registers and should be stored starting from an odd numbered register [ppc96]. But in the ARM, there is no such requirement.

This effects the program to generate the prologue for each call.

3.6 Endian and Word Order

The PowerPC and ARM(Xscale) both can be configured to either Big-endian or Little-endian byte ordering. The PowerPC default is Big-endian whilst ARM's is Little-endian.

There are two important features to be considered in this porting project.

1. Byte ordering

   Shown in Figure 3.3, the ARM and PowerPC have different byte ordering. Therefore the load and store programs must be changed, especially to load and store the short data types, whose length is smaller than 32 bits. On the PowerPC the load starts from the highest bits of the memory. On the ARM, it starts from the lowest bits.

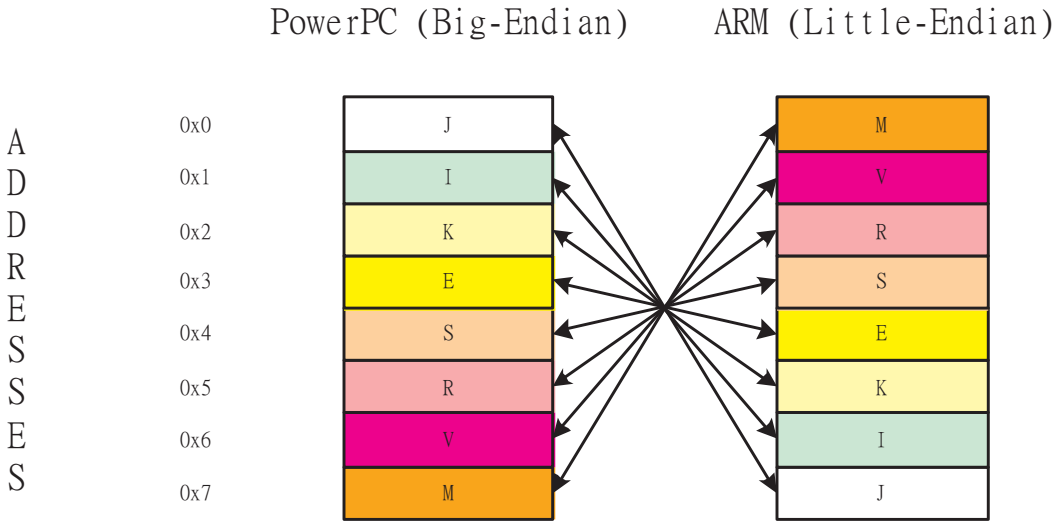

Figure 3.3: Comparison of the Byte Ordering between the PowerPC and ARM

2. Word ordering

   In most cases, word ordering is the same as byte ordering. But in ARM Linux, the word ordering of Long and Double are different. Long follows the Little Endian (Figure 3.4) order while Double is Big Endian (Figure 3.5).

For this reason, some parts of the Jikes RVM have been changed to be platform independent, such as bootImageWriter. Load and store long and double should use different code.

3.7 Register Zero

On the PowerPC, if register 0 is used as an operand, it means the value 0 instead of the value of the memory, addressed by register 0. According to the Application

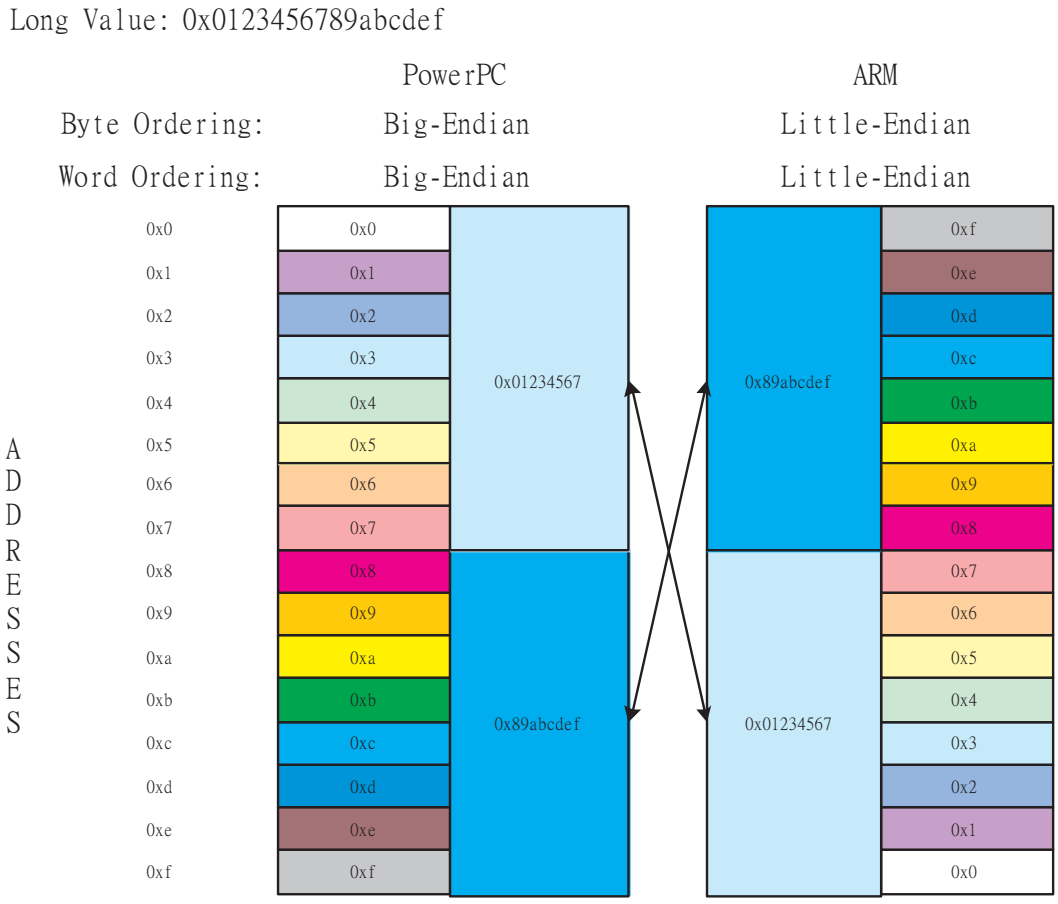

Figure 3.4: Comparison of the Word Ordering of Long Value between the PowerPC and ARM

