University of Manchester – School of Computer Science

Emulating the ARM Architecture
Using a Java Dynamic Binary Translator

A dissertation submitted to the University of Manchester

for the degree of Master of Science

in the Faculty of Engineering and Physical Sciences

Author: *Michael Baer*

Submitted in: *2007*

Contents

LIST OF ABBREVIATIONS **4**

LIST OF FIGURES **5**

ABSTRACT **7**

DECLARATION **8**

COPYRIGHT STATEMENT **9**

1 INTRODUCTION **10**

1.1 CLIENT 10
1.2 SUBJECT 10
1.3 MISSION 10

2 BACKGROUND **12**

2.1 EMULATION AND DYNAMIC BINARY TRANSLATION 12
2.1.1 Interpretation 13
2.1.2 Binary Translation 14
2.1.3 Optimisations in Binary Translation 16
2.2 THE JIKES RESEARCH VIRTUAL MACHINE 19
2.2.1 The Java Virtual Machine Architecture 19
2.2.2 The Jikes Research Virtual Machine 21
2.3 PROCESS VIRTUAL MACHINES 26
2.4 THE ARM ARCHITECTURE 28
2.4.1 Programmer’s Model 32
2.4.2 IO and Memory Model 37
2.4.3 Architecture Versions 38

3 PEARCOLATOR ARCHITECTURE **40**

3.1 INTEGRATION INTO THE JIKES RESEARCH VIRTUAL MACHINE 40
3.2 PEARCOLATOR CLASS ARCHITECTURE 42
3.3 INITIALIZATION 43
3.4 LOADER 44
3.5 EXECUTION CONTROLLER 49
3.6 DECODER 51
3.7 OS EMULATION 55
3.8 PROFILING 58
3.9 MEMORY 59

3.10 FAULTS 60
3.11 PROCESS SPACE 61

4 THE ARM EMULATOR 62

4.1 GENERAL ARCHITECTURE 62
4.2 THE ARM- AND THUMB-DECODER....64
4.3 INSTRUCTION REPRESENTATION 67
4.4 ARM TRANSLATOR 68
4.4.1 Scheme Selection 68
4.4.2 Implementation 70
4.4.3 Conditional Instructions 71
4.4.4 Condition code handling 72
4.5 THE ARM INTERPRETER 74
4.6 THE ARM DISASSEMBLER 75
4.7 OPERATING SYSTEM EMULATION 76
4.7.1 Linux Operating System Support....76
4.7.2 Angel Debug Monitor Support 77

5 EVALUATION OF THE ARM BACKEND 78

5.1 FUNCTIONALITY EVALUATION 78
5.2 PERFORMANCE EVALUATION 78
5.2.1 Execution Controllers....79
5.2.2 Condition Code Evaluation 81
5.2.3 Memory Model 82
5.2.4 Inlining Options 83
5.2.5 Profiling and Indirect Jump Prediction 84
5.2.6 Overall Emulator Performance 86

6 CONCLUSION....87

6.1 CONCLUSION....87
6.2 FUTURE WORK....88

7 REFERENCES....89

8 APPENDIX 94

8.1 APPENDIX A....94
8.2 APPENDIX B 95
8.3 APPENDIX C 96

Word Count: 24875

List of Abbreviations

Abbreviation	**Long Form**
ABI	Application Binary Interface
ARFF	Attribute Relation File Format
APCS	ARM Procedure Call Standard
BURS	Bottom Up Rewriting System
CISC	Complex Instruction Set Computer
CMP	Chip Multiprocessor
CPI	Clocks per Instruction
CPSR	Current Program Status Register
EABI	Embedded Application Binary Interface
ELF	Executable and Linkable Format
GC	Garbage Collector
gdb	GNU Debugger
GOT	Global Offset Table
HIR	High-Level Intermediate Representation
ISA	Instruction Set Architecture
JNI	Java Native Interface
JRVM	Jikes Research Virtual Machine
JVM	Java Virtual Machine
LIR	Low-Level Intermediate Representation
MIR	Machine Intermediate Representation
MMU	Memory Management Unit
OS	Operating System
PIC	Position Independent Code
PLT	Procedure Linkage Table
RISC	Reduced Instruction Set Computer
SMP	Symmetric Multiprocessing
SPSR	Saved Program Status Register
SWI	Software Interrupt
TIB	Type Information Block
UML	Unified Modelling Language
XML	Extended Markup Language
VM	Virtual Machine

List of Figures

FIGURE 1 – OVERVIEW OF INTERPRETER COMPONENTS 13
FIGURE 2 - JAVA VIRTUAL MACHINE RUNTIME DATA AREAS 20
FIGURE 3 – JRVM OPTIMISING COMPILER OVERVIEW 24
FIGURE 4 – THE JIKES ADAPTIVE OPTIMIZATION SYSTEM....25
FIGURE 5 – COMPONENTS OF A PROCESS VIRTUAL MACHINE....27
FIGURE 6 - ARM REGISTER LAYOUT 30
FIGURE 7 – THE ARM 3-STAGE PIPELINE....32
FIGURE 8 – RELATION OF PEARCOLATOR TO THE JIKES RESEARCH VIRTUAL MACHINE....40
FIGURE 9 – PEARCOLATOR OVERVIEW AS A UML PACKAGE DIAGRAM 43
FIGURE 10 – PERSPECTIVES ON AN ELF FILE....45
FIGURE 11 – DEPENDENCIES BETWEEN ELF SHARED OBJECTS....48
FIGURE 12 – BUILDING DYNAMIC BASIC BLOCKS OF INTERPRETED INSTRUCTIONS....52
FIGURE 13 – TRANSLATION OF A SIMPLE INSTRUCTION SEQUENCE INTO A TRACE 53
FIGURE 14 – DEFAULT PEARCOLATOR FILESYSTEM CONFIGURATION....57
FIGURE 15 – ENTITY RELATIONSHIP DIAGRAM OF THE PEARCOLATOR PROFILING DATA MODEL....59
FIGURE 16 - ARCHITECTURE OF THE PEARCOLATOR ARM BACKEND 62
FIGURE 17 – COMMUNICATION BETWEEN ARM BACKEND COMPONENTS. 64
FIGURE 18 – DATA FLOW DIAGRAM OF THE DECODER CONSTRUCTION PROCESS 66
FIGURE 19 – SAMPLE INSTRUCTION DEFINITIONS 67
FIGURE 20 - CONFLUENCE BETWEEN GUEST AND HOST TRANSLATION FOR THE PEARCOLATOR BINARY TRANSLATOR....69
FIGURE 21 – INLINING AN INTERPRETER DURING TRANSLATION....69
FIGURE 22 – COMPARISON OF EXECUTION TIME FOR DIFFERENT BINARY TRANSLATION SCHEMES 70
FIGURE 23 – TRANSLATION OF A CONDITIONAL INSTRUCTIONS....72
FIGURE 24 – EXECUTION TIME OF DHRYSTONE FOR DIFFERENT EXECUTION CONTROLLERS AND DIFFERENT NUMBER OF DHRYSTONE LOOPS (ARM 32-BIT CODE) 79
FIGURE 25 – EXECUTION TIME FOR DIFFERENT NUMBER OF DHRYSTONE ITERATIONS AND VARIOUS STAGED EMULATION THRESHOLDS. 81

FIGURE 26 – RELATIVE EXECUTION SPEED OF LAZY AND IMMEDIATE EVALUATION OF CONDITION CODES FOR DIFFERENT NUMBERS OF DHRYSTONE ITERATIONS 82
FIGURE 27 – TRANSLATOR PERFORMANCE FOR DIFFERENT MEMORY MODELS 83
FIGURE 28 – INFLUENCE OF INLINING DIFFERENT TYPES OF BRANCHES INTO A PEARCOLATOR TRACE 84
FIGURE 29 – EFFECT OF PROFILING FOR DIFFERENT INLINING TECHNIQUES............................85
FIGURE 30 - EFFECT OF PROFILING FOR DIFFERENT INLINING TECHNIQUES 85
FIGURE 31 – COMPARISION OF PEARCOLATOR PERFORMANCE WITH A COMMERCIAL EMULATOR AND NATIVE EXECUTION 86
FIGURE 32 – ARM DECODER DECISION TREE..95
FIGURE 33 – THUMB DECODER DECISION TREE..96

Abstract

Binary translation enables the execution of binary code from one processor architecture on a different architecture by translating the respective machine code. Pearcolator is a dynamic binary translator written in Java, which runs on top of the Jikes Research Virtual Machine. It has been developed by the Advanced Processor Technologies group, which explores architectures for Chip Multiprocessors (CMP) as well as appropriate operating system designs and compiler technologies. Pearcolator allows running legacy applications on the group's Jamaica processor architecture.

In this thesis, Pearcolator is enhanced with a backend to run programs for the ARMv4T architecture. Furthermore, it is reengineered to support interpretation, profiling, dynamic linking and a generic software component model. The interpreter support is leveraged to implement staged emulation, i.e. the dynamic switching between interpretation and translation, within Pearcolator.

Using the new Pearcolator component model, the performance impact of different strategies for program execution, flag management, memory access, code inlining and profiling were investigated. It has been found that staged emulation yields a three times performance increase in the best case. Furthermore, it is shown that lazy flag evaluation is not always the best performing flag emulation strategy. The performance of these strategies is dependant on the instruction set. Choosing immediate flag evaluation for ARM code and lazy flag evaluation for Thumb code improves performance by up to 10%. Similarly, the right choices for the memory model and inlining of code during binary translation deliver a significant speedup. Profiling further increases the translator's speed by about 30%.

The thesis produced the first open source ARM emulator written in Java. The final emulator performance proved to be five times faster than a commercial emulator, but still several orders of magnitude slower than native execution.

Declaration

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

Copyright Statement

i. Copyright in text of this dissertation rests with the author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the author. Details may be obtained from the appropriate Graduate Office. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.

ii. The ownership of any intellectual property rights which may be described in this dissertation is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

iii. Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of Computer Science.

1 Introduction

1.1 Client

The Advanced Processor Technologies Group within the University of Manchester conducts research into the design of novel processing architectures. Its focus spans from software approaches, esp. compiler optimisations, to developing new hardware architectures.

The Jamaica Project Group is part of the Advanced Processor Technologies Group. It explores architectures for Chip Multiprocessors (CMP), appropriate operating system designs and compiler technologies that leverage parallelism within applications. The group has developed a CMP architecture that is able to run a customized version of the Jikes Research Virtual Machine.

This thesis has been written in the Jamaica Project Group under the supervision of Prof. Ian Watson and Dr. Ian Rogers.

1.2 Subject

The thesis focuses on the dynamic binary translator Pearcolator, which is a development of the Jamaica Project Group. Binary translation enables the execution of binary code from one processor architecture on a different architecture by translating the respective machine code. Pearcolator runs on top of the Jikes Research Virtual Machine, thus enabling the CMP architecture developed by the Jamaica Project Group to run legacy applications from different systems. This is an important strategic asset, as it increases the application base and therefore the acceptance of a novel architecture.

Pearcolator supports multiple source architectures through different backends. A PowerPC and an X86 backend have been developed previously [Burcham2004] [Matley2005]. However, though both backends used similar techniques, they were essentially developed as separate applications and made no efforts to unify Pearcolator into a single system.

1.3 Mission

This thesis aims to improve the architecture of the dynamic binary translator Pearcolator, to enhance it with a new backend and to investigate possible performance

gains that have not been analyzed by the previous authors. More specifically, the following tasks will be accomplished.

1. The Pearcolator architecture will be revised to unify the previous PowerPC and X86 backends. A template architecture, which promotes the reusability of components and facilitates the implementation of new backends, will be developed and implemented.
2. Pearcolator will be enhanced with support for interpreters. This feature will be driven by a staged emulation framework, which dynamically switches between translation and interpretation to improve execution performance.
3. A new backend for the ARM processor architecture will be implemented into Pearcolator. The backend shall support translation and interpretation of the ARM 32-bit and the Thumb 16-bit instruction set.
4. The new architecture and the ARM backend will be used to evaluate the performance of Pearcolator. Especially the influence of dynamic switching between translation and interpretation and the benefits of lazy evaluation of condition codes will be explored.

The thesis will first introduce the technologies that were used during its realization, continue by describing the new architecture as well as the ARM backend, which have been implemented into Pearcolator, and finally investigate the performance of the system.

2 Background

2.1 Emulation and Dynamic Binary Translation

Emulation is the *process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality* [Smith2005, p. 27]. The term was first used in 1962 in conjunction with IBM testing the compatibility of their new products with their predecessors. In 1965, IBM shipped the System/360– it contained the world's first emulator, which ran programs that were originally written for the IBM 7070 machine.

According to the above definition, emulation involves two systems[1]. Commonly, the system that is being emulated is called the emulation *source* while the system that the emulator is running on is called the emulation *target*. Some authors also use the terms *guest* to denote the source and *host* to denote the target system in conjunction with binary translation. Usually, source and target refer to two distinct computer architectures or instruction sets. However, some systems may apply the same techniques that are subsequently described in cases where source and target refer to the same entity. [Smith2005, p. 63] calls this same-ISA emulation. For instance, the dynamic binary optimizer Dynamo, developed by HP Labs, re-translates a binary for the HP PA-8000 or IA-32 processor while running on the same processor. Though not strictly an emulator, Dynamo applies optimisations known from emulation to the binary, thereby increasing execution speed by up to 20% [Bala1999, p.12].

Though various aspects of a system can be emulated, the following paragraphs focus on the emulation of conventional instruction sets[2]. Naturally, performance is one of the primary concerns during emulation. In emulation, execution performance is usually achieved by weighing off the amount of pre-processing applied to the program and

---

[1] The term *subsystem* from the above definition is omitted in the remainder of this document and only the term system is used instead.

[2] Virtual instruction sets, such as Java Bytecode, may have special properties that can be exploited to apply more advanced emulation techniques [Smith2005, p. 28].

runtime performance on the host. In that regard, one usually distinguishes interpretation and translation.

2.1.1 Interpretation

An interpreter is a computer program that analyzes and executes another computer program at runtime, without translating the interpreted program itself into machine language. Interpretation usually involves a cycle in which an instruction is retrieved from the source program, processed and executed before the next instruction is retrieved. Figure 1 shows an overview of a typical interpreter. The interpreter holds an image of the source program's code and data in memory, as well as code to perform the interpretation and a *source context block*, which stores the state of the guest that is usually held in hardware registers.

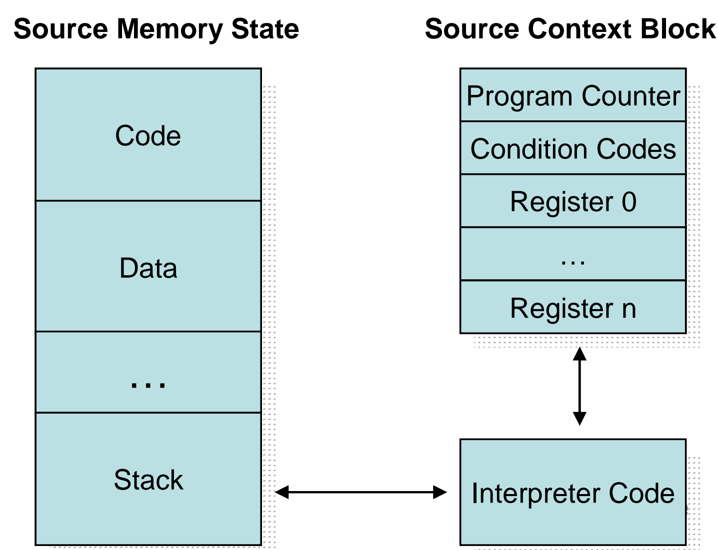


Figure 1 – Overview of interpreter components
Inspired by [Smith2005, p. 30]

Appendix A gives a graphical representation of the most common interpreter types. A simple decode-and-dispatch interpreter *operates by stepping through the source program, instruction by instruction, reading and modifying the source state according to the instruction* [Smith2005, p. 30]. More specifically, the interpreter runs a *dispatch loop*, which uses a `switch` statement[3] to distinguish between different instruction types. The loop invokes an individual *interpreter routine* for each instruction type,

[3] Or an equivalent structure, depending on the language of implementation.

which retrieves further information from the instruction and performs the actual execution. Though such an interpreter is easy to understand and write, a number of advanced interpretation techniques have been developed, which lead to improved performance for certain applications. These techniques are described next.

A common approach to building a high performance interpreter is to append the branch to the next interpreter routine directly to the end of the each single interpreter routine. This is equivalent to inlining parts of the dispatch loop into the interpreter routines. This technique, called *threaded interpretation*, disposes of the dispatch loop and thereby also a number of jumps related to it[4]. Because there are usually patterns in the instruction stream (e.g. a compare instruction is often followed by a conditional branch), threaded interpretation also leads to regular execution patterns, which are often more amenable to branch predictors.

Nevertheless, threaded interpretation still requires some kind of dispatch code, which will associate a binary instruction with the address of the respective interpreter routine. *Direct threaded interpretation* replaces the instructions in the memory image of the source binary with the address of the respective interpreter routine. Though this requires some pre-processing, it minimizes the dispatch time upon repeated instruction execution. A similar, popular technique is *predecoding*, where pieces of information are extracted from the instruction and put into more accessible fields [Smith2005, p. 35]. Machine instructions are usually highly compressed and not necessarily memory-aligned in the manner preferred by the host machine. Predecoding extracts the necessary information to execute an instruction from its machine representation and stores this information within memory-aligned, easy accessible structures. The predecoded information is saved for later reuse and allows skipping the decoding phase on repeated execution.

2.1.2 Binary Translation

Predecoding translates source instructions into an easily accessible format, but still uses central interpreter routines to execute the instruction. Binary translation takes the

---

[4] More specifically, the return from the interpreter routine and the return to the top of the dispatch loop.

approach further. Instead of translating from source instructions to an intermediate representation, binary translation describes the *process of converting the source binary program into a target binary program* [Smith2005, p. 49]. In essence, binary translation will create custom target code for each occurrence of a source instruction. However, this translation is not trivial and poses a number of problems for the translator, the most common of which will be introduced in the following sections.

A basic problem in binary translation is the *code discovery* problem. It describes a translator's inability to discover which sections of a binary actually contain executable code and to distinguish this code from data. Even though many binary formats designate regions containing program code, it is not uncommon for compilers to include additional (read-only) information within these regions. For instance, the MIPSPro C compiler inserts a mask before each procedure that denotes, which registers the procedure saves onto the stack and at which offset from the stack pointer those registers are saved [Huffman1997, p. 106]. Other compilers may include constant values in literal pools at the beginning of a function or introduce padding bytes in order to align code with word or cache line boundaries. Especially when instruction sets have variable sized instructions, it may be impossible to distinguish data from code by just performing a static, syntactical analysis of the instruction stream. Similar problems arise in the presence of indirect jumps, whose targets may be hard to determine.

Indirect jumps also give rise to a second problem, known as the *code location* problem. When translating a source binary to a target system, it is almost inevitable that the target code will reside at a different memory address than it would on the source system. Reasons for that include (but are not limited to) different memory architectures, operating system constraints, different instruction sets or simply optimisations performed by the binary translator that change the code structure. It is the translator's task to make sure that target addresses for indirect jumps within the source executable are fixed up to comply with the memory layout on the target system. A related problem is *self-referencing code* [Smith2005, p. 62], where a program reads values from its text segment. A translator also has to make sure that these reads, which are often used to access constants, return the expected value.

Depending on the amount of emulation desired, *self-modifying code* and *precise traps* may also cause a problem for binary translators [Smith2005, p. 62]. Self-modifying code is concerned with programs that write to code regions, thereby altering

their behaviour. Precise traps try to simulate a system's exact reaction to traps, such as memory faults and hardware or software interrupts. However, both problems are less prominent in environments where the source operating system shields the program from traps and prevents the use of self-modifying code.

A common solution to these problems is to use *dynamic translation*. The idea behind this technique is *to translate the binary while the program is operating on actual input data, i.e. dynamically and to* [..] *translate new sections of code incrementally, as the program reaches them*. In contrast, *static translation* aims *to translate a program in its entirety before beginning emulation* [Smith2005, p. 52, 55]. The advantage of dynamic translation is that it not only allows to lazily discover and translate code as the program tries to execute it, but also that self-modifying code can usually be handled by triggering a recompilation once a code region gets modified. However, dynamic translation comes with a substantial runtime overhead, as the program has to be translated on the target machine. This also means that code to perform this translation and manage previously translated code has to be present on the target machine.

2.1.3 Optimisations in Binary Translation

In contrast to interpretation, binary translation opens the opportunity for applying a whole new set of optimisations to the target code. These optimisations can provide a vast speed-up, making binary translators a popular option for high-speed emulation. A few general techniques are presented in this section. Note that most techniques used in compiler backends and Just-In-Time compilers can also be applied to binary translation. In fact, Just-In-Time compilation is actually a special case of binary translation [Altman2000, p. 40f].

One of the most obvious code optimisations during binary translation is to map parts of the source context block (see section 2.1.1) to target machine registers, thus providing faster access to the context block and allowing some operations to be executed without any memory accesses at all. With this technique, *the speed of execution of the translated code [..] starts becoming comparable to the original source cod*e [Smith2005, p. 52]. Some registers on the target machine are typically reserved for

usage by the translator[5]. If the number of registers available on the source machine is higher than the number of available registers on the target machine, the mapping of source to target registers must be carefully performed as to maximize performance. Previous solutions for register allocation in compilers can be applied to this problem. Depending on the amount of register pressure[6], graph colouring or linear scan register allocation algorithms are commonly used.

During dynamic translation, instructions are commonly grouped into blocks that are translated and scheduled together. The number of instructions within a block is an important performance issue. When the translated blocks are too small, the overhead of scheduling these blocks increases. On the other hand, when the blocks are too large, the effort for translation and optimisation increases, while the resuse of translations might be limited. The natural unit of translation in dynamic binary translators is a dynamic basic block [Smith2005, p. 56]. A static basic block is a sequence of instructions *that can be entered only at the first of them and exited only from the last of them* [Muchnick1997, p. 173]. Dynamic basic blocks usually start after a branch and follow the line of execution until a branch is encountered. They do no stop at branch labels and therefore tend to be larger than static basic blocks. *Translation chaining* increases the size of the executed portions even further by chaining several dynamic basic blocks into a larger *trace*. A trace is usually not stopped by static but only by dynamic jumps, because their target address might be hard to determine until the jump is actually performed. The best size for a trace depends on the actual translator.

Indirect jumps often cause a large evaluation overhead during the execution of programs that have been binary translated. This is particularly an issue for object oriented programs, which tend to make heavy use of indirect jumps to implement polymorphism. I*ndirect jump prediction* or *inline caching* [Smith2005, p. 66] can mitigate this problem. This technique uses profiling to gather information about previously seen target addresses for each indirect jump. The most common jump targets

---

[5] This thesis uses the term “translator” as a short form for “binary translator”. This is in accordance with contemporary writing on the subject, e.g. [Altman2000].

[6] Register pressure describes the inverted ratio of free registers and variables that are to be allocated to these registers.

are included within a trace. At an indirect jump location, the translator inserts code that checks, if the dynamic jump target is included within the trace. In case it is, it can directly branch to that target. Otherwise, a default handler is invoked that will return control to the binary translator.

