Dynamic Binary Translation on the .NET Platform

by

Patrick Andrew Wright

BSc, Victoria University of Wellington, New Zealand, 1994

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisory Committee

Dynamic Binary Translation on the .NET Platform

by

Patrick Andrew Wright

BSc, Victoria University of Wellington, New Zealand, 1994

Supervisory Commitee

Dr. Nigel Horspool, (Department of Computer Science)

Co-Supervisor

Dr. Micaela Serra, (Department of Computer Science)

Co-Supervisor

Abstract

Emulation is the practice of simulating one computer system on another. There are many methods of implementing an emulator. They exist on a performance continuum from simple interpretation to dynamic binary translation extended with various optimizations. Optimizations are diverse, including just in time compilation, large translation units, shadow stack, register mapping and many more. The goal of this thesis is to develop a high performance, portable emulator for the ARM v4 architecture without requiring substantial code analysis. This thesis describes the implementation of a dynamic binary translator translating to an intermediate language targeting a virtual machine. Targeting a virtual machine ensures that the emulator is portable. Optimizations implemented include forming large translation units and branch straightening in hot regions. The particular combination of translating to intermediate form for a virtual machine, and creating large translation units from hot regions does not seem to appear in the literature. The performance of the described dynamic binary translator exceeds the performance of an interpreter on the same platform by an order of magnitude. Code analysis was only used to straighten branches in hot regions. While many popular dynamic binary translation optimizations are not readily applicable when using a virtual machine target, the performance achieved shows that using virtual machine as translation target is viable method of implementing dynamic binary translator.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I wish to acknowledge all the people who have helped and guided me during my studies. I would like to express my sincere gratitude to my supervisors, Dr. Nigel Horspool and Dr. Micaela Serra. Lastly I want to thank my family members for their understanding and endless support.

Dedication

For Jenny.

1 Introduction

This thesis investigates dynamic binary translation, which is a particular form of emulation. Emulation may be described simply as the practice of simulating one computer system on another. Emulation is useful for:

- Preservation (the ability to execute programs from an obsolete system).
- New system development – develop a system emulator as a precursor to developing the hardware.
- Software development when no hardware is available. E.g. an emulator is built in to Apple Computer's Xcode development environment.
- Education, e.g. the UVic ARMSim#, an emulator for the ARM7TDMI processor.
- Migration, providing the ability to run programs from a previous platform. E.g. Apple computer provided an emulator for Motorola when they migrated to PowerPC, and again when they migrated to Intel processors.
- Performance tuning, e.g. an emulator may give access to count clock cycles for memory access and other operations, which can be difficult even when hardware is available.

The goal of this thesis is to develop and describe a high performance, portable emulator for the ARM v4 architecture. An existing interpreter will be extended to create a dynamic binary translator (DBT) that implements some of the techniques discussed in the recent literature. The implemented DBT runs on three platforms[1] and is more than ten times faster than the interpreter when compared using three benchmark programs. These results are achieved without requiring static code analysis.

This thesis will first introduce the area of emulation, followed by a review of the technologies used to implement the dynamic binary translator and the current literature in the area. The implementation of the DBT and the incorporation of concepts from the literature will be described and the results achieved discussed.

[1] Microsoft Windows, Linux and Apple Macintosh OSX

2 Background

2.1 Emulation

A broad definition of emulation is “the process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface or functionality” [**1**]. The same authors give another definition, which is more useful for the purpose of this thesis “In terms of instruction sets, emulation allows a machine implementing one instruction set, the target instruction set, to reproduce the behaviour of software compiled to another instruction set, the source instruction set”.

Emulation is an important enough area that large corporations have invested time and money to create emulators. This is often done to allow software migration to new hardware, for example Apple Computer provided the Rosetta emulator with OS X v10.4.4 to allow Intel based machines to run software compiled for the earlier PowerPC machines.

There are several common techniques that are used to create emulators. These techniques exist on a continuum of complexity, performance and resource requirements. Interpretation is the simplest, slowest, least resource intensive technique while binary translation lies toward the other end of the continuum requiring more resources but delivering superior performance. Several other techniques including threaded interpretation and pre-coding lie between the two extremes.

2.1.1 Interpretation

Interpretation is a widely used technique for implementing high-level programming languages, for example Perl, Ruby and Python are currently popular interpreted languages. However in the context of emulation the goal is to interpret machine code rather than a high level language. The operation of a simple interpreter may be defined as “stepping through the source program instruction by instruction, reading and modifying the source state according to the instruction” [**1**].

A simple interpreter may be described as having 3 parts:

1. Simulated memory area containing code, data and the stack.
2. A context block that stores the simulated state of the source machine.

3. The interpreter codes.

The interpreter operates by loading an instruction from the simulated memory area, decoding the instruction and updating the state and/or memory based on the decoded instruction. This approach is known as a *decode-and-dispatch* interpreter because there is a central loop that *decodes* the instruction and *dispatches* it to a routine that updates the memory and state of the simulated machine [**1**].

A simple interpreter may be enhanced in several ways. Common approaches include:

- Indirect Threaded interpretation removes the central dispatch loop and replaces it with code to move directly to the next instruction at the end of each instruction interpretation routine. The goal of this is to reduce the number of branch instructions executed. The key component of this technique is the dispatch table, which maps instructions to routines that interpret them. When interpretation of the current instruction is complete the next instruction is decoded and the address of the routine to interpret it is obtained from the dispatch table. This is referred to as indirect threading because of the indirection of the dispatch table.
- When a source instruction is interpreted multiple times, some of the work that is done to extract the meaning of the instruction is repeated. The precoding technique captures this repeated work and stores the instruction in an intermediate form that allows the instruction to be interpreted more easily. The operands and other information from the instruction are extracted into the fields of the precoding. The precoding is based on the instruction type so one precoding may be shared across multiple instructions. The interpreter routine to execute the decoded instruction retains the mechanism from Indirect Threaded interpretation that loads the next instruction, but now the dispatch table stores the pointer to the routine that executes the pre-coded instruction. It has been suggested that this technique is better suited to CISC architectures where the instruction decode is a more complex process due to variable instruction lengths and layouts [**1**].
- Direct Threaded Interpretation is based on precoding but discards the dispatch table. Instead of storing the address of the routine to execute the pre-coded instruction in the dispatch table the address of the routine is held in the structure that contains the pre-coded instruction.

2.1.2 Binary Translation

Binary translation is the process of converting the instructions of the source program into instructions for the target and update the simulated state of the source machine. Each instruction in the source program is mapped to a specific translation on the target that simulates the operation of the source instruction. Two things separate binary translation from precoding:

1. The translated form is directly executable on the target and requires no interpreter routine.
2. Each translation fragment maps to a specific source instruction, block or region.

Binary translation is a hard problem in the absence of a high level representation of the source program because all knowledge of the program must be derived from the in memory representation of the source program. In a static context this presents several significant challenges [**1**]:

- Code discovery
- Code location
- Self referencing code
- Self modifying code

2.1.2.1 Code Discovery

Code discovery is the problem that it is hard to know in any given block of memory exactly which bytes represent code and which represent data. This is a more significant problem with CISC instruction sets, because of the inconsistent instruction lengths and layouts, but is relevant to RISC also. For example the bytes following a jump instruction may or may not be code, or be reachable if they are code.

Some common object/executable file formats separate code and data e.g. the Portable Executable (PE) format on Windows. In the case of the PE format the operating system maps text areas as execute/read-only and data areas as no execute/read write. However there may be read-only data in the text area so code discovery remains an issue.

2.1.2.2 Code Location

Code location is the issue of knowing where in the target memory code is located. The problem can be illustrated with an indirect jump such as the following ARM v4 instruction:

```
MOV PC, R2
```

At runtime this causes the processor to branch to the location held in register 2, at translation time the address is usually not known and sometimes cannot be known. We can assume that the contents of R2 is an address in the memory block that contains the code and data. The issue is that the address in R2 is a source address, which means that a mechanism to determine the translated address at runtime must be implemented.

2.1.2.3 Self Referencing Code

Self-referencing code is where the program reads data from its code area. This is a variation of the code location problem in that the address that is being read from must be translated to a target address.

2.1.2.4 Self-Modifying Code

Self-modifying code is code which writes into its code area at runtime, so potentially instructions may change after they have been translated. Again as with the code location problem, it's also necessary to be able to find the correct location to write to.

Data execution prevention (DEP) in modern operating systems normally prevents execution of any writeable memory areas, except for privileged applications such as just-in-time (JIT) compilers. The purpose of DEP is to improve the security of the operating system against attacks such as buffer overflow exploitation.

2.1.3 Dynamic Binary Translation

One way to overcome some of these issues is to perform translation on the source program when it is executing with actual data. This is dynamic translation, as code is discovered it is translated.

A simple DBT has several parts [**1**]:

- An emulation manager that controls the process at a high level.
- An interpreter.

- A binary translator that converts one or more source instructions to one or more target instructions.
- A source code block – an address range in memory that contains the code and data of the source program.
- A target code cache.
- A mechanism to map the source program counter to target program counter.

The general procedure is to build some minimal translation unit, which is called by the emulation manager in place of interpreting individual instructions. This minimal translation unit may be described as a dynamic basic block [**1**], which is defined as a block of instructions that starts with the first instruction executed after a branch or a jump and ends with the next branch or jump instruction. This differs from a static basic block in that it may contain multiple entry points. It also means that a series of instructions may be present in multiple DBBs.

Static Basic Blocks

```
       add          Block 1
       load
       store
loop:  load         Block 2
       add
       store
       brcond
skip
       load         Block 3
       sub
skip:  add          Block 4
       store
       brcond
loop
       add          Block 5
       load
       store
       jump
indirect
```

Dynamic Basic Blocks

```
       add                  Block 1
       load
       store
loop:  load
       add
       store
       brcond skip

       load                 Block 2
       sub
skip:  add
       store
       brcond loop

loop:  load                 Block 3
       add
       store
       brcond skip

skip:  add                  Block 4
       store
       brcond loop
```

Figure 1 Static versus Dynamic Basic Block [1]

The execution of the translator may be described as follows:

1. From the program entry point instructions are interpreted until execution reaches the first branch or jump instruction.

2. The branch or jump instruction ends the dynamic basic block (DBB).
3. The DBB may be translated and stored in the translated block cache (TBC). The source program counter to target program counter map is updated to point to the translated block. Now the block can be executed without falling back to the emulation manager between translated instructions.
4. Before an instruction following a branch or jump is interpreted the emulation manager checks the TBC for a translation for the address of the next block:
    a. If the address is a miss in the TBC the next block is interpreted until execution reaches a branch or jump instruction which ends the DBB.
    b. If the address is a hit in the TBC the translated block is executed. At the end of the translated block control passes back to the emulation manager.

A primary goal of Dynamic Binary Translation is to significantly improve performance when compared to an interpreter as well as being easier to implement than a Static Translator. While a DBT may require additional memory to store the translated code cache and extra execution time to create the translated instructions it has performance potential well beyond the interpreter. DBT has been widely used to implement emulators [**2**] including:

- Apple Computer, M68K to PowerPC
- Transitive Corporation, QuickTransit Motorola Power PC to Intel x86 (also SPARC to x86, x86 to Power Architecture)
- DEC, FX!32 x86 to DEC Alpha
- HP ARIES, HP 9000 HP-UX to HP Integrity HP_UX 11i
- Sun Microsystems Wabi x86 to SPARC

3 Previous Work

Current work in dynamic binary translation is focused in several areas:

- The reduction of the overhead associated with binary translation [**3**], [**4**].
- The ability to target many different kinds of applications without being limited to static program code [**5**].
- The ability to observe the simulated system state at any point during a simulation [**6**].
- The ability to easily change the target architecture from one instruction set architecture (ISA) to another [**7**].

However the main focus of research is on improving the performance of binary translation through reducing the associated overhead. This overhead has been characterized in several ways:

- The overheard is made up of translation overhead and execution overhead. Most processor time is spent executing translated code so optimizing the translated code to reduce the execution overhead is important [**4**].
- A less generalized view of the overhead is that it is comprised of overhead from initialization, cold code translation, profiling, and hot trace building, all of which should be targeted to reduce their impact [**3**].
- Another view is that there is overhead in translating code and in executing translated code and a balance must be struck between the two. Up to some limit improving the quality of the translated code pays off with a reduction in overall execution time [**8**].