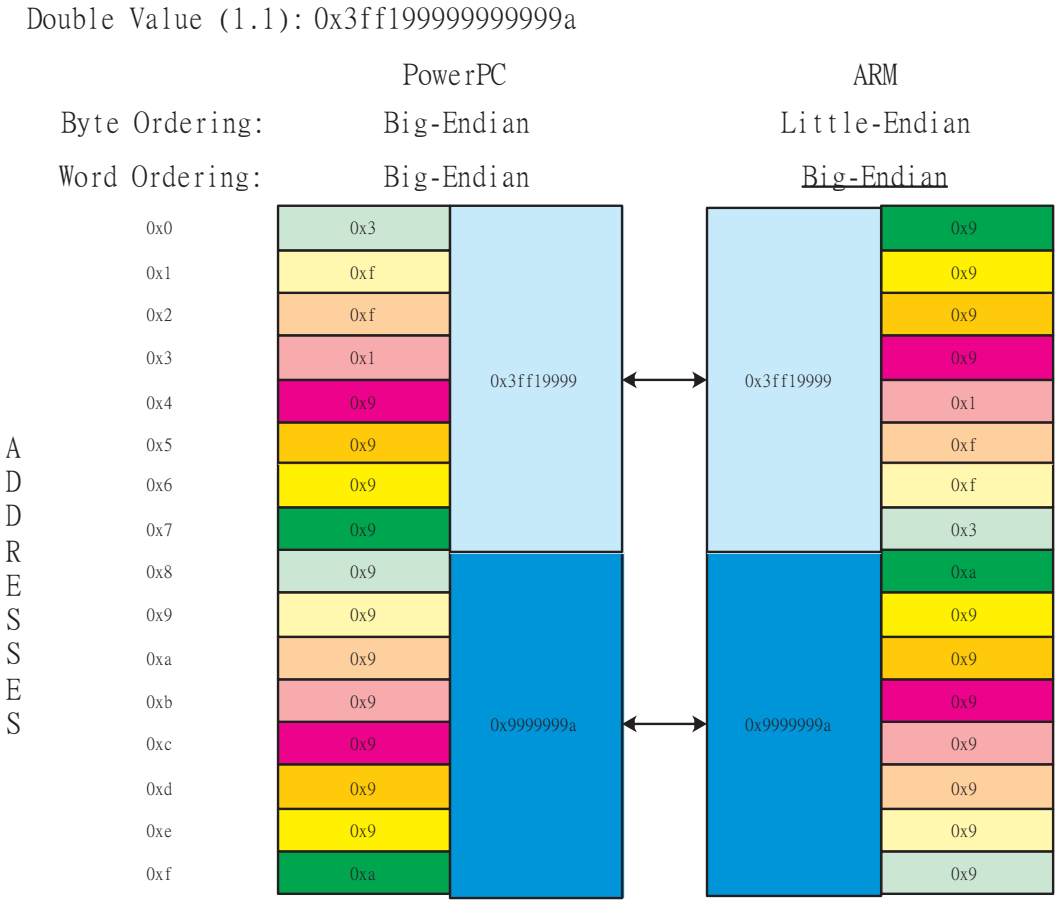

Figure 3.5: Comparison of the Word Ordering of Double Value between the PowerPC and ARM

Binary Interface of the PowerPC, register 0 is not used to transfer parameters between methods. Register 0 is only used in the prologue program.

On the ARM, register 0 has been used as a parameter register. The first argument of a method is stored on it. Since register 0 is used by the prologue program on the PowerPC, the register 0 has been replaced by 'scratch register 0' (S0).

3.8 Limited length of offset and immediate operand

Most of the data processing instructions in the PowerPC can have an 16-bit immediate operand. But on the ARM, most instructions can have only an 8-bit immediate value. There is a similar difference between data transfer instructions, ARM has a 12-bit offset, while the PowerPC has 16-bit offsets. As a result, when the immediate or offset are too long for one instruction, it is necessary to use several instructions to implement a similar function to the one instruction of PowerPC. Basically the solution is that when the length of the offset or immediate is longer than the limitation of an instruction, the value is loaded to the scratch register (S0 and S1), then used as the operand to complete the operation. The most common example is to load a constant to a register. A 32-bit constant needs four move and shift instructions to implement.

3.9 System Trap

In the Jikes RVM, the system trap mechanism is used to tell the operating system that a fatal run-time error has happened in the Jikes RVM. For example, dividing by zero, or trying to write a value to a negative address.

In the PowerPC, the trap would generate a software interrupt and then handle it using a SIGTRAP signal handler. Linux will catch this signal and execute the necessary signal handler program. This mechanism is similar to event listeners. At boot time a series of signal handlers are setup in advance. If a signal is generated, the relative pre-set handler would be invoked by the operating system.

There are three potential design solution for the System Trap an ARM as follows:

Solution 1, to use a similar SIGTRAP mechanism to PowerPC and IA32. This is difficult.

Solution 2, to use the Java call directly. Compared with solution 1, this solution is simple and straightforward, so it consumes less system performance. But it cannot handle a system error. To make this port more robust, it is necessary to install some system error handlers to process some system level errors.

Solution 3, cause a memory fault deliberately when it is necessary to stop the Jikes RVM, such as store a value into a negative address on purpose.

Currently, solution 3 has been used. Later solution 2 will be considered at the system optimisation stage.

3.10 Floating Point

A minimal port of ARM's Jikes RVM still needs floating point support. For example, the memory management unit uses floating point to manage the heap. There are two kinds of floating point solutions:

1. Hardware Floating Point

   The compiler emits opcodes designed to be used with a hardware floating point coprocessor (FPU). The FPU usually has a set of extra arguments for its use, and the compiler may pass floating point arguments to functions through those registers. This should gain higher performance than a software floating point solution by using the computing power of the FPU.

2. Software Floating Point

   In this case, there is no hardware support of floating point. The compiler convert floating point operations into function calls and a special library is used to provide all functions performing the required operations. In this case, standard registers are used instead of the floating point registers. All the floating point arguments have to be passed through standard registers.

The Xscale processor, used on this project, has no hardware floating point support. If using the software floating point solution, it should gain better performance, but refer to the traditional compiler implementation of ARM Linux, and considering the compatibility issues. In this port, hardware floating point has been used. Later it could be optimised to use soft floating point.

How does hardware floating point work without FPU hardware? The CPU will raise an invalid instruction exception each time a FPU opcode is encountered.

Then the Linux kernel will trap this exception, look at the given FPU instruction and emulate it in software. Relatively, it has worse performance than a software implementation. But it can reuse the existing floating point trap and process mechanism in the ARM Linux kernel, whilst it can improve the compatibility between the different ARM Linux platforms of this Jikes RVM port.

On the ARM, compared with the floating point instruction set of the PowerPC, the ARM has some special features.

Firstly, The ARM cannot use a register as an offset operand in the floating point instruction sets, it only can use an immediate value as offset operand, which means that it is impossible to use the same approach as the fixed point instruction sets which can load the offset to a register, then use the register as offset operand, when the practical offset is bigger than the limitation of an immediate offset operand. The solution is to add the base register to the offset, save the result to a scratch register, then use the scratch register as baseline register and zero as immediate offset operand.

Secondly, in the floating point data operation sets, there is an instruction to round a floating value to an integer. On the PowerPC, the destination register is a floating point register whist on ARM, it is a general purpose register. This has been considered in this porting project.

3.11 Summary

This chapter has introduced nine major problems and solutions on this porting project. All of them arise from the different features of the PowerPC and ARM processors. Most of the problem presented in this chapter were discovered by debugging. In the following chapters, we discuss the testing (section 4.3) and debugging (section 5.3) issues.