Finally, another well known source of performance problems are condition codes or flags, which characterize the result of a previous instruction. Common condition codes indicate whether an operation produced a carry, the result is zero or if an overflow occurred. They are frequently tested during the execution of conditional instructions. There are different approaches to updating the condition codes: some architectures feature special *compare* instructions that will explicitly update the conditions codes[7], other architectures implicitly update the condition codes after each arithmetic operation[8] and some architecture do not have a distinct set of condition codes at all[9]. *In general, computing all the condition codes for a given source instruction takes many target instructions, often more than emulating the rest of the instruction, and it can slow emulation considerably* [Smith2005, p. 71]. The problem is prominent on architectures that set condition codes implicitly with every instruction. A common solution is *lazy evaluation of condition codes*. In that scheme, instead of updating the condition codes at the same rate as the source architecture, only the operands and the operation that will produce the condition codes are saved. Instruction that need the condition codes can then produce them from the saved information. This technique is based upon the observation that many architectures update the condition codes more frequently than they are actually read. The usefulness of lazy evaluation depends on the condition code update/usage ratio of the source architecture as well as on how well the source and target ISA match in their handling of condition codes. In some situations, it might be beneficial to always evaluate the condition codes or to use a subset of the condition codes created by the target architecture [Smith2005, p. 74].

[7] For example, the PowerPC ISA provides a special set of arithmetic operations that will also update the condition codes [Frey2003, p. 26].

[8] The Zilog Z80 (which is binary compatible to the Intel 8080) updates condition codes with every arithmetical operation [Zilog2001, p. 75ff]

[9] The MIPSPro architecture does not have a special condition code register, but instead uses compare instructions that write their results into a general purpose register [Huffman1997, p. 52ff].

2.2 The Jikes Research Virtual Machine

2.2.1 The Java Virtual Machine Architecture

The Jikes Research Virtual Machine (JRVM) is a virtual machine targeted at executing programs for the Java Virtual Machine (JVM) Architecture. Therefore it is beneficial to introduce this architecture first. *The Java virtual machine is the cornerstone of the Java and Java 2 platforms. [... It] is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time* [Sun1999, ch. 1.2]. The instruction set of the JVM is called Java bytecode. The JVM must not be confused with the Java Compiler. The Java Compiler translates the Java Programming Language into Java bytecodes, while the JVM executes those bytecodes.

Figure 2 illustrates the general structure of the Java Virtual Machine. The JVM may execute multiple threads, which share a common memory area but also have separate per-thread data areas. The JVM is a stack machine and subsequently Java bytecodes are a zero-operand ISA. This unusual definition[10] of the JVM ensures that it does not favour a specific real architecture.

Each thread consists of a program counter, a stack and a native stack. The program counter identifies the currently executed bytecode. The stack holds all operands that are managed by the JVM, while the native stack is available to non-Java code, which can be called from Java using the Java Native Interface (JNI).

[10] The definition is unusual in the sense that most real-world target architectures for the JVM do not support stack operations, but rather contain a varying number of registers. However, this kind of definition can also be found in other virtual instruction sets, such as P-Code.

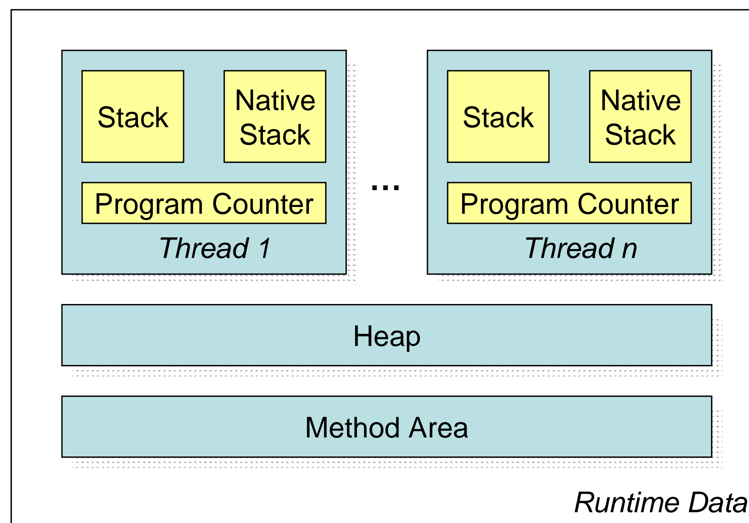


Figure 2 - Java Virtual Machine Runtime Data Areas

Naturally, the stack layout is a crucial part of a stack machine's definition. The Java Virtual Machine Stack is organized into frames, one for each method in a program's call stack. A frame contains all parameters and local variables of that method, an operand stack and a reference to the constant pool, which is part of the read-only method area and maps identifiers to constants [Sun1999, ch. 3.5.4]. Data on the stack is typed. The JVM distinguishes between primitive and reference types. Primitives types are the `boolean` type, numeric types (`byte`, `char`, `short`, `int`, `float`, `long`, `double`) and the `returnAddress`[11] type. Reference types denote references to instances of classes, array types and interfaces on the heap as well as the `null` reference, which references no object. In contrast to languages such as C, the size of all types is well defined. The JVM's stacks are split into slots of 32 bits, which can contain most data types[12]. The operand stack is controlled by the application and contains the operands for bytecode instructions.

The JVM separates memory into a heap and a method area. The method area works similar to a UNIX process' `TEXT` segment [Sun1999, ch. 3.5.4]. It stores the constant pool, field and method data and the Java bytecodes, which form the executable part of a Java program. The heap is a memory area, *from which memory for all class instances*

---

[11] In contrast to all other JVM data types, the returnAddress type does not correspond to any Java programming language type [Sun1999, ch. 3.3.3].

[12] Except for the long and double types, which occupy two slots.

and arrays is allocated [Sun1999, ch. 3.5.3]. The Java bytecode instruction set only features instructions that allocate objects, but none that deallocate them. Therefore, the heap is *managed by an automatic storage management system (known as garbage collector)* [Sun1999, ch. 3.5.3]. The garbage collector manages object allocation, movement, deallocation of unused objects and heap compaction. [Sun1999] intentionally does not specify implementation details about the garbage collector. That allows different garbage collectors to be used in different environments, while retaining compatibility to the JVM specification.

Java bytecodes are a mixed stream of instruction bytes and operand data. The instruction byte identifies the operation that is to be executed. It implicitly defines the number of operands that are following in the bytecode stream and those that have to be retrieved from the operand stack. This specification makes for a very compact instruction set, with a maximum of 256 instructions. However, only slightly more than 200 instruction codes are used and as bytecode instructions are typed, the actual number of different instruction classes is even lower. In contrast to other ISAs, the Java bytecode is directly targeted at running an object oriented Java program. Therefore, it is not surprising to find instructions that access object members, allocate objects, do type checking or perform synchronization within the instruction set. Finally, the Java bytecode ISA only allows type-checked memory access. This feature makes Java bytecodes immune to many of the security problems faced by traditional ISAs.

Efforts to build hardware implementations of the JVM architecture include Sun Microsystem’s picoJava microprocessor, the ARM926EJ-S processor, which uses the Jazelle Direct Bytecode Execution engine and other implementations. However, the emphasis of the JVM Architecture is still on execution on a wide variety of systems whose ISA is not Java bytecode. Consequently, the JVM Architecture has to be emulated on these systems.

2.2.2 The Jikes Research Virtual Machine

The Jikes Research Virtual Machine is a Java bytecode Just-in-Time compiler. It originated in the Jalapeño research project, conducted in 1997 at IBM’s Thomas J. Watson Research Center. IBM recognized that existing JVMs were not built to cope with requirements of high-performance servers, such as *SMP scalability, continuously running JVMs, [limited] GC pause times, thread limits* and optimisations concerning the

use of libraries [Alpern1999, p. 3]. The Jalapeño project was targeted specifically at server machines and meant to fulfil these requirements. After serving as a research environment at IBM for many years, the software was released as an open source project called the Jikes Research Virtual Machine in 2005.

Compared to other virtual machines, the JRVM has two distinguishing features [Alpern1999, p. 1]:

1. **Written in Java**
   The JRVM is mostly written in Java. Though there are previous references to JVMs being written in Java[13], the JRVM is self-hosted and does not need another JVM to run.

2. **The *widespread use of compilers and compiler technologies***
   Instead of providing both an interpreter and a JIT compiler as in other JVMs, bytecodes are always translated to machine code before they are executed [Alpern1999, p. 1]. As servers are the main target of the JRVM, it was anticipated that programs are long-running [Alpern1999, p. 3], which made forgoing an interpreter acceptable. Furthermore, the JRVM features two different compilers and an Adaptive Optimization System (AOS), making it possible to adapt compilation effort to execution requirements.

Writing a JVM in Java poses two additional challenges: Booting the VM and accessing the underlying hardware. The first problem is solved by building a bootstrap that can execute enough of the VM to enable it to compile itself. During the compilation process, a partially running version of the JRVM is created and an image of that process is stored into a file (Boot Image). On the target machine, the Boot Image Runner loads the Boot Image into a new process and branches into it to start execution of the JRVM.

In order to enable low-level hardware access for the JRVM, code replacement is used. If certain function calls, contained within the `VM_Magic` class, are encountered during the compilation, they are replaced with fixed target machine code sequences.

[13] For example, see the *JavaInJava* Project at http://research.sun.com/kanban/JavaInJava.html

This pattern allows the JRVM to be written mostly in Java, only resorting to platform-dependent code, where necessary.

At the heart of the JRVM lie two compilers, the baseline and the optimizing compiler. As *the overall optimization strategy is to compile only* [Smith2005, p. 320], these compilers present different compilation-runtime speed tradeoffs. The baseline compiler does not use an intermediate language, but rather translates Java bytecodes directly into equivalent machine code. The resulting code will emulate the Java stack in memory, bypassing any opportunities for optimisation. This results in fast code production, but slow runtime performance.

Alternatively, the sophisticated optimizing compiler puts much more emphasis on producing quality code, sacrificing compilation performance instead. Figure 3 gives an overview of the different optimisation stages within the optimizing compiler. Initially, Java bytecode is converted into HIR (High-level Intermediate Representation[14]), the first of three internal intermediate representations. Though the JVM architecture defines a stack machine, all intermediate representations are register-based. Not only do register-based intermediate representations allow a closer fit to the target instruction set, but they also facilitate code motion and transformation, leading to better code optimisations [Alpern2000, in ch. "A Dynamic Optimising Compiler"]. HIR groups instructions into extended basic blocks. In contrast to regular basic blocks, Jikes' extended basic blocks are not terminated by method calls or instructions that possibly throw exceptions. Upon them, *simple optimization algorithms with modest compile-time overheads are performed* [Brewer2003, slide 44], including dead-code elimination, common sub-expression elimination and copy propagation.

[14] The expression "high level" refers to the representation having instructions and semantics that are close to Java bytecode.

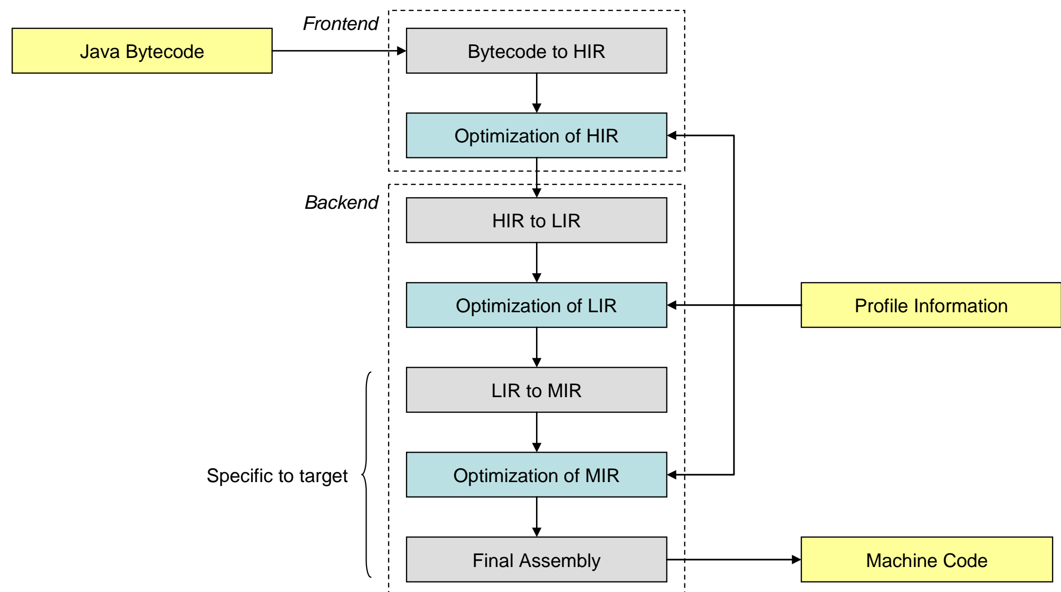


Figure 3 – JRVM Optimising Compiler Overview

Next, HIR is converted to a low-level intermediate representation (LIR). Though similar in principle, LIR is much more specific regarding JRVM internals. Most HIR instructions expand into several LIR instructions. For instance, the `invokevirtual` HIR instruction, which calls a virtual method, is translated into three LIR instructions: the first instruction retrieves the class' Type Information Block (TIB)[15] pointer, the second one locates the address of the appropriate method and the third instruction performs the actual branch.

It is obvious that the same code is much larger in LIR than it was in HIR. In fact, LIR can be two to three times larger than corresponding HIR [Alpern2000, in section "Low-level optimization"]. Due to its compactness and target-platform independency, HIR is the main optimisation target within the JRVM. However, the breakdown of HIR instructions into several LIR instructions offers new optimisation opportunities. Therefore, selected optimisations, such as common sub-expression elimination, are performed on LIR.

As a final step, the code in LIR is translated into machine intermediate representation (MIR). MIR is specific to the target machine's instruction set. For that conversion, a

[15] The Type Information Block manages all type information that does not vary with regard to individual instances. This includes the object's memory layout, its virtual method table and interface information.

dependency tree is created for each basic block, modelling the relationships between the instructions within the basic block. This dependency tree is passed to a Bottom-Up-Rewriting-System (BURS) to select the most appropriate machine instruction for a set of LIR instructions. BURS is *based on term rewrite systems, to which costs are added* [Nymeyer1997, p. 1]. An external grammar specification, which details the capabilities of the target machine's instruction set, maps sets of LIR to MIR instructions. Similar to other parser generators, a tool is used that builds a BURS parser from that grammar. After running the parser on the input LIR, it outputs equivalent MIR for the target machine.

Finally, live variable analysis is applied to determine the lifetime of registers. This information is used to drive a simple linear-scan register allocator. The optimizing compiler adds prologue and epilogues to function calls and, as a last step, outputs machine instructions for the target machine.

The described process can be performed using three different optimisation levels. Higher levels use additional optimisations to generate more efficient code, but are also costly in terms of compilation time. Therefore, it is necessary to determine which optimisation level to apply to which section of code. The Adaptive Optimisation System (AOS) within the JRVM performs this task.

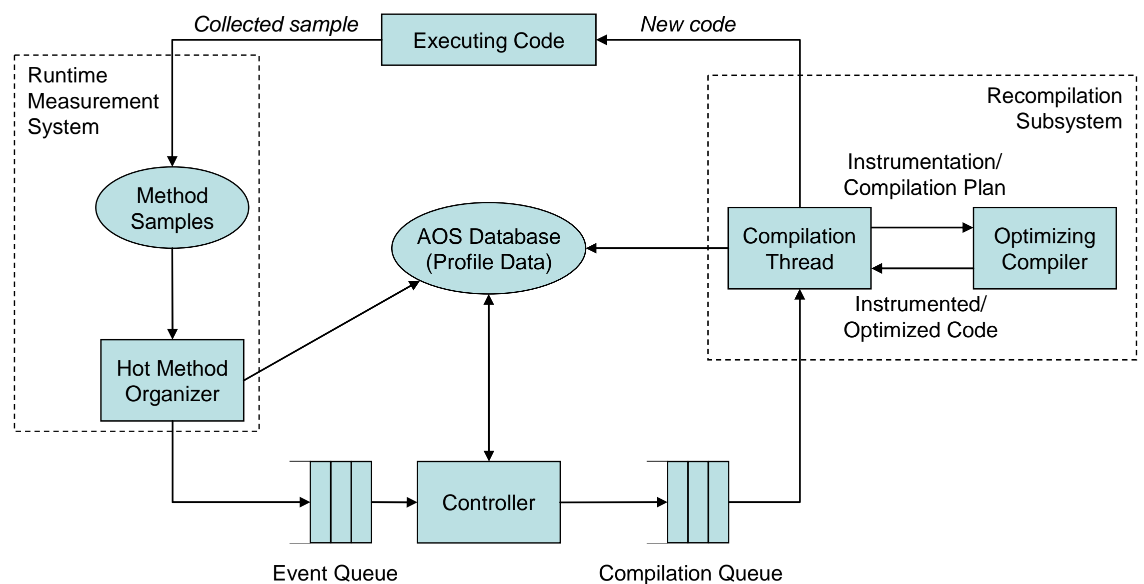


Figure 4 – The Jikes Adaptive Optimization system
Inspired by [Smith2005, p. 321]

Figure 4 given an overview of the Jikes Adaptive Optimization system. Naturally, the central part of the system is a database of profiling data, which is continuously being updated during the execution of a program. As Jikes has a non-preemptive scheduling model that relies on yield points, it is easy for the AOS to add profiling code at these yield points. The profiling information is available to the AOS as well as to the optimizing compiler.

The AOS can increase a method's performance by recompiling it at a higher optimisation level. It is the task of the Runtime Measurement Subsystem to organize and analyze the samples retrieved from the profiling activities. Using that data, it can evaluate how much a method might benefit from recompilation.

The Controller steers the JRVM optimizations. *It instructs the measurement subsystem to initiate, continue or change profiling activities* [Smith2005, p. 322]. Furthermore, it chooses the methods which are to be recompiled and also determines the required optimization level. Those methods are finally forwarded to the Recompilation Subsystem, which controls the activity of the Optimizing Compiler. After the chosen methods are recompiled, On-Stack-Replacement[16] is used to switch to the optimized version.

2.3 Process Virtual Machines

Most computer systems are designed to only run programs that have been compiled for a specific ISA and operating system. One use of emulation in conjunction with virtual machines is to overcome this restriction. A *process virtual machine* architecture is tailored for solving this problem. *By using a process VM, a guest program developed for a computer other than the user's host system can be installed and used in the same way as all other programs on the host system* [Smith2005, p. 83]. A process virtual machine runs as a process on top of an operating system and encapsulates a guest process with a runtime layer, which controls the guest process' execution and manages the communication with the host operating system.

[16] On-Stack-Replacement is a compiler technique that allows code of a running method to be exchanged by a newly compiled version.

Pearcolator uses the features of the JRVM to build a process virtual machine with *extrinsic compatibility*. This means that application compatibility partially relies on properties of the guest software. More specifically, Pearcolator only provides limited Operating System emulation, which restricts its support for guest software.

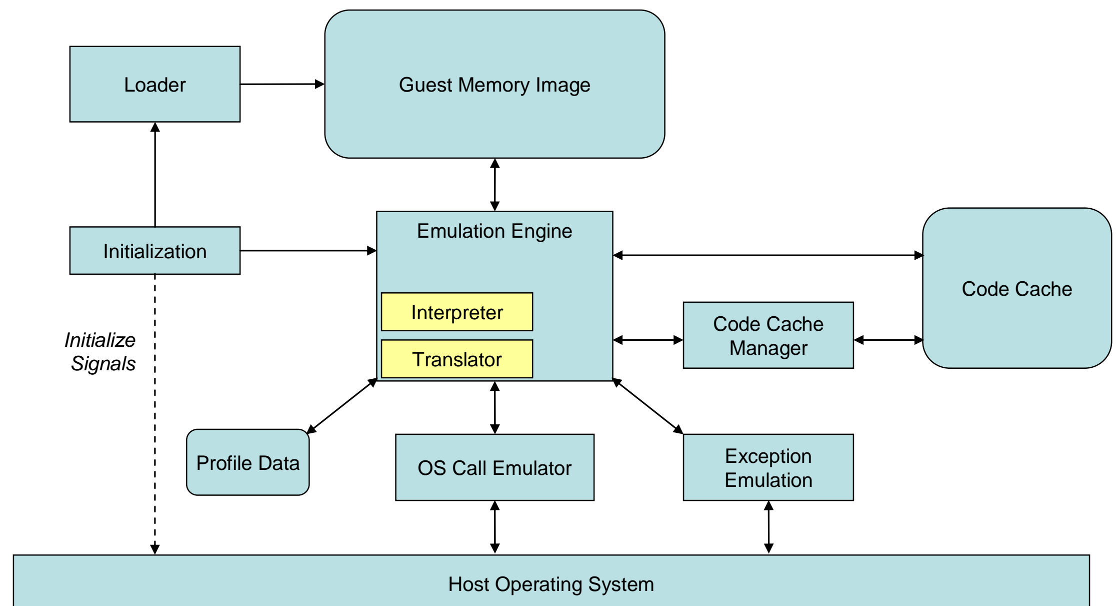


Figure 5 – Components of a process virtual machine
Inspired by [Smith2005, p. 86]

Figure 5 shows the typical components of a process virtual machine:

1. Initialization

The initialization routine sets up the different components of the process virtual machine and establishes communication with the host operating system. It parses user arguments and invokes the loader.

2. Loader

It is the loader's task to read the guest executable and initialize a memory area containing the guest program's code and data. This may involve loading and linking other modules, as required by the guest executable. Note that the guest executable's code is not loaded as an executable memory segment, but that it is rather considered as "input" data for the emulation engine.

3. Emulation Engine

The emulation engine is what drives the guest program's execution. As the process virtual machine's central component, it arranges the collaboration of all other

components. It uses binary translation or interpretation to execute the instructions contained within the guest memory image.

4. **Profile data**

   A process virtual machine may use profile data to optimize the runtime behaviour of the guest program. Most commonly, profile data is used to switch between translation and interpretation. During binary translation, it may also be used to support various optimizations.

5. **OS Call Emulator**

   As the guest program executes an operating system call, the OS call emulator has to emulate the expected operating system behaviour. Depending on the emulated and the host operating system, this may involve translating the guest OS call into a host OS call and marshalling data between both environments. After the call has been executed, the call's result has to be translated into the format expected by the guest.

6. **Code Cache and Code Cache Manager**

   Especially when using binary translation, it is necessary to store translated blocks of code and retrieve them for later reuse. The code cache stores the appropriate target code while the code cache manager decides which code portions to replace in case the cache grows too large.

7. **Exception emulation**

   This component manages the emulation of exception conditions. Depending on the host operating system, this may involve intercepting signals or interrupts from the host and translating them into the format expected by the guest. Exception emulation may be tightly coupled to the emulation engine, which must emulate the *precise guest state (including program counter, register values and trap conditions) when an exception occurs* [Smith2005, p. 87].

2.4 The ARM Architecture

The ARM Architecture is based upon a microprocessor developed by Acorn Computers Limited between 1983 and 1985. This microprocessor, called the ARMv2, was the world's first commercial Reduced Instruction Set Computer (RISC). To market the new processor architecture, ARM Ltd. was formed in 1990 as a joint venture with Apple Computer and VLSI Technology. Since then, ARM Ltd. has become *the*

industry's leading provider of 32-bit embedded RISC microprocessors with almost 75% of the market [Allison2002].