3.1 Reducing Translation Overhead

While this is described as needing to be balanced with the execution overhead, there appears to be very little work being done to reduce it. One way to reduce the translation effort is to simply translate only the 'hot' blocks [**9**], [**6**].

3.2 Reducing Execution Overhead

The approaches to improving the quality of translated code are many and varied, some address improving the efficiency of running the translated code, others address the issue of code expansion and attempt to reduce the number of target instructions that are generated for each source instruction. Proposed approaches for reducing execution overhead include:

- Increase translation unit size
- Shadow stacks
- Indirect branch target caching/software indirect jump prediction
- Duplicate and rearrange code
- Map source registers to target registers
- Optimize condition code handling
- Take advantage of powerful instructions in the target ISA that may replace several source instructions
- Cache decoded instructions for reuse by the interpreter

The first three of the above can be grouped together as mechanisms to reduce calls to the emulation manager. The emulation manager in a binary translator performs a similar role to the dispatcher in an interpretive simulator. When a translation unit's execution is completed the emulation manager determines whether execution can continue from the translation cache or whether further source code must be translated.

3.2.1 Translation Unit Size

A common approach to reducing calls to the emulation manager is to execute larger units of code. Larger units of code may improve performance in two ways:

1. Give fewer points at which to return to the emulation manager (minimize context switching).
2. Provide the compiler of the translated code greater scope for optimization (improve code quality).

While some DBT systems have used single instructions [**5**] as the translation unit, it is more common that the unit of translation is the basic block, [**6**], [**9**] or dynamic/extended basic block [**10**], [**11**].

Generating larger translation units is an active area of research. The general approach is to profile the execution of the program and monitor the number of times that each basic block is executed. Blocks that are executed more times than a configurable threshold value are marked as hot blocks. Various-schemes have been proposed for making use of the hot block profile.

The Edinburgh High-speed Simulator (EHS) implements four execution modes to investigate the effects of translation unit size [**12**]. The system maintains a counter to track the total number of interpreted blocks and only performs translation after some number of blocks have been interpreted. They refer to the interval between two successive translations as an epoch. During an epoch the simulator builds an execution profile for each physical page. The baseline mode counts the number of times each block on the page is interpreted and translates the 'hot' blocks. This is referred to as basic block (BB) mode. There are three additional large translation unit (LTU) modes with translation units of increasing size. In LTU mode a control flow graph (CFG) is created for each physical page, the page may contain several separate CFGs, one or more combined CGs or a combination of separate and combined CFGs. The first increment of translation unit size use strongly connected components (SCC) as translation units. An SCC is a collection of basic blocks that have hot control flow graph (CFG) arcs between them, and any attached linear regions. Tarjan's algorithm is used to extract the strongly connected components from the page. The next increment in translation unit size is the CFG where each CFG within the page forms a translation unit. The final translation unit is the page that contains one or more CGFs. The page LTU has the best simulation performance, but the performance difference between the large translation unit approaches is minimal.

Translation Unit	**Performance Improvement over BB**
SCC	1.63
CFG	1.64
Page	1.67

Table 1 EHS Performance improvement

A similar approach is to create hot regions, which are arbitrary CFG sub-graphs constructed from hot basic blocks and the blocks neighboring them that are also hot [**13**] [**8**].

Another mechanism is to patch the end of a translated block with a jump to the start of the next BB [**14**], [**15**]. This may be the simplest approach since no profiling is involved. A similar approach is lazy block linking, where blocks are only linked together when a context switch occurs [**16**].

A significant difference between these approaches is whether the large execution units are formed by joining together smaller translation units [**13**] or compiled directly to larger translation units [**12**].

Creating dynamic basic blocks often results in code duplication as more than one entry point may exist to the same piece of code. This code duplication has been measured and found to only increase the generated code size minimally [**4**], [**13**].

3.2.2 Shadow Stack

When a function is called the return address is usually stored in a register or pushed onto the stack to enable the epilogue of the called function to jump to the saved location. In a DBT returning from a function call may entail returning to the emulation manager to determine the address of the next instruction to execute. The shadow stack is a mechanism to avoid the cost of mapping the source return address to the target return address when returning from a function call. The address of the translated block to return to is saved on the shadow stack, while the address of the source address to return to is stored in the source stack. On return from the function call, if the address from the shadow stack matches the translation of the address from the source stack, the target address from the shadow stack can be used to access the next translated block directly. If the addresses don't match or there is no translated block for the address, then the call goes to the emulation manager [**1**], [**16**]. The shadow stack technique is only applicable to DBTs that maintain both a source program counter and a target program counter (TPC). Many of the DBTs that are JIT compiled do not maintain a TPC [**6**], [**11**]. This idea can be extended to preemptively translate the code at the return site if it has not already been translated [**17**].

3.2.3 Indirect Branch Target Caching/Software Indirect Jump Prediction

Software indirect jump prediction exploits the observation that indirect branch targets seldom or never change. By profiling the execution of the code the common destinations for an indirect branch may be recorded and used to directly access the translated block. The simulation can fall back to the emulation manager if the target is an address that has no translation. This can be achieved by adding a series of if statements that check the return address to the end of the function body [**1**]. A similar but more expensive operation at run time is to use the indirect branch target address to look up Target PC values which allows the translated branch target to be executed directly without returning to the emulation manager [**17**], [**16**].

3.2.4 Map Source Registers to Target Registers

A simple way to manage the translation of source registers is to map them to memory. However this causes significant overhead as each read or write from a register requires a memory operation. One way to avoid this overhead is map source registers directly to target registers for each register use in a basic block [**14**], or within a region [**8**].

3.2.5 Optimize Condition Code Handling

Simulating the condition code flags in memory incurs significant overhead in a similar manner to simulating registers in memory. It is possible to avoid this overhead if the effect of the instruction on the condition code flags in the source ISA matches the behavior in the target ISA [**8**].

3.2.6 Take Advantage of the Target ISA

In some case the target ISA contains powerful instructions that may map to several source ISA instructions. For example in the ARM v4 ISA the MLA maybe used to simulate the sequence of MUL and ADD instructions in the Intel x86 ISA. Patterns of instructions in the source ISA suitable to be mapped to a single target instruction may be found by constructing directed flow graphs for each basic block and the searching for sub-graphs that match the pattern that can be replaced by a single instruction [**18**].

3.2.7 Cache Decoded Instructions

In general there is little mention in the literature of the interpretation stage of DBT. However, one scheme for improving the efficiency of the interpretation stage is to cache the decoded instructions for future use, saving the cost of repeated decodes [**6**].

3.3 Alternatives to Dynamic Binary Translation

As discussed previously performing binary translation in a static context has a significant number of challenges. Combining static and dynamic binary translation can overcome the difficulties of creating a static translation and create a translation with performance that exceeds a typical dynamic binary translator.

3.3.1 Hybrid Static/Dynamic Translation

One approach to combining dynamic and static translation is to statically decode the program, adding instructions into a single translation unit until a branch to a location that cannot be statically determined is encountered. When the entire program has been translated and is executing, dynamic branches are profiled and if the destination of the branch is not in the current translated block a 'miss' is generated. If a sequence of instructions generates too many misses it is recompiled using the collected branch destination information [**11**].

3.3.2 Hybrid Instruction Set Compiled Simulation

Another approach to combining static and dynamic translation is to perform a static analysis to create a decoder for each type of instruction that is discovered in the program [**19**]. The compiler optimizes the decoder for each instruction type when the translator is compiled. The compilation time for the translator for the program is minimized because only the instructions discovered in the program have translators compiled, not the whole of the ISA. At run time instructions are executed one at a time giving the flexibility of interpretive simulation.

3.4 The ARM v4 Instruction Set Architecture and Binary Translation

The ARM ISA has several features that provide additional challenges for binary translation [**20**].

3.4.1 PC Relative Addressing

PC relative addressing is an instance of the code discovery problem. Large constants may be loaded from the text section by using the PC as the base address for the LDR (Load Register) instruction. The PC that is used must be the source PC. PC relative addressing may also be used to implement switch statements [**20**], [**17**].

3.4.2 Shifter Operand and Shifter Carry Out

The ARM v4 ISA provides several addressing modes for instruction operands, as described in section 4.1.2.8. While some of these modes generate an operand from a straightforward immediate or register value, others may apply one of several shift operations to generate the operand value. Instructions using an addressing mode that is performing a shift to calculate the operand value require additional code in the translation to perform the shift. Using this type of addressing in an instruction also sets the C flag if there is a Carry Out from the shifter operand. So translated code also has to provide a mechanism to set the C flag [**20**].

3.4.3 Condition Flags

Many instructions in the ARM ISA may set the condition codes, the comparison instructions, CMN, CMP, TEQ and TST, as well as arithmetic, logical or move instructions. Exactly how the flags are set depends on the instruction. Flag setting introduces considerable translation overhead to any instruction that sets the flags. The translator requires additional functionality to handle the specific flag setting for the various instruction groups [**20**].

3.4.4 Conditional Execution

Nearly all instructions in the ARM v4 ISA contain a conditional execution prefix as described in section 4.1.2.1. This prefix determines if the instruction will be executed based on the settings of the flags in the Current Program Status Register (CPSR), described in section 4.1.1.3. The action of checking the CPSR flags can introduce a considerable overhead to the translation of any particular instruction. The number of flags checked depends upon the particular condition code. Any instruction that has a condition code prefix other than "always" requires that the translator generate code that implements the condition code check. On some other architectures similar condition code mechanisms exist and there may be ways to use the target architectures condition code flags to emulate those of the source architecture, in place of generating a translation that checks the simulated CPSR [**8**].

4 “An Approach to Dynamic Binary Translation”

In this thesis we investigate the specific case of implementing a portable dynamic binary translator for the ARM v4 ISA. The translator is portable because it runs on both the Microsoft .NET Common Language Runtime (CLR) and the Mono open source implementation of the ECMA C# and CLR standards. Mono is supported on the Linux, OS X and Windows operating systems[2]. The translator implements the usual features of a DBT as well as some of the optimization techniques described in Chapter 3 Previous Work that are suited to CIL as a translation target, including:

- Large translation units
- Code layout changes
- Code duplication
- Conditional execution optimization
- Branch condition inversion

The ARMSim#[3] emulator previously developed in the Department of Computer Science at the University of Victoria was implemented in C# on Microsoft’s .Net platform. The interpreter extracted from ARMSim# provided a readily available starting point to develop a DBT from. Choosing to use the ARMSim# interpreter made using the same development environment the default choice. In turn, choosing to develop the DBT using CIL as the target constrained the implementation in other ways. Many of the techniques outlined in Chapter 3 Previous Work are not suited to an intermediate language as a translation target or cannot be readily implemented on a stack machine.

- Register mapping is not a useful technique in this context as there are no registers to map to.

---

[2] http://www.mono-project.com/Main_Page

[3] ARMSim# is an ARM7TDMI emulator that provides simulation of state including the 16 general purpose registers and the CPSR, the L1 Cache, both code and data, main Memory, and the Stack. ARMSim# also includes an assembler and a linker, so that when a file is loaded it is automatically assembled and linked [**40**].

- CIL does not contain powerful instructions that multiple ARM assembler instructions may be mapped to.

The approach taken in this thesis combines several techniques from Chapter 3. Instructions are translated in isolation and combined to form DBBs. DBBs are coalesced to form large translation units that span hot regions in the control flow graph.

The particular combination of translating to an intermediate language for a virtual machine and forming large translation units from hot traces does not seem to appear in the literature.

4.1 The ARM v4 Architecture

Implementing a translator for the ARM processor requires some understanding of the features of the device that a programmer may use. There are many versions of the ARM architecture. This thesis describes a translator for the ARM7TDMI processor, which implements the ARM v4 architecture. The ARM is a RISC processor [**21**] and as such implements these typical features:

- The instruction fields have fixed lengths and are uniform across different instructions. Instruction decoding is simplified because the same mechanism can be used to decode different instructions, wherever instructions share the same layout. For example all data processing instructions share the same addressing mode options in the shifter operand.
- Addressing modes are simple with the address for the load or store instructions being determined from a combination of values in registers and immediate values in the instruction.
- A large uniform register file, which means there are a large number of registers and they are all a uniform 32 bits.
- A load/store architecture, which means that instructions do not manipulate memory directly, rather instructions manipulate registers. Memory is loaded into registers, manipulated by instructions and stored from registers back to memory.