Chapter 4

Result Analysis

This chapter will examine the achievements of this project and the methodology to validate the quality of this port.

4.1 Achievement

This porting project has finished most of the programming work, however it still can not make 'HelloWorld' run yet. The project passed the third milestone (see figure 5.1) and is still ongoing. The latest progress can be tracked on the project web site[1].

From the figure 4.2, all the PowerPC instructions, which have been used by the baseline compiler subsystem, have been ported to ARM instructions. 202 *emit* methods have been replaced the ARM implementation and all of them passed the test, which will be described in section 4.3.1.

See Table A.1, 91.55% of bytecode *emit* methods have passed the test. The debugging is still ongoing, to complete the remaining 12 bytecode tests.

4.2 Limitations of implementation

The objective of this project is to implement a port of Jikes RVM to the ARM. A prototype system has been developed, but there are a lot of potential improvements which could be made.

[1]In the project, we use an online discussion group - http://groups.yahoo.com/group/jikesRVM2ARM/, to support the project progress management.

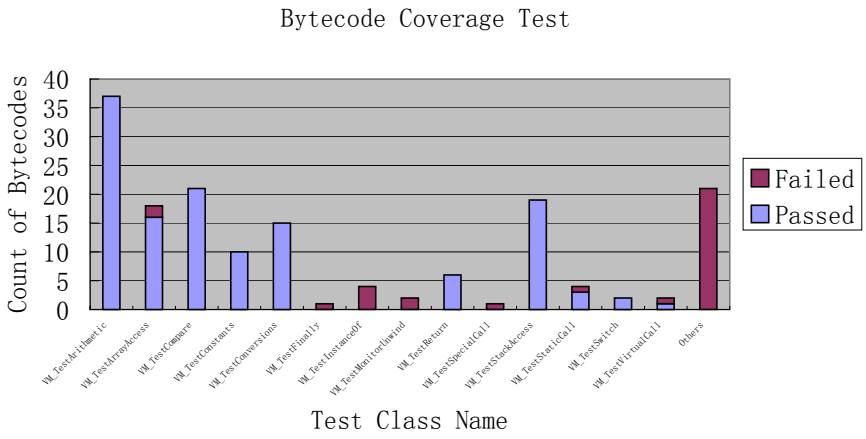

Figure 4.1: Bytecode Coverage Test A.1

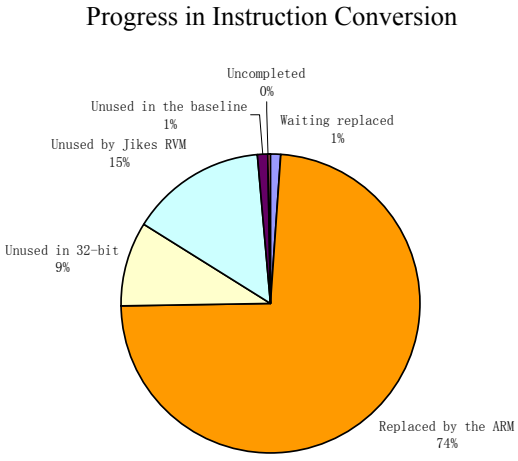

Figure 4.2: Process in Instruction Conversion A.3

Firstly, not all the system has been tested. Only some functional unit tests have been completed. The aim of this project was to make the 'HelloWorld' program run with minimum modification. It is very hard to find out the bugs without complete module and system tests.

Secondly performance has not been addressed. Both the VM_Assembler and VM_Compiler porting reused the logic of the PowerPC's implementation focusing on the correctness of the logic. A source code review may be helpful to modify the design to simplify the logic to fit the ARM architecture.

Thirdly, the assembler only generates ARM code. A complete solution for the ARM architecture should also have an assembler for Thumb code.

Fourthly, the baseline subsystem has no implementation of the VM_Disassembler, which is used to disassemble the binary machine code to assembly code.

In section 6.2, the above limitations will be discussed further.

4.3 Validation and testing

Testing is critical to discover the bugs in this project. As mentioned, in Chapter 3, debugging is very hard on the Jikes RVM. Discovering the bugs in the early stages of the porting would save a lot effort compared to finding them out among tens of thousands of machine code instructions. There are two important classes in the baseline compiler subsystem: VM_Assembler and VM_Compiler, which are the key test targets in this project.

4.3.1 Testing the VM_Assembler

The function of VM_Assembler is to produce binary machine code from the assembly code. In order to verify the correctness of the binary machine code the gcc assembler on the ARM/Linux was used as the benchmark.

The PowerPC's VM_Assembler can disassemble the binary machine code by using a disassembler. This means that it is necessary to implement a disassembler to make it work. Therefore, a disassembler is essential for this purpose, while it would be much more complex to port the disassembler at the same time. First, it is necessary to implement a complex and large dissembler class, which will increase the work load significantly. Secondly, if there are any mistakes found in the binary machine code generation, there will be at least two possible points where some bugs could be hidden. Either the assembler produced the wrong binary machine

code, or the disassembler generated the wrong assembly code based on the correct binary machine code. Sometime there are some bugs in both the assembler and disassembler. This would increase the difficulty of identifying the location of the bug.

Currently the solution is that the system implements some simple methods in the VM_Assembler to print out the assembly code for each emitXXX() method. The binary machine code and assembly code are generated in the same short method, which makes it easier to keep consistency between binary machine code and assembly code compared with the disassembler solution.

The Further work section will discuss the implementation of the disassembler further.

4.3.2 Testing the VM_Compiler

Testing the VM_Compiler is more complex than the VM_Assembler. It is hard to find a suitable test tool to verify the assembly code generated by VM_Compiler for each bytecode. The project used a run time execution result to test the VM_Compiler. The basic idea is to make VM_Compiler and VM_Assembler compile the test class to machine code, and then put the machine code in the boot image file. When the boot image is executed, the result is checked to see whether it is the same as predicted. This project reuses the bytecode test source codes in the Jikes RVM source code package.

In order to perform the test, first of all, it is necessary to make the VM.sysWrite() work. VM.sysWrite() is a magic call, which invokes the print() in the C library on ARM/ Linux. Without a fully working-well VM_Assembler, it is necessary to make the stack and call invoking mechanism work.

Secondly, there is a need to change the names of the test codes. They must be prefixed with 'VM_', so that, the boot image writer knows that the machine code of this class should be written in the boot image file.

Finally, after the bytecode test for the primitive data type is passed, it can do some object relative bytecode test. The test code should be executed after the memory management subsystem is initiated.

4.4 Summary

- All the *emit* methods in the VM_Assembler of PowerPC have been analysed and sorted. All the assembly code of PowerPC, which are necessary for the baseline compiler subsystem, has been replaced by the same function of the ARM's assembly code. The VM_Assembler has been successfully ported.
- 130 bytecodes passed the test, however 12 bytecodes can not yet pass the test. Most of VM_Compiler has been ported.