RISC is a microprocessor design philosophy that favours a simpler instruction set. While Complex Instruction Set Computer (CISC) architectures aim to provide powerful and varied instructions, RISC architectures focus on executing a small set of instructions. This makes them easier to implement and more amenable to optimization. RISC architectures often achieve competitive performance through higher clock rates, lower Clocks per Instruction (CPI) ratios and other advanced features.

RISC designs usually share three common features that are visible from a programmer's point of view [Furber2000, p. 24f]:

1. **Load-store architecture**
   Only load and store instructions access memory, while all other instructions work on register operands. This reduces the complexity of the instruction set.
2. **Fixed instruction sizes**
   In machine code, all instructions are represented by the same number of bits. Fixed instruction sizes allow a whole instruction to be retrieved by a single memory fetch, increasing the efficiency of the instruction decoder.
3. **Large amount of registers**
   The original RISC design featured 32 general purpose processor registers, far more than even modern CISC microprocessors, such as the Intel P4 Architecture, offer. The vast number of registers reduces the performance penalty that comes with separate instructions for memory operand fetches.

By constraining the architecture in the described way, RISC designs can be implemented on smaller die sizes and in shorter time, while still maintaining high performance due to the ability to hardwire the instruction decoder and pipeline the instruction execution [Furber2000, p. 25ff].

The ARM architecture is a RISC microprocessor design. It features a load-store architecture with fixed-length 32-bit instructions and a 3-address instruction format. However, it is not a pure RISC design. Some popular RISC features have not been implemented into the ARM architecture: delayed branches, register windows and the execution of instructions in a single-cycle. Instead, CISC instructions have been

included for a small number of commonly used operations, namely multi-register data transfers and compare-and-swap instructions.

	Privileged Modes					
		Exception Modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast IRQ
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR	SPSR	SPSR	SPSR	SPSR

Figure 6 - ARM Register Layout
Inspired by [ARM2000, p. A2-4]

As illustrated in Figure 6, the ARM processor has 16 general purpose user-mode registers, each 32-bit in size. An additional register, the Current Program Status Register (CPSR) maintains a number of flags:

1. **Four arithmetical flags**

   The negative, zero, carry and overflow flag can be set by arithmetic operations. Many processors update the flags after each arithmetical operation. On ARM, most instructions come in two versions, which either do or do not update the arithmetical flags.

2. **Two Interrupt flags**

   The ARM architecture supports two types of hardware interrupts: normal interrupts (IRQs) and fast interrupts (FIQ), with the latter taking precedence over the former. Two bits within the CPSR can be used to mask both types of interrupts.

3. **One Thumb flag**

   Some ARM processors support two instruction sets: the “regular” ARM instruction set with 32-bit instructions and the Thumb instruction set, which provides compressed 16-bit instructions. The Thumb instruction set has been developed to provide better code density. Section 2.4.1 discusses it in more detail. This flag denotes which instruction set is currently executed.

4. Five operating mode flags

The processor's operating mode determines the layout of the register map, availability of certain instructions and can influence memory access privileges, if the system is equipped with a Memory Management Unit (MMU). The ARM processor distinguishes seven operating modes, all of which are listed in Figure 6. The *undefined instruction* and *memory abort* modes are used when trapping undefined instructions or illegal memory accesses (prefetch or data aborts). The undefined instruction trap is also invoked when an instruction for a missing coprocessor is encountered. Therefore it is commonly used to perform software emulation in systems where no floating point coprocessor is installed. The *interrupt* and *fast interrupt* modes are entered to handle the respective hardware interrupts, while software interrupts or system calls are handled in *supervisor mode*. Interrupts are only raised between executions of two instructions [ARM2000, p. A1-4], while aborts can occur during instruction execution, as long as enough state is preserved to restart the current instruction. S*ystem mode* is equivalent to supervisor mode, but permits access to all user mode registers. Finally, *user mode* offers the least privileges and is the mode that user processes are usually running in. When an exception is processed, the processor changes its operating mode accordingly and executes a branch instruction from the *exception vector*[17] to reach the exception handler. The exception type serves as an offset into the exception vector. In addition to the 16 general purpose registers, the ARM processor also contains 15 shadow registers for system-level programming and exception handling. Depending on the processor's operating mode, the shadow registers overlay some of the general-purpose registers. During normal operation, they are not visible to the programmer. Using shadow registers, the ARM processor saves enough state before processing an exception to allow execution to be resumed later as if the exception did not occur. In order to support context switching, ARM also provides special instructions that allow accessing the user mode registers from privileged modes [Furber2000, p. 310].

[17] The exception vector is commonly stored at the beginning of the memory map. It is expected to contain branch instructions, though it may, strictly speaking, contain any ARM instruction.

The ARM architecture features a three stage instruction pipeline[18], which improves processor performance by overlapping the execution of adjacent instructions. The ARM pipeline is split into a fetch, decode and execute stage, omitting the *operand fetch* and *operand store* stages from the classical RISC pipeline [Clements1991, p. 279]. At each pipeline stage, an instruction uses either the memory, the data path or the processor's decode logic, as shown in Figure 7. Note that the before-mentioned CISC instructions may stall the pipeline, as they can occupy the decode and execute stages for more than one cycle. The ARM pipeline is partially visible to the programmer. An instruction reading the program counter (PC) register during the execute stage will actually retrieve the address of the next but one instruction.

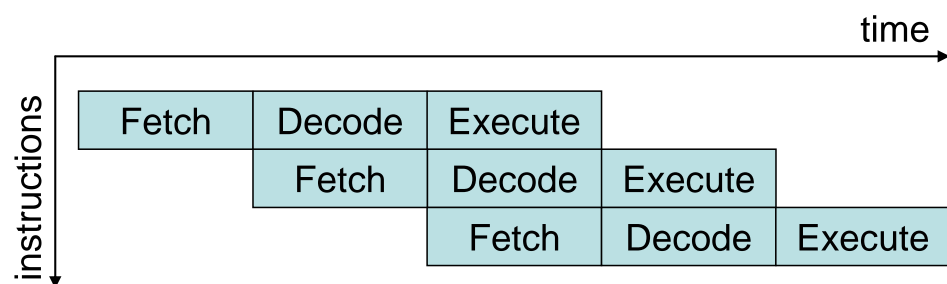


Figure 7 – The ARM 3-stage pipeline

2.4.1 Programmer's Model

Most ARM processors support two instruction sets: the regular 32-bit ARM and the optional 16-bit Thumb instruction set. This section describes the features of the regular ARM instruction set, unless stated otherwise.

The ARM instruction set is a fixed-length instruction set, with each instruction being 32-bits long. *The ARM instruction set can be divided into six broad instruction classes*:

- *Branch instructions*
- *Data-processing instructions*

[18] The ARM9 processor family improves upon the performance of previous ARM processors by using a five-stage pipeline and simulating a Harvard architecture (using a separate data and instruction cache). Nevertheless, for backwards compatibility the ARM9 still simulates the "program-visible" parts of the three-stage pipeline as described here.

- *Status register transfer instructions*
- *Load and store instructions*
- *Coprocessor instructions* and
- *Exception-generating instructions* [ARM2000, p. A1-5].

Almost[19] all ARM instructions can be executed conditionally, depending on the values of the arithmetical flags. The condition is determined by a 4-bit *condition field* within the binary instruction encoding and allows

- *tests for equality and non-equality*
- *tests for <, <=, > and >= inequalities, in signed and unsigned arithmetic*
- *each condition code flag to be tested individually* [ARM2000, p. A1-5].

Branches are expensive, because they flush the instruction pipeline. The ARM instruction set also offers conditional execution of instructions, which does not suffer from this drawback and is therefore commonly used as a alternative for conditional branches and to unroll loops. Code 1 illustrates this by comparing two assembly functions for copying memory regions. The left one relies on conditional branches, while the right one uses conditional execution to unroll the copy loop. Not only does the unrolled version execute 25% less instructions than the normal version, its longer instruction sequence also makes much better use of the ARM pipeline, leading to a better CPI ratio.

```
MemMove_Normal
LDR         r4, [r1], #1
STR         r4, [r2], #1
ADDS        r3, r3,   #-1
BGT         MemMove_Normal
```

```
MemMove_Unrolled
CMP         r3, #2
LDR         r4, [r1], #1
STR         r4, [r2], #1
LDRGE       r4, [r1], #1
STRGE       r4, [r2], #1
LDRGT       r4, [r1], #1
STRGT       r4, [r2], #1
ADDS        r3, r3,   #-3
BGT         MemMove_Unrolled
```

Code 1 - Normal and unrolled ARM Assembly code for a Memory Copy operation[20]. The usage of conditional execution is highlighted.

[19] Architecture versions up to ARMv5 supported conditional execution for all instructions [ARM2000, p. A3-5]. Enhancements to the ISA have since then occupied the encodings of some conditional instructions.

Although the ARM architecture is built upon RISC principles, it provides a high number of addressing modes, for arithmetic as well as load and store instructions. Table 1 gives an overview of the available addressing modes. Most of this flexibility is provided by the barrel shifter, which sits on the processor's B-bus and can perform shift operations on one of the two ALU operands. Additionally, the barrel shifter can also provide a carry flag for logical operations that do not usually create a carry (e.g. AND, OR, etc.).

Instruction Class	Addressing Modes
Data Processing	Immediate Register Logical shift left by register / immediate Logical shift right by register / immediate Arithmetic shift right by register / immediate Rotate right by register / immediate Rotate right with extend
Load / Store	Immediate offset / pre-indexed / post-indexed Register offset / pre-indexed / post-indexed Scaled register offset / pre-indexed / post-indexed
Load / Store Multiple	Increment address before / after transfer (optional write back) Decrement address before / after transfer (optional write back)
Load / Store Coprocessor	Immediate offset / pre-index / post-indexed Consecutive memory reads

Table 1 – ARM Addressing Modes

Adhering to the RISC principles, ARM instructions commonly take three 32-bit operands. Additionally ARM can access other operand types using powerful load-and-store instructions. For instance, the single-register load-and-store instructions also accept other data types, such as signed and unsigned half-words (16 bits) and bytes. Other data transfer instructions can transfer an arbitrary number of registers (including the CPSR) in a single instruction.

The exceptional characteristic of ARM coprocessor instructions is that their exact meaning is not defined by the instruction set, but rather by the coprocessor itself. In the ARM architecture, coprocessors are supported using a flexible "plug-in" system. Each coprocessor within a system is assigned a unique id. Coprocessors listen to the instructions executed by the processor, ignoring all ARM instructions and instructions

[20] Both code snippets assume that at least one word is to be copied. The ARM processor also provides multi-word data transfer operations. For the purpose of this example, these were omitted.

for other coprocessors. Once they encounter an instruction that they are supposed (and able) to execute, they will signal this to the CPU, stalling the instruction pipeline until the coprocessor instruction has been processed. The ARM ISA provides instructions that allow the *processor to initiate a coprocessor data processing operation, ARM registers to be transferred to and from coprocessor registers* and *addresses [to be generated] for coprocessor load and store instructions* [ARM2000, p. A3-25].

Most instructions accept any of the 16 general purpose registers as a target or operand. Though the registers can be used in such a versatile manner, there are two hardware restrictions and a couple of software conventions that apply to register usage. The hardware restrictions are:

1. **Register 15 serves as the program counter**

   Though register 15 (r15) contains the program counter, it can still be used as an operand, as long as the ARM pipeline behaviour is taken into account. When reading from r15, the address of the next instruction but one is returned[21], while writing to r15 flushes the pipeline and branches to the written address. Furthermore, the Branch-And-Exchange instructions (`BX` and `BLX`) may also use the first bit of the new program counter value to determine the target instruction set.

2. **Register 14 serves as the link register**

   Like most instruction set architectures, the ARM architecture provides support for function calls with an instruction that will branch to a destination address and put the address of the instruction following the branching instruction into the link register. The ARM architecture features three different Branch-And-Link instructions[22], all of which use register 14 as the link register. Nevertheless, the register may be used as a general purpose target and source operand register with all other instructions.

---

[21] For instructions that occupy any of the pipeline stages for longer than one cycle, the result of reading from the program counter is implementation defined.

[22] Architecture version 5 provides the BL and BLX instruction, the latter of which is available in two addressing modes.

Additionally to these restrictions, there are a number of software conventions, which are commonly followed. They are captured within the ARM Procedure Call Standard[23] (APCS). The APCS defines register usage conventions, procedure call conventions and the stack layout [STD1998, p. 6-4ff].

The APCS register layout defines the first four registers as “scratch registers”, i.e. registers that do not need to be preserved by functions. The following six registers (r4-r9) serve as variable registers. They need to persist across function calls. The remaining six registers have special uses, though they can be treated as variable registers, if their special functionality is not required by the program. The stack limit register (r10) contains the maximum address that the stack can grow to. It can be used to perform software stack overflow checking. The frame pointer (r11) contains a pointer to the *stack frame* (or activation record) of the active function. The stack frame stores a function’s local variables, return address and parameters passed into the function. Register 12 contains the Intra-Procedure-call scratch register. The linker is often required to insert a veneer between a calling and a called function. In dynamic linking, the veneer may be part of the Procedure Linkage Table (PLT); in static linking it may be a piece of code that compensates for the ARM `BL` instruction being unable to address the whole 32-bit address space [ARM2000, p. A4-10]. The Intra-Procedure-call register can be used as a scratch register by the linker. Furthermore, it *can also be used within a routine to hold intermediate values between subroutine calls* [Earnshaw2006, p. 14]. Finally, the *stack pointer* (r13) holds the address of the top of the stack. By default, the APCS uses a full descending stack.

Code density is a measure for *the amount of space that an executable program takes up in memory* [Computer Desktop Encyclopedia: Code Density]. In embedded systems, code density is important to enable low power consumption. Higher code density not only results in less memory being needed for a particular piece of software, but also allows significant power savings by reducing cache activity, which can amount to about 22% of the total energy expended by a system [Gupta2002]. RISC systems traditionally

[23] See [Earnshaw2006] for a complete definition of the APCS.

suffer from lower code density than CISC systems [Dandamudi2005]. The Thumb instruction set addresses this disadvantage on the ARM architecture.

Many ARM processors incorporate a second instruction set, the Thumb instruction set. Thumb offers a restricted functional subset of the ARM instruction set with each Thumb instruction being only 16 bits long. The goal is to provide higher code density, by limiting instructions to those that are frequently used by compilers.

The Thumb instruction set is not a complete ISA and relies on *recourse to the full ARM instruction set where necessary* [Furber2000, p.188]. Apart from certain subtleties, all Thumb instructions can be translated into equivalent ARM instructions [ARM2000, p. A6-2]. In contrast to the ARM instruction set, Thumb mostly contains 2-address instructions that are executed unconditionally. Thumb also hard codes the APCS assumption of register 13 being the stack pointer into the instruction set. Furthermore, the accessible register set is restricted to the eight registers for most instructions. While ARM instructions set the condition codes optionally, Thumb instruction always set the condition codes when exeucting data processing instructions.

Despite these restrictions, Thumb has an impressive track record. A typical Thumb program requires *70%* of the space of ARM code, while needing *40%* more instructions. Nevertheless, when being run from 16-bit memory Thumb code is about *45%* faster than ARM code. Finally, using Thumb code needs about *30%* less energy than equivalent ARM code [Furber2000, p. 203].

2.4.2 IO and Memory Model

The ARM architecture uses a linear address space of 2^{32} bytes. The word size is 32 bits with the endianess being configurable by a processor input pin. ARM usually expects word addresses to be aligned at 4-byte boundaries. Some architecture versions are also able to access 16-bit half-words aligned at 2-byte boundaries. Accesses that are not aligned according to these rules are called *unaligned accesses*. The behaviour of an ARM processor for an unaligned access differs on the specific system and the instruction used. Generally, the results are either unpredictable or cause an alignment

exception[24]. However, some instructions specifically ignore the bottom address bits or use them to control a rotation of the loaded data[25].

Similar to other RISC processors, ARM does not provide special instructions to communicate with I/O devices. Instead, it uses memory-mapped I/O, where regions of memory are overlapped with I/O device registers. To avoid the related caching issues, the ARM System Control Coprocessor supports marking regions of memory as uncacheable and unbufferable.

2.4.3 Architecture Versions

Having been developed from 1985, the ARM architecture underwent several architectural revisions. The very first version (ARMv1) mostly served as an evaluation architecture for Acorn. It was only used as a second processor to the BBC microcomputer and manufactured in *very small numbers* [Furber2000, p. 147]. Nevertheless, it is today known as the first commercially exploited RISC architecture [Furber2000, p. 147]. The ARMv2 architecture was used in the Acorn Archimedes computer. As the ARMv1, it only had a 26-bit address bus, but did already feature a multiplication instruction. The ARM3 chip introduced the ARMv2a architecture version. It enhanced the previous design with two atomic compare-and-swap instructions and established a standard for the System Control Coprocessor, now widely integrated as coprocessor 15 in most ARM systems. Furthermore, it was the first ARM processor to include a 4kb cache. The following architecture version, ARMv3 was the first version to use a 32-bit address space. It also introduced new aborts (undefined instruction and memory aborts) as well as 64-bit multiplication for certain architecture revisions. The ARMv4 is *the oldest version of the architecture supported today* [ARM Website]. Its main innovation was the introduction of the Thumb instruction set in the ARMv4T architecture revision. As it was the first architecture version that was built upon a formal specification, certain instruction combinations were deprecated (specifically reading the program counter in instructions that spend more than one cycle in the *execute* pipeline phase). With the release of the ARMv5T architecture, new

[24] This only applies to systems containing a MMU capable of checking access alignments.

[25] This behaviour is exhibited by the LDR and SWP instructions.

instructions were added to the instruction set that were *greatly improving compiler capabilities and the ability to mix and match ARM versus Thumb routines* [ARM Website]. The most recent architecture versions, ARMv6 and ARMv7 are specifically targeted at advanced applications. They provide Single Instruction Multiple Data instructions for multimedia applications, the TrustZone security extensions and support for dynamic compilers.

3 Pearcolator Architecture

In the previous two years, a PowerPC and an X86 version of Pearcolator have been produced by other MSc projects. However, these two versions were developed in separate branches and made no attempt to unify Pearcolator into a single code base. The new Pearcolator, described in this thesis, implements a common structure and unifies the PowerPC, X86 and ARM versions. This chapter explains the general structure of the newest Pearcolator version, how Pearcolator integrates with the JRVM and describes those new features that are not exclusively related to the ARM backend implementation.

3.1 Integration into the Jikes Research Virtual Machine

Pearcolator is a dynamic binary translator that is built on top of the Jikes Research Virtual machine. It uses the optimizing compiler within the JRVM to perform the binary translation, while simultaneously being a Java program that is executed by the JRVM. Figure 8 illustrates the relationship between Pearcolator and the Jikes Research Virtual machine.

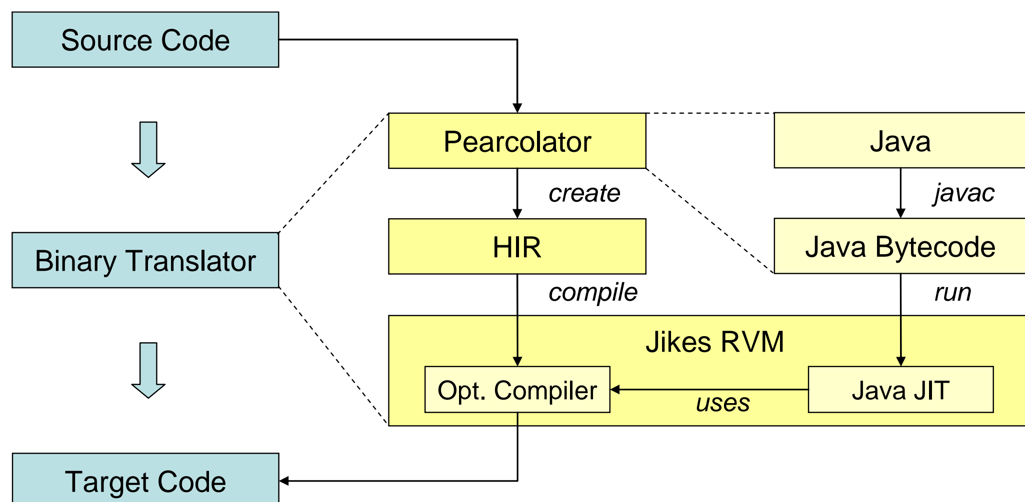


Figure 8 – Relation of Pearcolator to the Jikes Research Virtual Machine

Pearcolator's target architecture is the High-Level Intermediate Representation (HIR), which is used as an input for the JRVM optimizing compiler. For this reason, Pearcolator can translate programs to any target architecture that is supported as a

platform for the JRVM[26]. At the same time, the translated binary can take advantage of the advanced optimisation features offered by the JRVM optimizing compiler. Not only does this increase translation quality, but it also reduces maintenance efforts, as any advances made in the JRVM are automatically applied to the Pearcolator binary translator. However, though Pearcolator is a Java program, the high dependency on the JRVM prevents the binary translator from being run on other virtual machines at the moment.

Therefore, Pearcolator is not distributed as a standalone Java program. Instead, it integrates into the JRVM build process and is compiled as part of the boot image. The boot image is automatically compiled at the highest optimisation level. This allows Pearcolator to circumvent the startup costs, which Java programs usually have when first executing on a Just-In-Time compiler.

In order to use the optimizing compiler for non-Java code, Pearcolator will register “fake”, non-existent Java methods within the JRVM. Each method represents a trace[27] and may include one or more source instructions. The method’s name, which needs to be unique, is derived from the memory address at which the trace’s first instruction resides. As its only parameter the method receives an object handle, which allows the trace to access the source state. The source state consists of the source context block as well as the source memory image. By convention, the method shall return the program counter value after executing the trace. When the JRVM tries to compile any of these non-existent methods, i.e. when it tries to compile a trace, Pearcolator intercepts this request and forwards it to a component that translates the source instructions into equivalent HIR. This component, which is highly dependent on the source platform, is called the Pearcolator backend. The translated HIR is further processed by the optimizing compiler as described in section 2.2.2, finally leading to target machine code.

---

[26] The JRVM currently supports execution on the IA-32 Linux, PowerPC 32 and 64 AIX/Linux/OS X platforms. Work on building a port for IA-32 Microsoft Windows is in progress.

[27] A chain of basic blocks, see section 2.1.3.

Within a trace, the binary translator reads parts of the source state into local registers, performs calculations and writes values from the local registers back into the source state. All of these operations are performed in the HIR ISA, which provides a set of high-level primitives to express loads, stores, arithmetical operations and control flow instructions. Additionally, Pearcolator offers a set of helper routines that create HIR code for commonly needed functionalities.

Integrating Pearcolator at the HIR-level into the JRVM optimizing compiler has two important side effects. Firstly, it means that translated code is always compiled at least at the minimum optimization level. The JRVM baseline compiler, which tries to execute Java bytecode at the minimum translation cost, is not available to Pearcolator. This shortcoming will be addressed in later chapters by implementing an interpreter into Pearcolator. Secondly, the Adaptive Optimisation System can monitor the execution of translated code, thereby being able to dynamically recompile long-running traces at higher optimisation levels.

3.2 Pearcolator Class Architecture

The new Pearcolator architecture adapts the generic process virtual machine architecture described in section 2.3 and shown in Figure 5 (see p. 27). However, as Pearcolator is a framework for binary translation research and supports multiple source ISAs, it does not implement the template architecture as classes, but rather as packages with most packages providing multiple implementations of the expected functionality. Consequently, the packages communicate using well-defined interfaces. The overall Pearcolator architecture is shown as a UML Package Diagram in Figure 9.

