Native Code Execution Within a JVM

A thesis submitted to the University of Manchester for the degree of Master of Science in the Faculty of Science and Engineering.

2004

Richard George Matley

Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

A project is presented which has developed an emulator, PearColator, to execute programs compiled for the PowerPC instruction set architecture [1]. It is written in Java and runs as a component of the IBM Jikes Research Virtual Machine. This enables it to take advantage of the sophisticated optimising compiler and adaptive compilation architecture of the RVM.

This work was carried out as part of the Jamaica project which is designing a chip-multiprocessor architecture. The long term goal of PearColator is to use the parallel compilation and execution capabilities of the Jikes RVM (on which work is also being done by the Jamaica group) to offer much better execution of legacy programs on this type of architecture than current emulators which are better suited to current single processor technology.

The performance of PearColator is evaluated and compared with other PowerPC emulators and with execution of a program on actual PowerPC hardware. PearColator was used to run a benchmark on a 2.16 GHz AMD AthlonXP processor and the same benchmark was run on a 600 MHz G3 PowerPC processor. PearColator was found to be approximately seven times slower.

The PearColator project is ongoing, so the last chapter discusses the future work to be done.

Declaration

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

Copyright

Acknowledgements

To my supervisor Prof. Ian Watson and to the indispensable Dr. Ian Rogers. To all the people who have worked on the Jikes RVM without which this project would never have happened.

To my family for putting up with me spending another year as a student.

This dissertation was produced under SuSE Linux using LaTeX, xfig, and GLE[1]; as such I am indebted to Donald Knuth, Leslie Lamport, Linus Torvalds, the GNU project and many others.

[1]Graphics Layout Engine, *http://glx.sourceforge.net*.

The Author

Having graduated from the University of Manchester with a BSc in Physics I then joined the Nonlinear and Condensed Matter Physics Group and gained my PhD researching convection in liquid helium. After this I decided on a change of direction and began the MSc Computer Science course. The latter part of this has been the work described in this dissertation, undertaken as part of the Jamaica Project in the Advanced Processor Technologies Group.

And now that I've introduced myself, I should like to have some idea of what's going on.

Major General Stanley; The Pirates of Penzance,
W. S. Gilbert

Quidquid latine dictum sit, altum viditur.
(Whatever is said in Latin seems profound.)

Anonymous

1 Emulation and Dynamic Binary Translation

1.1 Introduction

An *emulator* is a system that allows a computer to execute programs which are in a machine code native to a different platform.[1] Commonly this is an *interpreter*, which analyses a program one instruction at a time and immediately executes the required action. This is potentially very slow, particularly in that one instruction may be executed many times during the course of a program and must be interpreted each time.

An alternative approach is *binary translation* which translates machine code from one instruction set architecture (ISA) to another. This new machine code is natively compatible with the hardware on which it is to be executed, and can be run repeatedly, if necessary, with no further translation needed.

There are two types of binary translation: *static binary translation* translates a whole binary file into a different ISA; *dynamic binary translation* translates 'on the fly', translating only those instructions which are actually needed in the flow of the program[2]. The process of static binary translation has problems of incompleteness due to issues of self-modifying code, dynamic linking, and discovery of code to be translated. Static binary translators usually require some human intervention during translation or some model which attempts to describe how compilers generate code from the original high level language. The dynamic approach overcomes this, but there is a trade-off between the time spent translating and the time spent executing the translated code. A better translation may reduce code execution time, but the extra time spent translating should not be allowed to outweigh this benefit [2].

[1] i.e. in an instruction set architecture other than that of the computer on which they are running

[2] Or at least making a sensible attempt to avoid translating code which will not be executed.

1.2 Dynamic Binary Translators

1.2.1 Terminology

The architecture on which a binary executable program was compiled to run is called the *subject machine*. The dynamic binary translator presents a *subject environment*. This may replicate the hardware of a raw subject machine directly, or provide an operating system environment with linking/loading and emulation of system calls.

The dynamic binary translator itself runs within the *target environment*. A program executed in the subject environment is called a *subject program*. The DBT translates *subject code* into *target code* [3].

Note that this use of the terms subject and target is not universal, sometimes the terms client and host are used, while target can be used in the sense that subject is used here.[3]

1.2.2 Types of Dynamic Binary Translators

As is illustrated in figure 1.1, there are multiple possibilities for the relationship between the translator and the operating system(s) present.

In the first type, there is an operating system running in the usual way on top of the hardware, and the dynamic binary translator runs as a process on top of the OS. This requires the DBT to provide a subject environment which replicates the operating system which the running program expects, not necessarily the same as that which is actually present on the hardware. System calls must be passed through to the operating system or emulated in some suitable way.

In the second, the DBT is executed directly by the hardware and provides a subject

[3]For example in reference 4.

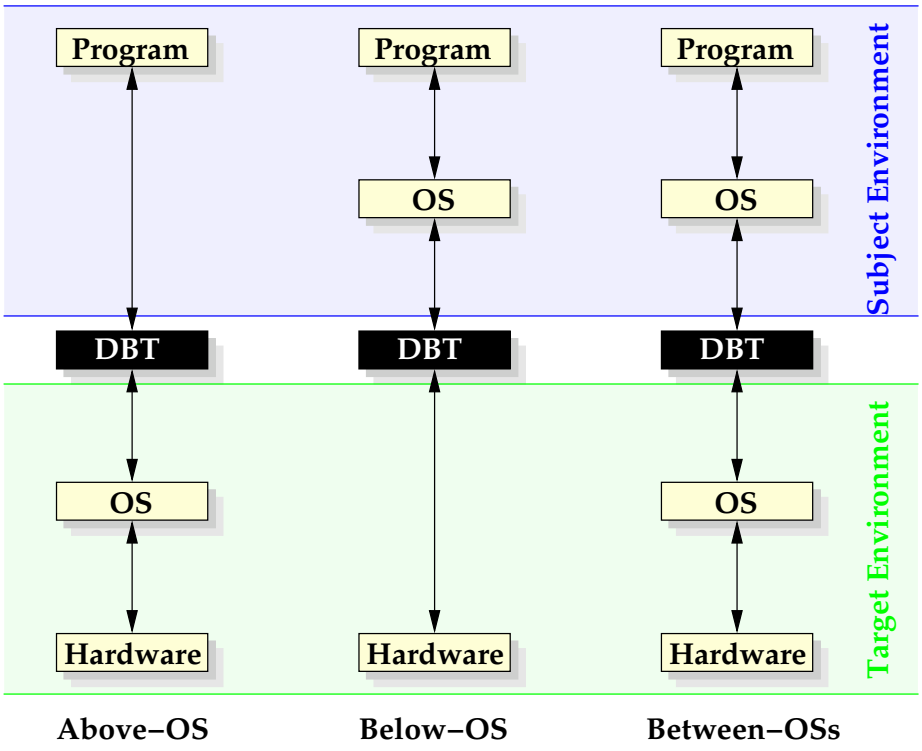

Figure 1.1: *Three different types of dynamic binary translators.*

environment which replicates different hardware. An operating system is run on top of the DBT, and is itself translated by it. This approach is commonly used by translators built into the hardware. Both AMD Athlon and Intel Pentium processors translate IA-32 instructions into simpler instruction sets used internally [5,6].

The third configuration has one operating system running on the hardware, a DBT executing as a process within it, and a second operating system running in translation on the DBT. This second OS is then responsible for running application programs [3].

Possibly the best known dynamic binary translator is FX!32, which was produced by Digital Equipment Corporation [7]. It is an above-OS DBT which allows MS Windows programs for the Intel IA-32 architecture to run under Windows NT on Digital's Alpha architecture, as an aid to users migrating from the Intel to the Alpha platform.

The approach taken is that the first time a program is executed using FX!32, it is interpreted. This is slow but allows a profile to be built of the sections of code

which have been emulated. After the user exits from the program, the translation is performed as a background task, translating only the code which was actually executed during the interpreted phase, or which can be reached by simple branches. In future executions of the same program, the stored translator output code is used, falling back into interpretation if an untranslated area of code is entered. These areas are then translated after execution has finished. Since only 32-bit Windows applications are supported, FX!32 can trap system calls and they are executed by the underlying Windows NT/Alpha operating system.

1.3 This Project in Context

1.3.1 The Need for Another Emulator

Many dynamic binary translators already exist (FX!32 has been mentioned and a few others will be discussed shortly), so the question is raised: why write another one?

The answer lies in the particular needs of projects such as Jamaica and in the use of the IBM Jikes Java Research Virtual Machine (formerly Jalapeño) as the basis of the translation [8,9].

1.3.2 Jamaica

The Jamaica project exists to develop a new chip-multiprocessor architecture with hardware support for light-weight threads, and the necessary software environment for its use [10]. Conventional dynamic binary translators do not support the parallel execution needed to take advantage of this type of architecture. Other work within the research group is investigating advanced techniques of parallelisation within the Jikes RVM. An emulator which runs as part of the RVM will be able to take full advantage of this, and the existing optimisation work which has been

incorporated into the RVM.

1.3.3 The Jikes RVM as the Basis for an Emulator

The Jikes RVM is designed as a testbed for virtual machine technology. It was originally written to execute Java byte-code by dynamically translating (rather than interpreting) into machine code. It includes an optimising compiler which performs many stages of optimisation to give high performance.

PearColator adapts the Jikes RVM to translate not byte-code but rather machine code in an ISA not native to the hardware platform on which the RVM is running. This allows a dynamic binary translator to be written which takes advantage of the existing and ongoing work on optimisation, particularly parallelisation. Conventional dynamic binary translators are not usually as sophisticated as virtual machines. Another advantage of the whole emulator, including its memory access, being in Java is that of improved security (such as in array access) [1].

The DBT is of the above operating system type. It creates a PowerPC/Linux subject environment. As it is based on the Jikes RVM, it can run in any of the target environments supported by the RVM. The one used to develop the project is Intel IA-32/Linux. Since the RVM can run on AIX and OS X operating systems on PowerPC hardware, PearColator could be used to execute Linux programs on these platforms.

1.3.4 Other PowerPC Emulators

PowerPC emulators currently existing (or in development) include QEMU, PearPC, and SoftPear [11–13]. Unlike the Jikes RVM based PearColator DBT, all these are written in C or C++. The performance of two of these emulators is compared with that of PearColator in chapter 5.

1.3.4.1 QEMU

QEMU is a dynamic binary translator which can emulate several CPU architectures, including PowerPC [11]. It has two modes of operation [4]. *Full system emulation* which is what is referred to as a between-OSs DBT in figure 1.1, emulating a full system of a computer and peripherals. *User mode emulation* is an above-OS type DBT allowing Linux programs compiled for one architecture to run on another.

The operating principle of its translator is to break down each subject code instruction into a few simpler instructions. Each of these is implemented in C code. The object code resulting from compiling this set of simple instructions is used to make a dynamic code generator which concatenates the simple instructions to build up a full instruction. A pass is made through the generated simple instructions to find cases where condition codes are set by an instruction but never used. In these cases the code setting is eliminated.

1.3.4.2 PearPC

The PearPC project is developing a between operating systems emulator of the PowerPC hardware running on various target environments [12]. It can be used to run an Apple operating system on Intel IA-32 hardware. It uses both an interpreter and a dynamic binary translator.

1.3.4.3 SoftPear

The specific goal of the SoftPear project is to enable programs for the Apple OS X operating system to run on Intel IA-32 hardware, rather than its intended PowerPC platform [13]. SoftPear runs on a target operating system with the OS X user interface and libraries running on top of it, to enable OS X applications to be run.

1.4 The Jikes RVM

PearColator is written as an addition to IBM's Jikes Research Virtual Machine [14, 15].

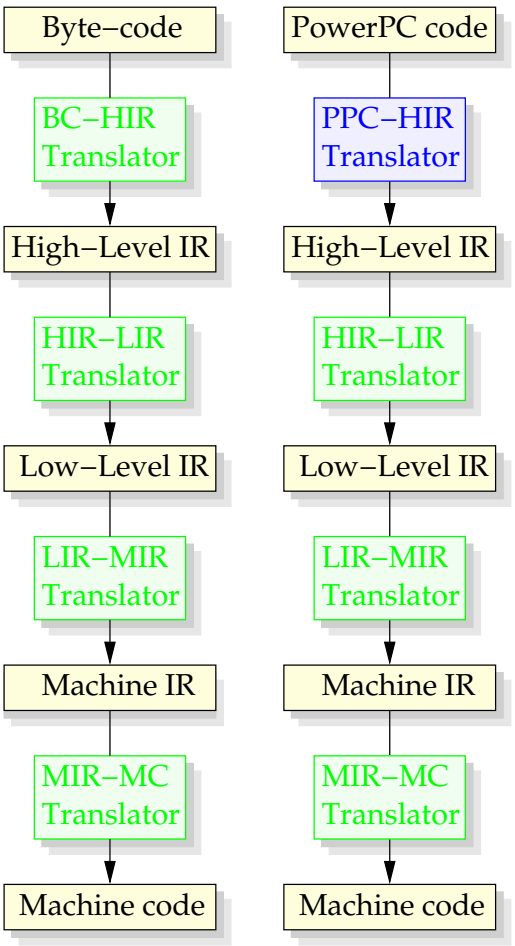

Figure 1.2: *The Jikes RVM optimising compiler translates (left) from Java byte-code into machine code in stages. PearColator replaces (right) the first translation stage.*

The RVM has two compilers. The *baseline compiler* translates from Java byte-code directly into machine code for execution. It is intended to give code which is correct, quickly generated, but not necessarily fast in execution.