4.1.1 Programmer's Model

4.1.1.1 Modes

There are seven processor modes in the ARM architecture, for the purpose of this thesis we are only interested in User mode, which is normal program execution mode. The other six modes are generally only used by the operating system.

4.1.1.2 Registers

Although the ARM processor has 31 general-purpose registers, only 16 are visible to the programmer in user mode. The remaining non-visible registers are used for speeding up exception processing [**21**]. Exceptions are outside the scope of this thesis so I will not mention them further. The use of the registers is defined by the ARM calling convention [**22**] as follows:

- Registers 0 to 11 or 12 are available for the use of the programmer
- Register 12 is the intra procedure call scratch register (IP)
- Register 13 is by convention used as the stack pointer (SP)
- Register 14 is the Link Register (LR) used to hold the return address of the Branch and Link instruction, which is used when making a function call
- Register 15 is the program counter (PC)

The programmer may change the PC directly; this is sometimes described as the program counter being exposed [**17**]. Changing the PC directly is the equivalent of a branch instruction. However the PC should only be read or written according to the specified rules. [**21**][A2-7] ARM v4 and earlier have a three-stage instruction pipeline so the PC contains the address of the next instruction to fetch, two instructions after the instruction being executed. [4]This is important for calculating branch destinations when translating instructions [**21**][A1-5].

Each column in Figure 2 represents a single clock cycle, so in each cycle the processor:

- Fetches an instruction.
- Decodes the instruction fetched in the last clock cycle.
- Executes the instruction fetched two cycles ago.

[4] More recent versions of the architecture have deeper pipelines, but the 2 instruction offset is maintained for reasons of compatibility

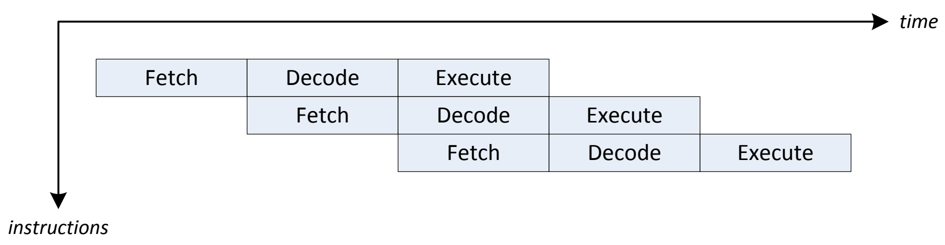

Figure 2 Three stage instruction pipeline (ARM 7 TDMI)

4.1.1.3 Current Program Status Register

The current program status register (CPSR) shown in Figure 3 is a special purpose register outside of the general-purpose registers 0 through 15. The bits of interest in the CPSR are the highest 5 and the lowest 8 (bits 8 through 26 are defined as Do Not Modify).

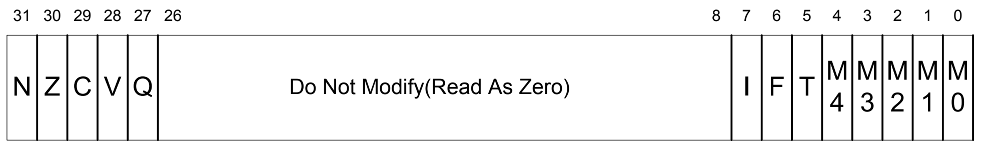

Figure 3 Format of the CPSR

Bits 31 to 28 are the condition code flags, N, Z, C and V. These flags are modified by comparison instructions, as well as arithmetic, logical and move instructions. Many of the arithmetic, logical and move instructions may be flag setting or flag-preserving depending on whether the instructions S bit is set. If the S bit is set:

- The N flag is set to 1 if the result of the instruction is negative when viewed as 2’s complement signed integer. Otherwise the N flag is 0.
- The Z flag is set to 1 if the result is zero; otherwise it is set to 0.
- The C flag is set in one of four ways depending on the instruction. The C flag is the carry flag, so if the operation would have produced a carry then the flag is set.
- The V flag is the overflow flag and is generally set if an operation causes a signed overflow, but it may be set for other reasons.

Bit 27 is the Q flag and is used in some variants of the ARM architecture that support Enhanced DSP Extensions.

The bottom 8 bits of the CPSR are known as control bits. The I bit and the F bit enable and disable interrupts, the T bit is set when in thumb mode which is an alternative operation mode supported by the ARM processor that uses 16 bit instructions. The M bits indicate the processor mode. The control bits are not modifiable in User mode code so they will not be discussed further.

4.1.1.4 Memory

In the ARM architecture v4 memory is a single address space of 2^32 8 bit bytes, with unsigned integer addresses from 0 to 2^32 – 1. This address space may also be viewed as 2^20 32 bit words aligned on 4 byte boundaries.

4.1.1.5 Data Types

The ARM processor itself supports 3 data sizes:

1. Word, which is normally aligned to 4 byte boundaries. The 4 bytes making up the word with the address A, are at A, A+1, A+2, A+3.
2. Half-Word, which is normally aligned to 2 byte boundaries.
3. Byte.

The data types are extended to include double word values when the floating point coprocessor is present and quad word values when the DSP unit is present.

4.1.1.6 Memory Alignment

The ARM processor operates most efficiently when data access is aligned, the address for a word is word-aligned and the address for a halfword is halfword-aligned. The assembler provides a directive for the programmer to align data. Alignment is not compulsory but there is a performance penalty for unaligned access.

4.1.2 Instructions

Version 4 of the ARM architecture supports two different instructions sets. By default instructions are 32 bits. However there is a second instruction set, called the Thumb instruction set which is a re-encoded 16 bit subset of the 32 bit instruction set. The Thumb instruction set is not in the scope of this thesis.

The Architecture Reference Manual [**21**] [A1-5] states “The ARM instruction set can be divided into six broad classes of instruction”:

- Branch instructions
- Data-processing instructions
- Status register transfer instructions
- Load and Store instructions
- Coprocessor instructions
- Exception-generating instructions”

4.1.2.1 Conditional Execution

Before considering the classes of instructions it is important to note that most instructions may be executed conditionally. The top 4 bits of nearly all instructions is the condition code. This code determines which flag or combination of flags from the CPSR must be checked in order for the instruction to be executed.

```
Loop:
CMP r1, #0
SUBGT r1, r1, #1
BNE loop
```

Figure 4 Conditional execution

The small loop shown in Figure 4 demonstrates conditional execution. The compare (CMP) instruction sets the flags in the CPSR based on the results of the comparison as follows:

- N flag is set to the MSB of the result of r1 – 0
- Z flag is set if r1 – 0 = 0
- C flag is set if r1 – 0 >= 0
- V flag is set if r1 – 0 generates a 32 bit signed overflow[5]

The SUBGT instruction is conditionally executed on checking the flags in the CPSR. For GT the flags are checked as follows:

- Z flag is clear

[5] Subtraction causes an overflow if the operands have different signs, and the first operand and the result have different signs.(Seal, 2000)

- N flag equals V flag

Before the BNE instruction is executed the flags in the CPSR are checked as follows:

- Z flag is clear

The default condition code prefix is *always* (mnemonic AL) which instructs the processor to execute the instruction regardless of the setting of the flags. The remaining 14 conditions cover:

- Tests for equality and non-equality.
- Tests for <, <=, > and >+ inequalities in both signed and unsigned arithmetic.
- Each flag to be tested individually.

[**21**][A1-5]

One purpose of conditional execution is to reduce the number of branch instructions. Reducing branches is beneficial because they cause the instruction pipeline to be flushed. A conditional instruction that is not executed is equivalent to a no operation (NOP) instruction [**21**].

4.1.2.2 Branch Instructions

The ARM processor has several different branch instructions as well as allowing direct setting of the program counter which has the same effect as a branch (There are very specific rules as to how the program counter may be changed).

1. B is the plain branch instruction.
2. BL is the branch and link instruction, effectively a subroutine call. Setting the L bit causes the instruction to save the return address in the link register (register 14). At the end of the subroutine the link register can be copied into the program counter to return from the subroutine.
3. BX is the branch and exchange instruction, this instruction branches to a destination that is held in a register. This instruction allows the option of changing into Thumb mode.

The other significant difference between 1 – 3 and 4 above is that 1 – 3 branch to an immediate value that is encoded in the instruction, whereas 4 is indirect and branches to a destination held in a register.

4.1.2.3 Data Processing Instructions

There are 16 data processing instructions, which perform operations including addition, subtraction, logical and, logical or etc. All the data processing instructions, except for move and move negative, take two source operands. The first source operand is always a register. The second source operand is called the shifter operand and is calculated using one of eleven modes. All the data processing instructions may update the condition code flags in the CPSR depending on whether the S bit is set. Exactly how the flags are set is specific to the instruction.

4.1.2.4 Status Register Access Instructions

These two instructions allow the value of the CPSR to be loaded into a register or updated from a value in a register.

4.1.2.5 Load and Store Instructions

Load instructions retrieve a value from memory into a register, store instructions save a value from a register to memory. The instructions come in two forms:

1. Operates on 32-bit word or 8-bit unsigned byte values
2. Loads or stores 16-bit unsigned halfwords and loads 8-bit byte or 16-bit halfwords with sign extension.

Similarly to the data processing instructions the load and store instructions have several addressing modes available to determine the memory address to read from or write to.

4.1.2.6 Coprocessor Instructions

The ARM processor has a set of instructions specifically for communication with coprocessors. These instructions fall outside the scope of this thesis.

4.1.2.7 Exception-Generating Instructions

The binary translation application described in this thesis makes minimal use of the software interrupt instruction (SWI) to terminate a program. The SWI instruction is normally used to allow user mode to transfer control to privileged Operating System code. Breakpoint (BKPT) is the other exception generating instruction; this instruction is not available in the translator described in this thesis.

4.1.2.8 Addressing Modes

In the ARM v4 architecture there are five addressing modes associated with five different classes of instructions.

- Mode 1 - Data processing operands
- Mode 2 - Load and store word or unsigned byte
- Mode 3 – Miscellaneous Loads and stores
- Mode 4 – Load and store multiple
- Mode 5 – Load and store coprocessor

4.1.2.9 Setting of CPSR Flags Depends on Specific Instruction:

The MOV instruction sets the flags in the following manner:

- N Flag = MSB of destination register
- Z Flag = if destination register == 0 then 1 else 0
- C Flag = shifter carry out
- V Flag unaffected

Whereas the SUB instruction does the following

- N Flag = MSB of destination register
- Z Flag = if destination register == 0 then 1 else 0
- C Flag = NOT *Borrow From*[6](operand a – operand b)
- V Flag = *Overflow From*[7](operand a – operand b)

However, how instructions set the flags fall into several groups so that it is possible to maintain a small collection of methods to generate the flag checking translations.

[6] "Returns 1 if the subtraction specified as its parameter caused a borrow (the true result is less than 0, where the operands are treated as unsigned integers), and returns 0 in all other cases. This delivers further information about a subtraction which occurred earlier in the pseudo-code. The subtraction is not repeated." (Seal, 2000)

[7] "Returns 1 if the addition or subtraction specified as its parameter caused a 32-bit signed overflow. Addition generates an overflow if both operands have the same sign (bit[31]), and the sign of the result is different to the sign of both operands. Subtraction causes an overflow if the operands have different signs, and the first operand and the result have different signs."(Seal, 2000)

This is a very brief outline of the ARM processor. For further information, refer to the ARM Architecture Reference Manual [**21**].

As discussed at the start of this chapter the previously implemented ARMSim# interpreter runs on Microsoft's .NET platform. This made the most obvious choice of target instruction set one of the dynamic code generation technologies supported on the .NET platform. The ability to easily generate and execute code dynamically also makes the .NET framework attractive as a translation target. The .NET framework provides several mechanisms for generating dynamic code:

1. CodeDOM
2. Reflection.Emit
3. ExpressionTrees

4.1.3 CodeDOM

The CodeDOM is a library of types that provide representations for many types of source code items. A multi-step process allows using this library to create assemblies that can be stored in memory (or on disk) and executed dynamically.