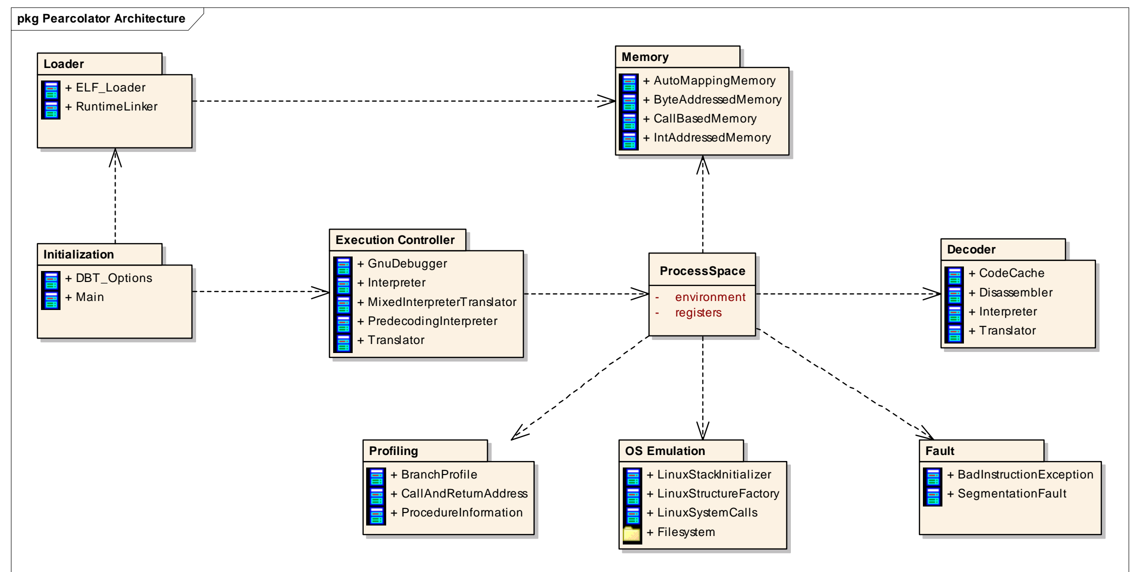


Figure 9 – Pearcolator Overview as a UML Package Diagram[28]

The general package functions have been discussed in section 2.3. The following sections introduce the Pearcolator-specific implementations.

3.3 Initialization

The `Main` class provides an entry point into Pearcolator, which initializes all parts of the Binary translator. First, `DBT_Options` uses a parser built upon the *state pattern*[29] to analyze the command line options that have been provided to Pearcolator. Available command options set various configuration settings, including:

- The executable file and arguments to the executable,
- Debug levels for different Pearcolator components and
- Execution and profiling strategy.

---

[28] The package diagram only gives an overview about the most prominent components. Package names have been adapted slightly to be more readable, the original package names naturally follow the Java package naming convention. See [http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html] for more details on the Java package naming conventions.

[29] The state pattern allows *an object to alter its behaviour when its internal state changes* [Gamma1994, p. 305ff]. It is commonly used to process input in environments while making sure that all possible cases are handled appropriately.

`DBT_Options` will parse that information into static, typed variables thereby validating the user's input. Then control will be passed to the Loader, which reads the executable into the memory image.

3.4 Loader

UNIX and UNIX-like systems, including Linux, store executable files in the Executable and Linkable format (ELF) [Levine2000, p. 206]. The ELF file format was introduced with UNIX System V Release 4 and quickly became popular among UNIX systems. Today, ELF is the standard binary format on most Unix-based operating systems, such as Solaris and Linux [Haungs1998].

Among other features, ELF supports multiple platforms[30] and relocation. Relocation describes *the process of adjusting program address to account for nonzero* [..] *origins and resolving references to external symbols* [Levine2000, p. 149]. It is necessary when an executable is loaded to a memory address other than zero or when the executable uses shared libraries. Shared libraries contain code that is not included within an executable, but is loaded by the operating system once the program executes. Common libraries, such as the C standard library, are usually provided as shared libraries that can be used by multiple programs. Relocations that are performed when a program is loaded are called dynamic linking or runtime linking [Levine2000, p. 205]. In contrast, executables that do not require dynamic linking are called statically linked.

Though Pearcolator provides facilities to support different binary formats, previous versions only offered support for loading statically linked ELF executables. However, most Linux executables are dynamically linked. Therefore, the Pearcolator Loader has been rewritten to support dynamic linking. To understand the new loader, an overview of the ELF format is necessary.

The layout of an ELF file is shown in Figure 14. The file starts with a header that contains a magic number and identifies necessities for reading the file, such as byte order, architecture and the location of index structures that allow descending further into

[30] The ELF format can be read and interpreted on different platforms without having to know the platform that a specific ELF was originally compiled for. However, this does not imply binary compatibility.

the file's structures. *ELF files have a* [..] *dual nature* [Levine2000, p. 62ff]. A loader may treat an ELF file as a sequence of segments that are indexed by the program header table, while linkers treat the file as a set of sections that are described by the section header table.

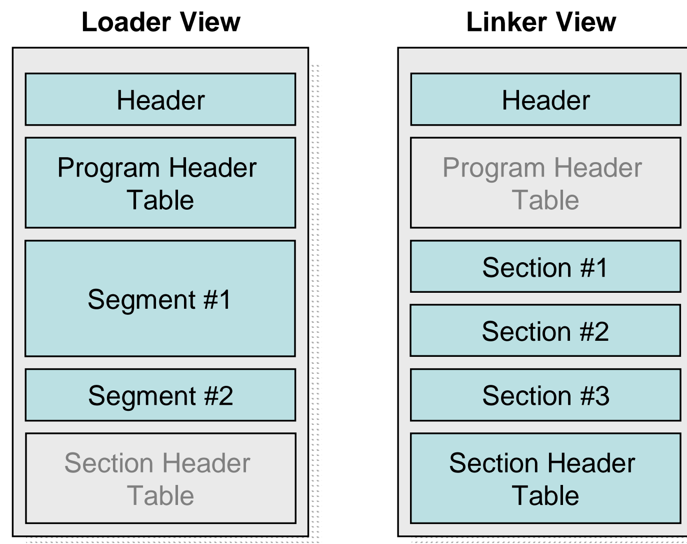


Figure 10 – Perspectives on an ELF file

When loading an ELF file, the loader iterates over the entries in the ELF program header table and maps all loadable segments into memory. Each entry in the program header table describes a single segment with its type, its position in the file, information on where the segment expects to be loaded into memory and access descriptor bits, that describe whether the segment shall be readable, writeable or executable within memory. The loader reads the data on a segment from the file and writes it to the appropriate memory location. Some segments may also reserve additional free space at their end. For instance, this is useful for expressing `.bss`[31] sections without actually wasting space in the executable.

Sections allow a more fine-grained view of an ELF file. Similar to the program header table, the section header table describes the location, length, content and other attributes for the sections in an ELF file. However, sections are usually smaller than

[31] The .bss memory section stores uninitialized program data. Therefore, memory needs to be reserved for it in the executable image, however no initial values for the .bss section needs to be stored within the image.

segments. A single segment may contain several sections, as long as they share the same access rights and are meant to be mapped into memory serially[32]. Table 2 gives an overview of sections that appear frequently in executables. Most executables also contain more than one section of any given type. For instance, a program that requires initializers to be run on start up defines at least one `PROGBITS` section with regular code and one that has to be executed by the loader after mapping the ELF file into memory.

Section type	**Meaning**
`PROGBITS`	Program data or code.
`NOBITS`	No space is allocated for the section in the file itself; however space for that section shall be allocated in memory when the file is loaded.
`SYMTAB / DYNSYM`	Sections of these types contain symbol tables. The symbol table for the dynamic linker (`DYNSYM`) is contained within a separate section.
`STRTAB`	Contains a string table, which associates names with symbols.
`REL / RELA`	Contains relocation information.
`DYNAMIC`	Holds information for the dynamic linker.
`HASH`	Holds a runtime symbol hash table, which helps in finding a symbol within the symbol table, given only its name.

Table 2 – Commonly found ELF section types

For shared libraries to work properly, two problems have to be solved: movement of code and access to functions and variables from another shared library.

Because there are an unlimited number of libraries but only a limited amount of memory addresses, libraries cannot rely on being loaded at a fixed memory address. Rather, they are positioned dynamically by the loader, possibly at different addresses each time they are loaded. Consequently, libraries rely on position independent code (PIC)[33]. However, using only PIC may introduce severe restrictions for some architectures or may even be impossible for others. ELF relocation handles this problem by allowing non-PIC code to be written as if the library was loaded at address zero and relying on the runtime linker to correct these addresses if the library is loaded at another

[32] This is usually the case as linkers arrange the sections accordingly.

[33] Position Independent Code does not use absolute addresses. Instead, addressing is usually performed relative to the program counter. Therefore, it can be loaded anywhere into the address space [HP1997, p. 260ff].

address. The tables in the REL and RELA section contain information about which addresses have to be corrected.

To access addresses (i.e. data and functions) across libaries and executables, ELF introduces the notion of symbols. An ELF symbol associates a name with a numeric value, usually an address. Each library can define symbols and access symbols defined by other libraries. Using special REL and RELA table entries, a library can use the value of a symbol defined by a different library to change an address within its own memory image. To avoid name clashes, libraries may define symbols with different visibilities. Most ELF executables use a Global Offset Table (GOT) in which they store all addresses that are imported from other libraries. Because the GOT is stored at a fixed offset from the beginning of the library that contains it, the library can often access the GOT using PIC. Of course, the GOT needs to be populated by the dynamic linker at load time[34].

The Pearcolator runtime linker starts by determining all required libraries for a given program. Libraries can also depend on other libraries, leading to a dependency graph[35] as shown in Figure 11. Pearcolator traverses the dependency graph in a depth-first manner, mapping libraries into memory as it encounters them. As a library is loaded, Pearcolator also parses its Dynamic Section, Symbol Table, String Table, REL, RELA and Hash Table into Java representations.

---

[34] The GOT also stores function addresses. However, on some systems these are not resolved at load time, but rather resolved dynamically when the program first calls the respective function. To achieve this, each GOT entry is initialized with the address of code that will resolve the address of the actual function when it is first called. After resolving the correct address, the address in the GOT will be replaced. The area of code containing these initialization functions is called the Procedure Linkage Table (PLT). Using the PLT may provide better runtime behaviour for functions that are only rarely called, but need to be linked nevertheless.

[35] The dependencies do not form a tree, because there may also be circular dependencies between shared objects.

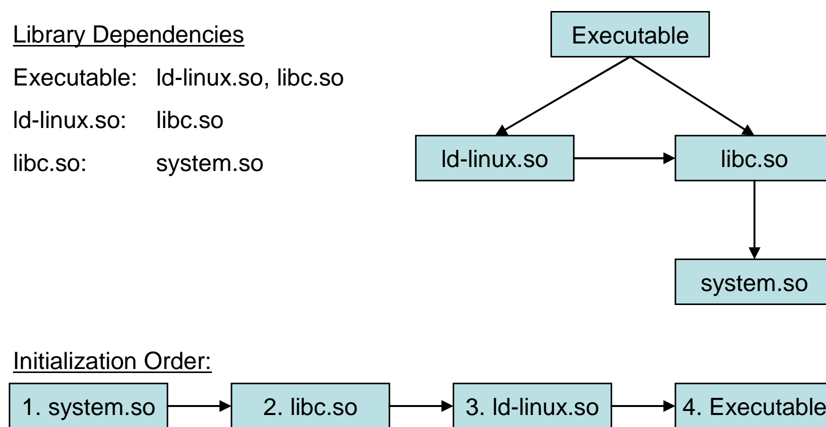


Figure 11 – Dependencies between ELF Shared Objects

In a second stage, relocations are performed for each library. A library's `REL` and `RELA` tables contain a list of all relocations that are to be performed for that library. Each entry in the `REL` and `RELA` tables contains a relocation type, which defines what kind of relocation shall be performed, where it is to be performed and which symbols needs to be resolved to find the relocated address. During relocation, unresolved symbols from one library may be resolved by finding an appropriate global symbol in another library.

The meaning of relocation types are platform specific – therefore, each Pearcolator backend must provide an implementation of relocation types that are only found on the respective platforms. Pearcolator uses the *template method pattern*[36] to implement this requirement efficiently. A general dynamic linker implementation, which performs the loading, mapping of memory, parsing of the above-mentioned structures and controls the workflow is provided. Each platform can derive from this implementation to provide the platform-specific relocation behaviour.

Some executables require that data within read-only segments, which are loaded into read-only memory pages, is relocated. For that reason, Pearcolator removes such protections during the relocation process and restores them after all relocations have been performed.

---

[36] The template method pattern provides a generic implementation of an algorithm, but leaves the individual implementation of certain phases to subclasses [Gamma1994, p- 325ff]. In the given context, it is the translation algorithm that is provided as a template, while some of the architecture specific phases have to be implemented by individual backends.

Finally, libraries may specify initialisation functions, which are to be called before the library can be used properly. These functions are supplied as a function pointer in the `DT_INIT` section or as a list of function pointers in the `DT_INIT_ARRAY` section. Before the initialisation function of a library is called, the initialisation functions of all libraries that it depends on must have been called. However, no order is defined for circular dependencies [SCO2003, ch. 5]. Again using the template method pattern, the Pearcolator dynamic linker passes the addresses of the initialisation functions in a suitable order to the platform-dependant linker part for execution.

3.5 Execution Controller

Execution Controllers are vaguely similar to the Emulation Engine shown as part of the generic process virtual machine architecture in chapter 2.3. Execution controllers implement different strategies to drive the components in the *Decoder* package (see section 3.6). Pearcolator separates control of the decoders from the actual decoding and execution functions to allow easy implementation and testing of new execution strategies. Execution controllers choose whether to use interpretation or translation and generally also determine the pace of the execution. That makes them suitable for implementing different caching strategies or switching between interpretation and translation.

The new Pearcolator version implements the decoders shown in Table 3. An evaluation of their performance may be found in chapter 5.2.1.

Execution Controller	**Implemented Strategy**
Interpreter	Performs simple interpretation.
Profiling Interpreter	Performs runtime profiling during interpretation.
Predecoding Interpreter	Performs threaded, predecoded interpretation.
Profiling Predecoding Interpreter	Adds runtime profiling to the threaded, predecoding interpreter.
Dynamic Translation	Performs binary translation.
Staged Emulation	Optimizes execution time by switching between binary translation and interpretation.
GNU Debugger	Waits for an instance of the GNU Debugger to connect to Pearcolator and lets the debugger determine the execution speed.

Table 3 – Execution Controllers implemented in Pearcolator

The *interpreter controller* interprets every instruction in the source binary separately. It does not perform any caching, but rather decodes each instruction every time it is

encountered. Depending on a configuration option, branch profiling may be performed during interpretation using the *profiling interpreter*.

This *predecoding interpreter controller* implements a Java version of a threaded, predecoding interpreter (see section 2.1.1). As Java does not support arbitrary control transfers (GOTO etc.), this implementation creates dynamic basic blocks of interpreted instructions that are delimited by conditional branches. Any dynamic basic block containing a minimum number of instructions is cached. If the dynamic basic block is to be executed again (i.e. its first instruction is to be executed), it is retrieved from the cache. This saves decoding time and can speed up interpretation considerably. An additional variant of this execution structure exploits the fact that dynamic basic blocks are delimited by conditional branches to perform runtime profiling.

The *dynamic translation controller* only uses binary translation and collects translated instructions into traces. The size of a trace depends on a number of configuration options. It can vary from single instruction traces, with a separate trace for each instruction, to traces that incorporate a whole program. The backend, which does the actual translation, may use the branch profiling information collected by Pearcolator to optimise the trace structure. Naturally, the dynamic translation controller will use a *code cache* to store previously translated traces and retrieve them, when appropriate.

The *staged emulation controller* tries to optimize execution times. It generally uses the predecoding interpreter controller, but switches to binary translation if it assumes that a dynamic basic block might benefit from that. In taking this decision, the controller takes the size of the block as well as its execution frequency into account.

The *GNU debugger controller* is a controller that does not strive to achieve high execution speeds. Instead, it opens a TCP/IP port and waits for an instance of the GNU debugger[37] to connect to Pearcolator, as if it was a remote system. Gdb uses the *gdb Remote Serial Protocol*[38] to communicate with *remote stubs*, remote gdb instances commonly used to debug embedded systems. This execution controller can read the

---

[37] The GNU debugger is a popular debugging tool that is maintained as part of the GNU project.

[38] See the gdb user manual, appendix D for a comprehensive definition of the Remote Serial Protocol. The manual is available at http://sourceware.org/gdb/documentation. Last checked 30th July 2007.

Remote Serial Protocol and steer the execution of the source program accordingly, allowing the user to debug it in gdb as if it was running on a real, remote target machine.

3.6 Decoder

The decoder package hosts templates for three components: a *disassembler*, an *interpreter* and a *translator*. It also provides definitions, helper classes and interfaces that allow these components to interact with other parts of the Pearcolator architecture. All three components are optional, though any working Pearcolator backend shall at least implement one interpreter or translator[39]. The execution of these components is controlled by an `ExecutionController` instance.

A disassembler is a *software that converts machine language back into assembly language* [Computer Desktop Encyclopedia: Disassembler]. For Pearcolator, it is an optional feature that may be implemented by a backend to facilitate debugging guest code. The `Disassembler` interface can be used to disassemble a stream of instructions, while the `Disassembler.Instruction` interfaces defines an object representation for a single disassembled instructions. Some parts of the `Disassembler.Instruction` interface deliberately share the same signature with functions in `Interpreter.Instruction` – this allows an easy implementation of a combined interpreter and disassembler.

Similar to the disassembler, the interpreter and its interfaces `Interpreter` and `Interpreter.Instruction` define the necessary methods for implementing an interpreter in Pearcolator. The interface definition has two important properties: It defines an object representation of a single instruction and it allows querying whether an instruction always has a fixed successor instruction (i.e. whether it is not a conditional or indirect jump). Though none of these features is strictly necessary to create an interpreter, they do leverage Pearcolator's functionality as a research platform. Defining an object representation for an instruction allows separating the decoding from

[39] At the moment, all Pearcolator backends implement a translator and a disassembler, while only the ARM backend also features a working interpreter.

the execution of instructions. Instruction caching and asynchronous decoding are two possible uses for this property. The second property, querying for a fixed successor instruction, allows interpreter instructions to be combined into dynamic basic blocks, within which all instructions have to be executed once a block's first instruction is executed. Figure 12 illustrates the opportunities that this opens for the interpreter: instead of dealing with single instructions, predecoding interpreters can work on blocks of instructions, with only the top address being stored for each block. Furthermore, this scheme allows conditional branches to be easily and efficiently profiled without requiring specific support by the backend. The profiling predecoding interpreter controller (see chapter 3.5) implements this functionality.

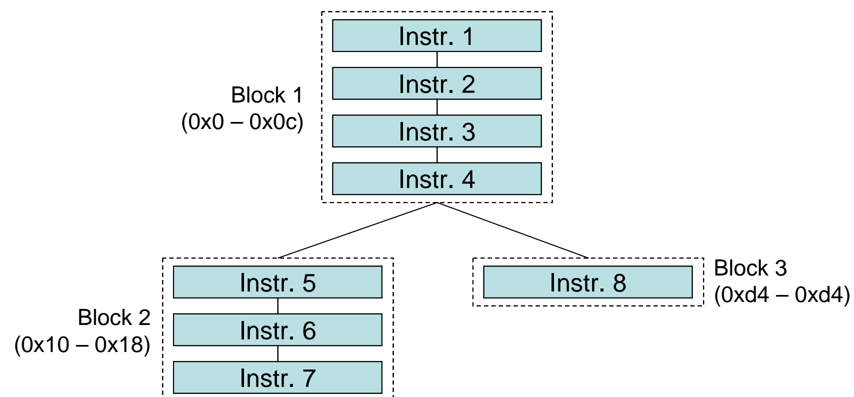


Figure 12 – Building dynamic basic blocks of interpreted instructions

The translator is the most complex part of Pearcolator. Though the translator is heavily dependent on the specific backend, the decoder package provides a comprehensive framework which provides not only methods for commonly used functionality, but rather a translation scheme with template methods that support the implementation of that scheme. The following paragraphs describe the translation scheme. Of course, the package's interfaces also allow the implementation of other translation schemes.

Pearcolator defines a template for the translation process using the template method pattern. In order to build a trace, the backend needs to implement a small set of functions that are called by the class `CodeTranslator`, which controls the translation process and also includes a large number of methods that help in translating common instructions.

The most important function that the backend has to implement translates a single given instruction into the trace. `CodeTranslator` calls this function to construct the

trace incrementally. Figure 13 shows the basic structure of a trace. Every box on the right side of Figure 13 corresponds to a HIR basic block. The trace starts with a `prefill` section, in which trace-wide initialisations are performed and code is created that fills symbolic registers from the source context block. Each instruction in the instruction sequence is translated into its own HIR basic block. More complex instructions (e.g. instruction 3 in Figure 13) may also be translated into more than one basic block. In order to end a trace, the address of the instruction following the most recently executed one is put into a symbolic register, which will be returned from the trace. A `finish` block finally writes the values of the symbolic register back into the source context block and ends the trace's execution. When the trace is ended, registers that have been filled in the `prefill` phase but not been used will be removed from both, the `prefill` as well as the `finish` block.

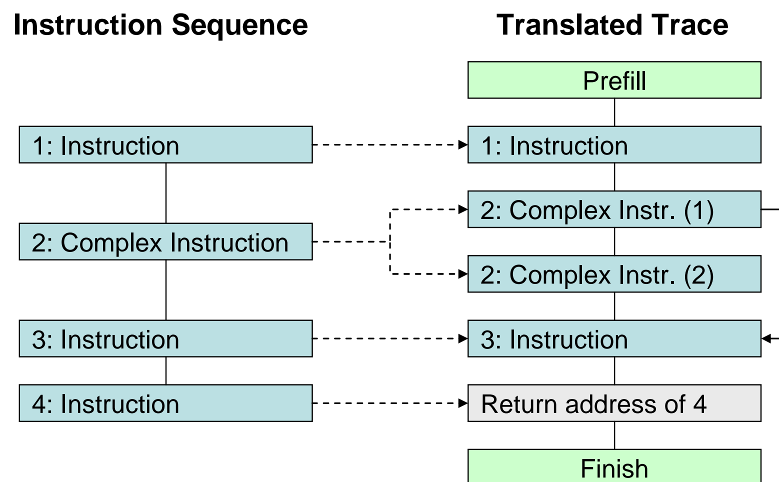


Figure 13 – Translation of a simple instruction sequence into a trace

Branches are a vital part of most ISAs and therefore need to be supported accordingly. A trace can include static[40] as well as dynamic[41] branches and calls[42]. In

[40] Static branches are branches whose target address can be deducted at statically at translation time, i.e. the target address does not depend on the contents of a register or memory location. Note that conditional branches do qualify as static branches, as long as the target address of the conditional jump is known at translation time.

[41] Dynamic branches are all branches that are not static branches.