Chapter 5

Engineering Experience

This porting project is largely an engineering project rather than a pure research one. Most of the time and energy have been spent on hunting bugs. The most significant engineering challenge on this project is how to avoid generating bugs by minimising the changes. To speed up the progress of this project, we tried to make some improvement in the following three aspects:

- Exception Driven Porting
- Effective Debugging
- Cross Compiling

5.1 Milestones of this porting project

To make this project easy to be managed, six milestones have been set.

1. Build up the development environment.

   There are some preparations, which should be done before the formal porting starts.

   (a) Create a series of environment values and directories for the ARM.

   (b) Copy the machine dependent code from the PowerPC to the ARM. The code includes the code under the src/arch/PowerPC directories and the Jikes RVM preprocessor directives on the PowerPC.

(c) Insert the exception "Please implement [the name of method]" in the beginning of each method, which has been copied from the PowerPC implementation in the previous step.

(d) Make the cross compiling work, including cross compiling GNU CLASSPATH and Jikes RVM.

2. Make the jbuild successfully generate the boot image.

   The target of this stage is to generate a bootable boot image file. At this stage, exception driven modification (see section 5.2) is used to identify the machine code, which needs to written into the boot image file.

   The conclusion of this stage is that the boot image file has been built without any exceptions. All the machine code, written into boot image file, has been verified by GCC assembler (see section 4.3.1). In the other words, VM_Assembler has been mostly correctly ported.

3. Insert the Bytecode test into the VM.boot()

   This step verifies the correctness of the assembly code sequences generated by VM_Compiler (see section 4.3.2). To make the test work, there are several preparations need to be done.

   (a) Make the bootstrap program work.

   Some assembly code, on the src/tools/bootImageRunner/bootThread.C, need to be modified.

   (b) Make the VM_sysWrite work.

   In this project, most of the tests are dependent on the output. It is necessary to make the Jikes RVM print out something which is necessary for the further porting.

   This simple method uses the Magic call mechanism. It invokes the print() in the C++ library. This means that it is necessary to make the whole method Prologue/Epilogue and invocation of the C++ library method mechanism work.

   Then the bytecode test can be run in the VM.boot(). This uses the bytecode test code to test whether correct machine code has been generated at run-time. This tests the logic of the VM_Compiler. At the end of this

milestone, all the bytecode test programs should be passed without any errors.

4. Make the Jikes RVM launch correctly.

   The objective of this milestone is to execute the boot image without any errors. In this stage, the whole Java Virtual Machine runtime system should be loaded into the memory.

5. Run 'HellowWorld'.

   To run 'HelloWorld' in the Jikes RVM successfully, all the subsystems must work correctly. It is necessary to run most of the bytecode for 'HelloWorld'.

6. Run Dhrystone [Oka02] and Spec JVM'98 [SPE98].

   Dhrystone and Spec JVM'98 are going to be used as the benchmark to test the performance of Jikes RVM.

5.2 Exception Driven Modification

The porting of the VM_Assembler is driven by the exceptions. The process is a loop.

1. Run the build image program (jbuild).

2. If an unimplemented exception is thrown.

   This exception will identify which machine dependent method is been invoked by the build image program.

3. Check the PowerPC and ARM instruction manual, and then replace the method which emits the PowerPC machine code with the one which can generate ARM machine code. Sometimes the name of the emit method must be changed to follow the naming convention of the ARM. In this case, VM_Compiler should be changed to use the new method name.

4. Run the build image program again, and use GCC to verify whether the machine code has been correctly planted.

5. After this machine code has been corrected ported, go to the first step, to port another machine code.

5.3 Debugging the Jikes RVM

As a Java Virtual Machine written in Java, Jikes RVM is started by a C program that launch the boot image, created by running the Java compiler programs to compile themselves. Currently the boot image does not include a symbol table, which can be used by some debug tools, such as gdb. Use of gdb to debug the boot image file of Jikes RVM just shows a list of assembly machine code, without any information about which Java methods and bytecodes have generated the machine code sequence.

To debug the Jikes RVM with gdb, it is necessary to refer to several information sources to identify the Jikes RVM run-time status. The most useful information is in the file named RVM.map. It is a map file for the JTOC (Jikes RVM Table of Contents) of Jikes RVM. The whole Jikes RVM method call mechanism is based on the JTOC table. The JTOC table is big table; it stores all the pointers to the static methods and fields. There are some skills which proved useful during this project.

Firstly how to find the start of a method in a sequence of machine code. A NOP instruction is plugged in between each method, to make it easy to identify the start of the method.

Secondly, for an address in the Jikes RVM, how to identify to which method this address belongs. A python script was obtained from Mr. Chris Hoffmann [Hof04]. The logic of this script is to find out a method name on the RVM.map with the closet start address to a particular address.

Thirdly, we need to find out which line of Java code is related to a particular address. There is not a straight forward way, but there are two useful clues.

- One is to find a sequence of codes which get a static value from the JTOC table. The JTOC not only stores the offset of the start of the methods; but also the offset of the static values. In the ARM port of Jikes RVM, the base of JTOC is stored in register 9. A location in the program can be identified by finding the closet instruction which loads a value using register 9 as the base register, and then identifying which value it wants to load from JTOC by the offset. After knowing which constant the machine code is going to access, one can scan the Java source code of the method, identified by the second skill. In most case, this decreases the scope of the Java source to further identify from which line of Java code the machine

code was compiled.

- The other helpful clue is the log file generated by the VM_Assembler and VM_Compiler. In this log file, there are details of each bytecode and the machine code emitted for it. At this stage, it is necessary to carefully match the nearby machine code to identify from bytecode the particular machine code came. Then the bytecode can be related to the Java source code.

5.4 Cross Compiling

Cross compiling [1] is a technology used in this porting project. Using exception driven modification, whenever a piece of code has been modified, it is necessary to rebuild and link the boot image file. However the performance of Xscale/Linux is very poor, it takes more than eight hours to build and link. To work more effectively, we use cross-compiling: to build the boot image on a high performance machine (Intel IA32 with 2 Xeon processors) and then link the image file on Xscale/Linux. Because the whole boot image writing subsystem is implemented in Java, it is platform independent. As a result, it takes only one hour to finish the rebuilding of Jikes RVM. There are three special technical issues, which would be invoked in cross compiling.

1. Cross compile GNU CLASSPATH

   First of all is to cross compile the GNU CLASSPATH. The GNU CLASSPATH library is used both to build the boot image and by the run-time Jikes RVM. Most of the GNU CLASSPATH is written in Java; the rest is C++ code. Because there are some bugs in the Jikes compiler on Xscale/Linux, it failed to compile Java code in the GNU CLASSPATH. We compiled the Java code on the IA32 machine, and just compiled the C++ code locally. On the local Xscale/Linux, we use the -enable-Java=no, to skip the compiling of Java code. Then we copy the class file from the GNU CLASSPATH, which is created on the IA32/Linux, to the GNU CLASSPATH for the Xscale/Linux directory.