While CodeDOM has been used as a translation target for binary translation it's a poor choice if simulation performance is an important goal. CodeDOM requires the use of a high level language such as C# as the translation target and each source instruction may generate as many as 1500 native target instructions [**11**].

4.1.4 Reflection.Emit

Another possibility for generating code dynamically is the System.Reflection.Emit library. A major benefit of using IL as the translation target is that it allows access to all of the features of the CLR. Using IL it is possible to generate methods, classes and assemblies. Assemblies may be saved to disk, as well as executed dynamically.

However using CIL directly makes the job of the programmer considerably more difficult for two reasons. The first is that it is hard to ensure the correctness of the CIL, "… unfortunately Reflection.Emit makes it very easy to generate invalid IL …" and "But the correctness and validity of IL is a far more subtle matter" [**23**].

Because the emitted CIL is not available until runtime, the only way to investigate the validity of the generated code is to capture the emitted output using WinDBG or a similar console debugger.

The second challenge with emitting CIL directly is that while the ARM processor is a register machine, the CLR is a stack machine. This means that the process of translating ARM instructions to CIL is a two part process where the instruction must be translated to the equivalent CIL instruction or instructions, but also the register manipulation performed by the instruction must be translated to the equivalent stack machine sequence. This introduces a large amount of code overhead to each translated instruction that the programmer is responsible for[8].

ARM Asm **CIL**

```
sub r1, r2, r3        Ldarg_0                    // push reference to class instance
                      Ldfld, _registerInfo       // push ref to the registers array
                      Ldc_I4, (int)rd            // push index of rd
                      Ldarg_0                    // push reference to class instance
                      Ldfld, _registerInfo       // push ref to the registers array
                      Ldc_I4, (int)rn            // push index of rn
                      Ldelem_U4                  // push the value from rn
                      Ldarg_0                    // push reference to class instance
                      Ldfld, _registerInfo       // array info
                      Ldc_I4, (int) rm           // push index of rm
                      Ldelem_U4                  // push the value from rm
                      Sub                        // perform subtraction
                      Stelem, typeof(uint)       // store element in rd
```

Figure 5 Stack manipulation overhead

Figure 5 shows the translation of a simple subtraction instruction to CIL. Translating to the stack machine requires inserting instructions to configure the stack from the simulation context block, performing the desired operation and storing the result from the stack back to the context block. The stack machine nature of this code can be seen in that the initial items pushed onto the stack are the register array and the index of the register in which to store the result, and the last instruction stores the result.

An obvious limitation of translating one instruction at a time is that the stack is reconfigured for each instruction, generating overhead when the results from one instruction and that state of the stack may be useful to the next instruction – hence the idea of translating blocks of instructions

[8] See Appendix C for an example of code expansion due to simulating register access and CPSR flag setting and checking.

rather than isolated instructions. In practice this may require two passes to generate the IL, removing redundant store and load instructions on a second pass.

The emitted CIL is used to can be used in various ways. Used in the most minimal way it can generate a dynamic method, which is stored in memory and discarded when it is no longer required. Alternatively it can be used to generate an assembly containing modules, types and methods that may be saved to disk and used outside the scope of the currently executing program.

In contrast with the high level code generated by the CodeDOM the IL generated by Reflection.Emit is at a similar level of abstraction to the source ARM assembly code. The number of instructions required to emulate a specific ARM instruction may be significant, especially if the condition codes are involved but it still generates many, many fewer instructions than the CodeDOM [**11**].

4.1.5 Dynamic Language Runtime

The Dynamic Language Runtime (DLR) [**24**] is a library that sits on top of the CLR and provides a set of runtime services for supporting dynamic languages[9]. These services include:

- Expression trees which are used to represent the semantics of a programming language.
- Call site caching, which is a mechanism where operands and operations are cached so that if the same operation and operand types are seen subsequently the cached version can be used. This saves repeatedly looking up the same method.
- Dynamic object interoperability, a set of classes and interfaces is provided that enable representation of dynamic objects and operations by language implementers.

4.1.5.1 Expression Trees

Expressions and Expression Trees form the basis of the translation mechanism in the binary translator described in this thesis. Expressions are implementations of the abstract Expression class. The System.Linq.Expressions namespace contains a large number of Expression classes that implement various programming language constructs. It also includes an Expression class that provides static factory methods to create the various node types. E.g. The Expression.Add

[9] Examples include Ruby, Python, Lisp, Smalltalk, Lua and many more.

factory method creates an instance of the BinaryExpression class that represents an addition operation without overflow checking. A key feature of expressions is that any operands of an expression are themselves expressions. This enables representing any source code as a tree of expression objects. An additional advantage to using the Expression class is that the class manages the IL stack transparently.

An expression tree is a tree of objects where each node in the tree is an instance of an Expression object. An Expression tree is "also a representation of a program that can be manipulated at runtime" [**25**].

For example Figure 6 shows a simple C# assignment translated to an Expression tree.

C# **Expression tree**

```
_cFlag = 1;                 Expression.Assign(
                            Expression.Field(Expression.Constant(this), _cFlagInfo),
                            Expression.Constant(1))
```

Figure 6 Translation from C# to Expression tree

To obtain an expression that represents the field in the class, the Expression.Field method requires a reference to the class that contains the field, which is supplied by Expression.Constant(this). The field itself is accessed via a FieldInfo object, which has been created previously. Expression.Constant(1) supplies the value to assign to the field and the Expression.Assign on the outside performs the assignment of the constant to the field.

The expression syntax compares favorably to the CIL syntax for the same operation shown in Figure 7.

C# **CIL**

```
_cFlag = 1;                 Ldarg_0             // push a ref to the class
                            Ldc_I4_1            // push the value to store
                            Stfld, _cFlagInfo   // store value into field
```

Figure 7 Translation C# to CIL

Despite the fact that the expression tree code is making 4 calls to CLR API functions there is little or no difference in the performance between the two implementations. This was determined empirically by writing a small test program to repeatedly assign a value to a field. The variability between runs exceeds the difference between the two implementations. See Appendix B for a small test program demonstrating this.

To execute an Expression tree, it is compiled into a delegate, which can then be called. A delegate is a type safe callback mechanism, or put more simply 'a handle to piece of code that can be called' [**25**]. The delegate provides C# with a mechanism that is similar to a function pointer in C/C++. A delegate can also be defined as a class that holds a reference to a method. The signature of the delegate must match the signature of the method.

The delegate may be:

- Created at runtime.
- A custom delegate provided by the programmer.
- One of the generic delegate types provided by the runtime. Two categories of generic delegate types are available, Func<T> delegates which return a value and Action<T> delegates which do not.

The CLR executes the delegate by calling its invoke method. A delegate can have several methods chained to it, both static and instance methods.

```
var assignDelegate = Expression.Lambda<Action>(assignExpression).Compile();
assignDelegate();
```

The Compile method generates CIL that is available to be JIT compiled when the delegate is called.

One of the primary purposes of expression trees is to implement dynamic programming languages. This also makes them ideal for creating a DBT. Both IronRuby[10] and IronPython[11] are implemented on the DLR, and work in a similar manner. Both use a tokenizer and a parser to generate an abstract syntax tree that is either an expression tree in the case of IronPython, or converted into an expression tree in the case of IronRuby. In these dynamic languages the expression trees are first interpreted, to reduce the start up time, before being compiled.

[10] http://ironruby.codeplex.com
[11] http://ironpython.net

In the case of DBT on the CLR the Expression tree simplifies implementing the instructions from the target language because there is no need to translate from the register architecture of ARM assembler to the stack architecture of the CLR's IL. The JIT compiler manages the stack.

4.1.5.2 Nested and Block Expressions

The Expression tree syntax becomes difficult to read once expressions exceed a certain size, however the combination of the fact that most Expression methods take at least one Expression parameter and that the Block Expression allows expressions to be grouped means that simple expressions may be combined to form more complex ones.

```
var accessR1 = Expression.ArrayAccess(
                    Expression.Field(Expression.Constant(this), _registerInfo),
                    Expression.Constant(IndexR1));
var assignR1 = Expression.Assign(accessR1, Expression.Constant(7));
var accessR2 = Expression.ArrayAccess(
                    Expression.Field(Expression.Constant(this), _registerInfo),
                    Expression.Constant(IndexR2));
var assignR2 = Expression.Assign(accessR2, Expression.Constant(8));
var block = Expression.Block(assignR1, assignR2);
```

Figure 8 Block expression

4.1.6 How the Dynamic Code is Executed

Both Reflection.Emit and Expression trees can be used to add methods to the existing assembly. These methods are compiled to native code by the CLR's just-in-time compiler when the method is called. The CodeDOM is a much less dynamic approach requiring the creation of an assembly containing a class.

4.2 CLR Optimization

A significant benefit of targeting a runtime such as the CLR is the optimizations that the runtime itself provides. The optimizations provided by the runtime also improve as new versions are released. There have been significant improvements to the CLR between .NET 2.0 and .NET 4.5.

This is not an exhaustive list, but rather some of the documented optimizations included in the CLR JIT compiler [**26**], [**27**]:

- Multi-core JIT [**27**] Added in .NET 4.5, when an application is started a background thread running on a different processor is also started to perform JIT compilation. The goal of running the compilation on a separate thread is to avoid the compilation overhead in making method calls. When a method is required in the application, it has already been compiled on the background thread. To know which methods are required by the application and in what order the application generates an execution profile that is stored. The execution profile is used by the background thread to determine which methods to compile and the order in which to compile them.

- Method inlining. The method call is replaced by the body of the method. The CLR has many rules governing which methods may be in-lined, which vary with the specific version of the CLR. The trend has been to allow more inlining of methods between the initial release of the CLR and the latest .NET 4.5 release.
- Assertion propagation (what is known to be true), this is a generalization of constant propagation.
- Constant folding, e.g. y = 3 * 2 is changed to y = 6 by the compiler.
- Branch straightening via prediction which is reordering of code so that the likely path taken in a branch is moved to the fall through position [**26**].
- Tail call, if the last code element in method1 is a call to method2 call then the stack frame of method1 may be reused by method2 rather than adding another frame to the stack. This optimization may be shared between the language to IL compiler and the JIT IL to native compiler by the language compiler creating IL code that the JIT compiler is more likely to apply the tail call optimization to [**28**].
- Value type handling has been improved by implementing "value type scalar replacement" which is essentially converting value types that are suitable into a collection of scalars. Other improvements have been made that allow methods with value type parameters, local variables or return value to be inlined [**29**].

4.3 Challenges

There are three major challenges to resolve in order to implement a DBT for ARM v4 ISA with CIL as a target.

4.3.1 Hardware Simulation Challenges

At first glance the translation from one instruction set to another might seem quite straightforward, however in the case of ARM v4 assembler to the System.Linq.Expression syntax there are a number of challenges. Many of these challenges are simply challenges of dealing with code expansion; a single ARM instruction may expand into tens of Expressions. This code expansion is generally driven by the need to simulate features of the ARM v4 ISA in software:

- Since CIL is executed on a stack machine the registers of the ARM processor must be simulated.
- Many ARM instructions make use of addressing modes that are encoded in the instruction and the calculations performed by the addressing mode must be simulated.
- Simulating the CPSR flags and the setting of the flags.
- Accessing the simulated memory block.
- Simulating conditional execution.

See Appendix C for an example showing code expansion in a translated block.

4.3.2 When and What to Translate

Translation is expensive in terms of the computation time required to translate the code, so there is little point in translating code that is unlikely to be executed enough to recoup the cost of the translation. After a DBB is discovered it isn't translated until some decision is reached regarding the expected pay back from translating the block. Various mechanisms have been used to decide when to translate from a simple execution count [**7**], to a heuristic based on the block size and the execution count [**6**]

4.3.3 Translating Program Flow

A very similar problem is deciding which regions of an executing program are hot and worth creating large translation units for. In the case of forming large translation units (LTU) there is the additional challenge of discovering which code should be included in the LTUs. Forming LTUs also opens up other optimization possibilities such as code reordering. Forming LTUs may also be regarded as translating the program flow. When the unit of translation is the DBB the emulation manger controls the flow of the executing translation. At the end of each DBB control is transferred back to the emulation manager to determine which block to execute next. The formation of LTUs requires that this flow of control be translated. An additional challenge to maximizing the performance of LTUs is to ensure that the translation doesn't become so large that it adversely affects the JIT compilation time.