PearColator uses the *optimising compiler*, which has a much more complicated translation mechanism, giving code which executes quickly. This is illustrated in figure 1.2.

Java byte-code is translated into the Jikes RVM's own *High-Level Intermediate Representation (HIR)*. This is a format which is independent of the architecture on which the RVM is executing. From here it is transformed through stages of Low-Level and Machine-Specific Intermediate Representations (LIR and MIR), becoming increasingly specific to the target architecture, finally reaching machine code. This machine code is then executed.

The figure is a great simplification of the working of the translator. Each of the stages shown can include many steps of optimisation [8, 9]. An *optimisation plan* is followed which determines which steps are to be taken to optimise the generated

code.

The PearColator approach takes advantage of all this existing technology to achieve a high performance translator. The first translation stage is replaced by one which takes PowerPC machine code as its input. The rest of the process happens exactly as it does when running byte-code.

The performance obtained using PearColator is discussed in chapter 5, where it is compared with that of other PowerPC translators.

1.5 Summary

- Emulation allows a computer to execute programs compiled to machine code for an architecture other than its own.
- There are a number of approaches to this. This project developed a dynamic binary translator, PearColator, which executes PowerPC programs.
- PearColator is written as a part of the Jikes Research Virtual Machine. This enables it to take advantage of the RVM technology currently existing or being developed, in particular techniques of parallelism with a view to use on future multiprocessor architectures.

Q: How many IBM CPUs does it take to do a logical right shift?
A: 33. 1 to hold the bits and 32 to push the register.

Q: How many IBM CPUs does it take to execute a job?
A: Four; three to hold it down, and one to rip its head off.

Unknown, from the fortune program

2 Emulating the PowerPC Architecture

2.1 PowerPC/Linux as a Subject Environment

The PowerPC/Linux platform has a number of features which make it a good choice of (initial[1]) subject environment for PearColator. The PowerPC is a load-store RISC architecture enabling the transfer of data to and from the (emulated) memory to be separated from the instructions which act upon data held in registers.

Linux executable binary files are in the ELF format which has a simple layout and is therefore straightforward to load into memory [16]. Additionally, unlike the format used by OS X, binaries can be statically linked. Consequently it has been possible to write the initial versions of PearColator to handle only statically linked binaries. It is intended that support will be added later for dynamic linking using the program **ld.so**, just as the Linux kernel does (see section 6.3).

2.2 The Architecture of the PowerPC

2.2.1 Overview

The architecture exists in 32 bit and 64 bit implementations; PearColator emulates the standard 32 bit version and the following discussion relates to this implementation.

There are three levels within the PowerPC architecture. PearColator emulates the one to which user level programs conform—*User Instruction Set Architecture (UISA)*. This defines the user-level instruction set and registers, data types, and memory and programming model [17].

The standard byte-ordering is *big-endian*, and PearColator uses this ordering. Within a value, the most significant bit is denoted bit 0.

[1]The design of PearColator is intended to make it possible to add further subject environments in the future.

Memory access instructions can operate on word (32 bit), half word or byte values. Multiple byte values do not need to be aligned to natural boundaries, unlike on some architectures including, in one mode, the POWER architecture of which PowerPC is a development.

The floating-point arithmetic follows the IEEE-754 standard for 32 bit single precision and 64 bit double precision values.

2.2.2 Register Set

The UISA architecture model includes the following registers:

- 32 *general purpose* registers (GPRs) holding 32 bit integer values,
- 32 *floating-point* registers (FPRs) holding 64 bit double precision values,
- five *special purpose* registers:
  - the *condition register* (CR),
  - the *floating-point status and control register* (FPSCR),
  - the *XER register* (XER),
  - the *link register* (LR),
  - the *count register* (CR).

General Purpose Registers These are used as the source and destination operands for integer instructions.

Floating-Point Registers These are used as the source and destination operands for floating-point instructions. They hold double precision values, and load and store instructions are provided in the instruction set to transfer double precision

Bit	*Meaning*
0	LT—set when result is negative.
1	GT—set when result is positive (non-zero).
2	EQ—set when result is zero.
3	SO—summary overflow, a copy of the SO bit of the XER at the completion of the instruction.

Table 2.1: *Condition register field CR0 values (integer instructions).*

Bit	*Meaning*
0	FX—floating-point exception.
1	FEX—floating-point enabled exception.
2	VX—floating-point invalid exception.
3	OX—floating-point overflow exception.

Table 2.2: *Condition register field CR1 values (floating-point instructions, other than compare). All bits are copies of the similarly named bits of the XER at the completion of the instruction.*

values without conversion. There are also instructions which convert a double precision value to single precision and store it as a 32 bit value in memory, or load a 32 bit value and convert to double precision.

Condition Register This holds flags indicating result conditions and is used for testing and branching. It is divided into eight fields, each of four bits (CR0–CR7, with CR0 being the first four bits of the register). When an integer instruction sets a condition code based on its result, it uses CR0, with codes as in table 2.1.

When a floating-point instruction (other than floating point compare) sets a condition code, CR1 is used, with codes as in table 2.2.

Compare instructions can set any field of the condition register, with the codes as in table 2.3.

Bit	*Meaning*
0	Less than or floating-point less than.
1	Greater than or floating-point great than.
2	Equal or floating-point equal.
3	Summary overflow (copy of SO bit of XER) or floating-point unordered (if either operand is a Not a Number).

Table 2.3: *Condition register field values set by compare instructions.*

Bit	*Meaning*
0	SO—Summary overflow, set whenever an overflow is generated, stays set until explicitly cleared.
1	OV—Overflow, set whenever an overflow occurs.
2	CA—Carry, set or cleared by instructions which specify carrying.
25–31	Byte count, number of bytes to be transfered by load/store string word instructions.

Table 2.4: *Definitions of the bits of the XER.*

Floating-Point Status and Control Register Bits 16–19 hold another copy of the condition codes, for an explanation of the meanings of the other bits see reference 17.

XER Register Few of the bits of this register are used, and these are defined in table 2.4.

Link Register This contains the target address for branch conditional to link register instructions. When a branch and link instruction causes a branch to occur, the link register is set to the address of the next instruction after the branch. This is the mechanism for entering and returning from a section of code (function/method call and return).

Count Register This can be used to hold a loop count which is decremented by the execution of a branch instruction containing a code to indicate this. It can alternatively be used to hold a branch target address.

2.2.3 The Instruction Set

The PowerPC is a load/store RISC architecture. The only instructions which access memory are load and store instructions; all others act only on the contents of registers. All instructions are of 32 bit length. An instruction can specify up to three registers, two for source operands and one as the destination. This allows a binary operation (one with two source parameters) to be performed in a single instruction. Alternatively an immediate value can be given as a source operand, encoded in the instruction.

The instruction set can be divided into the following groups of instructions [17]:

Integer These comprise arithmetic, compare, logical, rotate, and shift instructions.

Floating-Point This group includes arithmetic, compare, and move instructions.

Load/store Instructions exist to load and store integer and floating-point values from/to memory. Multiple integers can be transfered using a single instruction.

Flow control These include branch instructions and those which perform logical operations on the condition register.

Other instructions Synchronisation, memory/cache control, external device control.

2.2.4 Addressing Modes

The following addressing modes are supported for memory access and branching:

- *Register indirect*: the address is given in a register,
- *Register indirect with immediate index*: the address is the sum of a value from a register, plus an offset given as an immediate value,
- *Register indirect with index*: the address is the sum of the values from two registers.

2.2.5 Instruction Format

Each instruction is identified by a *primary opcode* represented by the first six bits of the instruction. In some cases a set of instructions have the same primary opcode and are distinguished by the *secondary (or extended) opcode* (the last eleven bits in these instructions). Each instruction can be broken down into parts, representing opcode(s), operands, and sometimes signals to update the condition and/or XER registers.

There are several different *instruction forms* which define how the various bits of the instruction should be interpreted.

For example, the **addi** (add immediate instruction) is a *D-Form* instruction (one which contains an immediate value) and its bits are interpreted as shown in figure 2.1 [17].

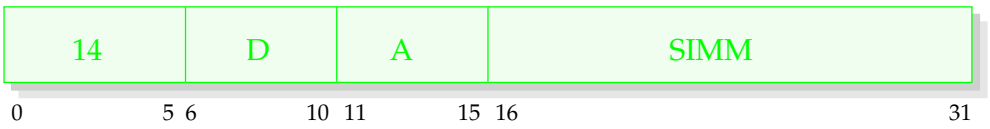

Figure 2.1: *An example D-Form instruction,* ***addi****.*

In this case, the opcode is 14, bits 6–10 give the number of the register to be used as the destination operand (rD), bits 11–15 give the number of the register to be used

as one of the source operands (rA) and bits 16–31 give an immediate value which is the other source operand. The value in rA is added to the immediate value (which is treated as a 16-bit signed value and is sign extended to give a 32-bit representation of the same number[2]); the result is stored in rD. Note that in this instruction and many others, if rA is given as 0, this is interpreted as the value zero, rather than as register r0.

A more complicated format is the *XO-Form* (X-Form instructions having three register operands, and the 'O' indicating overflow signalled in the XER register), of which an example is shown in figure 2.2. This instruction is **adde**, **adde.**, **addeo**, or **addeo.** depending on bits 21 and 31.

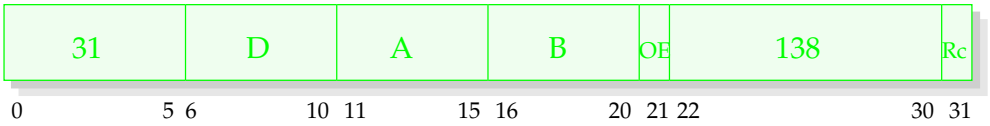

Figure 2.2: *An example XO-Form instruction,* ***adde****x.*

This has the primary opcode 31 and secondary opcodes given by bits 21–31. Here D and A have the same meanings as in the previous example; the second source operand is the register rB. If the Rc bit is set, the condition register field CR0 is updated (and the instruction mnemonic contains '**.**'). If the OE bit is set, the XER is updated (and the mnemonic contains '**o**').

The effect of this instruction is to add the contents of rA and rB, plus the carry flag (CA bit of the XER), putting the result into rD.

2.3 Emulation

Before the actual translation of PowerPC instructions can begin, it is necessary to set up a subject environment within which the subject program will run, replicating that of real PowerPC hardware running Linux.

[2] i.e. bits 0–15 of the 32-bit value are set equal to bit 16 (the sign bit of SIMM).

The main issues here are:

- A *process space* must exist covering the 4 GB of virtual memory address range accessible to a Linux process, including code, heap, and stack areas.
- Variables are used to store the values which, on a real PowerPC, would be held in its *registers*.
- *System calls* must be handled.

2.4 The Process Space Object

2.4.1 Overview

A new class has been added to the Jikes RVM called **com.ibm.JikesRVM.ppcEmulator.PPC_ProcessSpace**.