[1] Cross compiling is the procedure for building a program for a platform different from the one on which the cross compiler runs. "Platform" does not only mean the hardware architecture but also software platforms, e.g. the process for building the GNU/Hurd operating system from sources on a running Linux for the same hardware architecture is also a cross compiling[ND03].

2. Cross generate the RVM.image

   The programs to generate the boot image are written in Java. This program is platform independent. We use a high performance PC server to generate the boot image.

3. Local linking the Jikes RVM.

   To link the boot image and boot strap code, it is necessary to use GCC on the local Xscale/Linux machine.

4. Ignore recompiling GNU CLASSPATH

   To further speed up the rebuilding process, the shell scripts have been changed to skip recompiling the GNU CLASSPATH. As an result, it save more than 60 minutes to rebuild the Jikes RVM each time.

5.5 Summary

This chapter has introduced the technical skills applied on this porting project. It is difficult to debug the Jikes RVM in such a porting project. It would be helpful to make the Jikes RVM generate a symbol table, which could be used by some debugging tools, such as gdb in this kind of porting project.

Chapter 6

Conclusions and future work

6.1 Conclusions

This thesis has described the design and implementation of a port of the Jikes RVM to the ARM architecture. This port starts from the PowerPC implementation. The PowerPC and ARM processors share some RISC architecture features, differences between them have been considered in this project.

According to time and resource limitation, this project applied the methodology of 'change, only when necessary' and used testing and debugging to discover the problems and find out the solutions. However, without effective tools, the project progress was relatively slow.

By now, most of the programming work has been done. From the bytecode coverage test (see table A.1), only 8.45% of bytecodes cannot pass the tests. Some debugging work still needs to be done.

6.2 Future work

This project managed to implement a minimum implementation of the Jikes RVM for the ARM processor. There are two areas of further work which are necessary.

- Further improve the ARM port
- Research on the Jikes RVM with the ARM series processors.

6.2.1 Engineering

Robust Jikes RVM

As a simple port of the Jikes RVM, we have ignored the Garbage Collector and Performance Management Unit Support mechanism.

Firstly, the effort can be put into implementing a complete Jikes RVM.

Secondly, some optimisation on the Magic calls can be considered. To save time, in this porting project, we overused the Magic calls to implement some complex byte codes. The Magic calls invoke the C library. If this can be replaced by a sequence of assembly code, it would improve the performance.

Thirdly, the other potential point to improve the performance is to examine the pipeline and cache issues to optimise the assembly machine code emitted for each byte code.

Baseline compiler for Thumb code

On the ARM processor, there are two kinds of instruction sets: the ARM code and the Thumb code. The ARM code is 32-bit instructions with relative higher performance; Thumb code is 16-bit high-density code, which may be viewed as a compressed form of a subset of the ARM instruction set. Normally thumb code requires 70% of the space of ARM code. Thumb code has higher code density, and can save energy on the fetch operations. In an energy critical situation, a mix of ARM and Thumb code could be one of the best solutions: to use the ARM code for performance critical methods and Thumb code for the other codes.

Optimise compiler design

The optimise compiler subsystem is the core of the Jikes RVM. Most of the research on the Jikes RVM is using the optimising compiler system. To port the existing parallel optimisation algorithm to the ARM port would be a very interesting topic for the JAMAICA research group.

6.2.2 Research Future Work

Dynamically generate the Thumb code and ARM code

As mentioned, in the background section 2.4.1, the ARM processor is famous for its low power oriented design. One of the most important techniques for power

saving is the Thumb code. On the C compiler, programmers would decide which part of source code should be compiled into the ARM code or the Thumb code. The best practice is to compile the performance critical program into the ARM code and the rest to the Thumb code. Normally it requires a lot of performance test and analysis to trade off between them.

On the virtual machine, the real-time profile information makes it possible to dynamically compile the source to ARM code or Thumb code. For large systems written in Java it should be possible to use a dynamic measure of performance to choose the appropriate machine code. In this way, it can give us more opportunity to gain the best trade off between the performance and power consumption at the system level. It is very important for some power critical systems, such as mobile equipments.

Dynamically switch off the spare processors or parallel execute

As with other series of microprocessors, the ARM processor family has developed the chip-multi-processor – MPCore, as discussed in section 2.4.1. The multi-chip processor should improve the performance by parallelling computing, but would consume more energy than a single core processor. Currently many chip-multi-processor parallel computing algorithms are based on a speculative prediction mechanism. If the prediction is correct, it uses parallel computing to improve the performance, otherwise it will achieve similar performance to a single core processor. A failed prediction will cost little or nothing from some view points. But in a power critical situation, the failure will waste energy. Currently the ARM chip multiprocessor supports a turn-off function, which means that it can choose the number of active processors at run time. In some cases, it is worth using parallel computing to improve the performance, when the ratio of successful prediction is high enough; otherwise it is more sensible to switch off some processors and just use one processor to save power. This requires the run-time system to be smart enough. If the trade off can be considered based on the run-time execution profile, it would give the system more chance to gain the optimum trade-off.

Appendix A

Appendix results

A.1 Table of the Bytecode Coverage Test

Test Class	*Number*	*Passed*	*Failed*
VM_TestArithmetic	37	37	0
VM_TestArrayAccess	18	16	2
VM_TestCompare	21	21	0
VM_TestConstants	10	10	0
VM_TestConversions	15	15	0
VM_TestFinally	1	0	1
VM_TestInstanceOf	4	0	4
VM_TestMonitorUnwind	2	0	2
VM_TestReturn	6	6	0
VM_TestSpecialCall	1	0	1
VM_TestStackAccess	19	19	0
VM_TestStaticCall	4	3	1
VM_TestSwitch	2	2	0
VM_TestVirtualCall	2	1	1
SUM	142	130	12
Percentage		91.55%	8.45%

Table A.1: Bytecode Coverage Test

Table A.2: Bytecode Coverage Test Detail

Bytecode	*Test Class Name*	*Status*
aaload	VM_TestArrayAccess	Passed
aastore	VM_TestArrayAccess	Passed
aconst_null	VM_TestConstants	Passed
aload	VM_TestStackAccess	Passed
areturn	VM_TestReturn	Passed
arraylength		
astore	VM_TestStackAccess	Passed
athrow		
baload	VM_TestArrayAccess	Passed
bastore	VM_TestArrayAccess	Passed
caload	VM_TestArrayAccess	Passed
castore	VM_TestArrayAccess	Passed
checkcast		
checkcast_final		
checkcast_resolvedClass		
d2f	VM_TestConversions	Passed
d2i	VM_TestConversions	Passed
d2l	VM_TestConversions	Passed
dadd	VM_TestArithmetic	Passed
daload	VM_TestArrayAccess	Passed
dastore	VM_TestArrayAccess	Passed
dcmpg	VM_TestCompare	Passed
dcmpl	VM_TestCompare	Passed
dconst_0	VM_TestConstants	Passed
dconst_1	VM_TestConstants	Passed
ddiv	VM_TestArithmetic	Passed
deferred_prologue		
dload	VM_TestStackAccess	Passed
dmul	VM_TestArithmetic	Passed
dneg	VM_TestArithmetic	Passed