The final challenge of translation is purely a challenge for the programmer, ensuring that the generated translation is an accurate representation of the source program.

5 Implementing a Dynamic Binary Translator

Faculty and students in the CS department at the University of Victoria developed the ARMSim# ARM simulator, which runs on the .NET framework. This simulator is based on a simple fetch and decode interpreter, and as mentioned previously the decoder from the interpreter makes an ideal starting point for implementing a dynamic binary translator. Many additional components were required to create a functioning DBT.

5.1 Hardware Simulation

While the ARMSim# emulator contains a simulation of the processor state the implementation is tightly coupled to the graphical user interface. For this reason the registers and memory of the processor had to be implemented in a manner suitable for use in a translator.

5.1.1 Registers

Since the CLR is implemented as a stack machine there are no registers available to map the registers of the ARM architecture to. The registers of the ARM architecture must be simulated in a Context Block, described previously. A simple approach to implementing the context block is to make it globally accessible. Translated data instructions may access the global context block for many reasons:

- Loading data from source registers.
- Storing data in destination registers.
- Accessing the program counter to calculate offsets for immediate addressing.

The System.Linq.Expression namespace provides a mechanism to access an array by index. This provides a way to implement translation of the actions of data transfer instructions on the simulation of the ARM processor's registers. For example:

```
Expression.ArrayAccess
(
        Expression.Field(
                Expression.Constant(this),
                 _registerInfo),
        Expression.Constant(rd)
)
```

Figure 9 Accessing simulated registers

The Field expression on Line 3 provides the ArrayAccess expression on line 1 with the array to access. The Constant expression on line 4 is a reference to the object that the Field is a member of. The variable _registerInfo on line 5 is an instance of the System.Reflection.FieldInfo class that holds meta-data about the _registers array. This meta-data is used to access the array via Expression syntax. Fields may also be accessed by name. The second Constant expression on line 6 is the index of the element to retrieve, where the value of the index is supplied by a local variable, rd.

This Expression.ArrayAccess is used to obtain the value of a register for use as an input in another expression or as an assignment target when creating an assignment Expression.

5.1.2 Memory

The source program to execute must be stored in memory to allow access to the code and data. This is implemented as a simple array of bytes. To access the program code and data requires mapping from a source address to a target address, which is an array index value. This can be summarized as:

Index = (instruction address – program base address) >> 2

An expression tree to perform this mapping is shown in Figure 10.

```
1.Expression.ArrayAccess(
2.     Expression.Field(Expression.Constant(this), _programInfo),
3.            Expression.Convert(
4.                   Expression.RightShift(
5.                          Expression.Subtract(instructionAddr,
6.                                 Expression.Constant(_baseAddress)),
7.                          Expression.Constant(2)),
8.                   typeof(int))
9.);
```

Figure 10 Expression syntax mapping a source address to a target address

The Expression.ArrayAccess on line 1 accesses the array that stores the program, the variable _programInfo is similar to the _registerInfo variable above. The Expression.Convert on line 3 generates the index into the array. It converts the unsigned integer generated by the Expression.RightShift into a signed integer. The conversion is required because the .NET CLR doesn’t permit an array to be indexed by an unsigned integer.

The arrays used to model memory and registers could have been implemented using the System.Collections.Generic.Dictionary class. This was investigated because it removed the requirement to map the SPC to the TPC, but the impact on simulation performance was significant.

5.1.3 Translating Instructions

Translating an instruction requires that we generate an expression to:

1. Evaluate the condition code evaluation if the instruction is conditional.
2. Increment the source program counter.
3. Update the simulation state.
4. Set the condition codes if required.

Not all instructions require all steps. Many instruction translations simply increment the program counter and update the simulation state. For example:

```
SUB R3, R2, R1
```

Translates to the following in Expression syntax:

```
Expression.Block(
   Expression.AddAssign(
         Expression.ArrayAccess(
               Expression.Field(Expression.Constant(this), _registerInfo),
               Expression.Constant(ProgramCounter)),
         Expression.Constant((uint)4))
   Expression.Assign(
         Expression.ArrayAccess(
               Expression.Field(Expression.Constant(this), _registerInfo),
               Expression.Constant(rd)),
         Expression.Subtract(
               Expression.ArrayAccess(
                     Expression.Field(
                           Expression.Constant(this), _registerInfo),
                     Expression.Constant(rn)),
               Expression.ArrayAccess(
                     Expression.Field(
                           Expression.Constant(this), _registerInfo),
                     Expression.Constant(regM))
   ));
```

Figure 11 A simple instruction translation

Lines 2 through 6 form an expression to increment the program counter.

Lines 7 through 20 store the result of the subtraction expression in the simulated register, using the value of the local variable rd to index into the array that simulates the ARM processor's registers.

5.1.3.1 Indirect Branches

Indirect branches are those branches whose destination is not known until runtime. For example:

```
MOV PC, R2
```

The address loaded from R2 is a SPC address so it must be mapped to a TPC address.

```
1.Expression.Assign(
2.    Expression.ArrayAccess(
3.          Expression.Field(Expression.Constant(this), _registerInfo),
4.                Expression.Constant(pc))),
5.    Expression.ArrayAccess(
6.          Expression.Field(Expression.Constant(this), _programInfo),
7.                Expression.Convert(
8.                      Expression.RightShift(
9.                      Expression.Subtract(Expression.ArrayAccess(
10.                           Expression.Field(Expression.Constant(this),
11.                                 _registerInfo), Expression.Constant(r2)),
12.                           Expression.Constant(_baseAddress)),
13.                     Expression.Constant(2)),
14.               typeof(int))
```

Figure 12 Indirect branch translation

The indirect branch translation combines a simulated memory access with manipulating the simulated registers. The Expression.Assign on line 1 is the direct translation of the move instruction, assigning the value from the address in R2 to the PC. The nested expressions are necessary "boilerplate" code to access the simulated state. The memory access code duplicates the code in Figure 10 and the register access code duplicates the code in Figure 9.

5.1.4 Flag Setting and Condition Codes

5.1.4.1 Conditional Instructions

As discussed previously ARM instructions may be executed conditionally which means that the condition check must be translated. A simple optimization is to only generate flag checking code for those instructions that have a condition code other than always.

```
ARM               Expression Tree

MOVNE R3, R2      Expression.IfThen(
                        Expression.NotEqual(
                              Expression.Field(
                                    Expression.Constant(this),
                                    "_zFlag"),
                              Expression.Constant(1)),
                        Expression.Assign(
                              Expression.ArrayAccess(
                                    Expression.Field(
                                          Expression.Constant(this),
                                          "_registerList"),
                                    Expression.Constant(R3Index)),
                              Expression.ArrayAccess(
                                    Expression.Field(
                                          Expression.Constant(this),
                                          "_registerList"),
                                    Expression.Constant(R2Index))))
```

Figure 13 Translation of instruction with conditional execution.

Figure 13 illustrates the translation of a conditionally executed instruction. The simulated registers exist in an array that is an instance variable of the class that holds the simulated state. The indices of the accessed registers are passed into the method generating the translation as the variables R2Index and R3Index. NE or not equal is a simple translation as it checks only the Z flag from the CPSR, but the general case of translating a conditionally executed instruction is similar. One thing that can be seen from this example is the code expansion where 14 Expressions are required to translate the single ARM instruction, this is one of the challenges of translating assembler, it is very terse and the code expansion is challenging for the programmer to manage.

5.1.4.2 Flag Setting

Setting the simulated CPSR causes code expansion in a similar manner to checking the flags in the previous section. As described previously, how the flags are set is dependent on the specific instruction, but all are translated in a similar manner. It is also possible to extract and reuse the flag setting code, so that different instructions may use the same flag setting code. The SUB instruction with the S bit set, sets the flags as described in 4.1.2.9, its translation is shown in Figure 14.

ARM Expression Tree

ARM **Expression Tree**

SUBS R2, R1, #1

```
// set up operands for instruction
var operandA = Expression.ArrayAccess(
             Expression.Field(Expression.Constant(this),
      _registerInfo), Expression.Constant(indexR1)

var operandB = Expression.Constant(1);

// set up destination to store result
var result = Expression.ArrayAccess(
             Expression.Field(Expression.Constant(this),
                    _registerInfo),
             Expression.Constant(indexR2));

// perform subtraction and store result
var Reg2 = Expression.Assign(result,
             Expression.Subtract(operandA, operandB));

// set the simulated C flag
var setC = Expression.IfThenElse(
      Expression.LessThan(operandA, operandB),
      Expression.Assign(
             Expression.Field(
                    Expression.Constant(this),_cFlagInfo),
             Expression.Constant(0)),
      Expression.Assign(
             Expression.Field(
                    Expression.Constant(this),_cFlagInfo),
             Expression.Constant(1)));

// set the simulated Z flag
var setZ = Expression.IfThenElse(
      Expression.Equal(result,
             Expression.Constant((uint)0)),
      Expression.Assign(
             Expression.Field(
                    Expression.Constant(this),_zFlagInfo),
             Expression.Constant(1)),
      Expression.Assign(
             Expression.Field(
                    Expression.Constant(this),_zFlagInfo),
             Expression.Constant(0)));

// set the simulated N flag
var setN = Expression.Assign(
      Expression.Field(
             Expression.Constant(this),
             _nFlagInfo),
      Expression.Convert(
             Expression.RightShift(
                    result,
                    Expression.Constant(31)),
             typeof (int)));
```

ARM Expression Tree

```
// set the simulated V flag
var setV = Expression.IfThenElse(
      Expression.And(
            Expression.NotEqual(
                  Expression.RightShift(
                        operandA, Expression.Constant(31)),
                  Expression.RightShift(
                        operandB, Expression.Constant(31))),
            Expression.NotEqual(
                  Expression.RightShift(
                        operandA, Expression.Constant(31)),
                  Expression.RightShift(
                        result, Expression.Constant(31))),
      Expression.Assign(
            Expression.Field(
                  Expression.Constant(this), _vFlagInfo),
            Expression.Constant(1)),
      Expression.Assign(
            Expression.Field(
                  Expression.Constant(this), _vFlagInfo),
            Expression.Constant(0)));

// combine the instruction and flag setting translations
var translation = Expression.Block(Reg2, setC, setZ, setN, setV);
```

Figure 14 Translation of flag setting instruction

The overhead introduced by setting the flags is obviously significant. ARM code that contains many flag setting instructions may prove challenging to translate with good performance.

When creating large expressions such as this it is easy for them to become deeply nested and for their meaning to be obscured. The use of Expression.Block in combination with assigning comprehensible expressions to temporary variables and creating functions for frequently used expressions makes the code readable.

5.1.5 Register to Stack Machine

As outlined in section 4.1.4 dealing with the stack machine nature of the CLR is challenging when emitting CIL directly using the Reflection.Emit class. The use of the System.Linq.Expression class to generate dynamic methods overcomes this difficulty and frees the developer from having to manage the state of the CLR's stack. The remaining issue with dealing with the stack machine is generating the code to simulate the memory and register access as outlined above in sections 5.1.1 and 5.1.2.

5.2 What to Translate

The ARMSim# interpreter forms the basis for this investigation of binary translation. However, the ARMSim# simulator runs inside a graphical user interface, which makes it unsuitable for measuring performance. A message pump thread drives the GUI, running a loop. This interferes with measuring the performance of the interpreter in isolation. Instead the interpreter modules were extracted into a standalone application in order to enable more precise measurements.

The interpreter loop extracted from ARMSim# works as follows. Program code and data is loaded into a simulated memory area. The entry point is specified by an assembler directive and defaults to the start of the text area if it is not specified. The first instruction is fetched from this address. The program counter is incremented to point to the next instruction. The condition code prefix of the loaded instruction is checked against the simulated CPSR (current program status register) and if the instruction is to be executed it is passed to the decoder. The instruction is decoded and the result is applied to the simulation of the state of the target architecture, called the Context Block by [**1**]. Each instruction is dispatched and decoded every time it is executed. When the state has been updated the loop starts again, loading the instruction that the program counter is pointing to. The interpreter proceeds in this fashion until the execution halts. Interpretation in the simulator halts when an SWI instruction is encountered.