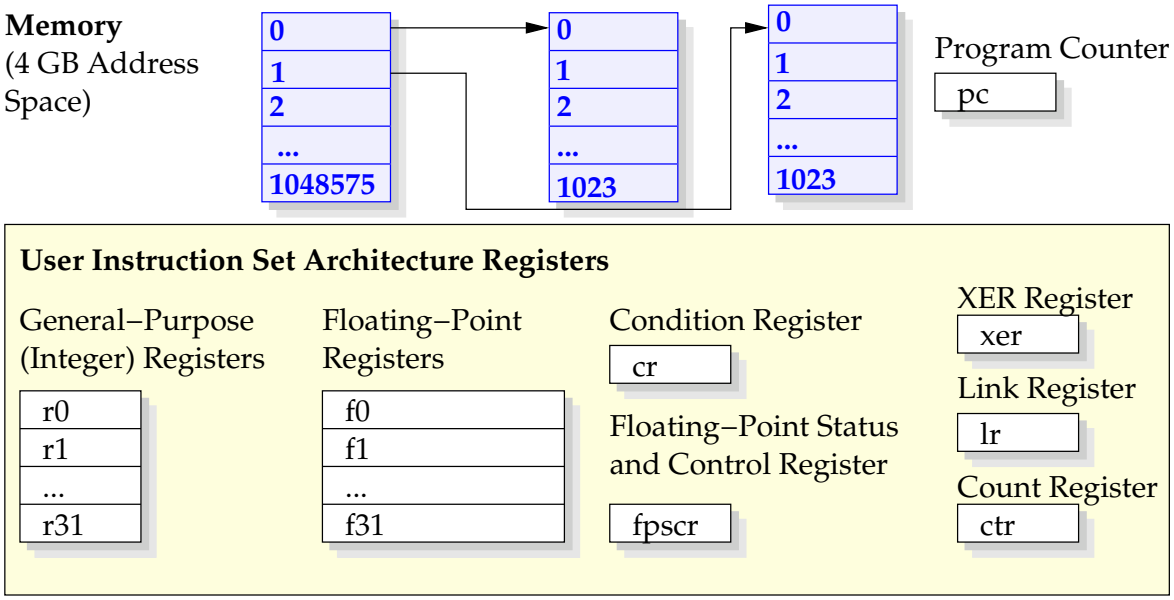

Figure 2.3: *The **PPC_ProcessSpace** class emulates the 4 GB address space in which a program executes, the program counter, and those registers used in UISA mode.*

As illustrated in figure 2.3, the memory space is emulated by an array of 2^{20} *page* arrays, each of 1024 integers; this gives the required 4 GB total. The individual page

arrays are allocated when needed using the method **PPC_ProcessSpace.mmap()** which replicates the semantics of the Linux mmap system call.

All the registers used in the User Instruction Set Architecture (UISA) model are emulated by Java integer variables, except the floating-point registers, which are of 64-bit size, so are Java type double. Although the PowerPC architecture has no program counter register, a variable is included in this class to hold the address of the current instruction.

2.4.2 Methods

In addition to the **mmap()** method, there is a corresponding **munmap()**. To support memory mapping, methods are provided to locate a suitable region of unallocated memory and to keep a record of which memory regions have been allocated (using another class **com.ibm.JikesRVM.ppcEmulator.PPC_MemoryRegion**).

Several methods are provided to read and write the contents of the memory array. The PowerPC instruction set requires word (32 bit), half word, and byte access, with the larger types not necessarily aligned to word/half word boundaries. This adds a little complexity since it is often necessary to access two consecutive integers from the array and read/write some bytes from each, leaving other bytes unchanged, resulting in misaligned access being slower than aligned access[3]

For example, suppose we wish to write the value 0x12345678 to address 0xffd. This word will be written to the four bytes at addresses 0xffd, 0xffe, 0xfff, and 0x1000. The first three of these addresses are the last three bytes of the first memory page; the other is the first byte of the second page. The process whereby these bytes are set while leaving the neighbouring bytes unchanged is shown in figure 2.4.

Each memory page contains 4096 bytes (1024 Java integers), so this spans the last word of the first page (**memory[0][1023]**) and the first of the second page

[3] As can be the case with real PowerPC hardware, and other architectures which allow misaligned access [17, 18].

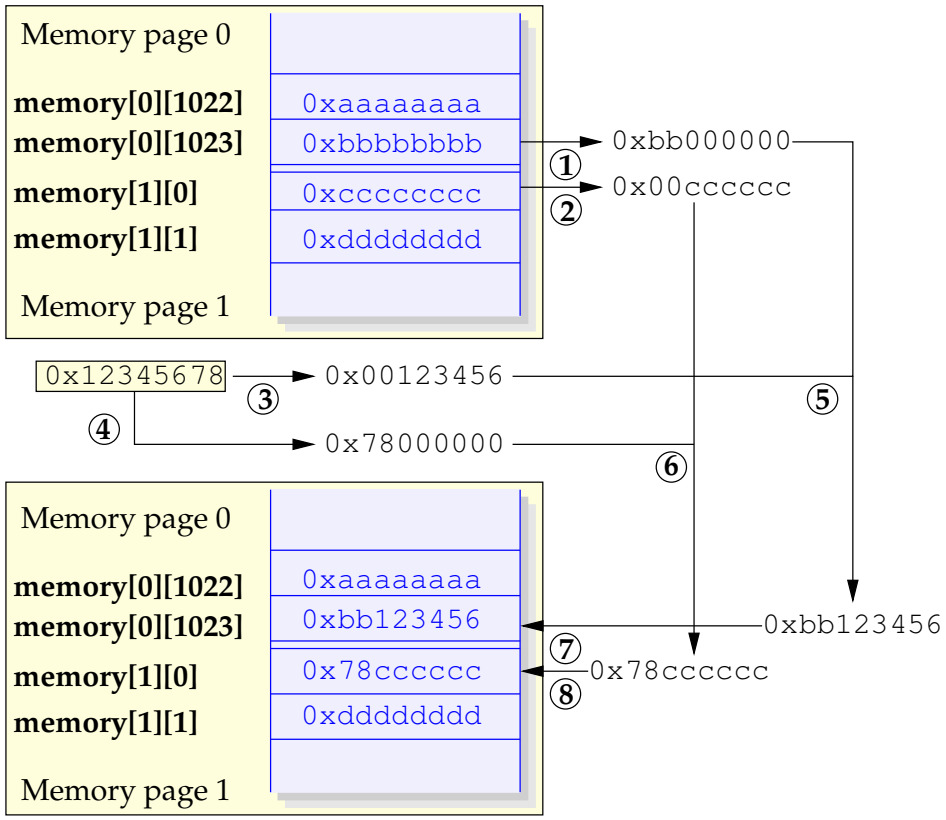

Figure 2.4: *The* ***setInt*** *method is used to write a value of 0x12345678 to the address 0xffd (4093).*

(**memory[1][0]**). The stages involved are 1,2: retrieve the two words and clear those bits to which we must write; 3: shift the word value which is to be stored right, to extract the part to be stored in the earlier word; 4: shift it left to extract the remaining part; 5,6: bitwise OR the results of stages 1 and 3, and of 2 and 4; 7,8: write back to memory. It can be seen that the word value 0x12345678 has been stored as required, leaving the neighbouring bytes unchanged.

Fortunately (in terms of the speed of the emulator) it has been found that memory access is usually aligned and most commonly in whole words, hence accessing the array of integers is simple and efficient.

It should be noted that these load/store methods in the **PPC_ProcessSpace** are not the usual way that PearColator handles memory. Usually the memory array is accessed directly from the HIR instructions generated by the translator. However, to handle misaligned access, similar algorithms are needed.

2.5 Loading an ELF Binary

2.5.1 Executable and Linkable Format

The standard binary format used for programs by Linux (and some other UNIX type operating systems) is the *Executable and Linkable Format (ELF)*[4] [16,19].

As figure 2.5 shows, an ELF file has a dual nature. One view is used by compilers, assemblers, and linkers. This regards the file as a set of sections with an ELF header at the start and a section header table (listing the sections) at the end; if a program header table is present after the ELF header, it is ignored. The second is used for execution of a program. Here the file is considered to be composed of segments, each typically corresponding several sections, with a program header table (listing the segments) following the ELF header. The section header table, if any, is ignored. At this stage PearColator runs statically compiled binaries and is interested only in the execution view.

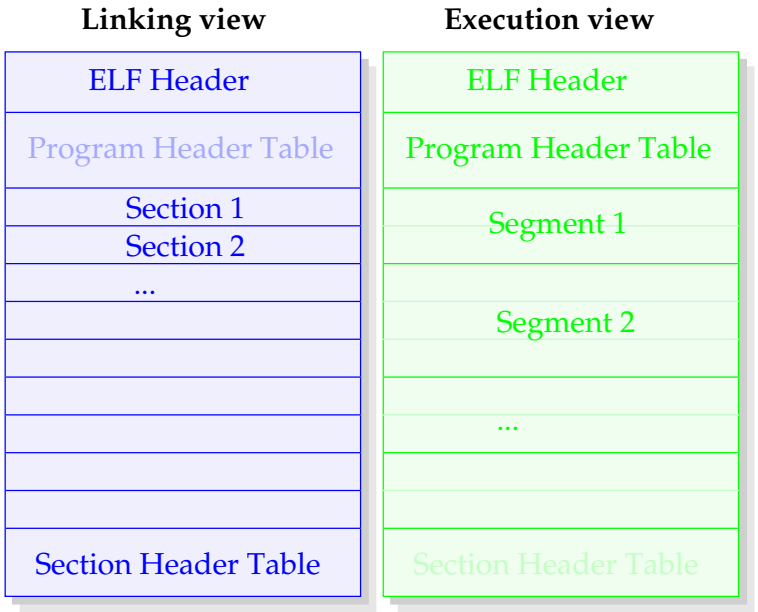

Figure 2.5: *An ELF file can be viewed in different ways for compiling/assembling/linking purposes and for execution purposes [16].*

[4] Or '... Linking ...' in some references.

2.5.2 Loading into Memory

PearColator has a class, **com.ibm.JikesRVM.ppcEmulator.PPC_ELFBinary**, which is used to read an ELF binary from a file and load it into the memory array, mapping it into the virtual address space as it would be if running natively on the intended PowerPC platform. Classes have also been written to hold the information from the ELF header and program header table.

The ELF header contains, amongst other information, the following [19]:

- The magic number identifying an ELF file.
- The address size, 32/64 bits.
- The byte order.
- The file type: relocatable, executable, shared object, core.
- The architecture type.
- The entry point, if executable.
- The positions in the file of the program header table and/or section header table.
- The size of each entry and number of entries in each of these tables.

Each ELF program header contains:

- The segment type, such as code or data.
- Its offset within the file.
- The virtual address to which to map the segment.
- The size of the segment in the file.

- Its size in memory (possibly larger than the file size, to allow space for uninitialised data).
- Read/write/execute flags.
- Required alignment.

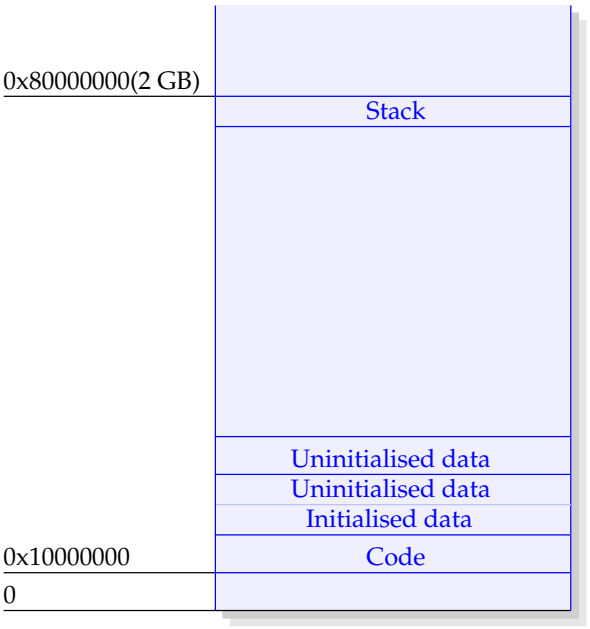

Figure 2.6: *The layout of a process in PearColator's process space.*

Using this information the ELF loader can correctly map the various segments into memory, copying code and initialised data and allocating the required space for uninitialised data.

Figure 2.6 shows the layout of the segments in memory for a simple program. There are four regions of memory used: one for the code, one for the initialised data and the beginning of the uninitialised data (up to the next memory page boundary), one for the remainder of the uninitialised data, and one for the stack.

Once this has been done, the process stack is set up, and the address of the start of the stack placed into general purpose register r1. [20–22]

Figure 2.7 shows the contents of the initial stack: it contains the command line arguments to the running program, a set of environment variables and the *auxiliary vector* [20].

Since PearColator is written as part of the Jikes RVM which runs on several operating systems, it cannot be assumed that the host OS will have suitable environment variables for use by the program executing within the emulator. For this reason a

set of values are provided by the **PPC_ELFBinary** class. As PearColator develops, facilities can be added to enable these to be customised.