[42] In this context, a branch is a change of control flow to a different address, while a call is a branch that writes the address of the instruction following the branch into a link register.

order to insert a branch into the trace, the translator calls one of the `appendBranch` functions offered by `CodeTranslator`. However, the framework will not insert code to perform the branch immediately. Instead, after translating all instructions into the trace, the framework will examine which branch target addresses have been compiled into the trace so far. For branches whose targets are already part of the trace, a trace-internal branch, which is directly expressed in HIR, is inserted. If a branch target is not part of the trace, the branch target can either be translated into the trace or a code stub can be inserted instead of the branch, which will end the trace and return the address of the branch target instead. An overridable function is invoked to determine whether a specific branch target should be included within the trace. By default, any branch target address will be included into the trace, except when

- the target address is the start address of another trace that is contained within the code cache,
- the branch is a call or return instruction or
- including the branch target might inflate the current trace to an undesired size.

Including the target of a dynamic branch into the trace is more difficult, because the target address is generally not known at translation time. However, dynamic branches are commonly found in modern programs[43], which is why it is desirable to be able to include them. Pearcolator achieves this aim by using Software Indirect Jump Prediction [Smith2005, p. 66f]. This scheme queries the running program's profile for all locations that the program previously branched to from a dynamic branch. This enables Pearcolator to build an HIR switch statement, similar to the one seen in Code 2. Notice how the statement's default branch handles unknown branch target addresses.

[43] For instance, C++ compilers may use dynamic branches to implement calls to virtual member functions.

```
switch (branch_target) {
      case 0x0004: goto LABEL_0x0004
      case 0x0012: goto LABEL_0x0012
      default:
            <Add missing branch target to program profile>
            <End Trace and return address branch_target>
}
```

Code 2 – Including a dynamic branch target into HIR

Not all Pearcolator functionality has to be expressed directly using HIR. System calls, interactions with the profiling systems or complex mathematics may be performed using calls to Java libraries. The previous Pearcolator design only allowed calls to specific, preselected functions. The new Pearcolator framework allows calls to arbitrary functions to be planted within a trace. This allows calling any Java library to execute complex instructions. The framework can even insert code that will call the interpreter to execute a particularly complex instruction instead of translating the instruction into HIR. System calls, which are described in chapter 3.7, are also handled by calling into regular Java code.

Some backends use different forms of laziness to increase the performance of translated code. Laziness refers to not performing a particular operation unless its result is actually needed. For instance, in the X86 architecture any of the four registers EAX, EBX, ECX and EDX *contain a number of sub-registers that can be referenced as if they were unique registers* [Burcham2005, p. 15]. However, changing a sub-register also alters the parent register and vice versa. Therefore, the X86 backend aims to lazily propagate changes between the parent- and sub-registers only if necessary. Pearcolator provides architectural support for lazy evaluation, though the specifics of resolving one lazy state into another have to be implemented by the backend.

Finally, the decoder package also implements a cache for translated traces and a Utility class that helps performing commonly needed translation functions, such as extracting bits and bit sequences from integral data types or calculating overflow and carry for subtraction and addition.

3.7 OS Emulation

Pearcolator performs *complete OS emulation* [Altman2000 p. 44]. Therefore, it first captures all calls into the operating system and then emulates the expected OS functionality. Most ISAs provide special instructions to perform calls to the operating

system (system calls). Pearcolator intercepts system calls and emulates them using Java code. As of now, Pearcolator primarily emulates Linux system calls, though the ARM backend also includes support for emulating the proprietary Angel Debug Monitor, which serves as a monitor on ARM embedded systems.

The communication between a program and the operating system is defined as part of an Application Binary Interface (ABI). The ABI defines how the operating system is entered by a user-mode program, as well as how and in which order parameters and return values are exchanged. An ABI is platform specific. In order to use the same code to handle Linux system calls across multiple architectures, it is necessary to abstract the details provided by the ABI. Pearcolator defines the `LinuxSystemCallGenerator` interface to transfers this information. By implementing it, any architecture can use the system calls already provided by Pearcolator.

Linux often uses specific structures to communicate with user programs. Instead of marshalling a structure member-by-member to a user program, Linux only returns the address at which the structure has been put into memory and expects the structure's layout to be defined implicitly by the architecture that the operating system is running on. Different architectures have different data alignments and different data type sizes. While previous Pearcolator versions did not tackle this problem, the new version introduces a generic, architecture-independent way of defining Linux system structures and transferring them to memory. Code 3 shows how a mixture of native Java types and Java annotations is used to express the layout of a Linux structure[44] in Java. In order to transfer the defined structure from and to the source memory image, the base class `Structure` uses reflection and the interface `StructureAdapter` to marshal single fields between the structure and the memory. `StructureAdapter` defines storage details, such as data type sizes and alignments. Different architectures must implement their own instances of `StructureAdapter` to be able to use the structures predefined in Pearcolator. Furthermore, the annotation system could be enhanced to express that a structure's member is only defined on certain architectures, thus enabling an even more

[44] As Linux is written in C, its structures are defined in C as well.

general use of the predefined structures. An *abstract factory pattern*[45] is used to create actual instances of the structures.

```
struct stat64 {                          class stat64 extends Structure {
  unsigned short st_dev;                   @_unsigned short st_dev;
  unsigned long __st_ino;                  @_unsigned long __st_ino;
  unsigned long long st_ino;               @_unsigned @_long long st_ino;
}                                        }
```

Code 3 – Example for the definition of a Linux structure (left) in Java (right)

In Linux, everything is a file. Therefore, Pearcolator needs a filesystem model that can cope with that complexity, while still being easy to implement. The new Pearcolator offers interfaces that try to fulfil this requirement: `FileProviders` allows querying files by path and `Files` are object representations of a single file.

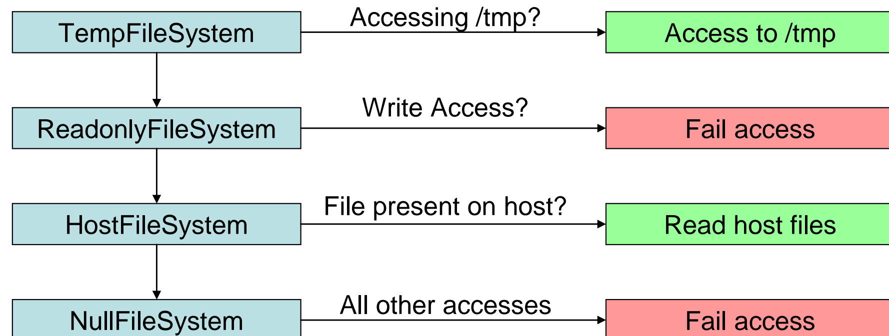


Figure 14 – Default Pearcolator Filesystem configuration

A Pearcolator filesystem is represented by a number of `FileProviders`, which are created by a Factory Method[46]. The `FileProviders` use the *chain of responsibility*[47] pattern to forward the request to open a file until a `FileProviders` either opens the file and returns its object representation or rejects opening the file (and stops forwarding

---

[45] An abstract factory pattern is a creational pattern that abstracts the instantiation of objects [Gamma1994, p. 87ff].

[46] A factory method is similar to an abstract factory in that it hides the instantiation of objects. However, a factory method is specifically targeted at letting subclasses provide an implementation of an interface [Gamma1994, p. 107ff].

[47] In the Chain of Responsibility pattern, a request is passed along a chain of objects, until an object is found that handles the request [Gamma1994, p. 223ff]. In the given case, it allows sophisticated file system structures to be composed from simple primitives.

the request). The chain of `FileProviders` is delimited by a Null Object[48]. Table 4 gives an overview of the `FileProviders` currently supported by Pearcolator and Figure 14 shows an example configuration of a file system, which uses the described components to allow write access to the temporary files and read access to any file on the host.

File Provider	Usage
`HostFileSystem`	Tries to open a file from the host's file system.
`ProcFileSystem`	Simulates the Linux /proc file system.
`ReadonlyFileSystem`	Denies all write requests to a filesystem.
`RemappingFileSystem`	Maps all access to a path A on the guest to a different path B on the host.
`TempFileSystem`	Allows write access only to the host's temporary storage location.
`NullFileSystem`	Null object that allows no file to be opened.

Table 4 – File Providers supported by Pearcolator

Similarly, the uniform `File` interface hides the differences between regular files, sockets, standard input/output and other files (such as those provided by the Linux `/proc` filesystem).

3.8 Profiling

Pearcolator provides general facilities to perform branch profiling. Using the interpreter interfaces described in chapter 3.6, Pearcolator can automatically profile the dynamic behaviour of a program during interpretation. Furthermore, backends can notify the profiler about the location of procedure call and return instructions that are encountered either during interpretation or translation. This additional information enables the profiler to anticipate the original structure of the program.

The profiling information is primarily used by the translator during the construction of traces. It allows better estimation of method boundaries and software indirect jump prediction to be performed. Furthermore, the translator tries to mirror a program's structure when building traces. This leads to traces whose structure is close to high-level language methods and can therefore reduce duplication of code segments into several

[48] The Null Object pattern is a popular alternative to using specific values for non-present objects (like null in Java) [Woolf1996]. Instead, an empty interface implementation is provided.

traces. The profiling information is also used to guide the translator's inlining decisions. Finally, having some information on the structure of a program can be used to asynchronously pre-translate parts of the program, which might be beneficial in a multiprocessor environment.

Profiling information can either be collected at run time or loaded from an XML file when Pearcolator starts. Figure 15 shows the data model used by the XML file.

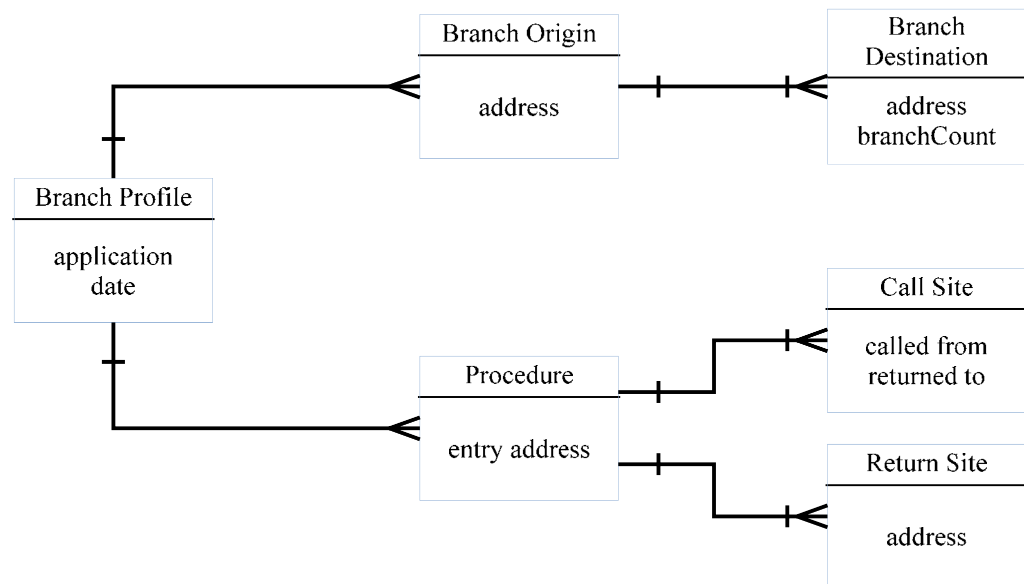


Figure 15 – Entity Relationship Diagram of the Pearcolator Profiling Data Model

3.9 Memory

The Pearcolator memory model is defined by the `Memory` interface. Signed and unsigned 8-bit, 16-bit and 32-bit quantities can be read and written using this interface. The interface further provides methods that will insert an HIR representation of a memory access into a trace, either by directly inlining the appropriate HIR instructions or by planting a call to the memory interface. This shall allow efficient memory accesses within traces. The interface also assumes that memory is divided into pages, with each page having any combination of the permissions read, write and execute. Current implementations of the `Memory` interface are shown in Table 5.

`ByteAddressedMemory` and `IntAddressedMemory` both implement pages as arrays of primitive data types and only create a page if it is actually mapped by the running application. Therefore, Pearcolator can simulate big memory spaces by allocating memory pages lazily. The size of a memory page and the endianess of the

memory can be configured for each memory implementation. When a page is mapped from a file, both implementations support using the `java.nio` package functions, with which Java enables a technique for mapping files into memory equivalent to the Linux `mmap` system call.

By providing further decorators, like `AutoMappingMemory`, the Pearcolator memory model can easily be enhanced to support memory-mapped IO devices or programmable Memory Management Units (MMU). An evaluation of the different integer- and byte-based memory implementation can be found in chapter 5.2.3.

Implementation	**Usage**
`CallBasedMemory`	An abstract base class for new memory implementations, which performs memory accesses during the execution of traces by using calls to the Java functions in the `Memory` interface.
`ByteAddressedMemory`	Organizes memory pages as arrays of bytes. Two implementations, for little and big endian, are available.
`IntAddressedMemory`	Organizes memory pages as arrays of integers. Two implementations, for little and big endian, are available.
`AutoMappingMemory`	A *decorator*[49] that will prevent faults on accesses to memory pages that were not previously mapped. Instead, an empty page is mapped into memory and the access is repeated.

Table 5 – Implementations of the Memory interface

3.10 Faults

As Pearcolator is written in Java, it seems natural to use Java exceptions as a means of fault handling. To enforce uniform interfaces, Pearcolator provides a package with exception classes for the most common faults. Backends can either use these classes directly or inherit from them, if more detailed fault information is required. Currently, faults for invalid memory accesses (Segmentation Fault) or Instructions (Bad Instruction Fault) are available.

---

[49] The Decorator Pattern is a structural software design pattern. It attaches responsibilities to an object dynamically, thus providing a flexible alternative to subclassing [Gamma1994, p. 139ff]. It is used here, because it easily allows modifying the behaviour of any memory implementation to map unmapped pages lazily when they are first accessed.

3.11 Process Space

The final component within the Pearcolator architectural model is the Process Space. This class works as a communication hub, combining the previously described components. Furthermore, it manages the source context block, i.e. the registers, flags and other state information particular to the source architecture. Pearcolator can emulate several processes at the same time by instantiating different Process Spaces for each of them.

During the initialisation process, it is the Process Space that decides which Pearcolator components to use in order to execute a program. This includes deciding on an appropriate memory model, selecting an operating system emulation and initializing the source context block. The Process Space can use information provided about the executable by the loader during this stage.

4 The ARM Emulator

4.1 General Architecture

The ARM backend for Pearcolator emulates the ARMv4T architecture. This chapter describes its implementation.

The X86 and PowerPC backends perform instruction decoding, disassembling, translation and interpretation[50] in a single class hierarchy. This approach has several disadvantages, most noticeably that the architecture is hard to maintain and that the whole backend heavily depends on the JRVM, thus making it unsuitable for execution in another Java Virtual Machine. The ARM backend seeks to mitigate this problem by taking an architecture-driven approach to designing the backend, putting an emphasis on portability and loose coupling of the backend's components.

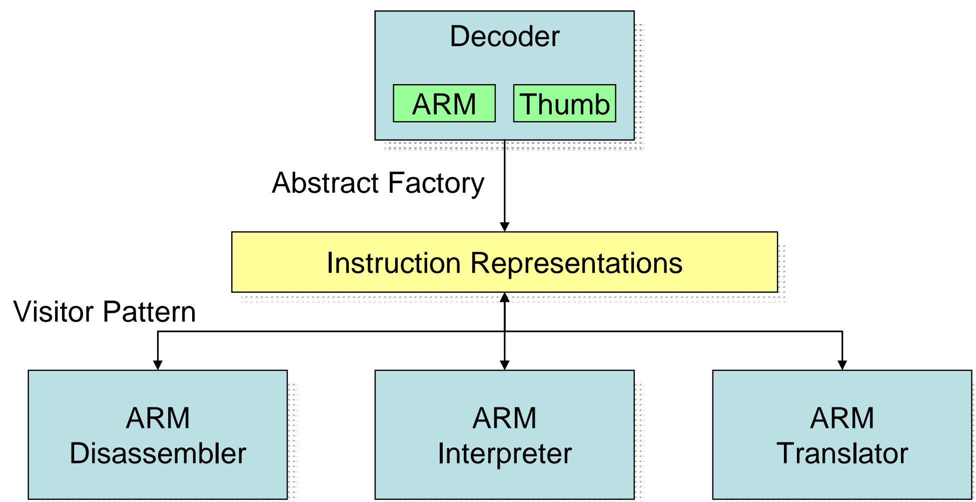


Figure 16 - Architecture of the Pearcolator ARM backend

Figure 16 shows a high-level abstraction of the ARM backend architecture. It consists of five major components, each with individual requirements:

1. **Decoder**

   The instruction decoder maps a binary instruction representation to a logical object representation. Because the ARM architecture defines two instruction sets, the 32-bit ARM instruction set and the compressed Thumb instruction set, the decoder

[50] Fully working interpretation is not available for the PowerPC Pearcolator backend.

must be capable of decoding both instruction sets. Furthermore, as it is frequently invoked, an efficient decoding algorithm has to be implemented.

2. **Instruction Representation**

   An instruction representation is an object representation of a decoded instruction. It shall hide the details of the binary instruction format and allow typesafe access to the options included in an instruction. Instruction representations are the output of the decoder. Instructions that belong to the same instruction class share a common instruction representation. ARMv4T processors contain a special decoder which transforms a Thumb instruction into an equivalent ARM instruction. Following this approach, the ARM backend will use the same instruction representation for both, Thumb and ARM instructions. This greatly simplifies the development and testing effort required in the translator, interpreter and disassembler.

3. **ARM Translator**

   This component translates the ARM instruction representations into HIR for the JRVM optimizing compiler. As such, it is the only component that is not compatible with other Java Virtual Machines. Naturally, a major concern for this component is the generation of correct and efficient HIR code.

4. **ARM Interpreter**

   The ARM interpreter is responsible for interpreting instruction representations. In contrast to the translator, it works independently of the Java Virtual Machine that it is running on. Nevertheless, it also seeks to provide efficient and correct emulation of ARM instructions.

5. **ARM Disassembler**

   The disassembler is responsible for converting an instruction representation, as created by the decoder, into a human-readable string. It is possible to invoke the disassembler from the interpreter or translator (for a currently processed instruction), without invoking the decoder again.

The backend components use two patterns as their means of communication: An abstract factory pattern and the visitor pattern[51]. As illustrated in Figure 17, a component that wants to process an ARM instruction first calls into the decoder. Next, the decoder calls into a, possibly custom, abstract factory to create an instruction representation, which is then returned to the caller. Finally, the visitor pattern can be used to access the instruction representation after it has been returned.

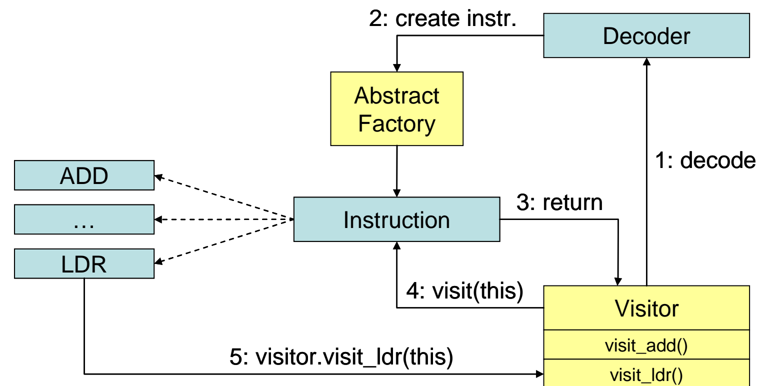


Figure 17 – Communication between ARM backend components.

The following section will give details about the implementation of the described components.

4.2 The ARM- and Thumb-Decoder

As the ARM instruction set has developed over time, its binary instruction encodings are not as regular as those in other processor architectures. Therefore, building an instruction decoder manually is error-prone and might possibly yield an inefficient implementation. To avoid these problems, emulator-builders can resort to software that

[51] The Visitor pattern decouples the implementation of a data structure from operations that are performed upon that data structure. It allows defining *a new operation without changing the classes of the elements, on which it operates* [Gamma1994, p. 331ff].

automatically builds a decoder from an instruction set specification[52]. However, these tools are often specific to a target environment[53].

The ARM backend generalizes upon the idea of building a decoder from an instruction set specification by using data mining instead of a specialised application to build the decoder. Data mining is *a problem solving methodology that finds a logical or mathematical description […] of patterns and regularities in a set of data* [Decker1995]. In contrast to other decoder-builders, the output of a data mining application is not an implementation of a decoder, but rather its implementation-independent description.

More specifically, classification will be used to derive rules that identify the appropriate instruction class from a binary instruction. When the instruction class is known, its fields can be extracted and an appropriate instruction representation can easily be created.

Building a decoder using data mining applications has two advantages. First, the decoder is proven to be correct and second, data mining applications strive to generate models of minimal complexity, leading to an efficient decoder description.

Figure 18 shows a data flow diagram of the decoder construction process. The process is split into three phases: instruction set specification, data mining and decoder implementation.

---

[52] For example, the UQBT Binary Translation Framework uses the New Jersey Machine Code Toolkit to build a decoder from a syntactic architecture specification. [UQBT: Adaptable Binary Translation at Low Cost, p. 62].

[53] For instance, the New Jersey Machine Code Toolkit can build decoders in C and Modula-3. [New Jersey Machine Code Toolkit, Reference Manual, p. 2]

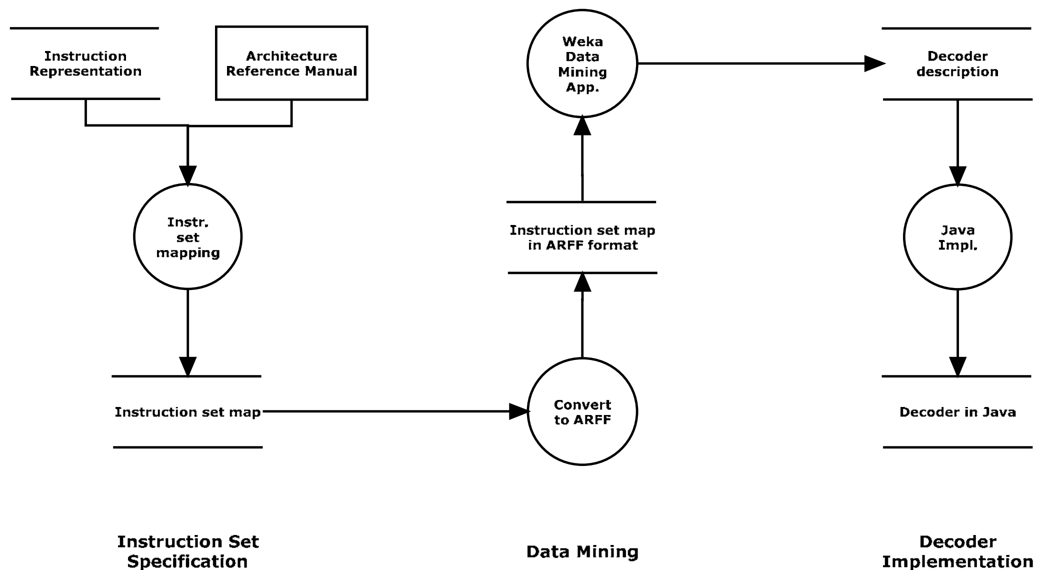


Figure 18 – Data flow diagram of the decoder construction process

During the instruction set specification phase, each instruction's encoding (as specified in [ARM2000, p. A3-2ff]) is mapped to an instruction representation. An instruction representation provides an object-oriented way of accessing the fields and options of an instruction.