Continued on next page

Bytecode	*Test Class Name*	*Status*
drem	VM_TestArithmetic	Passed
dreturn	VM_TestReturn	Passed
dstore	VM_TestStackAccess	Passed
dsub	VM_TestArithmetic	Passed
dup	VM_TestStackAccess	Passed
dup_x1	VM_TestStackAccess	Passed
dup_x2	VM_TestStackAccess	Passed
dup2	VM_TestStackAccess	Passed
dup2_x1	VM_TestStackAccess	Passed
dup2_x2	VM_TestStackAccess	Passed
f2d	VM_TestConversions	Passed
f2i	VM_TestConversions	Passed
f2l	VM_TestConversions	Passed
fadd	VM_TestArithmetic	Passed
faload	VM_TestArrayAccess	Passed
fastore	VM_TestArrayAccess	Passed
fcmpg	VM_TestCompare	Passed
fcmpl	VM_TestCompare	Passed
fconst_0	VM_TestConstants	Passed
fconst_1	VM_TestConstants	Passed
fconst_2	VM_TestConstants	Passed
fdiv	VM_TestArithmetic	Passed
fload	VM_TestStackAccess	Passed
fmul	VM_TestArithmetic	Passed
fneg	VM_TestArithmetic	Passed
frem	VM_TestArithmetic	Passed
freturn	VM_TestReturn	Passed
fstore	VM_TestStackAccess	Passed
fsub	VM_TestArithmetic	Passed
goto		Failed
i2b	VM_TestConversions	Passed
i2c	VM_TestConversions	Passed

Continued on next page

Bytecode	*Test Class Name*	*Status*
i2d	VM_TestConversions	Passed
i2f	VM_TestConversions	Passed
i2l	VM_TestConversions	Passed
i2s	VM_TestConversions	Passed
iadd	VM_TestArithmetic	Passed
iaload	VM_TestArrayAccess	Passed
iand	VM_TestArithmetic	Passed
iastore	VM_TestArrayAccess	Passed
iconst	VM_TestConstants	Passed
idiv	VM_TestArithmetic	Passed
if_acmpeq	VM_TestCompare	Passed
if_acmpne	VM_TestCompare	Passed
if_icmpeq	VM_TestCompare	Passed
if_icmpge	VM_TestCompare	Passed
if_icmpgt	VM_TestCompare	Passed
if_icmple	VM_TestCompare	Passed
if_icmplt	VM_TestCompare	Passed
if_icmpne	VM_TestCompare	Passed
ifeq	VM_TestCompare	Passed
ifge	VM_TestCompare	Passed
ifgt	VM_TestCompare	Passed
ifle	VM_TestCompare	Passed
iflt	VM_TestCompare	Passed
ifne	VM_TestCompare	Passed
ifnonnull	VM_TestCompare	Passed
ifnull	VM_TestCompare	Passed
iinc	VM_TestArithmetic	Passed
iload	VM_TestStackAccess	Passed
imul	VM_TestArithmetic	Passed
ineg	VM_TestArithmetic	Passed
instanceof	VM_TestInstanceOf	Failed
instanceof_final	VM_TestInstanceOf	Failed

Continued on next page

Bytecode	*Test Class Name*	*Status*
instanceof_resolvedClass	VM_TestInstanceOf	Failed
invoke_compiledmethod		
invokeinterface		
ior	VM_TestArithmetic	Passed
irem	VM_TestArithmetic	Passed
ireturn	VM_TestReturn	Passed
ishl	VM_TestArithmetic	Passed
ishr	VM_TestArithmetic	Passed
istore	VM_TestStackAccess	Passed
isub	VM_TestArithmetic	Passed
iushr	VM_TestArithmetic	Passed
ixor	VM_TestArithmetic	Passed
jsr	VM_TestFinally	Failed
l2d	VM_TestConversions	Passed
l2f	VM_TestConversions	Passed
l2i	VM_TestConversions	Passed
ladd	VM_TestArithmetic	Passed
laload	VM_TestArrayAccess	Passed
land	VM_TestArithmetic	Passed
lastore	VM_TestArrayAccess	Passed
lcmp	VM_TestCompare	Passed
lconst	VM_TestConstants	Passed
ldc	VM_TestConstants	Passed
ldc2	VM_TestConstants	Passed
ldiv	VM_TestArithmetic	Passed
lload	VM_TestStackAccess	Passed
lmul	VM_TestArithmetic	Passed
lneg	VM_TestArithmetic	Passed
loadaddrconst		
lookupswitch	VM_TestSwitch	Passed
lor	VM_TestArithmetic	Passed
lrem	VM_TestArithmetic	Passed

Continued on next page

Bytecode	*Test Class Name*	*Status*
lreturn	VM_TestReturn	Passed
lshl	VM_TestArithmetic	Passed
lshr	VM_TestArithmetic	Passed
lstore	VM_TestStackAccess	Passed
lsub	VM_TestArithmetic	Passed
lushr	VM_TestArithmetic	Passed
lxor	VM_TestArithmetic	Passed
Magic		
monitorenter	VM_TestMonitorUnwind	Failed
monitorexit	VM_TestMonitorUnwind	Failed
multianewarray	VM_TestArrayAccess	Failed
pop	VM_TestStackAccess	Passed
pop2	VM_TestStackAccess	Passed
prologue		
resolved_getfield	VM_TestFieldAccess	Passed
resolved_getstatic	VM_TestStaticCall	
resolved_invokespecial	VM_TestSpecialCall	Failed
resolved_invokestatic	VM_TestStaticCall	Passed
resolved_invokevirtual	VM_TestVirtualCall	
resolved_new		
resolved_newarray	VM_TestArrayAccess	Failed
resolved_putfield		
resolved_putstatic		
ret		
return	VM_TestReturn	Passed
saload	VM_TestArrayAccess	Passed
sastore	VM_TestArrayAccess	Passed
swap	VM_TestStackAccess	Passed
tableswitch	VM_TestSwitch	Passed
threadSwitch		
threadSwitchTest		
unresolved_getfield	VM_TestFieldAccess	Passed

Continued on next page

Bytecode	*Test Class Name*	*Status*
unresolved_getstatic	VM_TestStaticCall	Passed
unresolved_invokestatic	VM_TestStaticCall	Passed
unresolved_invokevirtual	VM_TestVirtualCall	Passed
unresolved_new		
unresolved_newarray		
unresolved_putfield	VM_TestFieldAccess	Passed
unresolved_putstatic		
Others	VM_TestFloatingRem	Failed
Others	VM_TestInterfaceCall	Passed
Others	VM_TestClone	Failed
Others	VM_TestInstanceOf	Failed
Others	VM_TestGC	Failed
Others	VM_TestClassInitializer	Failed