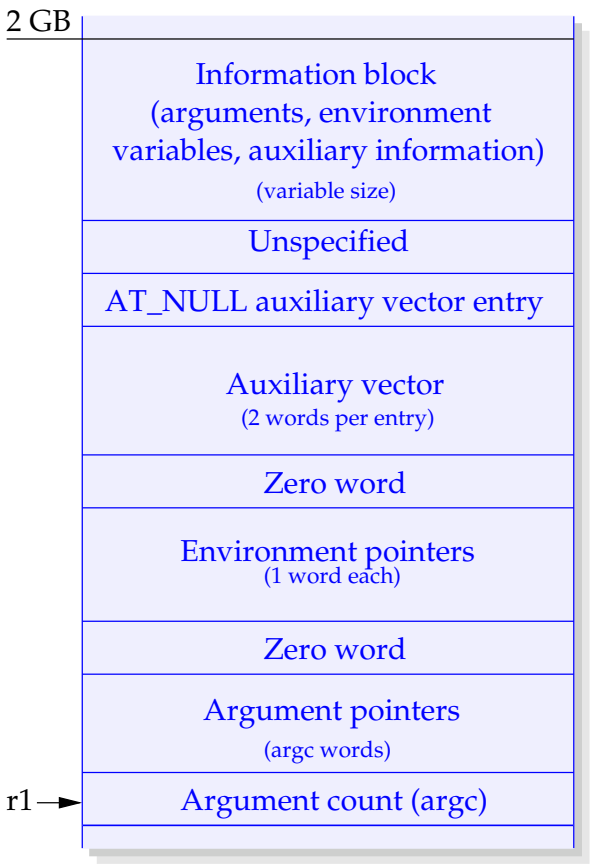

Figure 2.7: *The initial process stack, at the top of the user mode memory space [20] (i.e. the lower 2 GB of the total 4 GB).*

The auxiliary vector is a series of key-value word pairs. The meanings of the different items within the vector are given in references 20 and 21; they include the user and group ID of the user running the program, and some details of the CPU (here an emulated PowerPC) and cache; these are also given suitable values by **PPC_ELFBinary**. Also in the auxiliary vector are some values relating to where the binary is loaded into memory; these are taken from the ELF program header.

Once the process space has been created, the binary file loaded, and the stack initialised, the program is ready to be translated.

2.6 Summary

- The PowerPC platform and Linux operating system are good choices for an environment to be emulated offering a load/store architecture with constant length instructions and a binary file format which is simple to load into memory.
- General purpose, floating-point and special purpose registers have been emulated, to the level of the User Instruction Set Architecture.

- PowerPC instructions are of several forms, with up to three operands and there are three addressing modes for the operands.
- The class **PPC_ProcessSpace** provides the environment in which the PowerPC program is executed: registers, memory address space, and emulation of system calls.
- Before a program can be translated, it is loaded from file into this emulated memory, and the process stack is initialised.

For every complex problem, there is a solution that is simple, neat, and wrong.

H. L. Mencken

3 The PearColator Dynamic Binary Translator I

3.1 Introduction

This chapter discusses the PearColator DBT as it existed in the early stages of its development. The next chapter will cover the improvements that have been made and which have resulted in very substantial improvements in the performance of the translator.

3.2 Structure

Figure 3.1 demonstrates how the PearColator translator is structured and the interface between the new work and the existing Jikes RVM.

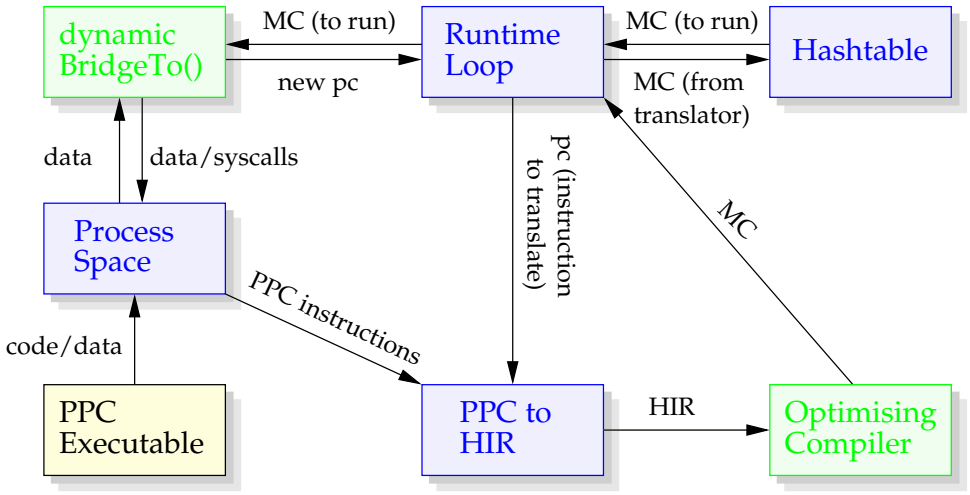

Figure 3.1: *The structure of the PearColator dynamic binary translator. The parts developed in this project are shown in blue. The green parts are sections of the existing Jikes RVM. The program being executed is shown by the yellow box. The data/code flows shown as arrows are explained in the main text.*

In addition to the **PPC_ProcessSpace** object which was described previously, the main features of PearColator are:

- A *runtime system* which controls the translation process (class **PPC_EmulatorRuntime**).

- The actual *translator* (class **PPC2IR**).
- A *Hashtable* containing blocks of machine code output from the translator.

The sequence of events involved in translating and executing a PowerPC program is (in its simplest form):

- Load the PowerPC code into the process space (see section 2.5).
- Repeat, under control of the runtime loop:
  - Search the hashtable for a block of machine code (MC) translated from the PowerPC instruction at the current program counter (pc).
  - If any is found, execute it using the method **VM_Magic.dynamicBridgeTo()**, the Jikes RVM's method of calling compiler output code, which returns the address of the next instruction to be translated/executed.
  - If not, this PowerPC instruction has not been translated yet, so:
    - Call the PowerPC translator, passing the program counter.
      - The translator reads the instruction in PowerPC machine code (PPC) from the process space memory emulation.
      - The PowerPC code is translated into Jikes RVM HIR, and passed on to the other translation and optimisation stages (standard parts of Jikes RVM).
      - The resulting (platform-native) machine code is passed back to the runtime system which stores it in the hashtable.
    - The newly translated code is executed.
    - The new program counter value indicates the next instruction.

3.3 Translating an Instruction

The PowerPC to HIR translator takes an instruction from the executable which has been loaded into the process space. From this 32 bit value it must decode the instruction and plant HIR instructions which give the same behaviour.

Figure 3.2 shows the decoding of one particular example instruction.

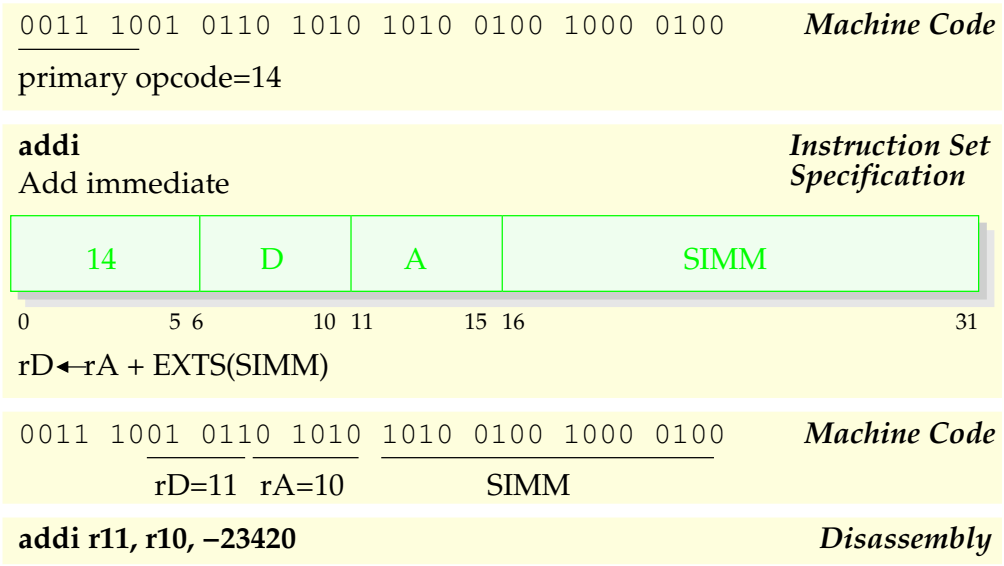

Figure 3.2: *Decoding a PowerPC instruction.*

Examination of the first six bits extracts the primary opcode value of 14, which identifies the instruction as **addi**. The format of this instruction is shown in the figure, together with the definition of the instruction. The last 16 bits of the instruction are interpreted as a signed immediate value. This is made into a 32 bit number by filling the first 16 bits with sign bits (bit 16). This is added to the contents of general purpose register rA and the result placed into register rD. In this case there is no further action; some instructions have the added complication of setting condition codes. In this example the source register is r10, the destination register is r11 and the immediate value is -23420.

3.3.1 System Calls

Some dynamic binary translators handle system calls by passing them through to the underlying operating system. This can be suitable for use in the case where the subject and target environments have the same operating system, but different architecture (although even in this case there can be differences in the layout of data structures involved which adds complication). PearColator, however, can run on a variety of operating systems, and is written in Java which, due to its platform-independence, cannot access (operating system specific) system calls directly.

The result of this is that PearColator must emulate system calls. A method is provided in the **PPC_ProcessSpace** class, called **doSysCall()**. When a PowerPC **sc** instruction is encountered, a call to **doSysCall()** is planted in the HIR. The method performs the necessary action, written in Java code.

Some of the system calls can be emulated very simply. For example, a **(**write) system call directed to standard output is mapped to **System.out.println()**. The **read** and **write** system calls applied to files are emulated by the use of Java's **RandomAccessFile** class. An array of these objects exists and the file descriptor numbers used in the programs being executed by PearColator act as array indices.

So far, just over twenty system calls have been implemented.

3.4 PearColator's Modes of Translation

In section 3.2, a sequence of events was given in which PearColator translates one instruction of a PowerPC program, executes it, then looks for a translation of the next instruction. In addition to this *single-instruction* mode of translation, PearColator has two other modes which translate a group of instructions into a block of machine code. This reduces the frequency with which control moves between the runtime system, the translator, and actual execution of translated code.

One of these translates a *block* of instructions. The translator reads in a PowerPC instruction, translates it into HIR and then does the same to the next instruction, building up a longer set of HIR instructions. When a branch instruction is reached, control is passed back to the runtime system, giving it the address of the next instruction needed (either the branch target or the next in order, in the case of a conditional branch for which the condition is not satisfied).

The last of the modes builds up a *trace* through the program. When an unconditional branch is encountered, translation continues uninterrupted with the target instruction. Conditional branches backwards to an earlier instruction within the trace are also handled without returning to the runtime system. Translation is stopped at a conditional branch to any instruction which is not earlier in the current trace. At this point control returns to the runtime system, which then searches the hashtable of code for an existing translation for the appropriate instruction.

In the next chapter, the development of PearColator beyond this initial version is described. Substantial changes have been made to the trace building procedure.

3.5 Jikes RVM High-Level Intermediate Representation

The essence of what PearColator does is to translate PowerPC machine code into the Jikes RVM's HIR, which is then translated in stages to platform-native machine code by the existing Jikes system. In this section the process of planting HIR instructions is explained in a little detail. The next section illustrates this with an example.

In the original Jikes RVM, a group of Java byte-code instructions corresponding to a Java method is translated into a group of HIR instructions. In PearColator, a single PowerPC instruction, a block of instructions, or a trace is translated. The resulting HIR is divided into *basic blocks*, each of which has a set of HIR instructions which

are to be executed in order.[1]

Any instruction which branches to another point in the code must be the last instruction in its block. Due to such branches, the order of execution of the basic blocks is not necessarily that in which they are ordered within the code. To handle this, whenever a new basic block of HIR instructions is created, two things must be specified: the position in *code order*, and the position in the *control flow graph*.

Code order is simply a linear sequence giving the order in which the basic blocks appear in a listing of the HIR. When execution reaches the end of a basic block, unless a branch is taken, it proceeds to the next block in code order.

The control flow graph reflects all the possible execution paths through the basic blocks. Each block is a node in the graph and each path from one block to any other which can be executed immediately following it is an edge. So, for example, if a block has no branch instruction at the end of it, control must pass to the next block in code order. If there is a conditional branch at the end of a block, it will have two 'out' edges, one to the next block in code order and one to the branch target.

3.6 A Very Simple Example Program

In this section an example program is presented, with the resulting translation.

Usually the programs run on PearColator are compiled from C, which links with the C library, giving a large executable. For simplicity an example written in assembly language has been selected: it does not do anything very useful, just sums the integers 1 to 5. PearColator has a debugging mode enabling register contents to be examined to show that it works properly. Figure 3.3 shows the code.