In the Data Mining phase, a data mining application's classifier is used to maps an instruction's binary encoding to its instruction representation. The Weka Data Mining Application[54] is well respected in the scientific data mining community and provides various implementations of classification algorithms. Therefore it has been selected for this task. A classifier can output its result in different formats. For this application, a decision tree seems the most appropriate format, because it can easily be implemented in Java code. Furthermore, in contrast to decision rules, the decision tree avoids expressions being evaluated multiple times. These two factors contribute to an efficient implementation.

Weka requires an input file in the *Attribute-Relation File Format*[55] (ARFF). The instruction set map is converted into the ARFF format using a custom tool. Before its conversion, the instruction set map used wildcards for bits that are irrelevant in

---

[54] See http://www.cs.waikato.ac.nz/ml/weka. Last checked on 26th July 2007.

[55] See http://www.cs.waikato.ac.nz/~ml/weka/arff.html for a formal specification. Last checked on 26th July 2007.

determining the appropriate instruction representation. During the conversion, these wildcards are expanded to create the set of all possible instruction encoding that share the same instruction representation. In cases where two definitions are clashing, the converter gives precedence to the more specific definition, which contains less wildcards. For example, given the set of definitions in Figure 19, the second definition would take precedence of the first one.

#	Instruction Class Bit	27	26	25	24	23	22	…
1	Data Processing	0	1	x	x	x	x	…
2	Undefined Instruction	0	1	1	0	x	0	…

Figure 19 – Sample Instruction Definitions

Though decision trees are generally well suited for these kinds of problems, they cannot exploit special features of the target language. For example, in Java it is possible to transform a comparison of multiple bits into a, possibly more efficient, `switch` statement. For the data mining application, each bit would be considered a separate attribute and therefore, would be evaluated individually. To allow the data mining tool to join adjacent bits, all combinations of three to four adjacent bits were duplicated into individual attributes during the conversion process. The data mining application can then consider these bits as a single attribute when creating the decision tree.

Finally, the decision tree output by the data mining application can trivially be transferred into a Java program. The said process is repeated for both, the ARM and Thumb instruction sets, creating an individual decoder for each. Both decoders are encapsulated within the ARM Decoder component. The images in Appendix B and Appendix C show the implemented decision trees.

4.3 Instruction Representation

The ARM backend relies on intermediate instruction representations for the communication between the decoder and the other components. An instruction representation is a class whose properties publish a binary instruction’s type, options and operands, therefore hiding the details of the binary encoding. Furthermore, as most Thumb instructions have equivalent ARM instructions, instruction representations also make the difference between Thumb and ARM instructions transparent to the interpreter and translator. Finally, Pearcolator’s `Interpreter.Instruction` interface assumes that a decoded instruction can be cached in memory (see section 3.6). Having instances of instruction representations easily allows doing that.

Table 6 shows the different instruction representations. They have been chosen to mirror similar operand numbers, options and semantics. Differences in addressing modes are hidden by the `OperandWrapper` class, which can represent immediate values, register values with optional offsets as well as the results of shifting a register by another register or an immediate operand.

Data Processing	64bit Multiplication	Move to CPSR/SPSR
Single Data Transfer	Atomic Swap	Coprocessor Data Transfer
Block Data Transfer	Branch	Coprocessor Data Process.
Software Interrupt	Branch and Exchange	Coprocessor Reg. Transfer
32bit Multiplication	Move from CPSR	Undefined Instruction

Table 6 – Instruction representations within the Pearcolator backend

4.4 ARM Translator

4.4.1 Scheme Selection

The ARM translator converts an instruction representation into HIR, which is then passed to the JRVM optimizing compiler. This process is controlled by the Pearcolator framework, as described in section 3.6. The ARM translator mostly needs to implement functions that access the source context block and features that translate a single instruction, given its address.

The traditional approach to writing a translator component, taken by all previous Pearcolator backends, is to extract the different fields from a binary instruction and write functions that create equivalent HIR code. For the ARM backend, the binary instruction would be exchanged for an instruction representation obtained from the ARM decoder. However, the unique situation that a binary translator is using the same facilities to perform binary translation and to execute its own code allows another translation approach, which had not previously been explored.

As Figure 20 illustrates, there is an equivalence between the way Pearcolator and the guest program are compiled by the JRVM. In the previously described translation scheme, the translator class itself would first be converted from Java bytecode into HIR, which is then compiled and executed. When the translator code is executed, it again generates HIR from the source program, which will be processed in the same way by the JRVM.

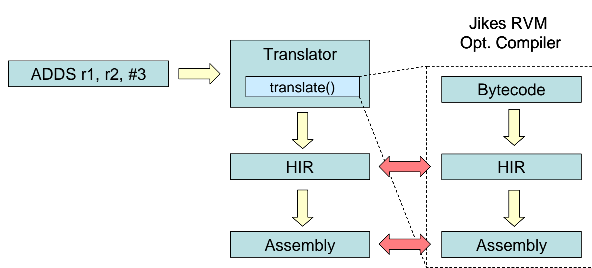


Figure 20 - Confluence between guest and host translation for the Pearcolator Binary Translator

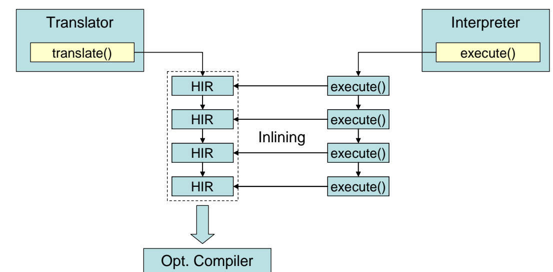


Figure 21 – Inlining an interpreter during translation

However, instead of going through the compilation process twice, it would be possible to make the `translate` method from Figure 20 behave like an interpreter. Consequently, the call to the `translate` method itself could be inlined into the trace, effectively replacing the translator with an interpreter, whose code is inlined and optimized. Figure 21 visualizes that scheme.

In order to investigate the feasibility and performance of this new approach, a vertical prototype[56] was developed. The prototype implements a subset of the ARM 32-bit instruction set, which allows it to run a loop with multiple data processing instructions within the loop. Several assembly programs that exercised the loop were created, each varying the number of instructions within the loop body or the number of iterations over the loop.

Figure 22 compares the execution times for running the said loop in different configurations. The loop has been benchmarked using both the traditional and the new translation scheme. It is apparent that the new scheme performs worse in all scenarios. This can be attributed to the lack of specialised optimisations applied to it – especially reads and writes from registers are highly optimised during the traditional translation, but less so in the newer scheme. However, more importantly it can be observed that, within the new scheme, increasing the number of loop iterations by 100 leads to a performance loss of about 40%, while only doubling the number of instructions incurs a performance penalty of 150%. It can be deduced that the compilation time required in the new scheme is far higher than in the traditional one. This was partially expected,

[56] A vertical prototype is a program in which a specific subset of features is fully implement, thus allowing their feasibility or to investigate design alternatives.

because the new scheme heavily relies on the optimising compiler to remove the inefficiencies of the interpreter code. Manual investigation of the created HIR confirms that the amount of generated HIR per instruction is much higher with the newer scheme. Though it is expected that the runtime performance of the new scheme could still be improved, the required compilation time would still remain an issue. As performance is one of the primary goals of binary translation, it has been decided to implement the ARM translator backend using the traditional scheme. However, it would still be possible to implement the new translation scheme as a separate execution controller within the new Pearcolator architecture.

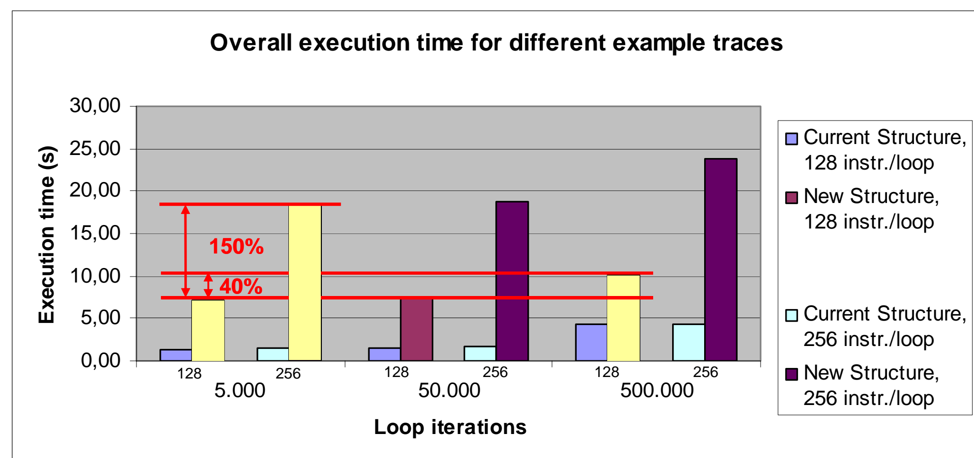


Figure 22 – Comparison of execution time for different binary translation schemes

4.4.2 Implementation

The general Pearcolator translation process has been outlined in chapter 3.6. Detailed descriptions of the translation process for two specific backend have been published previously [Matley2004, ch. 3 & 4], [Burcham2005, ch. 62]. Therefore, this report focuses on the differing specifics in the ARM translator.

The PowerPC and X86 backend translate instructions from their binary representation. The ARM backend provides a looser coupling of its components by separating the decoder from the translator, changing its input to instruction representations instead of binary instructions. Additionally, only one translator had to be implemented for ARM and Thumb code.

The ARM architecture allows using the program counter as a source or target register in many instructions. However, the ARM translator does not treat the program counter as a distinct register. Instead, it handles accesses to it depending on the access type.

Instructions that write into the program counter register are treated as indirect branches, allowing the branch optimisations illustrated in Code 2 (see p. 55) to be applied. [ARM2000, p. A9-4 & A9-10] propose two instructions that are to be used to return from a procedure call[57]. Though both are writing into the program counter register, the translator actually treats them as procedure returns instead of indirect branches.

Similarly, most instructions may use the program counter as a source operand. Reading the program counter exposes parts of the ARM pipeline, which is in the execute stage when registers are read. Therefore, the program counter always points at the next but one instruction (8 bytes from the current instruction in ARM mode, 4 bytes from it in Thumb mode). Also, a few instructions mask the first bits of the program counter, when it is accessed. The translator converts reads from the program counter to HIR constants, resolving the complexities of reading the program counter at compile time. This enables faster program execution and removes the need of managing a program counter register during execution. Furthermore, the optimising compiler has been enhanced with a new optimization phase, which performs constant-folding[58] and propagation[59] on HIR expressions involving constants[60]. It allows calculations involving the program counter –which are often used to read from the text segment– to be evaluated at compile time.

4.4.3 Conditional Instructions

ARM instructions can be executed conditionally. However, most instructions are actually executed unconditionally. Therefore, the translator has been designed so that every instruction is first translated as if it was unconditional. For conditional instructions, the translator inserts a prologue before the instruction, which conditionally

---

[57] The `MOV pc, lr` instruction and a block data transfer that contains the program counter in its registers list.

[58] Constant folding *refers to the evaluation at compile time of expressions whose operands are known to be constant* [Muchnick1997, p. 329].

[59] *Constant propagation is a transformation that, given an assignment $x \leftarrow c$ for a variable x and a constant c, replaces later uses of x with uses of c as long as intervening assignments have not changed the value of x* [Muchnick1997, p. 362].

[60] This work has mostly been implemented by Dr. Ian Rogers.

skips the instruction code. Due to this technique, the single translation functions do not need to implement conditional execution features.

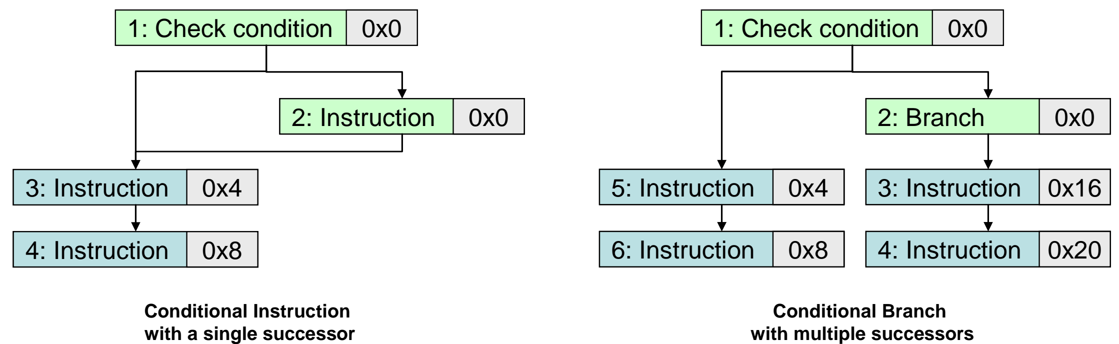


Figure 23 – Translation of a conditional instructions

Figure 23 compares two different translations for a conditional instruction. The left graphic shows a simple conditional instruction (block 1 and block 2), which does not change the program flow. Here, the successor of the conditional instruction (block 3) will be executed independently of whether the conditional instruction will be executed.

The right picture in Figure 23 shows a conditional branch instruction. Depending on whether the branch is executed or not, the conditional instruction has a different successor (either block 3 or block 5 in the graphic). In this situation, the translator has the option to include either both or only one of the successors within the translated trace. The ARM translator uses runtime profiling information to take that decision. The successor that is more likely to be executed is translated first and the basic blocks within the trace are ordered to speed up the execution of that successor. If the trace has not grown over a threshold when one successor has been translated, then the less likely successor instruction is inserted into the trace as well. If no runtime profiling information is present, the translator assumes that an instruction is likely to be skipped.

4.4.4 Condition code handling

Emulation of processor flags is one of the most expensive operations during binary translation [Smith2005, p. 71]. Therefore, the ARM backend supports a pluggable condition code handling architecture, which allows the easy implementation and evaluation of different algorithms. Two different algorithms have been implemented: *lazy evaluation* and *immediate evaluation* of condition codes.

Lazy evaluation of condition codes only calculates the value of a flag, when it is actually read. For each instruction that modifies the flags, code is created which copies

the instruction's operands into special *laziness registers*. The flags themselves are not evaluated. When a flag is read, the translator inserts code that reads the stored operands and creates the requested flag value from them. With each translated instruction, the translator stores a *lazy state*, which describes which flags are valid or invalid and which kind of operation stored its operands within the laziness registers. The lazy state is used to insert the correct code when required to resolve the flag values from the laziness registers.

Flag laziness lowers the runtime overhead, when many flags have to be updated with each instruction. This is convenient for architectures like the X86, which has six flags that are *updated by nearly every instruction with each update requiring various masks of the EFLAGS register* [Burcham2005, p. 41]. However, lazy evaluation also leads to multiple translations of the same source code with different lazy states. If a conditional instruction changes the lazy state, its successor instruction has to be translated twice – once for the situation in which the conditional had been skipped, and once if it had been executed. This leads to a similar situation as depicted on the right side of Figure 23, where blocks 3 and 5 would be separate translations of the same source instruction with different lazy states.

The ARM architecture defines four arithmetical flags, which are only updated if an instruction specifically requests so. Therefore, another approach to flag management is to evaluate flags immediately, thus saving the work of translation code pieces multiple times.

Instead of reserving three HIR registers for flag laziness[61], *immediate evaluation* uses four Boolean registers, with each register containing the state of one flag. Each flag is updated immediately when an instruction changes it. At first glance, this solution might seem more inefficient than using flag laziness. However, it provides several advantages:

---

[61] Because some ARM instructions do not set all condition codes, up to three lazy registers are required to hold all operands that are necessary to resolve all flag values.

1. **Use of dead code elimination**

   As each flag is stored within a separate register, the JRVM optimizing compiler can perform dead code elimination on a single-flag basis. This means that only flags that are actually going to be read will be evaluated.

2. **Use of fewer registers**

   Lazy evaluation requires three lazy registers and up to four Boolean registers to hold flags that have already been evaluated. In contrast, immediate evaluation only needs four registers to hold the actual flag values. Even when the evaluated flags are stored within a bit mask, lazy evaluation would still need four registers (three lazy registers and one for the bitmask), while immediate evaluation would need only one register.

3. **Fast condition code production**

   It takes up to three HIR instructions to fill the laziness registers with the operands. Additionally, further instructions are necessary to resolve the lazy state, once a flag needs to be read. Immediate flag evaluation can evaluate most flags using a single HIR instruction, while saving the three instructions that fill the laziness registers.

4. **Avoidance of multiple translations**

   With flag laziness, it is very common to have multiple translations of code sections. In particular, loops usually have to be translated twice, leading to a larger translation overhead. The flag handling in the ARM backend makes multiple translation of a single piece of code for different lazy states unnecessary.

From the previous reasons, it is not obvious which condition code handling algorithm leads to better runtime performance. Therefore, both systems have been implemented in the ARM backend. An analysis of their performance can be found in chapter 5.2.2.

4.5 The ARM Interpreter

Pearcolator provides general interfaces, `Interpreter` and `Interpreter.Instruction`, for the implementation of an interpreter. These interfaces and their cooperation with different Execution Controllers are described in chapter 3.6. The ARM interpreter is implemented in the spirit of these interfaces and following the aims for the general backend model, which are described in chapter 4.1.

As with the translator, the interpreter uses the ARM decoder to transfer a binary instruction into an instruction representation. It supplies a custom abstract factory to the decoder, which creates special instruction representations that implement the `Interpreter.Instruction` interface. The interpreter then performs its emulation based upon those instruction representations. As within the translator, this allows a single interpreter to be used for ARM and Thumb instructions. Furthermore, the interpreter works independently of the ARM translator, making it the one of the first Java-based, open-source ARM emulators[62].

The Pearcolator `Interpreter.instruction` interfaces assume that a predecoded representation of an instruction can be obtained from the interpreter, possibly to save the instruction for future reuse. The ARM interpreter leverages that concept by pre-calculating fixed values and offsets, where possible, when the pre-decoded representation is created. For example, the ARM architecture contains several types of block data transfer instructions, which can all be expressed as a single block transfer instruction by using different offsets from the start and end address. The ARM interpreter pre-calculates these offsets when a pre-decoded representation is created, thereby avoiding that calculation when the same instruction is executed again.

4.6 The ARM Disassembler

The ARM disassembler converts an ARM instruction into a human-readable string representation. Using a visitor pattern, the ARM disassembler can be applied to any instruction representation created by the ARM decoder. If a binary instruction is to be decoded, the disassembler invokes the decoder first to create an appropriate instruction representation and then transfers this representation into a string. The disassembler is a debugging component, which is used within the ARM interpreter and translator, as well as indirectly by the generic Pearcolator components.

---

[62] At the time of writing, no other open source ARM emulator written in Java could be discovered.

4.7 Operating System Emulation

The ARM backend supports the generic Linux emulation provided by Pearcolator and can emulate the platform specific ARM Debug Monitor “Angel”, which is supplied with ARM Development Boards and therefore commonly used.

4.7.1 Linux Operating System Support

Pearcolator provides generic support for emulating a Linux 2.6 environment. However, different Application Binary Interfaces (ABI) are defined for different platform-ports of the same Linux version. In order to hide a platform’s ABI details, the backend has to implement the `LinuxSystemCallGenerator` interface. The interface provides generic ways of accessing system call numbers, arguments and return values.

A complete documentation of the ARM ABI can be found in [LeeSmith2005]. It defines many important issues for compilers, such as structure layouts and data type sizes. There are two fundamentally different ABI definitions for the ARM architecture: the newer Embedded ABI (EABI) and the previous legacy ABI. The Pearcolator backend provides support for both ABIs.

In order to implement the `LinuxSystemCallGenerator` interface, details about a platform’s system call conventions are most important. Table 7 exemplifies the differences in the system call conventions between the two ARM ABIs. While the legacy ABI used to put the system call number into the `SWI` (Software Interrupt) instruction, the new EABI puts it into register r7. This yields better performance, as the OS is not forced to re-read a `SWI` instruction to extract the system call number. Furthermore, 64-bit parameters, which do not fit into a single register, were just split up into the next available register pair in the legacy ABI. The EABI further demands that such a register pair starts with an even register number, possibly leaving registers unused. Both ABIs return values in register r0.

	Embedded ABI location	**Legacy ABI location**
System call instruction	SWI #0	SWI #SYSCALL_NO
System call number	Register r7	Part of SWI command
Parameter #1	Register r0	Register r0
Parameter #2	Registers r1-r2	Registers r2-r3
Return value	Registers r0	Registers r0

Table 7 – Differences between the embedded and legacy ABI for a system call of the form: int func(int, long)

The ARM backend provides two adapters[63], which implement the two ABIs. Information in the ELF binary is used to distinguish between both formats and instantiate the correct adapter.

4.7.2 Angel Debug Monitor Support

Angel is an ARM debug monitor, which is supplied with ARM Development Boards. Using this debug monitor, a program can be transparently executed on an emulator, a development board with a serial connection to a debug host or independently on the target platform. Using a technique called Semihosting, Angel allows functions that are not available on the specific ARM platform to be executed on the host. Commonly, this includes features such as user input and output or access to files on the host.

Angel Debug Monitor 1.2 provides system calls that perform file and console I/O, provide access to timers and to the execution environment[64]. The Angel Debug ABI has a few simple conventions:

- The software interrupts `SWI 0x123456` (from ARM code) or `SWI 0xab` (from Thumb code) trigger a system call.
- The system call number is provided in register r0.
- A single parameter can be provided in register r1. Depending on the system call, this is either an argument value or the address of a memory structure, which contains all required arguments.
- The system call's return value is put into register r0.

Using these conventions, all Angel system calls as documented in [STD1998, p. 13-77ff] have been implemented.

---

[63] The Adapter Pattern is a structural software design pattern. It converts *the interface of a class into another interface, [which] clients expect* [Gamma1994, p. 139ff].

[64] This includes accessing the program's command line, heap size and exiting the program.

5 Evaluation of the ARM backend

The ARM backend was evaluated for functionality as well as performance. The functionality evaluation tests the compliance of the implementation with the ARM architecture specification, while the performance evaluation analyzes the speed of the backend.

5.1 Functionality Evaluation

The functionality of the ARM backend was constantly evaluated during the development. This was done using a set of regressions tests, which were extended whenever emulation for new ARM instructions was implemented. Constantly testing a growing set of functionality lead to a fast development cycle and ensured the quality of the final implementation. Table 8 provides an overview of the different programs that were used as regressions tests and of the areas that they tested.

Platform	**Program**	**Tested Area**
Linux	Linux Logo	• Linux system calls • /proc file system access • libc support
Linux	Hello World (dynamically linked)	• Linux system calls • libc support • Dynamic linking
Semihosting	Dhrystone Benchmark (ARM & Thumb version)	• Performance testing • ARM & Thumb ISA
Semihosting	nbench Benchmark (ARM & Thumb version)	• Performance testing • ARM & Thumb ISA • File access • Floating point math
Semihosting	ARM Monitor Program	• Context switching
Semihosting	Custom Test Program	• Rarely used instructions

Table 8 – Regressions Tests during the development of the ARM backend

5.2 Performance Evaluation

This section evaluates the performance of the new Pearcolator components and the ARM backend. The evaluations are performed using the following configuration:

- Intel Pentium 4 HT processor, clocked at 3 GHz with 1 MB Level 2 Cache
- 512 MB RAM
- SUSE Linux 10, Kernel version 2.6.18.2-34 (for Pearcolator)

- Windows XP, Service Pack 2, Build 2600.xpsp_sp2_qfe.070227-2300 (for the ARM RealView Developer Suite)

Unless stated otherwise, the Dhrystone benchmark is used in the following benchmarks. It has been compiled to an ARM program from version 2.1 of the C benchmark sources using the compiler included in the ARM RealView Developer Suite v. 2.2 at optimisation level 3 with inlining disabled. Performance in the Dhrystone benchmark is measured in Dhrystones/s with higher numbers meaning a better performance.