Many of the previously discussed DBT systems do not translate instructions until they are profiled and determined to be "hot" according to some metric. As mentioned previously, translators generally perform translation on blocks of instructions rather than on individual instructions in isolation, which requires that metadata is collected during execution to determine which blocks are worth translating. The initial unit of translation in the DBT described in this thesis is the dynamic basic block.

5.2.1 Discovering Dynamic Basic Blocks

Dynamic basic blocks are discovered as the program executes in the interpreter. The DBB ends when a branch or SWI instruction is encountered. To manage this process there is a nested emulation loop where the outer loop controls interpreting blocks and the inner loop manages interpreting the individual instructions within the block. For example:

```
while (!at end of program){
      if(this instruction is not recorded as starting a block){
            record the start of a block
            newBlock = true
      }
      while (!at end of block){
            interpret instruction
      }
      if(newBlock = true){
            record the end of the block
            add the block to the dbb list
      }
}
```

Figure 15 Dynamic basic block discovery algorithm

When the program exits a list of all the dynamic basic blocks that were executed has been generated. The entry address and exit address of each dynamic basic block are recorded in this list. Translation is expensive so we want to set a threshold for the number of times a block should be interpreted before it is translated. We add a counter to track the number of times a block has been interpreted; if the block count exceeds some threshold then the block is translated. For example:

```
increment block execution count
if(this instruction is not recorded as starting a block){
      Record the start of a block
      New Block = true
}
while(!at end of block){
      if(block execution count exceeds threshold){
            interpret and translate instruction
      }
      else{
            interpret instruction
      }
}
if(block execution count exceeds threshold) {
      Create translated block
}
```

Figure 16 Algorithm to decide when to translate a block

5.2.2 Translation

To translate a dynamic basic block the translated instructions in the block are combined into a BlockExpression. Then an Action delegate is compiled from the BlockExpression and stored in the translated block cache. For example:

```
BlockExpression block = Expression.Block(…);
translatedBlock = Expression.Lamda<Action>(block).Compile();
```

Figure 17 Compiling a translation

The translated block cache is implemented as a dictionary using the starting addresses of the translated blocks as keys. Control is returned to the emulation manager at the end of each translated block. Each translated block ends with a branch instruction, which sets the value of the Program Counter based on the condition test. When control returns to the emulation manager it checks the translated block cache using the Program Counter as a key. If the block address given by the Program Counter is a hit in the translated block cache the translation is executed, otherwise the interpreter executes the block. As program execution continues the proportion of translated blocks to interpreted blocks increases until the majority of instructions are being executed via their translations. Although the simple DBT has significant performance gains when compared to the interpreter it is mostly useful as a starting point for investigating mechanisms for further speeding up program execution. This basic DBT is similar to "Gear 1" described by [**8**] or QEMU [**14**].

Another reason that this may be considered a simple DBT is that data instructions are translated but control flow instructions are not.

5.2.3 When to Translate

Deciding when to translate is difficult because it requires looking into the future to see how many times an instruction will be executed. A pragmatic approach is to assume that once a block has been executed some number of times that it is likely that it will be executed many more times and therefore is worth translating. This approach is taken in many of the translators described in Chapter 3 [**7**], [**6**] and is the approach taken in this thesis.

5.3 Translating Program Flow

A translator that uses basic blocks as its final translation step is in a sense a hybrid. While all code that is executed may be translated code, between blocks it behaves much like an interpreter, dropping back to the emulation manager to determine which block to execute next.

5.3.1 Creating Large Translation Units

Executing the simulator using the profiler in Visual Studio 2010 Ultimate shows that the simple DBT described previously spends most of its execution time in the emulation manager accessing the translated block cache. One way to reduce this overhead is to implement a mechanism that coalesces hot blocks into hot regions and reduces the number of times that the simulation returns to the emulation manager. To determine which blocks in the program are hot requires profiling the execution of the program.

An ‘edge profile’ is created by recording the number of times each edge that leaves a block is taken. Since all blocks end with a branch instruction (disregarding the SWI instruction) there can only be one or two edges leaving a block. The edge profile records the starting block, the ending block and the number of times that the edge was taken. When execution of a block is complete, and before returning to the emulation manager, if the profile has an entry the edge the count is incremented, if there is no entry for the edge an entry is created and incremented. As pseudo code:

```
while(!atEndOfProgram) { // main emulation loop
    …
    if (!ProfileContainsEdge(blockStart, nextBlock)){
        AddEdgeToProfile(blockStart, nextBlock);
    }
    IncrementProfileCountForEdge(blockStart, nextBlock);
```

Figure 18 Profiling algorithm

Profiling the program execution and recording the number of times each edge between blocks is traversed generates a directed weighted graph. This graph is stored in a .NET dictionary (a hash table) with block starting addresses as keys and a dictionary for the value. The nested dictionary uses the addresses of adjacent blocks as keys and the number of times the edge has been traversed as the value.

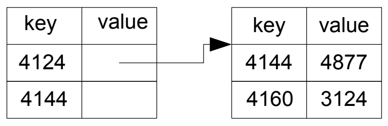

Figure 19 Adjacency list

Figure 19 shows a node in the graph, as it is stored in the hash table/dictionary. The block at address 4124 has 2 edges that leave it, to the blocks at addresses 4144 and 4160. The edge to 4144 has been traversed 4877 times and the edge to 4160 has been traversed 3124 times. The whole edge profile of a small program such as bubble sort may be displayed in a graphic form, for example:

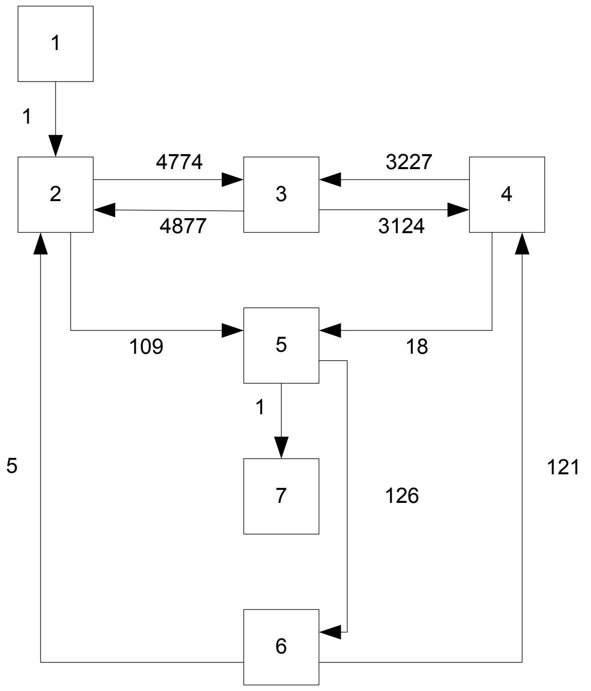

Figure 20 Directed weighted graph for bubble sort program

Although creating the edge profile is more involved than creating a block profile, it makes discovering hot regions in the profile more straightforward. Each time an edge count is incremented it is tested against a threshold value to determine if the edge is hot. When an edge is found to be hot, the address of the block with the hot edge is recorded. When the number of hot edges exceeds some threshold value, a hot region may be created. In pseudo code:

```
while(!atEndOfProgram) // main emulation loop
{
        …
        if (!ProfileContainsEdge(blockStart, nextBlock)) {
                AddEdgeToProfile(blockStart, nextBlock);
        }
        IncrementProfileCountForEdge(blockStart, nextBlock);
        if(EdgeIsHot(blockStart, nextBlock)) {
                AddToBlocksWithHotEdgesList(blockStart, nextBlock);
                hotEdgeCount++;
        }
        If(hotEdgeCount > hotRegionThreshold) {
                CreateHotRegion();
        }
```

Figure 21 Extended profiling algorithm to detect and create a hot region

5.3.1.1 Forming Hot Regions

Creating larger translation units requires translating the control flow instructions in addition to the already translated data handling instructions. The previously translated and compiled blocks could be combined to form regions. However the last instruction in each of these blocks sets up the program counter for return to the emulation manager. In the context of the hot region this instruction is likely to be both redundant and incorrect. Instead the translations of the individual instructions, which remain in the instruction translation cache, can be combined to form regions containing control flow. This assumes that instructions have not been modified since they were translated, so the translator has the limitation of no self-modifying code.

5.3.1.1.1 Labels and Gotos

The System.Linq.Expressions namespace includes Expressions for control flow including Expression.Label and Expression.Goto. An Expression.Label is inserted to provide a destination for control flow to branch to. An Expression.Goto takes an Expression.Label parameter whose location it transfers control to. For example:

```
Expression.Block(
      Expression.Label("start"),
      Expression.Expression.SubtractAssign(
            Expression.Field(Expression.Constant(this), _someValue),
            Expression.Constant(1)),
      Expression.IfThen(
            Expression.Equal(
                  Expression.Constant(0),
                  Expression.Field(Expression.Constant(this), _someValue)),
            Expression.Goto("start")
      )
)
```

Figure 22 An Expression.Block implementing a loop

To create the hot region, control flow is required within the region and to exit the region. When the interpreter first sees a block, an Expression.Label is created for the block and stored in a hash table using the block start address as the key. When creating the hot region these previously stored labels are inserted into the start of each block when the block is translated. As well, the labels are available to use in Expression.Goto, even when the destination block has not yet been added to the hot region translation. Similarly, to be able to insert Expression.Goto instructions that exit the region, an Expression.Label is required to supply to the Expression.Goto. This

Expression.Label("exit") is created at the beginning of creating the hot region translation, and inserted as the last expression in the Expression.Block that implements the hot region. An additional implication here is that when an Expression.Goto("exit") is encountered the PC must be set correctly.

5.3.1.1.2 Sub-graph or Connected Hot Blocks

Similar to the approach taken by other researchers [**12**] this thesis makes use of Tarjan's [**30**] strongly connected components (SCC) algorithm. The output from Tarjan's SCC algorithm is a collection of lists each containing the addresses of a set of blocks that are strongly connected. A block is strongly connected to another block if they are in the same directed cycle in the control flow graph.

This thesis relied on a simple threshold where once n edges had been traversed x times Tarjan's algorithm was run to find the SCCs. The components that contain one or more hot edges and more than one dynamic basic block are used to create translated regions.

Figure 23 below shows the SCCs from the directed weighted graph for the bubble sort program, inside the dotted line are the hot edges and blocks.

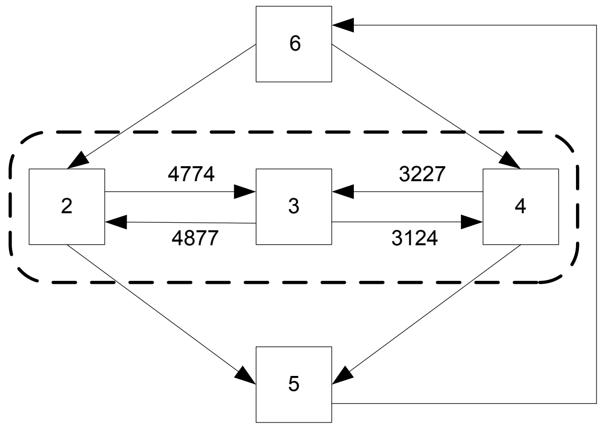

Figure 23 SCCs from bubble sort directed weighted graph

Combining the lists of SCCs and the translated instructions in the translated instruction cache, we have sufficient information to create translated hot regions.