The first two instructions set registers r10 and r11 to zero. Instructions 3–6 form a loop which increments r10, adds its new value to r11 and branches back to the start of the loop as long as the value in r10 is less than 5. Instructions 7 and 8 make the

[1] Or rather, translated to machine code instructions which are then executed.

```
.text
        .globl _start

_start:
        li  10, 0       (1)
        li  11, 0       (2)
_loop:
        addi  10, 10, 1 (3)
        add  11, 11, 10 (4)
        cmpwi  10, 5    (5)
        blt  _loop      (6)

        li  0, 1        (7)
        sc              (8)
```

Figure 3.3: *Simple example program.*

system call to exit the program.

When PearColator translates this program, it can form a trace up to and including instruction 6. The following high level intermediate representation is generated:

```
-13     LABEL0   Frequency:  0.0
-2 EG  ir_prologue              l0i(Lcom/ibm/JikesRVM/
    VM_CodeArray;,d), l1i(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;,x,d) =
-1     bbend                    BB0 (ENTRY)
0     LABEL2   Frequency:  0.0
-1     int_move                 t3i(I) = 0
-1     bbend                    BB2
0     LABEL4   Frequency:  0.0
-1     int_move                 t4i(I) = 0
-1     bbend                    BB4
0     LABEL5   Frequency:  0.0
-1     int_add                  t3i(I) = t3i(I), 1
-1     bbend                    BB5
0     LABEL6   Frequency:  0.0
-1     int_add                  t4i(I) = t4i(I), t3i(I)
```

```
-1     bbend                    BB6
0     LABEL7   Frequency:  0.0
-1     getfield                 t6i(I) = l1i(Lcom/ibm/
    JikesRVM/opt/PPCProcessSpace;), -408, <mem loc:
    Lcom/ibm/JikesRVM/opt/PPCProcessSpace;.xer>,
    <TRUEGUARD>
-1     int_ushr                 t5i(I) = t6i(I), 31
-1     int_cond_move            t8i(I) = t3i(I), 5, <, 8, 4
-1     int_cond_move            t7i(I) = t3i(I), 5, ==,
    2, t8i(I)
-1     int_or                   t9i(I) = t5i(I), t7i(I)
-1     getfield                 t11i(I) = l1i(Lcom/ibm/
    JikesRVM/opt/PPCProcessSpace;), -144, <mem loc:
    Lcom/ibm/JikesRVM/opt/PPCProcessSpace;.cr>, <TRUEGUARD>
-1     int_and                  t10i(I) = t11i(I), 268435455
-1     int_shl                  t12i(I) = t9i(I), 28
-1     int_or                   t11i(I) = t10i(I), t12i(I)
-1     bbend                    BB7
0     LABEL8   Frequency:  0.0
-1     int_move                 t16i(I) = 1
-1     int_and                  t14i(I) = t11i(I), -2147483648
-1     int_ushr                 t15i(I) = t14i(I), 31
-1     int_cond_move            t17i(I) = t15i(I), 1, ==, 1, 0
-1     int_add                  t18i(I) = t16i(I), t17i(I)
-1     putfield                 t3i(I), l1i(Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;), -56, <mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.r10>, <TRUEGUARD>
-1     putfield                 t4i(I), l1i(Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;), -60, <mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.r11>, <TRUEGUARD>
-1     putfield                 t11i(I), l1i(Lcom/ibm/
```

```
    JikesRVM/opt/PPCProcessSpace;), -144, <mem loc:
    Lcom/ibm/JikesRVM/opt/PPCProcessSpace;.cr>, <TRUEGUARD>
-1     putfield                t6i(I), l1i(Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;), -408, <mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.xer>, <TRUEGUARD>
-1     int_move                t2i(I) = 268435596
-1     int_ifcmp               t19v(GUARD) = t18i(I), 2, ==,
    LABEL5, Probability: 0.99
-1     bbend                   BB8
0     LABEL9   Frequency:  0.0
-1     bbend                   BB9
0     LABEL3   Frequency:  0.0
-1     goto                    LABEL1
-1     bbend                   BB3
-14     LABEL1   Frequency:  0.0
-3     return                  t2i(I)
-1     bbend                   BB1
```

Figure 3.4: *Translation of part of the example program into HIR.*

It can be seen that there are ten basic blocks, each beginning `LABEL`*n* and ending `bbend BB`*n*.[2] Basic blocks 2 and 4–8 correspond to instructions 1–6 in the PowerPC program (figure 3.3). Terms in the HIR like `t3i(I)` are temporary representations of register values (in this example an integer). The first four blocks contain HIR very similar to the PowerPC instructions. Block 7 sets the (temporary copy of the) condition register according to the result of the comparison of register r10 with the value 5. Block 8 examines the condition register for the code representing less than, the required condition. The penultimate instruction in this block sets the temporary register `t2i(I)` to the address of the next instruction (number 7 in figure 3.3),

[2]The numbering of the blocks reflects the order in which they were created, starting with the prologue and epilogue, this does not need to correspond to code order.

which will be executed if the condition for branching is found to be false. The final HIR instruction in this block causes a branch back to basic block 5, subject to the result of the condition register testing done earlier in the block.

When the translation of the trace is executed, backward branching occurs at this point as long as the condition is satisfied, without returning control to the runtime loop. When it is no longer true the remaining basic blocks of the trace are executed. Blocks 9 and 3 are effectively empty (and so are eliminated during optimisation), while the last block returns control to the runtime loop, passing the address of the next instruction.

In basic block 8 there are several **putfield** instructions; these write the values from the temporary registers back into the **PPC_ProcessSpace** object, in preparation for returning to the runtime loop (in case the condition for the backwards branch is not satisfied).

Figure 3.5 illustrates the control flow graph for this trace.

It can be seen that there is a sequential path through the basic blocks in code order, but also a backward branch from BB8 to BB5. The actual flow of control following BB8 is determined by the (translation of the) **blt** instruction.

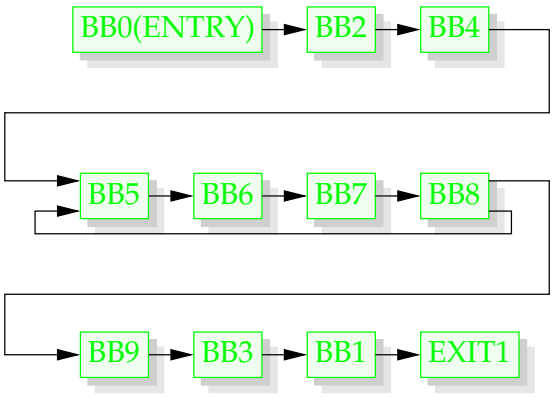

Figure 3.5: *The control flow graph for the first trace in the example program.*

Of course the computer on which PearColator is run does not execute Jikes RVM HIR directly; it must be translated into the correct machine code for the architecture. During this, many optimisations are carried out.

These stages, from HIR to machine code are carried out by existing parts of the Jikes RVM with no new PearColator code being needed. Figure 1.2 illustrates this in simple form; in reality, depending on the optimisation setting chosen for the

RVM, there can be over 140 steps in the process [9].

With an optimisation level of 1 chosen, the example trace under consideration leads eventually to the following IA-32 assembly language (after more than 50 stages of translation and optimisation):

```
-13     LABEL0   Frequency:  1.0
-1     ia32_cmp                AF CF OF PF SF ZF = esp(I),
    <[PR(I)]+-80>DW
-1     ia32_jcc                LE, LABEL4, <unused> AF CF OF
    PF SF ZF
-1     bbend                   BB0 (ENTRY)
-1     LABEL5   Frequency:  1.0
0     ia32_push                <[PR(I)]+-92>DW
0     ia32_mov                 <[PR(I)]+-92>DW = esp(I)
0     ia32_push                15542
0     ia32_mov                 <[esp(I)]+-20>DW = ebx(I)
0     ia32_mov                 <[esp(I)]+-16>DW = edi(I)
0     ia32_mov                 <[esp(I)]+-12>DW = ebp(I)
0     ia32_mov                 eax(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;,x,d) = edx(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;)
0     ia32_add                 esp(I) AF CF OF PF SF ZF
    <-- -20
-1     ia32_cmp                AF CF OF PF SF ZF = <[PR(I)]
    +-72>DW, 0
-1     ia32_jcc                NE, LABEL7, Probability: 0.0
    AF CF OF PF SF ZF
-1     bbend                   BB5
0     LABEL6   Frequency:  1.0
-1     ia32_mov                 edx(I) = 0
-1     ia32_mov                 ecx(I) = 0
```

```
-1     bbend                    BB6
0     LABEL1   Frequency:  100.0001
-1     ia32_cmp                 AF CF OF PF SF ZF = <[PR(I)]
    +-72>DW, 0
-1     ia32_jcc                 NE, LABEL9, Probability: 0.0
 AF CF OF PF SF ZF
-1     bbend                    BB1
0     LABEL8   Frequency:  100.0001
-1     ia32_inc                 edx(I) AF OF PF ZF <--
-1     ia32_mov                 <[eax(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;)]+-56>DW (<mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.r10>, <TRUEGUARD>) = edx(I)
-1     ia32_add                 ecx(I) AF CF OF PF SF ZF
    <-- edx(I)
-1     ia32_mov                 <[eax(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;)]+-60>DW (<mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.r11>, <TRUEGUARD>) = ecx(I)
-1     ia32_mov                 ebx(I) = <[eax(Lcom/ibm/
    JikesRVM/opt/PPCProcessSpace;)]+-408>DW (<mem loc:
    Lcom/ibm/JikesRVM/opt/PPCProcessSpace;.xer>, <TRUEGUARD>)
-1     ia32_mov                 <[eax(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;)]+-408>DW (<mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.xer>, <TRUEGUARD>) = ebx(I)
-1     ia32_cmp                 AF CF OF PF SF ZF = edx(I), 5
-1     ia32_mov                 edi(I) = 4
-1     ia32_mov                 ebp(B,+) = 8
-1     ia32_cmov                edi(I) <-- ebp(B,+), LT CF OF
    PF ZF
-1     ia32_cmp                 AF CF OF PF SF ZF = edx(I), 5
-1     ia32_mov                 ebp(I) = 2
-1     ia32_cmov                ebp(I) <-- edi(I), NE CF OF PF
```

```
    ZF
-1     ia32_shr                ebx(I) AF CF OF PF SF ZF
    <-- 31
-1     ia32_or                 ebx(I) AF CF OF PF SF ZF
    <-- ebp(I)
-1     ia32_shl                ebx(I) AF CF OF PF SF ZF
    <-- 28
-1     ia32_mov                edi(I) = <[eax(Lcom/ibm/
    JikesRVM/opt/PPCProcessSpace;)]+-144>DW (<mem loc:
    Lcom/ibm/JikesRVM/opt/PPCProcessSpace;.cr>, <TRUEGUARD>)
-1     ia32_and                edi(I) AF CF OF PF SF ZF
    <-- 268435455
-1     ia32_or                 edi(I) AF CF OF PF SF ZF
    <-- ebx(I)
-1     ia32_mov                <[eax(Lcom/ibm/JikesRVM/opt/
    PPCProcessSpace;)]+-144>DW (<mem loc: Lcom/ibm/JikesRVM/
    opt/PPCProcessSpace;.cr>, <TRUEGUARD>) = edi(I)
-1     ia32_mov                ebx(I) = edi(I)
-1     ia32_and                ebx(I) AF CF OF PF SF ZF
    <-- -2147483648
-1     ia32_shr                ebx(I) AF CF OF PF SF ZF
    <-- 31
-1     ia32_cmp                 AF CF OF PF SF ZF = ebx(I), 1
-1     ia32_set$b              ebx(Z) = EQ AF CF OF PF SF ZF
-1     ia32_movzx$b            ebx(I) = ebx(Z)
-1     ia32_inc                ebx(I) AF OF PF ZF <--
-1     ia32_cmp                 AF CF OF PF SF ZF = ebx(I), 2
-1     ia32_jcc                EQ, LABEL1, Probability: 0.99
    AF CF OF PF SF ZF
-1     bbend                   BB8
0     LABEL2   Frequency:  1.0
```

```
-1     ia32_cmp                AF CF OF PF SF ZF = <[PR(I)]
    +-72>DW, 0
-1     ia32_jcc                NE, LABEL11, Probability: 0.0
    AF CF OF PF SF ZF
-1     bbend                   BB2
-10     LABEL10   Frequency:  1.0
-3     ia32_mov                eax(I) = 268435596
-3     ia32_mov                ebx(I) = <[esp(I)]>DW
-3     ia32_mov                edi(I) = <[esp(I)]+4>DW
-3     ia32_mov                ebp(I) = <[esp(I)]+8>DW
-3     ia32_add                esp(I) AF CF OF PF SF ZF
    <-- 24
-3     ia32_pop                <[PR(I)]+-92>DW =
-3     ia32_ret                8, eax(I), <unused>
-1     bbend                   BB10
-1     LABEL4   Frequency:  0.0
-1 EG  ia32_int                <STACK OVERFLOW>
-1     ia32_jmp                LABEL5
-1     bbend                   BB4
0     LABEL7   Frequency:  0.0
0 EG  ia32_call               AF CF OF PF SF ZF =
    <0+1124082056>DW (<mem loc: JTOC @8132>, <TRUEGUARD>),
    static"com.ibm.JikesRVM.opt.VM_OptSaveVolatile.
    OPT_threadSwitchFromPrologue ()V"
-1     ia32_jmp                LABEL6
-1     bbend                   BB7
0     LABEL9   Frequency:  0.0
0 EG  ia32_call               AF CF OF PF SF ZF =
    <0+1124082064>DW (<mem loc: JTOC @8140>, <TRUEGUARD>),
    static"com.ibm.JikesRVM.opt.VM_OptSaveVolatile.
    OPT_threadSwitchFromBackedge ()V"
```

```
-1     ia32_jmp                LABEL8
-1     bbend                   BB9
-10     LABEL11   Frequency:  0.0
-10 EG  ia32_call                AF CF OF PF SF ZF =
    <0+1124082060>DW (<mem loc: JTOC @8136>, <TRUEGUARD>),
    static"com.ibm.JikesRVM.opt.VM_OptSaveVolatile.
    OPT_threadSwitchFromEpilogue ()V"
-1     ia32_jmp                LABEL10
-1     bbend                   BB11
```

Figure 3.6: *Final stage translation from the example program.*

It can be seen that the sequence of basic blocks has been changed significantly and that all the instructions are now specific to the IA-32 architecture. This is then assembled into a block of IA-32 machine code which the runtime environment places into its hashtable and executes. It remains in the hashtable during execution of the program in case the same code is to be executed again.