A.2 Table of the Process in Instruction Coversion

Table A.3: Progress in Instruction Conversion

The PowerPC Instructions	*Status*	*Comment*
_BC	Replaced by the ARM	
_BL	Replaced by the ARM	
ADC	Replaced by the ARM	
ADD	Replaced by the ARM	
ADD	Replaced by the ARM	
ADDCrLSL	Replaced by the ARM	
ADDI	Replaced by the ARM	
ADDI	Replaced by the ARM	
ADDICr	Replaced by the ARM	
ADDIS	Replaced by the ARM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
ADDIS12	Replaced by the ARM	
ADDItoc	Replaced by the ARM	
ADDs	Replaced by the ARM	
ADDsLSL	Replaced by the ARM	
ADFD	Replaced by the ARM	
ADFS	Replaced by the ARM	
AND	Replaced by the ARM	
ANDI	Replaced by the ARM	
B	Replaced by the ARM	
B	Replaced by the ARM	
BC	Replaced by the ARM	
BC	Replaced by the ARM	
BCCTR	Replaced by the ARM	
BCCTRL	Replaced by the ARM	
BCLR	Replaced by the ARM	
BCLRL	Replaced by the ARM	
BICLSL	Replaced by the ARM	
BLA	Unused by Jikes RVM	
BLA	Replaced by the ARM	
CMF	Replaced by the ARM	
CMP	Unused by Jikes RVM	
CMP	Replaced by the ARM	
CMP	Replaced by the ARM	
CMPAddrI	Replaced by the ARM	
CMPD	Unused by Jikes RVM	
CMPD	Replaced by the ARM	
CMPDI	Replaced by the ARM	
CMPI	Unused by Jikes RVM	3 arguments [1]
CMPI	Replaced by the ARM	
CMPI	Replaced by the ARM	

Continued on next page

[1] It is a three arguments instruction. On the ARM, the CMPI just have two arguments.

The PowerPC Instructions	*Status*	*Comment*
CMPL	Unused by Jikes RVM	3 arguments [2]
CMPLD	Unused in 32-bit	It is a 64 bit instruction.
CMPLD	Replaced by the ARM	
CRAND	Unused by Jikes RVM	
CRAND	Replaced by the ARM	
CRANDC	Unused by Jikes RVM	
CRANDC	Replaced by the ARM	
CROR	Unused by Jikes RVM	
CROR	Replaced by the ARM	
CRORC	Unused by Jikes RVM	
CRORC	Replaced by the ARM	
DCBF	Unused by Jikes RVM	
DCBF	Replaced by the ARM	
DCBST	Replaced by the ARM	
DIVD	Unused in 32-bit	It is a 64 bit-instruction.
DIVD	Replaced by the ARM	
DIVW	Replaced by the ARM	
DVFD	Replaced by the ARM	
DVFS	Replaced by the ARM	
EXTSB	Replaced by the ARM	
EXTSW	Unused by Jikes RVM	
EXTSW	Replaced by the ARM	
FABS	Unused by Jikes RVM	
FABS	Replaced by the ARM	
FCFID	Unused in 32-bit	It is a 64 bit-instruction.
FCFID	Replaced by the ARM	
FCTIDZ	Unused by Jikes RVM	
FCTIDZ	Replaced by the ARM	
FIXZ	Replaced by the ARM	
FMADD	Unused by Jikes RVM	
FMADD	Replaced by the ARM	

Continued on next page

[2]It is a three arguments instruction. On the ARM, CMPL just have two arguments.