5.3.1.1.3 Translating Branch Instructions

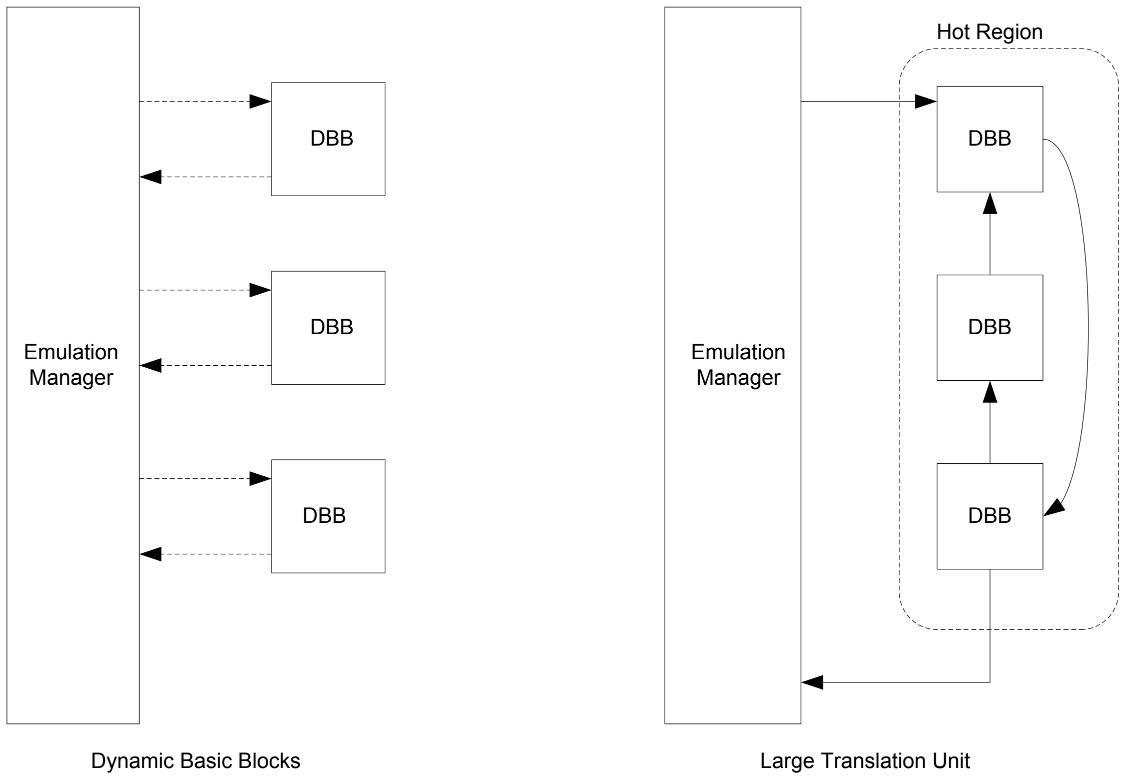

Figure 24 Control flow comparison between DBB translation and LTU translation

To form a LTU the branch instructions that end every DBB that is included in the LTU must be translated in the context of the LTU rather than simply returning to the emulation manager. To add a translated block to the region, we insert a label for the start of the block, and then add all of the instructions in the block up to the terminating branch instruction. The branch instruction can only be added when it has been analyzed to determine which of the 4 categories it falls into.

1. Neither edge leaving the block is hot:
   - Add the existing translated branch instruction and a goto expression to exit the region. The translated branch instruction from the translated instruction cache sets the program counter correctly.
2. The branch condition true edge is hot and the branch condition false edge is not:
   - If the block is the last block in the sub-graph list, insert a goto expression back into the region and fall through to the region exit.

   - If the next block in the sub-graph list is the branch target/hot edge, invert the branch condition so that control flow falls through on the hot edge and insert a goto expression to exit the region.
   - If the next block in the sub-graph list is not in the control flow for this block, then the branch target/hot edge requires a goto expression to a labeled block in the hot region, and the fall through edge requires a goto expression to exit the hot region.
3. The branch condition false edge is hot and the branch condition true edge is not:
   - If the block is the last block in the sub-graph list, invert the condition, set up a goto expression back into the region, and fall through to the region exit.
   - If the next block in the sub-graph list is the fall through/hot edge don't modify the instruction at all, set up a goto expression to exit the region if the condition is true and fall through on the hot edge.
   - If the next block in the sub-graph list is not in the control flow for this block, then the branch target requires a goto expression to exit the region and the fall through requires a goto expression to a labeled block in the hot region.
4. Both edges leaving the block are hot:
   - If the block is the last block in the sub-graph list, insert a goto expression for each edge back into the region.
   - If the next block in the sub-graph list is the branch target, invert the branch condition to fall through to the branch target instead and insert a goto expression to the original fall through address.
   - If the next block in the sub-graph list is the fall through address, then translate the control flow normally generating a goto expression for the branch target.

Notable in previous description is the optimization of inverting the branch conditions to reduce the number of goto expressions required to implement the translation. Reducing the number of goto expressions that are within the hot region has significant performance benefits. By inverting the branch conditions for branches with one hot edge the goto expressions can be confined to the cooler edges leaving the hot region. Inverting branch conditions also means that the code layout is no longer in the same order as the source program [**1**].

The hot region replaces the entry in the translated block cache for the first block in the region. When the emulation manager finds a hit in the translated block cache for the blocks starting address the translated region is executed.

5.4 Measuring Performance

To evaluate the various implementations and mechanisms, a method of measuring different execution times is required. The .NET framework provides a Stopwatch class, which provides an API that mimics a stopwatch. Included in the Stopwatch are methods including Start(), Stop(), Reset(), ElapsedMilliseconds() and ElapsedTicks(). The Stopwatch is simple to use and was used frequently to collect execution times. For example:

```
var _sw = new Stopwatch();
_sw.Start();
SomeMethodRequiringTiming();
_sw.Stop();
var elapsed = _sw.ElapsedTicks;
```

Figure 25 .NET Stopwatch class in use

ElapsedTicks are used to measure execution time since a tick has a precise time value. The duration of a tick is 1 second divided by the Stopwatch.Frequency. The Stopwatch.Frequency depends on the hardware the system is running on but it generally results in a tick that is significantly less than a millisecond. On an Intel Core 2 Duo T8100 running at 2.10 GHz the tick duration is approximately 69 nanoseconds.

The performance of the translator running the benchmark programs was measured using the Stopwatch. The translator ran each benchmark 20 times and the duration of each execution was recorded.

5.5 Testing

Accuracy of translation is of utmost importance in a dynamic binary translator to ensure that the emulator correctly models the behaviour of the source ISA. The interpreter that the dynamic binary translator is based on was used as a reference to ensure the correct behaviour of the translator. The effect of a translated instruction on the architected state of the emulator was manually compared to the effect of interpreting the same instruction. If both translated and interpreted instructions affected the architected state of the emulator in the same way the translation was regarded as being correct.

6 Benchmarking the Translator

As discussed previously instrumentation was added to the translator to determine the time taken for each execution. Several small programs were used to benchmark the performance of the simulator, including:

- Bubble sort, sorting 128 random values
- Sieve of Eratosthenes, finding all prime numbers less than 8190
- Generate the first 20 decimal digits of e.

See section 9.1 Appendix A for the ARM assembler listings of the benchmark programs.

To measure the performance of the translated code the translator must execute the generated code multiple times. The reasons for this are reviewed in sections 6.5 and 6.8.

6.1 Performance of Interpreter

The performance of the interpreter was recorded to provide a baseline to compare with the DBT implementations. In Figure 26 the cold code startup cost is seen in the first run where the time taken is approximately 100 000 CPU ticks more than subsequent runs.

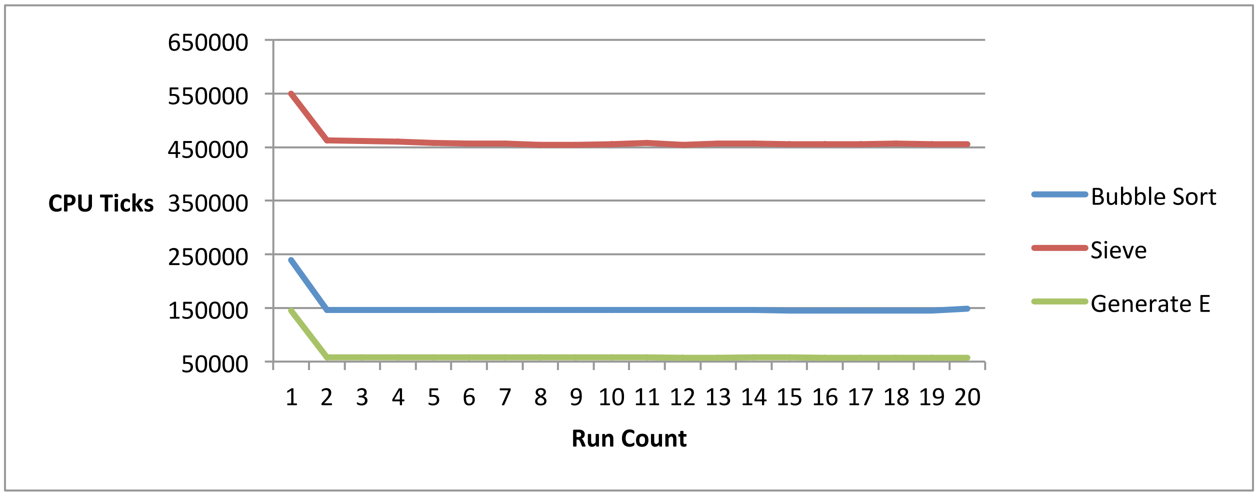

Figure 26 Interpreter performance

Figure 26 shows the average execution time over 1000 executions of the three benchmark programs running in the interpreter. The benchmark program is run 20 times per execution. This chart illustrates two interesting behaviors:

- Relatively constant execution time after the initial run.
- Relatively constant decrease in execution time between the first run and the subsequent runs.

Startup behavior is discussed in more detail in section 6.5.

6.2 Performance of Simple DBT

Figure 27 shows the average execution time over 1000 executions of the three benchmark programs running in the Dynamic Basic Block translator. As with the interpreter the benchmark program is run 20 times per execution. The most noticeable difference between this chart and Figure 26 above is the initial slow down and the subsequent speed up. The slowdown is caused by the overhead of translation, and the speedup is due to executing translated blocks rather than interpreting each block. The behavior of the first 5 runs is also significantly different to the interpreter; this is discussed further in section 6.7.

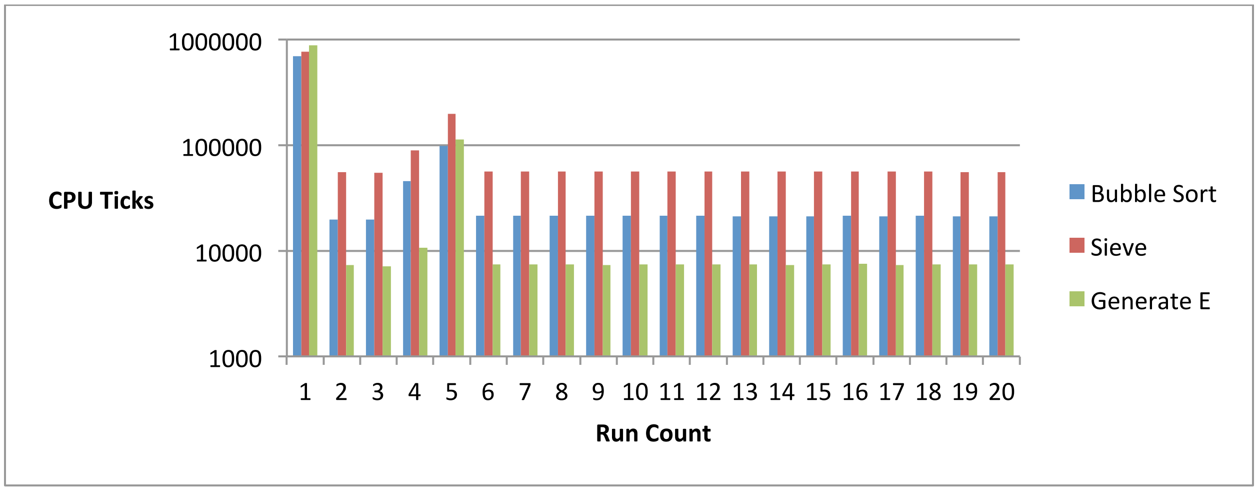

Figure 27 Block DBT Performance

6.3 Performance of DBT using LTU

Figure 28 shows the average execution time over 1000 executions of the three benchmark programs running in the Large Translation Unit translator. As with the interpreter and DBB the benchmark is run 20 times per execution. Noticeable in this chart is the longer first execution, almost double the time taken for the initial run in the DBB translator. This is caused by the additional overhead of creating large translation units. The speed increase in the 6th and subsequent runs of the benchmark programs is also notable when compared to the DBB translator. Additionally the LTU translator exhibits uneven execution times for the first five runs similar to the DBB translator. This is discussed further in section 6.5.