3.7 Summary

- The main components of PearColator are the process space, runtime system, hashtable of translated code, and the actual translator.
- The translator operates on single instructions, or on a block or trace of instructions.
- The hashtable keeps all the translations for future use.
- The instructions are translated into Jikes High-Level Intermediate Representation, which is then translated and optimised through many stages by the standard Jikes classes.

- The HIR is organised into basic blocks and the flow of control through them described by a flow control graph.

Every program has at least one bug and can be shortened by at least one instruction—from which, by induction, one can deduce that every program can be reduced to one instruction which doesn't work.

Anonymous

4 The PearColator Dynamic Binary Translator II

4.1 Introduction

The previous chapter described the first version of PearColator; this one covers the improvements which have been made to date. A great improvement in performance has been obtained, as will be shown in the next chapter.

4.2 Improved Code Modularity and Structure

As PearColator was originally written, the translation of a PowerPC instruction into HIR was done inside a very large switch statement in the class **PPC2IR**. This has now been rewritten to use a class **PPC_InstructionTranslator**, which has a subclass for each individual instruction in the PowerPC set.

The superclass has a method to translate each of the instruction forms (see section 2.2.5). These methods should never actually be called so they throw errors. Individual instruction subclasses override the method which is appropriate to their form with one which does the actual translation. Only these overriding methods should be called; the superclass methods are simply there in case a programming error leads to the a method being called for the wrong form.

Lookup tables are used to find the appropriate form and translator class for a given primary opcode. Where a secondary opcode exists, a second lookup table is used by the extended instruction translator, which in turn calls the translator for the particular instruction.

4.3 Lazy Evaluation of Condition Codes

Many instructions set a field of the condition register depending upon the result of the instruction (see the tables in section 2.2.2). Subsequently, this can be tested by a conditional branch instruction or moved to some other location.

In order to emulate this behaviour exactly, a fairly complex set of HIR instructions is necessary (basic blocks BB7 and BB8 in figure 3.4). This is slow because it involves making the comparison and setting CR bits accordingly, then later interpreting these in terms of comparison conditions.

To avoid this, PearColator uses lazy evaluation of the condition register. Instead of calculating the condition code following an arithmetic or compare instruction, variables are used to hold the values which are being compared (for a compare instruction), or the instruction result and the number zero (for instructions with the record (Rc) bit set).[1]

When a condition test is needed, it can be made using these values directly. It is necessary to know what type of comparison must be performed: signed, unsigned, or floating point. This is represented by a value in an object of class **PPC2IR.Laziness**. If it is desired to set the actual CR this can be also be done from these values.

If the condition code has been calculated and set in a field of the condition register, the **Laziness** object reflects the fact that lazy evaluation is not currently in use for that field. This is of importance at the start of a new trace or following a system call, since it is possible for the condition register to have been set to a value which does not follow the standard meanings of the bits, and so is not properly represented by the lazy state.

4.4 Execution Traces

Previously, HIR was generated by looping through a set of PowerPC instructions and translating each one until a point was reached where translation was stopped. This varied according to the translation mode (single-instruction, block or trace, see section 3.4).

The new PearColator is more sophisticated. A method is used,

[1] Comparison of the result with zero is the test for negative, positive, or zero.

PPC2IR.translateSubTrace(). For each instruction the appropriate translator is called, via look up tables. This plants the appropriate HIR and returns a value, usually the address of the next instruction, which is then translated. Alternatively a value of -1 may be returned to indicate that translation should stop (typically in these cases, the next instruction is not known at this point).

A mapping is maintained from a key formed from the program counter value and the lazy evaluation state to an HIR block (previously only the pc was used as the key for the mapping, since there was no laziness), and a count is kept of the number of instructions which have been translated in the current trace.

Conditional branch instructions are translated by planting HIR to evaluate the condition and jump to appropriate blocks of HIR. If no suitable HIR block exists for a jump target, this fact is registered, storing the instruction, program counter, and lazy state at the branch point. In the case of branch and link instructions, a list is kept of all the places in the code from which a branch can be made to any given target: this is used to guide the translation of return instructions. The return value from the translator method is the next instruction address (which will be executed if the branch is not taken), except in two cases:

- If the bits are set in the instruction to indicate 'branch always', in this case -1 is returned, to stop translation.
- If the instruction is a branch and link, and the number of instructions in the trace exceeds a certain value (dependent on the optimisation level): this is to prevent translating deeply into a region of code which might not actually be executed, depending on the result of the condition testing. In this case the return value to be passed back to the runtime system is set, the CR lazy state resolved, and the trace finished.

When translation is stopped by one of these possibilities, branch targets which have been registered as not yet translated can be resolved. The method **translateSubTrace()** is called again to translate these targets.

Whenever **translateSubTrace()** is called, it tests whether or not the number of instructions translated in the current trace has exceeded the same limit (dependent on the optimisation level) as in the case above. If it has, the trace is finished, with control passing back to the runtime system. Testing the number of instructions at this point ensures that traces end only at branch points, such as calls and returns. The intention is to generate traces which approximate to the methods in the original subject program.

4.5 Adaptive Compilation

One feature of the Jikes RVM which was not used by the earliest version of PearColator is adaptive compilation [9]. Whenever a method is found to be executed frequently the RVM estimates the cost and benefit to execution speed of recompiling at a higher optimisation level. If the result favours recompilation, it is performed, in parallel with code execution in a separate thread. This recompilation of the hottest (most frequently used) code regions is based on the conventional assumption of temporal locality—most of the execution time is spent in a small fraction of the total code.

PearColator has now been modified to allow recompilation of PowerPC code in a similar way. It is possible to configure the emulator to use optimisation level 0 initially and to compile at higher levels those code regions which are frequently used.

The trace length after which translation of a new subtrace is not started can be set independently for each of the optimisation levels. Tests have been conducted to find the optimum lengths. The results are presented in section 5.3.1.

4.6 Optimisation of Register Handling

At the start of each trace, all the register values are loaded into Jikes RVM temporary registers from the process space. At the end they are spilled back again. This involves unnecessary operations if, as is generally the case, not all are used.

To avoid this, an optimisation phase has been added. A record is kept of which registers are actually used, and the filling and spilling of the unused registers is eliminated during this optimisation phase.

4.7 Summary

- A number of modifications have been made to the original system described in the previous chapter.
- The code has been re-structured to improve its modularity.
- The condition codes are lazily evaluated.
- Unnecessary register spill/fill operations are eliminated.
- The formation of traces through the code has been improved to allow more sophisticated branching within a trace. It is possible to build traces through thousands of PowerPC instruction.
- The Jikes RVM's adaptive compilation system can be used.

Calvin*: "I like to verb words."*

Hobbes*: "What?"*

Calvin*: "I take nouns and adjectives and use them as verbs. Remember when 'access' was a thing? Now it 's something you do. It got verbed."*

Calvin*: "Verbing weirds language."*

Hobbes*: "Maybe we can eventually make language a complete impediment to understanding."*

Calvin and Hobbes cartoon strip, Bill Watterman

5 *Evaluating the Performance of PearColator*

5.1 The Testing Regime

The performance of PearColator has been investigated using the *Dhrystone* integer benchmark [23]. Tests were carried out on the early version of PearColator (described in chapter 3) and the improved system (chapter 4).

The benchmark performs a fixed set of integer operations a user-specified number of times. This number of executions was varied to allow a distinction to be made between the fixed[1] overhead of translation and the time take to execute the translated code (dependent on the number of loop executions).

During most of the tests (except those in section 5.3.1), PearColator was run on a computer with an AMD AthlonXP 2700+ CPU with a clockspeed of 2.16 GHz and 256 kB cache, running Linux kernel 2.6.4. For comparison the same benchmark was run (natively, without translation) on an Apple iBook computer with a 600 MHz G3 PowerPC processor.

5.2 Early Version

All three modes of translation available were tested—single-instruction, block, and trace. As was hoped, the more sophisticated modes, translating multiple instructions together, gave significantly better performance than was obtained by translating instructions one at a time. The Jikes RVM was run at optimisation level 2 for these tests; this uses a large number of optimisation stages.

Figure 5.1 shows the results for the three modes.

It can be seen that when the number of Dhrystone loops executed is small, the Dhrystones/second figure (D) is approximately proportional to the total number of loop executions (N). The total execution time of the benchmark is constant for all small numbers of loops, because it is dominated by the time take to translate the

[1] Independent of the number of loop executions.

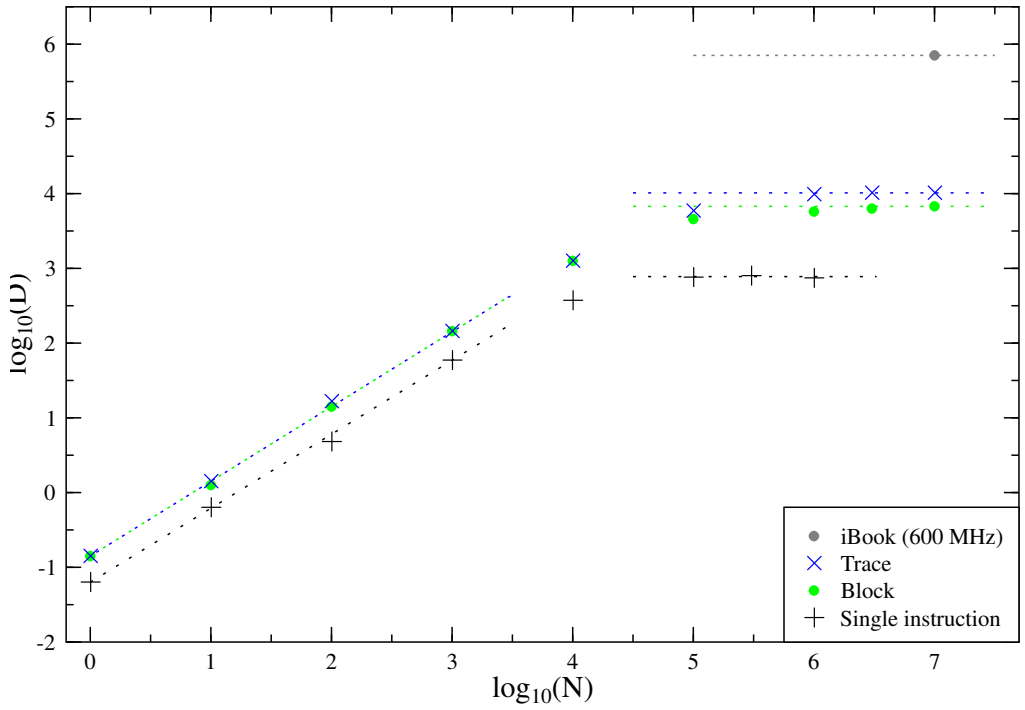

Figure 5.1: *Dhrystone benchmark performance of the early version of PearColator running on a 2.16 GHz PC. The data plotted shows the number of Dhrystone loops per second as a function of the total number of loops executed for the three translation modes. Also shown is the performance of a 600 MHz iBook (too fast to give reliable measurements at small N).*

code. At high N, however, the number of Dhrystones/second is almost constant: in this case the time taken in the actual execution of the translated code is dominant.