It is well understood that the Dhrystone benchmark is affected heavily by compiler quality [York2002]. Nevertheless, it is suitable for benchmarking Pearcolator, because it is only compiled once to a binary and is not used to compare different architectures. Furthermore, the Dhrystone benchmark has also been used to evaluate the X86 and Pearcolator backends [Burcham2005] [Matley2004]. Therefore, it seems appropriate to use the same benchmark to enable comparability. For any presented value, the benchmark has been run three times and the results were averaged.

5.2.1 Execution Controllers

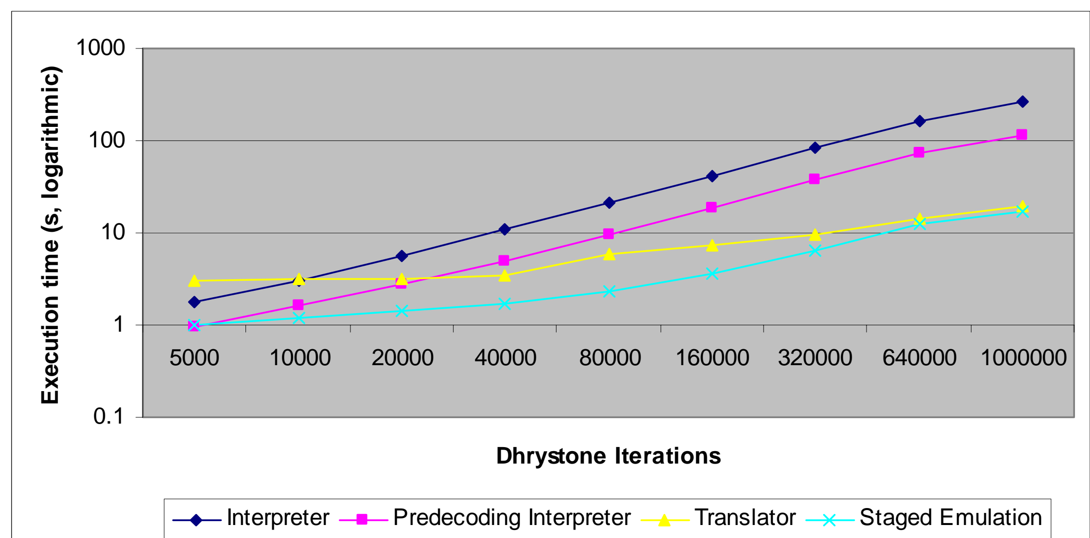


Figure 24 – Execution time of Dhrystone for different Execution Controllers and different number of Dhrystone loops (ARM 32-bit code)

Chapter 3.5 described how the new Pearcolator architecture supports execution controllers to perform different emulation strategies. Figure 24 gives an overview of the execution time that different execution controllers need to run a varying number of Dhrystone iterations.

The *naïve interpreter* is almost always the slowest execution strategy. For large number of Dhrystone iterations, it is about fourteen times slower than the translator. Only at less than ten thousand Dhrystone iterations does its speed become comparable to that of the translator. At 5,000 Dhrystone iterations it is about twice as fast as the translator.

Using the *predecoding interpreter* strategy improves upon the previous interpreter results. It executes the Dhrystone benchmark more than twice as fast as the naïve interpreter for most measured Dhrystone iterations. Compared to the translator, it is three times faster at its best performances and only six times slower at its worst.

The *translator* provides satisfactory performance when a piece of code is executed multiple times. Therefore, its performance quickly increases with the number of Dhrystone iterations. Though it starts as the worst-performing execution strategy at five thousand Dhrystone iterations, it outperforms the naïve interpreter at ten thousand Dhrystone iterations and the predecoding interpreter at twenty thousand iterations. In the long run, it provides the best steady-state performance of all execution controllers.

The *Staged Emulation* execution controller bridges the performance gap between the predecoding interpreter and the translator. It starts off by executing instructions using the predecoding interpreter. However, it keeps track of how often a dynamic basic block is invoked. Once a block has executed more than ten thousand instructions, the block is compiled using the translator. Future invocations of that block then execute the binary translated version. Staged emulation leads to a smooth transition between predecoding interpretation and translation. This makes it the fastest execution controller in all test cases, setting a new Performance mark for Pearcolator.

The threshold of ten thousand instructions has been validated experimentally. Figure 25 visualizes the performance implications of running the Dhrystone benchmark using Staged Emulation with different thresholds and for different numbers of Dhrystone iterations.

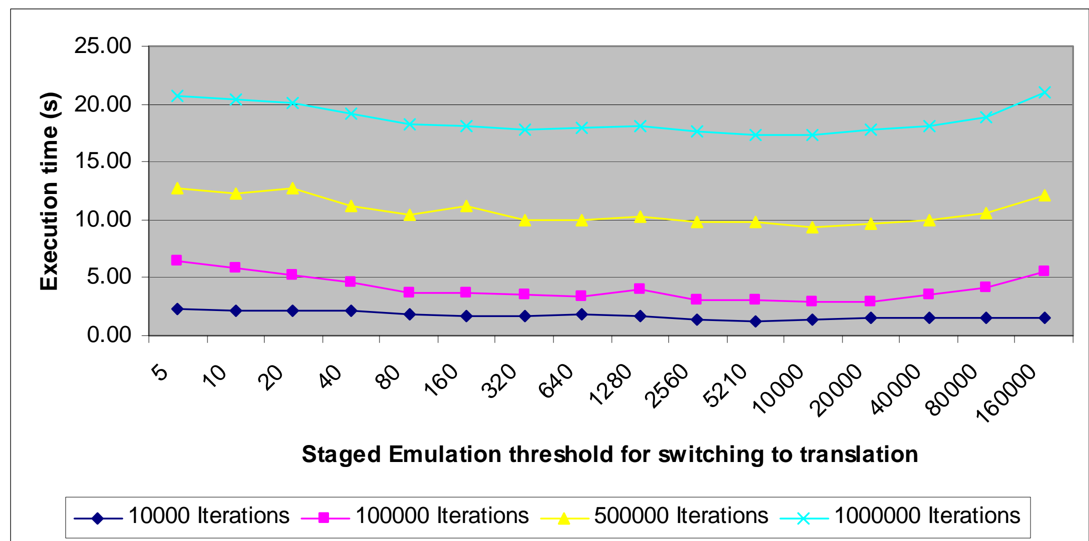


Figure 25 – Execution time for different number of Dhrystone iterations and various Staged Emulation thresholds.

5.2.2 Condition Code Evaluation

Though the X86 and PowerPC Pearcolator backends were both developed to use flag laziness, neither of the authors tested its effectiveness separately. [Matley2004 p. 67f] only documents performance increases in his second Pearcolator implementation that he also attributes to flag laziness – however, that version also included many other improvements, such as the introduction of traces. The ARM backend is the only Pearcolator backend that supports both, lazy evaluation and immediate evaluation of condition codes. Therefore, it is in a good position to analyze the differences between both implementations.

Figure 26 compares the performance implications of using lazy evaluation and immediate evaluation of condition codes. The diagram shows the relative speed of immediate evaluation compared to the performance of lazy evaluation in ARM 32-bit and Thumb code. As stated in chapter 2.4.1, ARM 32-bit code does not set the condition codes with each data processing instruction, but only when their evaluation is actually required. In contrast, Thumb code always sets the condition code with every data processing operation.

Surprisingly, the ARM 32-bit code using immediate evaluation of condition codes is mostly faster than its lazy evaluation equivalent. The reason is that the C compiler already decided whether the resulting condition codes of an instruction are actually needed and generated appropriate code. Therefore, lazy evaluation only adds

compilation overhead in Pearcolator, without offering a benefit. For less Dhrystone iterations, the overhead of compiling the same piece of code with different lazy states dominates the relative speed. In this situation, immediate evaluation is about 10% faster than lazy evaluation. This advantage fades as the number of Dhrystone iterations increases, because code regions can be reused and do not have to be compiled again. After about 500,000 Dhrystone iterations a steady state is reached, in which immediate evaluation and lazy evaluation perform similarly.

In contrast, Thumb code seems to have a greater benefit from lazy evaluation. It overcomes the initial overhead for compiling code with different lazy states at the same number of Dhrystone iterations as the ARM 32-bit code, but performs about 5% faster than immediate evaluation afterwards.

As a result of this investigation, Thumb code will always use lazy evaluation while ARM 32-bit code will default to immediate evaluation, which delivers superior performance for short-running programs.

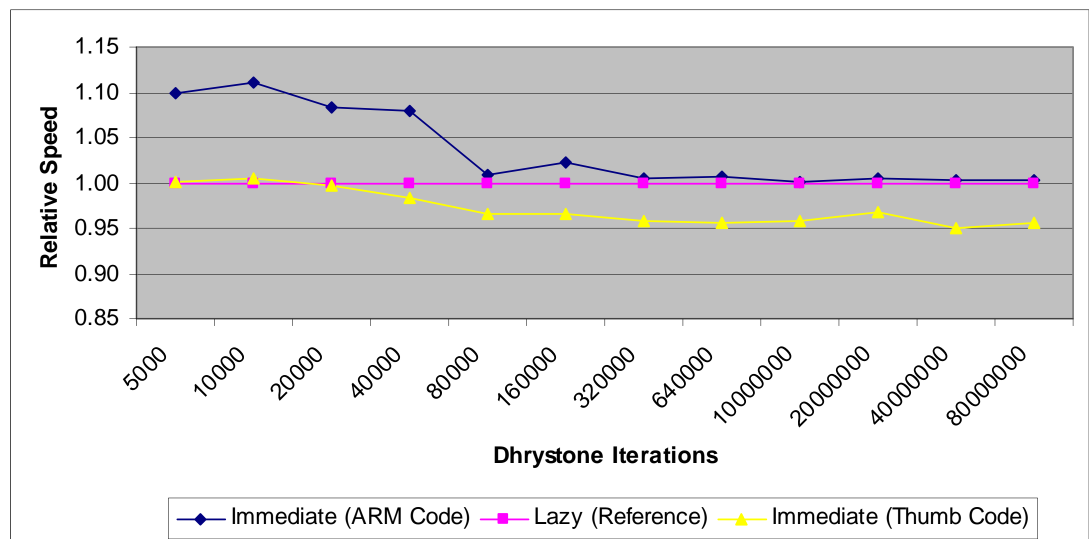


Figure 26 – Relative execution speed of lazy and immediate evaluation of condition codes for different numbers of Dhrystone iterations

5.2.3 Memory Model

Chapter 3.9 introduced the various exchangeable memory models supported by Pearcolator. The Dhrystone benchmark was run with one million iterations to benchmark their performance, measuring both the performance including compilation time and the steady state performance, excluding compilation time. The latter was

measured by configuring the benchmark to run five times, discarding the best and worst result and averaging over the remaining three runs.

Figure 27 compares the performance of the integer and byte-based memory. For each memory model, the access was implemented using either a function call to the `Memory` interface or inlining that call into every place were a memory access took place.

Overall, the integer based memory model performs about 50% to 60% better than the byte based memory. This is not surprising, since ARM has a 32-bit word length and therefore usually reads 32-bit at a time. With the byte based memory, this operation translates into four individual reads.

However, inlining memory access incurs a major performance hit. Especially the compilation time increases dramatically, as inlined memory accesses create significantly more HIR code than just a single call. In some cases, this even forced the test machine to start swapping during the compilation. The steady state performance for inlined memory accesses approaches the speed of non-inlined accesses. The remaining slight variation could not be attributed clearly. It might result from slightly worse usage of the instruction cache due to the increased size of the translated code.

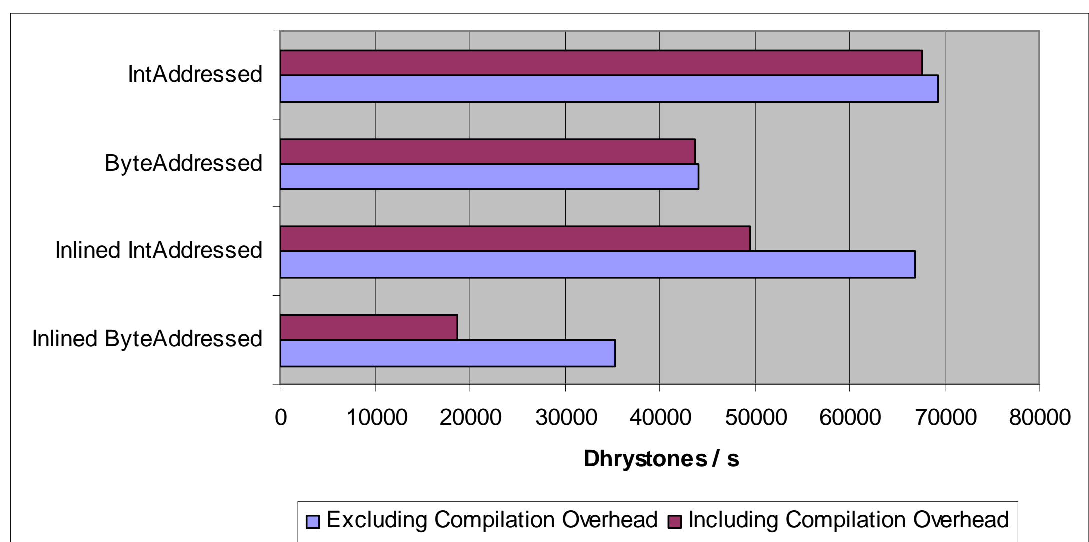


Figure 27 – Translator performance for different memory models (Dhrystone Benchmark, one million iterations, ARM 32-bit code)

5.2.4 Inlining Options

The new Pearcolator model provides an easy way of defining different strategies to decide whether a branch should be inlined into a Pearcolator trace or compiled into a

separate trace. This flexibility allows a general evaluation of the cost-performance benefit of using inlining during binary translation with Pearcolator. Obviously, the benefit of inlining is highly program specific. To achieve optimal performance, a custom inlining strategy may have to be implemented for each running program.

Figure 28 shows the effect of inlining different types of branches during the execution of the Dhrystone benchmark using ARM 32-bit code. The graph shows that the inlining of function calls and forward branches seems to have the greatest benefit for the benchmark. Generally, inlining as much code as possible into a single trace seems to yield the best performance. This is obviously influenced by the nature of the Dhrystone benchmark, which iterates frequently over a rather small piece of code. It can also be observed that all inlining methods have a similar compilation overhead / speedup ratio.

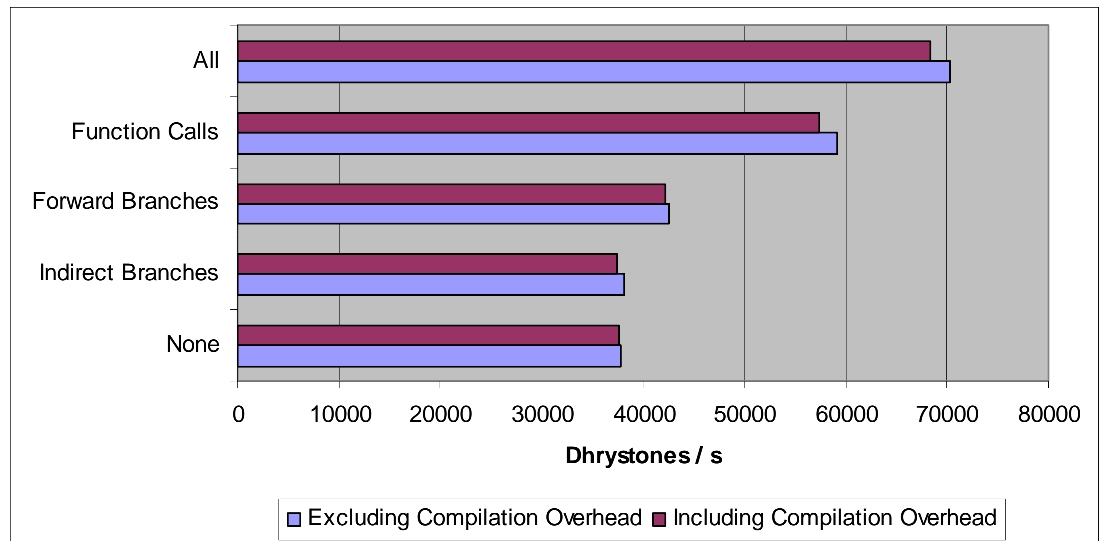


Figure 28 – Influence of inlining different types of branches into a Pearcolator trace (Dhrystone Benchmark, one million Iterations, ARM 32-bit code, Int-based memory)

5.2.5 Profiling and Indirect Jump Prediction

Inlining function calls yields a significant speedup in Pearcolator. Figure 29 shows the benefit of enabling different inlining techniques for Thumb code. When comparing this figure with the ARM 32-bit performance in Figure 28, it becomes obvious that, while the ARM code receives a major speedup for inlining of function calls, the Thumb code does not benefit from this optimisation at all. Looking at the Thumb Assembly code, the reason for this problem becomes apparent: Thumb performs all function calls as indirect branches. Because the respective branch target has to be known at compile time, simple function inlining cannot be performed in Thumb code.

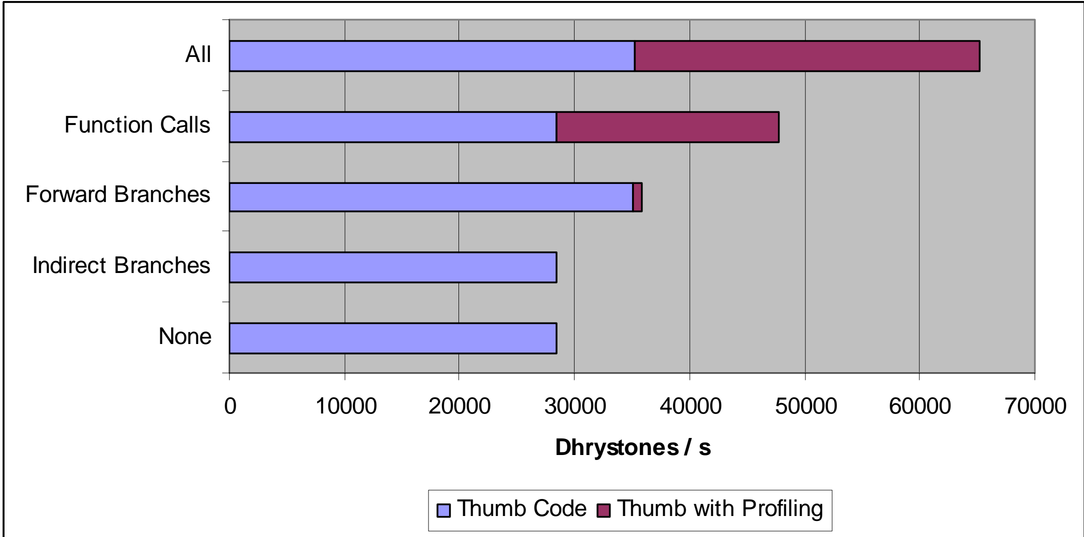


Figure 29 – Effect of profiling for different inlining techniques (Thumb code, Dhrystone Benchmark, one million Iterations)

The Pearcolator profiling system can remedy this disadvantage using indirect jump prediction. Any execution controller that performs interpretation (i.e. the interpreter controller, the predecoding interpreter and the staged emulation controller) also builds a profile of branches and their likelihood within the running application. The translator can resort to this information to perform indirect branch prediction and to optimize branches in HIR in general. As shown in Figure 29, profiling results in a performance increase of up to 85% for Thumb code.

ARM code also benefits from profiling, though to a lesser extend than Thumb. Figure 30 highlights that up to 33% of Dhrystone performance are gained by profiling.

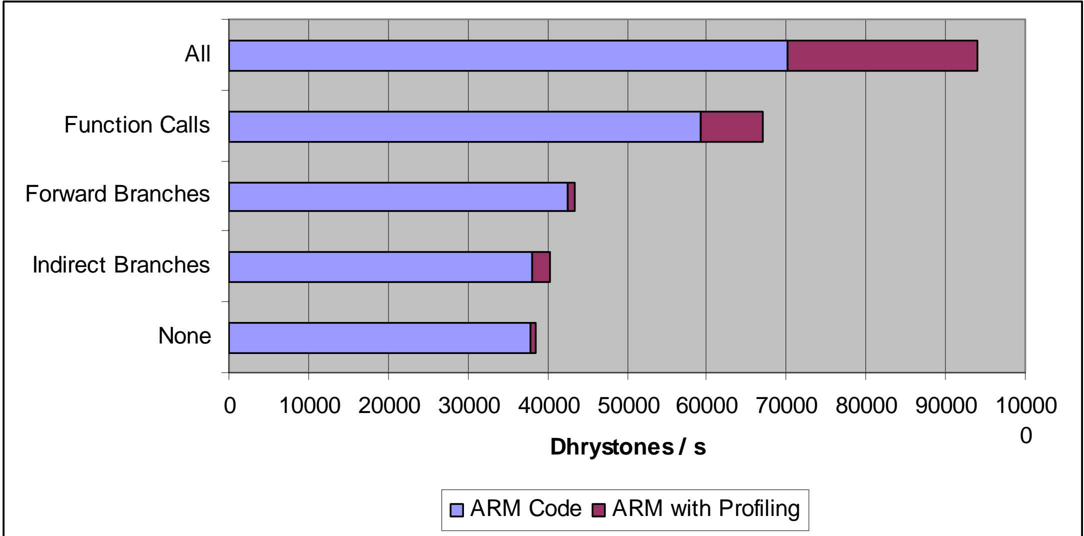


Figure 30 - Effect of profiling for different inlining techniques (ARM 32-bit code, Dhrystone Benchmark, one million iterations)

5.2.6 Overall Emulator Performance

Figure 31 compares Pearcolator's performance with that of the emulator included in the RealView developer suite and with the speed of native execution on an Intel XScale IOP80321 processor clocked at 600 MHz. The RealView developer suite can capture exhaustive performance data on an ARM processor as well as simulate a memory management unit. For the fairness of comparison, these features have been deactivated as Pearcolator does not perform full system emulation yet.

The graphic shows that Pearcolator is up to five times faster than the RealView developer suite. In the emulator as well as in Pearcolator, ARM code performs faster than Thumb code. However, the performance difference between both instruction sets is slightly less in Pearcolator than on the RealView emulator. Nevertheless, Pearcolator is still ten times slower than native execution on the XScale processor. Although it is difficult to compare different architectures, this hints at the fact that optimal emulation performance on Pearcolator is still to be obtained. Although the staged emulation system is a step into that direction, it does not yield any performance gains in a steady state situation, where Pearcolator's peak performance is measured.