The PowerPC Instructions	*Status*	*Comment*
FNMSUB	Unused by Jikes RVM	
FNMSUB	Replaced by the ARM	
ForwardBC	Replaced by the ARM	
FSEL	Unused by Jikes RVM	
FSEL	Replaced by the ARM	
ICBI	Replaced by the ARM	
ISYNC	Replaced by the ARM	
LAddr	Replaced by the ARM	
LAddrToc	Replaced by the ARM	
LAddrX	Replaced by the ARM	
LBZ	Unused by Jikes RVM	
LBZ	Replaced by the ARM	
LBZX	Replaced by the ARM	
LD	Unused in 32-bit	It is a 64 bit-instruction.
LD	Replaced by the ARM	
LDARX	Unused in 32-bit	It is a 64 bit-instruction.
LDARX	Replaced by the ARM	
LDFD	Replaced by the ARM	
LDFS	Replaced by the ARM	
LDRSB	Replaced by the ARM	
LDRSBX	Replaced by the ARM	
LDRSH	Replaced by the ARM	
LDRSHX	Replaced by the ARM	
LDtoc	Unused in 32-bit	It is a 64 bit-instruction.
LDtoc	Replaced by the ARM	
LDU	Unused in 32-bit	It is a 64 bit-instruction.
LDU	Replaced by the ARM	
LDUX	Unused in 32-bit	It is a 64 bit-instruction.
LDUX	Replaced by the ARM	
LDX	Unused in 32-bit	It is a 64 bit-instruction.
LDX	Replaced by the ARM	
LFDtoc	Replaced by the ARM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
LFDU	Unused by Jikes RVM	
LFDU	Replaced by the ARM	
LFDX	Replaced by the ARM	
LFStoc	Replaced by the ARM	
LFSX	Unused by Jikes RVM	
LFSX	Unused by Jikes RVM	
LHZ	Replaced by the ARM	
LHZX	Replaced by the ARM	
LI	Replaced by the ARM	
LVAL	Replaced by the ARM	
LVALAddr	Unused by Jikes RVM	
LVALAddr	Replaced by the ARM	
LWA	Unused by Jikes RVM	
LWA	Replaced by the ARM	
LWARX	Replaced by the ARM	
LWARX	Replaced by the ARM	
LWAtoc	Unused by Jikes RVM	
LWAtoc	Replaced by the ARM	
LWAX	Unused in 32-bit	It is a 64 bit-instruction.
LWAX	Replaced by the ARM	
LWZ	Replaced by the ARM	
LWZtoc	Replaced by the ARM	
LWZU	Replaced by the ARM	
LWZUX	Replaced by the ARM	
LWZX	Replaced by the ARM	
MFLR	Replaced by the ARM	
MFTB	Unused in the baseline	Used for performance measure.
MFTB	Replaced by the ARM	
MFTBU	Unused in the baseline	Used for performance measure.
MFTBU	Replaced by the ARM	
MLA	Replaced by the ARM	
MNFS	Replaced by the ARM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
MOVX	Replaced by the ARM	
MOVX	Replaced by the ARM	
MOVXS	Replaced by the ARM	
MTIP	Replaced by the ARM	
MTLR	Replaced by the ARM	
MUFD	Replaced by the ARM	
MUFS	Replaced by the ARM	
MUL	Replaced by the ARM	
MULHDU	Unused by Jikes RVM	
MULHDU	Replaced by the ARM	
MULHWU	Unused by Jikes RVM	
MULHWU	Replaced by the ARM	
MULLD	Unused in 32-bit	It is a 64 bit-instruction.
MULLD	Replaced by the ARM	
NEG	Replaced by the ARM	
NOP	Replaced by the ARM	
OR	Replaced by the ARM	
ORI	Replaced by the ARM	
ORIS	Replaced by the ARM	
patchConditionalBranch	Replaced by the ARM	
patchConditionalBranch	Replaced by the ARM	
patchLoadAddrConst	Unused by Jikes RVM	
patchShortBranch	Replaced by the ARM	
patchUnconditionalBranch	Replaced by the ARM	
ResetReg	Replaced by the ARM	
RLDINM	Unused by Jikes RVM	
RLDINM	Replaced by the ARM	
RLWINM	Unused by Jikes RVM	
RLWINM	Replaced by the ARM	
ShortBC	Replaced by the ARM	
SLAddr	Replaced by the ARM	
SLAddrI	Unused by Jikes RVM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
SLD	Unused by Jikes RVM	
SLD	Replaced by the ARM	
SLDI	Unused by Jikes RVM	
SLDI	Replaced by the ARM	
SLW	Replaced by the ARM	
SLWI	Replaced by the ARM	
SRA_Addr	Replaced by the ARM	
SRA_AddrI	Unused by Jikes RVM	
SRAD	Unused in 32-bit	It is a 64 bit-instruction.
SRAD	Replaced by the ARM	
SRADI	Unused in 32-bit	It is a 64 bit-instruction.
SRADI	Replaced by the ARM	
SRADIr	Unused by Jikes RVM	
SRADIr	Replaced by the ARM	
SRAW	Replaced by the ARM	
SRAWI	Replaced by the ARM	
SRAWIr	Unused by Jikes RVM	
SRAWIr	Replaced by the ARM	
SRD	Unused in 32-bit	It is a 64 bit-instruction.
SRD	Replaced by the ARM	
SRW	Replaced by the ARM	
SRWI	Replaced by the ARM	
STB	Unused by Jikes RVM	
STB	Replaced by the ARM	
STBX	Replaced by the ARM	
STD	Unused in 32-bit	It is a 64 bit-instruction.
STD	Replaced by the ARM	
STDCXr	Unused in 32-bit	It is a 64 bit-instruction.
STDCXr	Replaced by the ARM	
STDtoc	Replaced by the ARM	
STDU	Unused in 32-bit	It is a 64 bit-instruction.
STDU	Replaced by the ARM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
STDUX	Unused in 32-bit	It is a 64 bit-instruction.
STDUX	Replaced by the ARM	
STDX	Unused in 32-bit	It is a 64 bit-instruction.
STDX	Replaced by the ARM	
STFD	Replaced by the ARM	
STFDtoc	Replaced by the ARM	
STFDU	Unused by Jikes RVM	
STFDU	Replaced by the ARM	
STFDX	Replaced by the ARM	
STFS	Replaced by the ARM	
STFStoc	Unused by Jikes RVM	
STFStoc	Replaced by the ARM	
STFSU	Unused by Jikes RVM	
STFSU	Replaced by the ARM	
STHX	Replaced by the ARM	
STW	Replaced by the ARM	
STWCXr	Replaced by the ARM	
STWCXr	Replaced by the ARM	
STWtoc	Replaced by the ARM	
STWU	Replaced by the ARM	
STWUX	Unused by Jikes RVM	
STWUX	Replaced by the ARM	
STWX	Replaced by the ARM	
SUB	Replaced by the ARM	
SUB	Replaced by the ARM	
SUBFC	Replaced by the ARM	
SUBFCr	Replaced by the ARM	
SUBFE	Replaced by the ARM	
SUBFEr	Replaced by the ARM	
SUBFIC	Replaced by the ARM	
SUBFZE	Replaced by the ARM	
SUBI	Replaced by the ARM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
SUBICr	Replaced by the ARM	
SUFD	Replaced by the ARM	
SUFS	Replaced by the ARM	
SWP	Replaced by the ARM	
SYNC	Replaced by the ARM	
TAddrEQ0	Replaced by the ARM	
TAddrI	Replaced by the ARM	
TAddrLE	Replaced by the ARM	
TAddrLLE	Replaced by the ARM	
TAddrLT	Replaced by the ARM	
TAddrWI	Replaced by the ARM	
TDEQ0	Unused in 32-bit	It is a 64 bit-instruction.
TDEQ0	Replaced by the ARM	
TDI	Unused in 32-bit	It is a 64 bit-instruction.
TDI	Replaced by the ARM	
TDLE	Unused in 32-bit	It is a 64 bit-instruction.
TDLE	Replaced by the ARM	
TDLLE	Unused in 32-bit	It is a 64 bit-instruction.
TDLLE	Replaced by the ARM	
TDLT	Unused in 32-bit	It is a 64 bit-instruction.
TDLT	Replaced by the ARM	
TDWI	Unused in 32-bit	It is a 64 bit-instruction.
TDWI	Replaced by the ARM	
Trap	Replaced by the ARM	
TWEQ0	Replaced by the ARM	
TWI	Replaced by the ARM	
TWI	Replaced by the ARM	
TWLE	Replaced by the ARM	
TWLLE	Replaced by the ARM	
TWLT	Replaced by the ARM	
TWNE	Replaced by the ARM	
UMLAL	Replaced by the ARM	

Continued on next page

The PowerPC Instructions	*Status*	*Comment*
UMULL	Replaced by the ARM	
XOR	Replaced by the ARM	
XORI	Unused by Jikes RVM	
XORI	Replaced by the ARM	

Bibliography

[ARM00] ARM Limited. *ARM Architecture Reference Manual*, June 2000.

[Ear03] Richard Earnshaw. *Procedure Call Standard for the ARM Architecture*. ARM Limited, October 2003.

[Fur00] Stephen B Furber. *ARM system-on-chip architecture*. Addison-Wesley, Harlow, England, 2000.

[Hof04] Chris Hoffmann. Personal communication from the jikesrvm-researchers mailing list. 2004.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[IBM04] IBM Limited. *The Jikes Research Virtual Machine User's Guide*, April 2004.

[Int04] Intel Limited. *Intel XScale Core Developer's Manual*, Jan 2004.

[Mot01] Motorola Limited. *Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture*, 2001.

[ND03] Ross Moore Nikos Drakos. Adsl software router with firewalling and virtual private networking on embedded devices with linux on the example of a sega dreamcast gaming console. Technical report, Computer Based Learning Unit, University of Leeds; Mathematics Department, Macquarie University, Sydney, 2003.

[Oka02] Tetsuya Okado. Dhrystone benchmark in Java. `http://www.c-creators.co.jp/okayan/DhrystoneApplet/`, May 2002.

[ppc96] *The PowerPC Compiler Writer's Guide*. Warthman Associates, 1996.

[SPE98] SpecJVM benchmark. `http://www.spec.org/osg/jvm98/`, 1998.

[Ven97] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, December 1997.

[Ven98] Bill Venners. The HotSpot virtual machine. *Developer.com*, 1998.