Further tests were done to measure the effect of varying the level of optimisation performed by the Jikes RVM on the generated HIR. The trace translation mode was used for this and the results are shown in figure 5.2.

It can be seen that reducing the number of optimisation stages reduces the N-independent time due to the translation overhead (seen at small values of N) but that there is no significant variation in the execution time per loop iteration (the limiting value at large N).

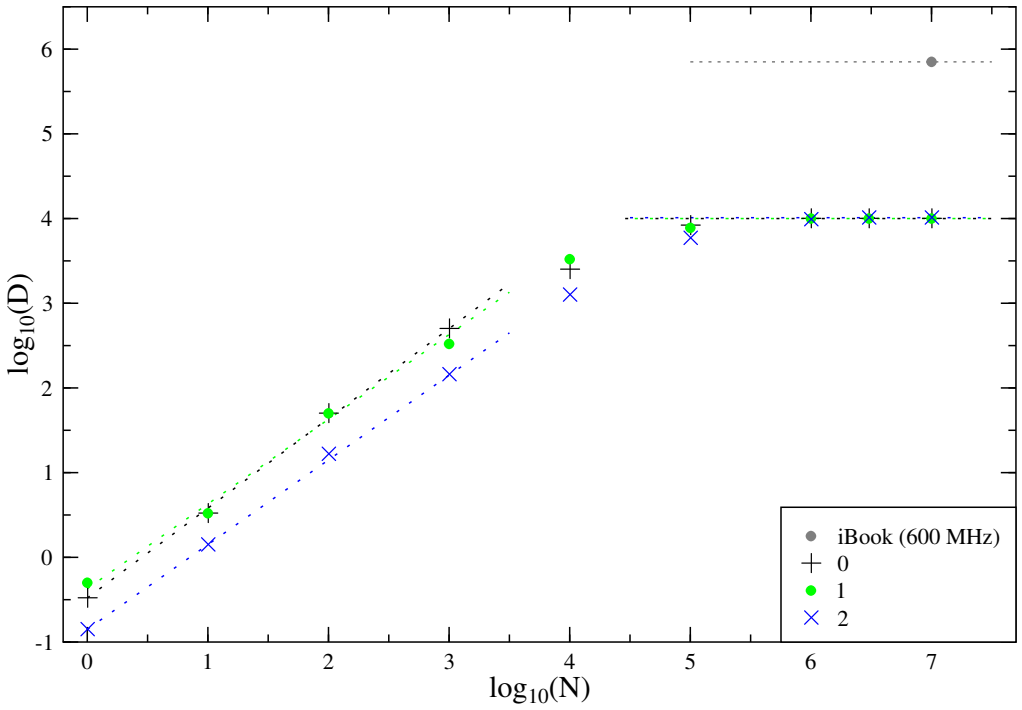

Figure 5.2: *Dhrystone benchmark performance of the early version of PearColator at different optimisation levels.*

5.3 Later Version

5.3.1 Trace Lengths

As explained in section 4.4, when the method **translateSubTrace()** is called it stops translation if the number of instructions already translated in the trace has exceeded a value which depends on the optimisation level. Results are presented here of the tests to find the optimum values of these limits.

The first stage test was carried out with adaptive compilation disabled and the optimisation level fixed at 0. This allowed the optimum trace length to be found for this level. Figure 5.3 shows the variation of the performance of PearColator with the trace length setting. The benchmark was 1 million runs of the Dhrystone loop. A different computer was used for the trace length tests from the one described above, having a 3 GHz Intel Pentium 4 processor with 1 MB cache.

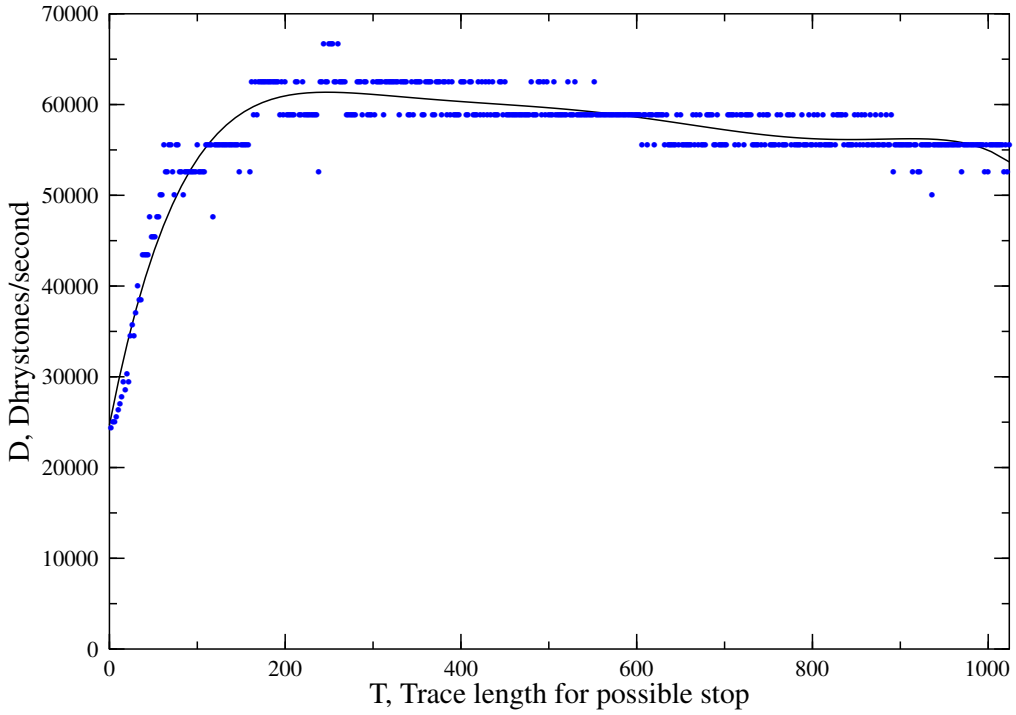

Figure 5.3: *Effect of trace length with optimisation level of 0 on performance of PearColator. Adaptive compilation was disabled. At each trace length setting, 1e6 iterations of the Dhrystone loop were performed. A sixth order polynomial least squares fit is shown.*

It can be seen that the performance peaks for a trace length of approximately 250 instructions.

A further test was conducted around this range of trace length. In each case several runs were conducted with the same trace length setting, each of 10 million Dhrystone loops, and the mean benchmark result plotted in figure 5.4.

This shows that in a small range around 250 instructions, there is little variation in the benchmark result. A 'round' figure of 256 was decided to be a suitable length to use.

A second stage of testing was conducted with adaptive compilation enabled. The initial optimisation level was set to 0 and the trace length for possibly stopping when at this level was set to 256 instructions. The trace length for levels 1 and 2 was varied, but in each case was the same for level 2 as level 1. To improve the accuracy

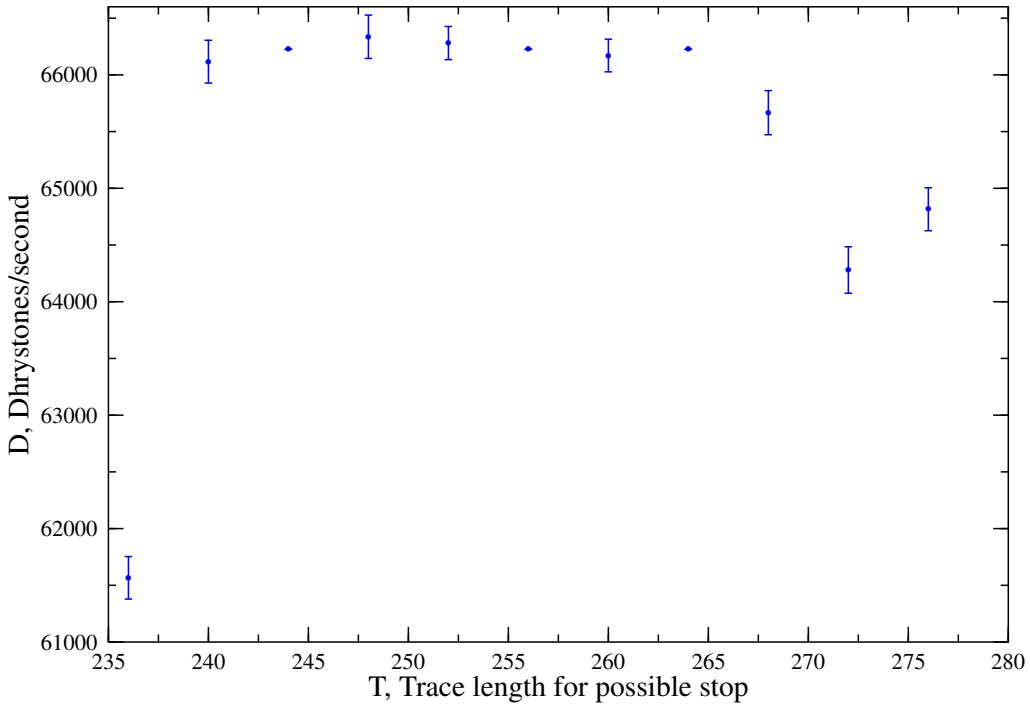

Figure 5.4: *Effect of trace length with optimisation level of 0 on performance of PearColator. Adaptive compilation was disabled. Each point shows the mean and standard deviation of several runs of 1e7 Dhrystone loop iterations.*

of the results, each test was conducted several times and the mean benchmark value calculated.

It can be seen in figure 5.5 that there is very little variation at all; note that the vertical scale is much enlarged. The error bars show the standard deviation: in some cases all runs gave the same result, so no deviation.

5.3.2 Comparison with the Old Version

Figure 5.6 shows the performance of the current PearColator on our standard test system, for comparison with the results in section 5.2. The trace lengths for possibly stopping translation were 256 instructions at optimisation level 0 and 1536 at the higher levels. Tests were conducted with adaptive compilation (with an initial optimisation level of 0) and without.

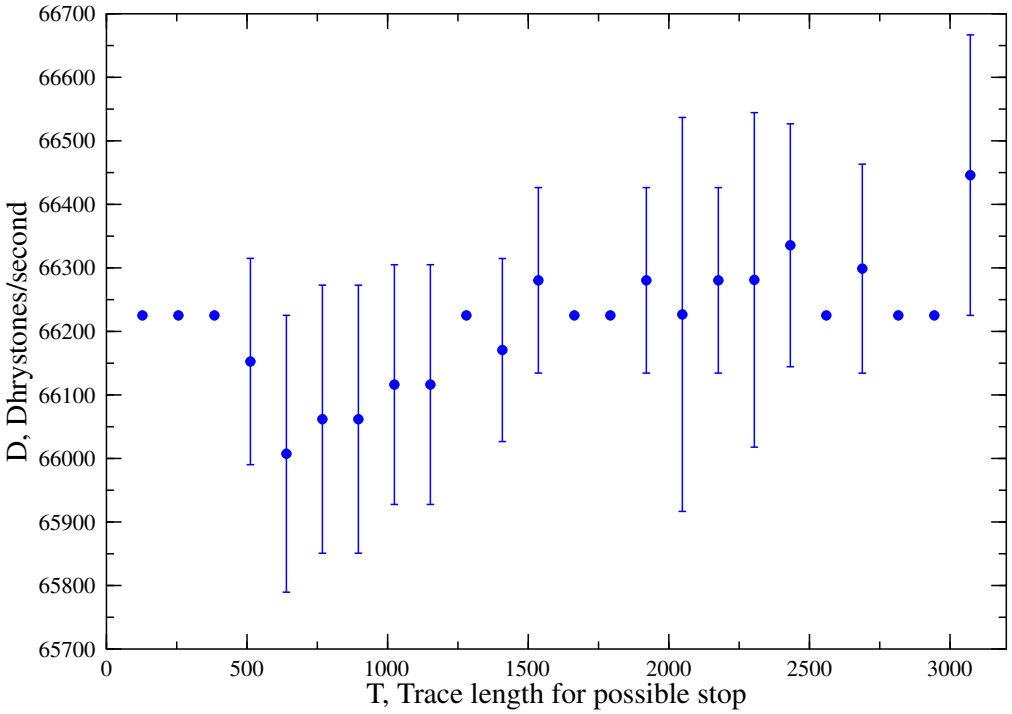

Figure 5.5: *Effect of optimisation levels 1 and 2 trace length on performance of PearColator. Adaptive compilation was enabled, with an initial optimisation level of 0.*

It can be seen that there is an extremely good improvement in the benchmark values (a factor of ten) . However the use of adaptive compilation does not currently offer any significant benefit.