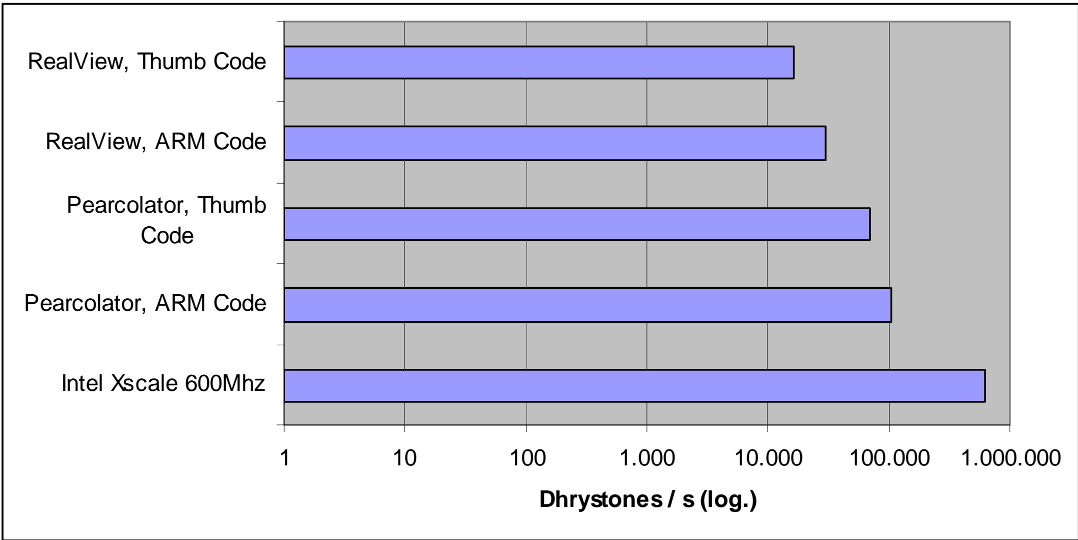


Figure 31 – Comparision of Pearcolator performance with a commercial emulator and native execution

6 Conclusion

6.1 Conclusion

In the course of this thesis, the dynamic binary translator Pearcolator was reengineered, significantly enhanced and equipped with an ARM backend. Regarding the goals that were stated in chapter 1.3, the following observations could be made:

Regarding goal 1: Pearcolator was redesigned with a modular component architecture, which is based upon the generic architecture of a process virtual machine. The new design promotes more software reuse within the binary translator, while making it easy to enhance it or exchange single components. The diversity of the implementations for execution controllers, memory, loaders and operating system emulator clearly shows the flexibility of the architecture. Furthermore, Pearcolator was enhanced with helpful features, such as support for dynamic linking.

Regarding goal 2: Generic support for interpreters was added to Pearcolator. Building upon the interpreter support, components for naïve interpretation, predecoding interpretation and staged emulation were developed and evaluated. Each of these components can perform program profiling, without the backend having to offer dedicated support for it. During the evaluation, staged emulation was shown to significantly enhance Pearcolator's performance.

Regarding goal 3: Pearcolator was enhanced with an ARM backend. The backend, consisting of a decoder, an interpreter, a disassembler and a translator, has been developed with a carefully chosen architecture, which even allows its components to be used independently of the Jikes Research Virtual Machine. The resulting software is the first open source ARM emulator implemented in Java. It supports both, the ARM and Thumb instruction sets, executes ARM Linux as well as semihosted Angel programs and has a performance that compares favourably with a commercial ARM emulator.

Regarding goal 4: The combination of all new components was used to evaluate open questions regarding Pearcolator's performance. By taking advantage of the modular architecture, the speedup delivered by different components could be quantified. Especially the benefit of using lazy evaluation could be analyzed, thus allowing a more intelligent decision of when to use lazy or immediate evaluation in the ARM backend.

In summary, the thesis satisfied the originally stated goals and provided valuable insights into the design of a dynamic binary translator. Furthermore, it put Pearcolator into a good position for future research.

6.2 Future Work

Though Pearcolator's performance has improved, it is still far from the performance of the emulated hardware. Therefore, Pearcolator's performance should be improved even further to extend its practical usability.

Because Pearcolator is based on the Jikes Research Virtual Machine, it would obviously benefit from any performance improvements to the latter. Especially the simulation of machines with many registers on architectures with less registers is not optimally served by the JRVM's linear scan register allocator. The implementation and evaluation of a graph coloring register allocator is a worthwhile goal, which would benefit both the JRVM as well as Pearcolator.

It was further shown that lazy evaluation does not provide a significant speedup in Pearcolator. To improve upon that, alternative approaches should be explored. As suggested in chapter 2.1.3, using flags that are already set by the host processor instead of calculating all flags explicitly, might potentially yield better performance.

The evaluation also discovered that Pearcolator's memory implementation has a significant impact on the overall program performance. Therefore, implementing tailored memory models seems like a promising performance source. Using low-level accesses, as done in the JRVM with `VM_Magic`, might lead to a significant speed up for memory interactions.

On the functionality side, performing full system emulation is a new goal that might be pursued. This includes the emulation of additional hardware and interrupts. Though the current ARM backend can interpret simple context-switching code, no such simulation is performed yet. Furthermore, the implementation of a more complete operating system emulation and the addition of different backends offer opportunities to enhance Pearcolator's functionality.

7 References

[Allison2002]
Andrew Allison (2002)
Merchant Market RISC Shipments in 2001
http://www.aallison.com/RISC2001.pdf
Downloaded on 23rd July 2007

[Alpern1999]
Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, Vivek Sarkar (1999)
Jalapeño - A Compiler-Supported Java Virtual Machine for Servers
Workshop on Compiler Support for Software System (WCSSS 99)
Atlanta, GA, May 1999, held in conjunction with PLDI 99
Also available online at http://citeseer.ist.psu.edu/alpern99jalapentildeo.html
Checked on 23rd July 2007

[Alpern2000]
Bowen Alpern et. al. (2000)
The Jikes Research Virtual Machine / The Jalapeno Virtual Machine
IBM Systems Journal, Vol. 39, No. 1, p. 211ff
Order No. G321-0137
Also available online at http://www.research.ibm.com/journal/sj/391/alpern.html
Checked on 23rd July 2007

[Altman2000]
Erik R. Altman, David Kaeli, Yaron Sheffer (2000)
Welcome to the Opportunities of Binary Translation
Computer, Volume 33, Issue 3, pages 40-45
ISSN: 0018-9162
Also available online at http://citeseer.ist.psu.edu/altman00welcome.html
Checked on 23rd July 2007

[ARM2000]
ARM Limited (2000)
ARM Architecture, Reference Manual
Document Number: ARM DDI 0100E
http://www.arm.com/community/university/eulaarmarm.html
Downloaded on 30th July 2007

[ARM Website]
ARM Limited
The ARM Instruction Set Architecture
http://www.arm.com/products/CPUs/architecture.html
Downloaded on 23rd July 2007

[Bala1999]
Vasanth Bala, Evelyn Duesterwald; Sanjeev Banerjia (1999)
Transparent Dynamic Optimization
HP Labs Technical Report (June 1999), HP Laboratories Cambridge
HPL-1999-77
http://www.hpl.hp.com/techreports/1999/HPL-1999-77.html
Downloaded on 23rd July 2007

[Brewer2003]
Shane A. Brewer (2003)
Jikes Intermediate Code Representation
Presentation at the University of Alberta in the course 'Advanced Topics in Compilers: Dynamic Re-Compilation' (CMPUT 605-JIT) in Winter 2003
http://www.cs.ualberta.ca/~amaral/courses/605-jit/jikesIR.ppt
Downloaded on 23rd July 2007

[Burcham2005]
Jonathan Kenneth William Burcham (2005)
An X86 emulator written using Java
MSc Thesis
University of Manchester, Faculty of Engineering and Physical Sciences

[Clements1991]
Alan Clements (1991)
The Principles of Computer Hardware, 2nd Edition
Oxford University Press, Oxford
ISBN 0-19-853765-4

[Computer Desktop Encyclopedia: Code Density]
Computer Language Company Inc. (2007)
Part of the Computer Desktop Encyclopedia
http://computing-dictionary.thefreedictionary.com/code+density
Downloaded on 23rd July 2007

[Computer Desktop Encyclopedia: Disassembler]
Computer Language Company Inc. (2007)
Part of the Computer Desktop Encyclopedia
http://computing-dictionary.thefreedictionary.com/disassembler
Downloaded on 23rd July 2007

[Dandamudi2005]
Sivarama Dandamudi (2005)
Guide to RISC Processors: For Programmers and Engineers
Springer, Berlin
ISBN 978-0-387-21017-9

[Decker1995]
Karsten M. Decker et al. (1995)
Technology Overview: A Report on Data Mining
Technical Report 95-02
Swiss Scientific Computing Centre, CSCS-ETH
Also Available Online http://citeseer.ist.psu.edu/73088.html
Last Checked on 22nd September 2007

[Earnshaw2006]
Richard Earnshaw (2006)
Procedure Call Standard for the ARM Architecture
ARM Limited
Document Number: GENC-003534
http://www.arm.com/pdfs/aapcs.pdf
Downloaded on 30th July 2007

[Frey2003]
Brad Frey, Ed Silha, Cathy May, Joe Wetzel (2003)
PowerPC User Instruction Set Architecture, Book 1, Version 2.01
IBM Corporation
http://www.ibm.com/developerworks/eserver/articles/archguide.html
Downloaded on 30th July 2007

[Furber2000]
Steve Furber (2000)
ARM System-on-Chip Architecture, 2nd edition
Addison-Wesley Longman, Amsterdam
ISBN 0-201-67519-6

[Gamma1994]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994)
Design Patterns: Elements of Reusable Object-Oriented Software
Addison Wesley Longman, Inc.
ISBN 0-201-63361-2

[Gupta2002]
Rajiv Gupta, Arvind Krishnaswamy (2002)
Profile Guided Selection of ARM and Thumb Instructions
Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems, pages 56 - 64
ISSN 0362-1340

[Haungs1998]
Michael L. Haungs (1998)
The Executable and Linkable Format (ELF)
http://www.cs.ucdavis.edu/~haungs/paper/node10.html
Last Updated on 21st September 1998
Last Checked on 22nd September 2007

[Huffman1997]
Larry Huffman, Wendy Ferguson (1997)
MIPSPro™ Assembly Language Programmer's Guide
Silicon Graphics, Inc.
Document Number: 007-2418-003
http://techpubs.sgi.com/library/manuals/2000/007-2418-003/pdf/007-2418-003.pdf
Downloaded on 23rd July 2007

[HP1997]
Hewlett Packard (1997)
HP-UX Linker and Libraries User's Guide (Technical Documentation)
Document Number: B2355-90655
http://www.docs.hp.com/en/B2355-90655/B2355-90655.pdf
Downloaded on 30th July 2007

[LeeSmith2005]
Lee Smith (2005)
Base Platform ABI for the ARM Architecture
ARM Limited
Document Number: GENC-005700 v2.0
http://www.arm.com/pdfs/bpabi.pdf
Downloaded on 30th July 2007

[Levine2000]
John R. Levine (2000)
Linkers and Loaders
Morgan Kaufmann, San Francisco
ISBN 1-55860-496-0

[Matley2004]
Richard George Matley (2004)
Native Code Execution Within a JVM
MSc Thesis
University of Manchester, Faculty of Science and Engineering

[Muchnick1997]
Steven S. Muchick (1997)
Advanced Compiler Design Implementation
Morgan Kaufmann, San Francisco
ISBN 1-55860-320-4

[Nymeyer1997]
A. Nymeyer, J.-P. Katoen (1997)
Code ceneration based on formal BURS theory and heuristic search
Acta Informatica, Volume 34, No. 8, p. 597 - 635
Also available online at http://citeseer.ist.psu.edu/206090.html
Checked on 23rd July 2007

[SCO2003]
The Santa Cruz Operation, Inc. (SCO)
System V Application Binary Interface, 17th December 2003
http://www.sco.com/developers/gabi/latest/contents.html
Downloaded on 23rd July 2007

[Smith2005]
Jim Smith, Ravi Nair (2005)
Virtual Machines, Versatile Platforms for Systems and Processes
Morgan Kaufmann, San Francisco
ISBN 1-558-60910-5

[STD1998]
ARM Limited (1998)
ARM Software Development Toolkit, Version 2.5
Document Number: ARM DUI 0040D
http://www.arm.com/pdfs/sdt250usrman.pdf
Downloaded on 30th July 2007

[Sun1999]
Tim Lindholm, Frank Yellin (1999)
Sun Microsystems, Inc.
The Java™ Virtual Machine Specification (2nd Edition)
Addison-Wesley Longman, Amsterdam
ISBN 0-201-43294-3
Also available online at http://java.sun.com/docs/books/jvms
Checked on 23rd July 2007

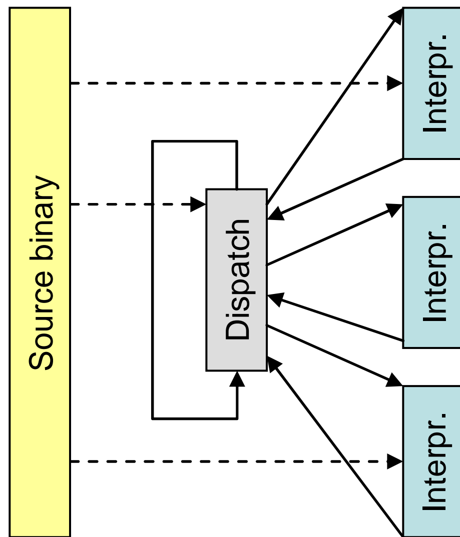
[Woolf1996]
Bobby Woolf (1996)
The Null Object Pattern
PLoP '96, University of Illinois
http://citeseer.ist.psu.edu/160174.html
Downloaded on 30th July 2007

[York2002]
Richard York (2002)
Benchmarking in context: Dhrystone
ARM Limited
http://www.arm.com/pdfs/Dhrystone.pdf
Downloaded on 16th August 2007

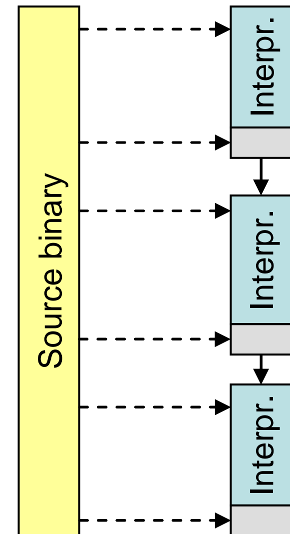
[Zilog2001]
ZiLOG Inc. (2001)
Z80 Family, CPU User Manual
Document Number: UM008001-1000
http://www.zilog.com/docs/z80/z80cpu_um.pdf
Downloaded on 30th July 2007

8 Appendix

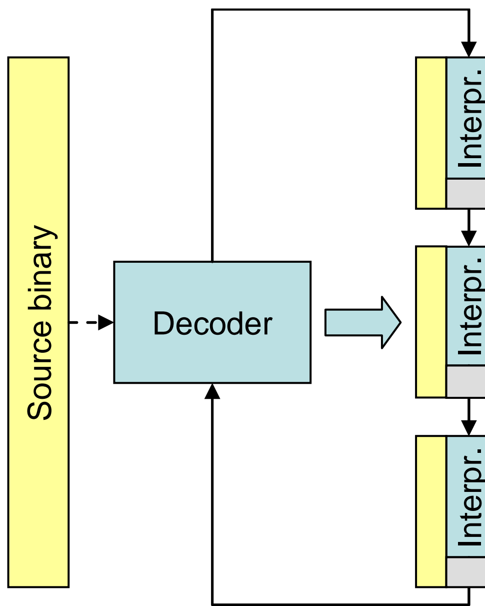
8.1 Appendix A



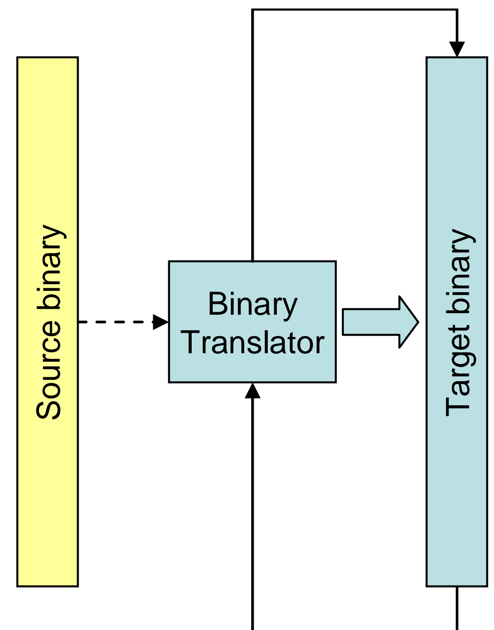
Decode-And-Dispatch Interpretation



Threaded Interpretation



Predecoded, Threaded Interpretation



Binary Translation

Table 9 – Overview of different emulator types

8.2 Appendix B

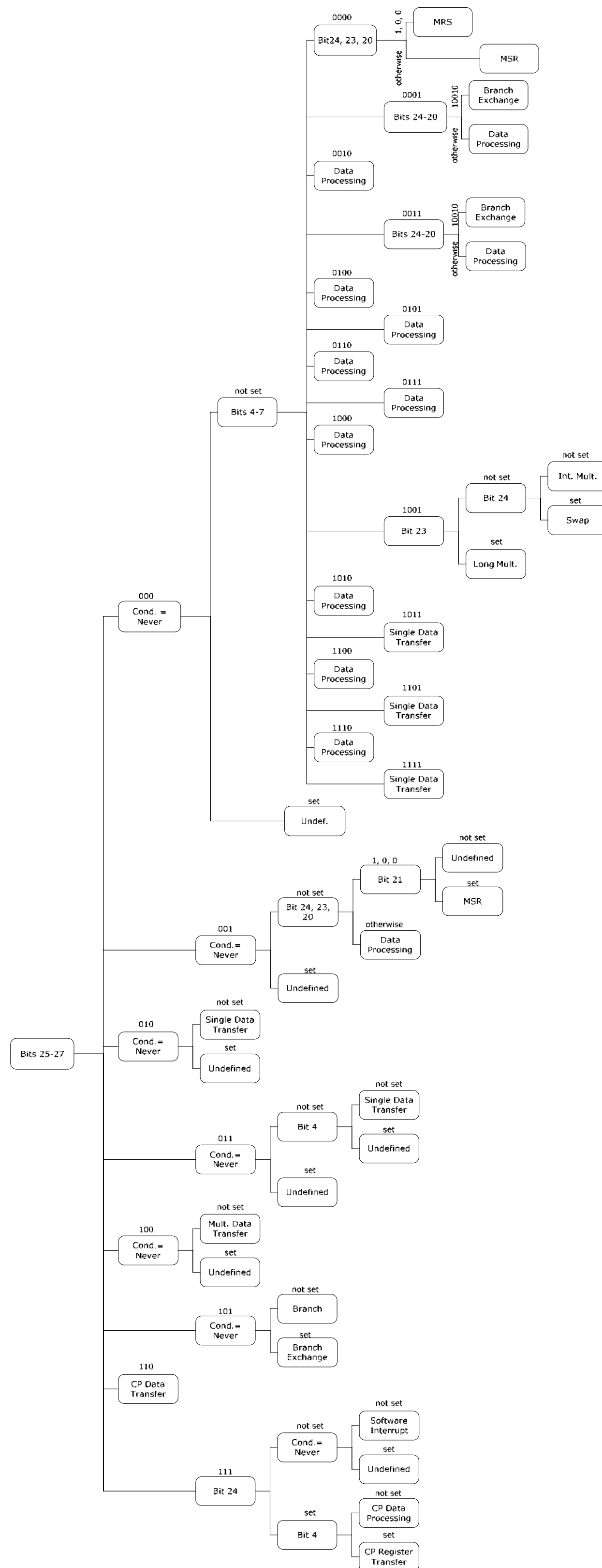


Figure 32 – ARM Decoder Decision Tree

8.3 Appendix C

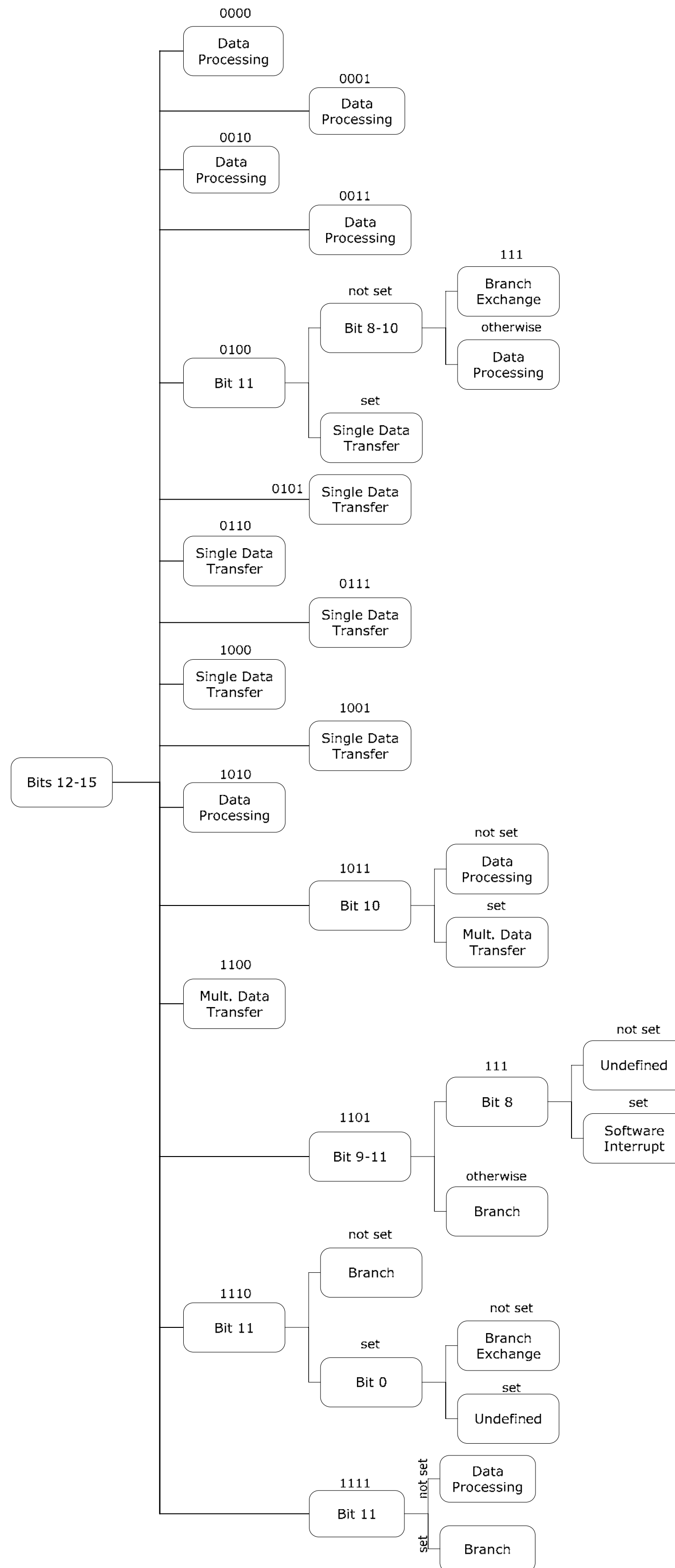


Figure 33 – Thumb Decoder decision tree