5.4 Comparison with Other Emulators and Native Execution

In this section, the performance of PearColator is compared with existing PowerPC emulators and with native execution of the Dhrystone benchmark on PowerPC and IA-32 hardware. There also exists a Java version of the benchmark and this is included in the comparison, running on the Jikes RVM and on the HotSpot virtual machine.

Figure 5.7 compares the Dhrystone results for PearColator with two other PowerPC

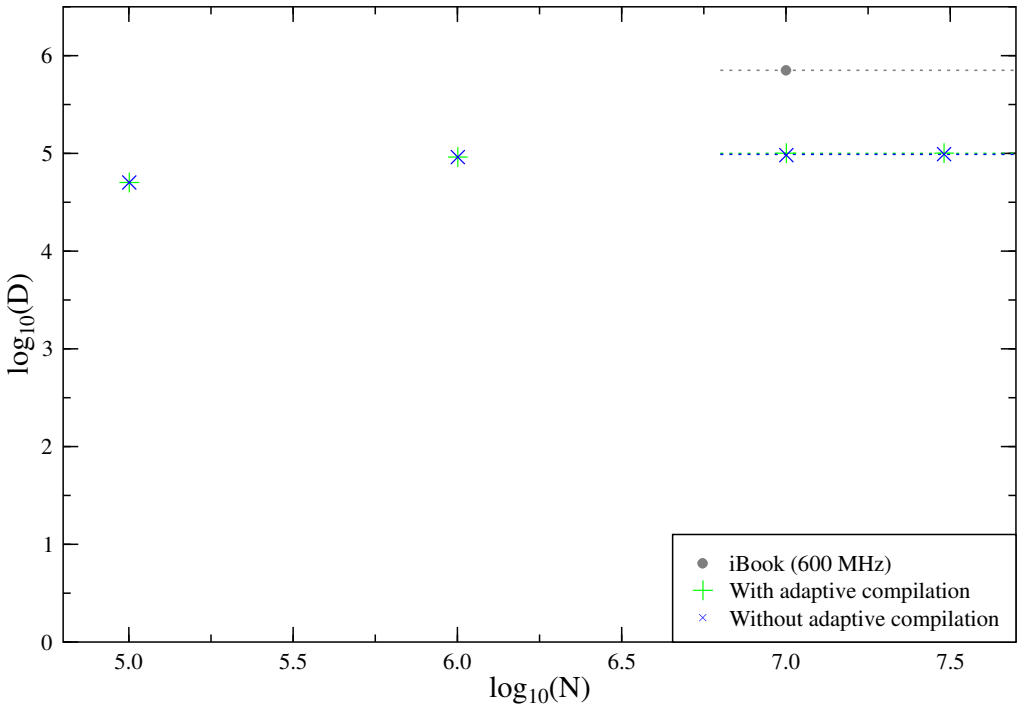

Figure 5.6: *Dhrystone benchmark performance of the current PearColator. At smaller values of N, execution was too fast for reliable timing.*

emulators, all running on the AthlonXP 2700+ test system. Also shown are the benchmark values obtained running Dhrystone compiled for native execution on this computer and on the 600 MHz G3 system. Lastly results are presented for the Java version of Dhrystone running on virtual machines on the AthlonXP.

It can be seen that all three emulators are much slower than native execution on the same hardware (AthlonXP). Although PearColator is the slowest currently, its performance is similar to that of PearPC. It is hoped that further improvements can be made to PearColator to increase its speed.

These results also show that the Java Dhrystone benchmark can be faster than the C original. This results from the use of compiler optimisations within the virtual machine which were not enabled in the C compiler (standard Dhrystone testing conditions).

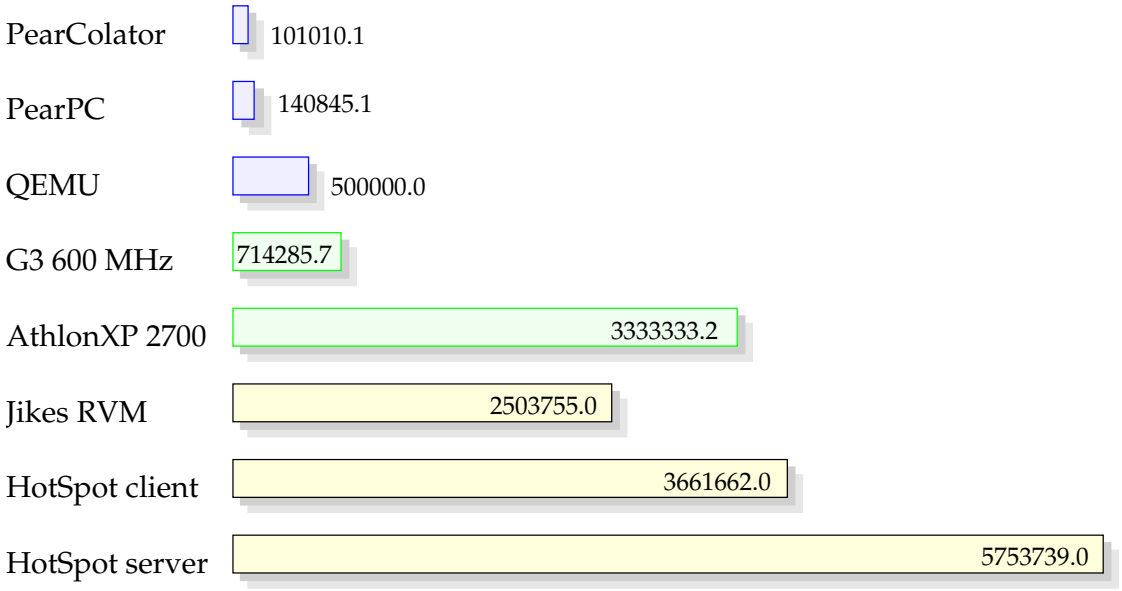

Figure 5.7: *Comparison of PearColator with other emulators and with native execution, Dhrystones per second for 1e7 iterations of the benchmark loop.*

5.5 Summary

- Considerable improvements have been made to the performance of PearColator.
- The threshold trace length at optimisation level 0 has a significant effect on the benchmark performance. Approximately 256 instructions was found to be the optimum length. No significant variation was found with the length of traces at the higher optimisation levels, showing that there is currently no benefit from the use of the adaptive system by PearColator.
- Using the Dhrystone benchmark, it is 28% slower than PearPC, and (when running on an AthlonXP 2700+) about seven times slower than native execution on a 600 MHz G3 PowerPC, but...
  - It is hoped that further improvements can be obtained.
  - On future chip-multiprocessor architectures, the PearColator approach should be much more suitable for parallel compilation and execution than traditional dynamic binary translators.

- PearColator has security advantages over many other emulators, including safeguards on memory access.

It is very difficult to prophesy, especially when it pertains to the future.

Patrick Kurzawe

6 Conclusions and the Future of PearColator

6.1 Overview

The approach to dynamic binary translation, using a virtual machine, taken by this project has been shown to be viable. Performance results have been achieved which are close to those of the PearPC emulator for the Dhrystone benchmark.

Although PearColator has achieved some very satisfying results, there is plenty of scope for further development of the project. These should enable PearColator to execute a wider range of PowerPC programs and it is hoped that there will be continued improvements in the performance obtained.

6.2 Completing the Instruction Set and System Calls

So far there remains a part of the instruction set which has not been implemented. In particular many of the floating point instructions are not yet provided with translators. Finishing this should simply be a routine matter of taking the time needed to write the translations.

Only a small number of system calls have been implemented—those needed to run the Dhrystone benchmark and few other programs. Eventually it will be necessary to implement the rest for PearColator to provide a complete PowerPC/Linux subject environment.

6.3 Dynamic Linking

So far PearColator requires the subject program to be statically linked against all necessary library code. This is not convenient as it is usual for Linux programs to be dynamically linked against shared libraries. The program **ld.so** is used by Linux to accomplish dynamic linking and PearColator will have this incorporated into it.

When the Linux kernel attempts to load an ELF binary file which identifies itself as

being dynamically linked, it actually loads and executes **ld.so**. A reference to the dynamically linked program is passed on the stack, so it can be loaded and linked by **ld.so** [19]. There should be no major difficulty involved in adapting PearColator to use this mechanism.

6.4 Parallelisation

It is intended that the main advantage of the approach used by PearColator over conventional dynamic binary translators will be its ability to take advantage of future multiprocessor architectures. Separately from the PearColator project, work is being carried out to include novel parallelisation features in the Jikes RVM. When these are available, PearColator can be made to operate using many threads for compiling and executing code.

6.5 Other Optimisations

It is hoped that PearColator will be able to take advantage of the adaptive system and its on-stack-replacement capability. This enables the replacement of a block of code with a better translation (if one exists) part way through execution.

There are plans to improve the trace building to improve the finding of those sections of code which will actually be used (rather than those which merely *could* be reached by conditional branches). Focusing on the order of compiling branch return targets and branch directions (forward or backward in code) is expected to be useful here.

6.6 Adaptation to Other Subject Architectures

In the more distant future it is possible that PearColator could be adapted to translate programs for other subject environments.

One such environment would be the OS X operating system on the PowerPC. Since PearColator already translates this instruction set, the necessary work would be to load programs in a different binary format (which does not support static linking), and to replace the handling of Linux system calls with something suited to this different operating system.

Translating another instruction set architecture would require much larger changes. If it were also of the load-store RISC type, the existing structure of PearColator should be portable to it. The most complicated development would be to adapt PearColator to an architecture such as IA-32, in which arithmetic and logical instructions can access memory directly and in which instructions are of varying length.

6.7 Summary

- There are still PowerPC instructions and Linux system calls which need to be implemented.
- Dynamic linking will be facilitated using **ld.so**.
- Enhanced performance is hoped for from better use of the adaptive system and more sophisticated setting of priorities in the translation of code.
- Major performance benefits are expected when PearColator is used with future parallel technology.
- The PearColator approach can be extended to other emulation subjects.

References

[1] Ian Rogers, Richard Matley, and Ian Watson. Dynamic binary translation with a Java Virtual Machine. Submitted to 2005 International Symposium on Code Generation and Optimization, 2004.

[2] Mark Probst. Fast machine-adaptable dynamic binary translation. In *Workshop on Binary Translation*, 2001.

[3] Ian A. Rogers. *Optimising Java Programs Through Basic Block Dynamic Compilation*. PhD thesis, The University of Manchester, 2002.

[4] Fabrice Bellard. QEMU internals. *http://fabrice.bellard.free.fr/qemu/qemu-tech.html*, July 2004.

[5] AMD. *Software Optimization Guide for AMD Athlon64™and Opteron™Processors*, September 2003.

[6] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. Intel Centrino mobile technology. *Intel Technical Journal*, 7(2), 2003.

[7] Jim Turley. Alpha runs x86 code with FX!32: Digital's emulation strategy could help boost Alpha/NT system sales. *Microprocessor Report*, 10(3):11–14, March 1996.

[8] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John

Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, San Francisco, California, June 1999.

[9] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, Minneapolis, Minnesota, October 2000.

[10] The Jamaica project. *http://www.cs.man.ac.uk/apt/projects/jamaica/*.

[11] QEMU CPU Emulator. *http://fabrice.bellard.free.fr/qemu*.

[12] PearPC PowerPC Architecture Emulator. *http://pearpc.sourceforge.net*.

[13] The Softpear Project. *http://softpear.sourceforge.net*.

[14] Jikes Research Virtual Machine. *http://www-124.ibm.com/developerworks/oss/jikesrvm/*.

[15] IBM. *The Jikes Research Virtual Machine User's Guide 2.3.1*, December 2003.

[16] Tool Interface Standards. *Executable and Linkable Format (ELF) 1.1*. *http://www.skyfree.org/linux/references/ELF_Format.pdf*.

[17] Motorola, Denver, Colorado. *Programming Environments Manual For 32-Bit Implementations of the PowerPC Architecture*, 2001.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann, San Francisco, California, third edition, 2003.

[19] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann, San Francisco, California, 2000.

[20] Free Standards Group. *Linux Standard Base Specification for the PPC32 Architecture 1.3*, 2002.

[21] Steve Zucker and Kari Karhi. *System V Application Binary Interface PowerPC Processor Supplement*. SunSoft, IBM, September 1995.

[22] SCO, AT&T. *System V Application Binary Interface 4.1*, March 1997.

[23] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

Index

A page number in bold type indicates the most important section relating to the entry. A page number in italics indicates the position of the definition of a term.