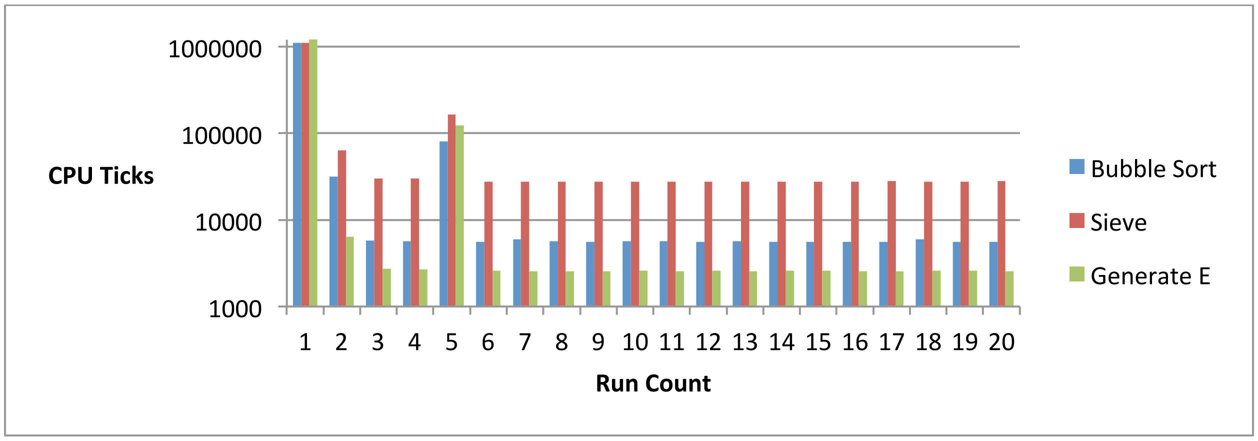

Figure 28 LTU DBT Performance

Also noticeable is the similarity in first execution times for all three benchmarks, this is discussed further in section 6.5.

6.4 Comparing Different Approaches

The performance improvement of the DBB translator over the interpreter and the LTU translator over the DBB translator can be seen immediately when they are plotted on the same axes.

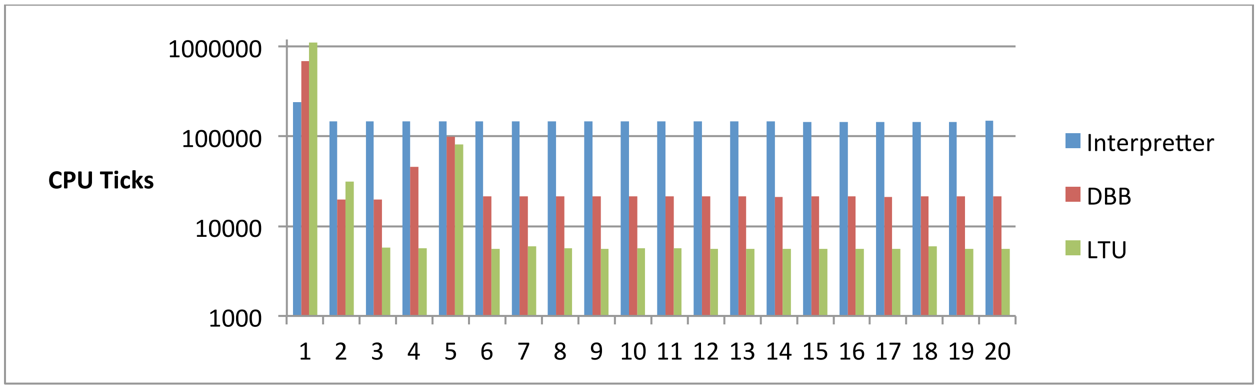

Figure 29 Bubble Sort benchmark execution times for different approaches

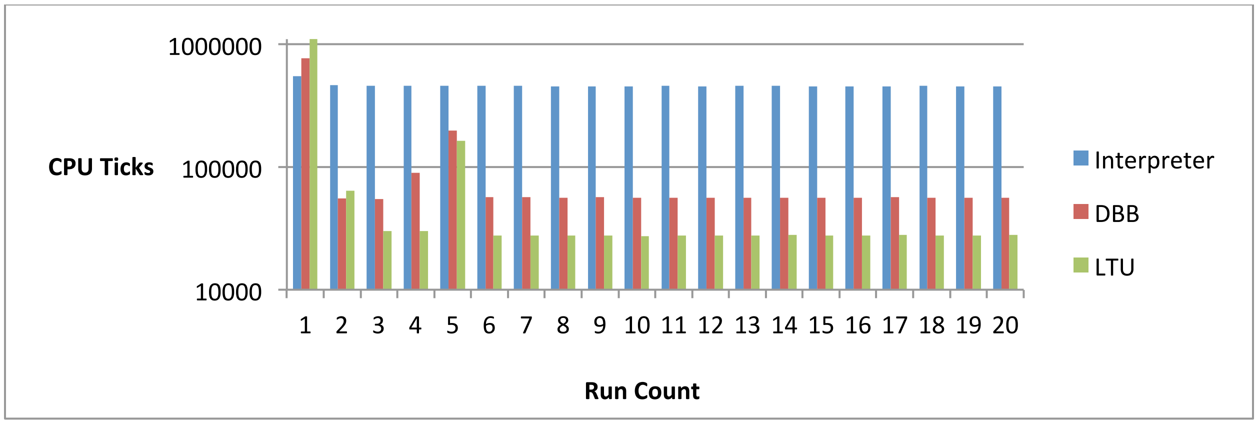

Figure 30 **Sieve of Eratosthenes benchmark execution times for different approaches**

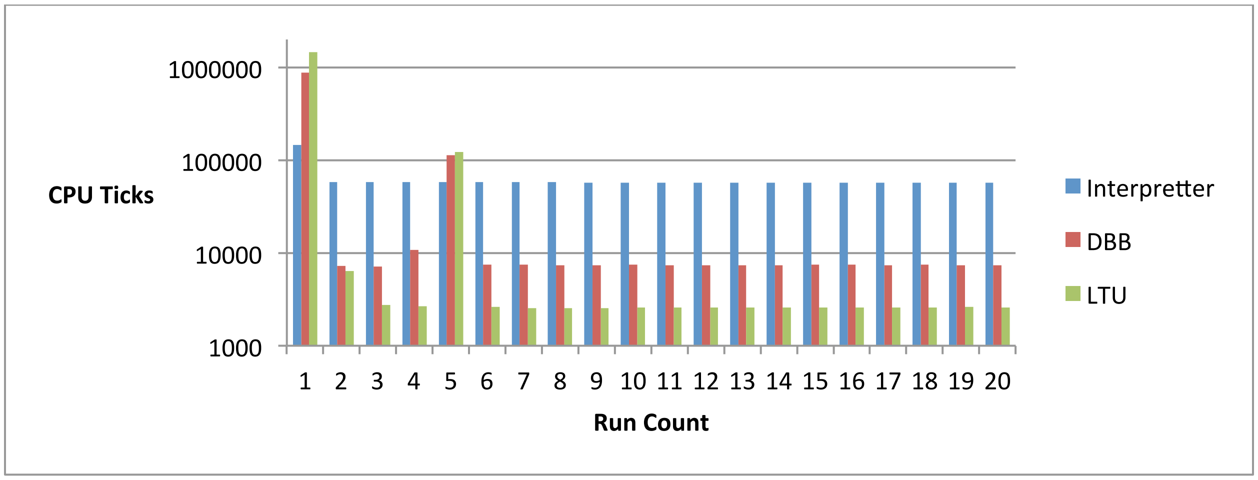

Figure 31 **Generate E benchmark execution times for different approaches**

Figure 29, Figure 30, and Figure 31 show the three different approaches on the same set of axes for each benchmark program. The cold start overhead can be seen for the first execution of all the approaches. Discarding the initial run the performance improvement of the DBB translator versus the interpreter is dramatic for all three benchmarks. The performance improvement of the LTU translator versus the DBB translator is a little less dramatic but equally significant.

Benchmark	**Interpreter**	**DBB (n times faster)**	**LTU (n times faster)**
Bubble Sort	146025	21391(6.82)	5655(25.82)
Sieve	456873	56254(8.12)	27720(16.48)
Generate E	57497	7417(7.75)	2577(22.31)

Table 2 **Average CPU ticks discarding start-up costs**

The performance increase for the average execution time, discarding the startup overhead is shown in Table 2. The numbers in brackets show the speed increase relative to the time taken to run the benchmark on the interpreter. The performance of the LTU translator versus the DBB translator is significantly better than the improvement seen between these two techniques in the Edinburgh High Speed (EHS) simulator [**12**]shown in Table 1. This is may be due to the EHS simulator being designed for both observability and performance.

6.5 Start Up

Regardless of the emulation technique used the first run of each of the benchmark programs is significantly slower than subsequent executions. There are several causes for the first run having poor performance relative to later runs, some are common to all emulation techniques and some are due to the translation mechanism in DBB and LTU translators:

- The code and dynamically linked libraries have to be loaded.
- There is significant overhead in translating the source to the target.
- There is additional overhead in creating Large Translation Units.

6.5.1 Loading

The interpreter makes the effects of loading code more obvious than either of the translators since all the code that is run in the first execution is also run in all subsequent executions. This means that the difference between the first and subsequent runs is due only to the application and dynamic library loading that is required the first time a program is executed. The simulator also loads the memory image of the source program for each execution, and it is likely that this is at least partially cached for subsequent executions.

Benchmark	**Cold Execution Ticks**	**Warm Execution Ticks**	**Difference**	**Source code size**
Bubble Sort	239082	146025	93057	1.88 KB
Sieve of Eratosthenes	594584	456873	137711	2.016KB
Generate E	145064	57497	87567	6.084KB

Table 3 Cold versus warm execution time

Table 3 shows that the difference between warm and cold execution is similar for each benchmark even though the execution times are significantly different. There does not appear to be any obvious relationship between the size of the benchmark program and start-up overhead.

6.5.2 Translation and Hot Traces

Both translation and creating hot traces add overhead to the first execution of a benchmark program.

Benchmark	Interpreter	DBB	LTU
Bubble Sort	239082	692489(2.90)	1107168(4.63)
Sieve	549584	769527(1.40)	1108079(2.02)
Generate E	145064	882041(6.08)	1466691(10.11)

Table 4 CPU Ticks for first run of each benchmark program

Table 4 shows the times taken for the first run of each benchmark using each simulation mechanism. The number in brackets is the slowdown relative to the interpreter for that simulation mechanism. The nature of the benchmark programs themselves is likely to influence the time that they take to execute. The Sieve benchmark seems to be very slow in the interpreter, possibly due to the memory intensive nature of the benchmark. The Generate E benchmark which is more compute intensive runs relatively quickly. It is difficult to compare between the benchmark programs without standardizing them in some way.

Differences between the benchmarks are also due to both the number of translated blocks, the number of blocks in the hot region and the decisions about when and what to translate, discussed further in section 6.6 and 6.7.

Benchmark	Total DBBs	Translated Blocks	Blocks in hot region
Bubble Sort	7	5	5
Sieve	10	6	5
Generate E	13	8	9

Table 5 Comparison of block counts in benchmark programs

The Generate E benchmark is particularly slow in the first execution when creating LTUs due to the fact that it translates any blocks that get included in the region that are not already translated. This benchmark is also significantly slower for the first execution of the DBB translator. This may be due in part to the fact that it is translating more blocks than the other benchmark programs.

6.6 When to Translate

Both the DBB translator and the LTU translator have to determine when it is appropriate to translate blocks or regions of source instructions into target instructions. For the DBB translator this was based on:

- The number of times an instruction has been executed.
- The number of times a block has been executed.

This was extended in the LTU translator to:

- The number of times an edge has been traversed.
- The number of edges that have been traversed more than some number of times.

In the LTU translator the translation threshold, the combination of the number of hot edges and the number of traversals required to make an edge hot can have a dramatic effect on the performance of the translated program. When the threshold is set too low for a particular program it is possible to fail to include edges in an LTU that are essential to the performance of the program.

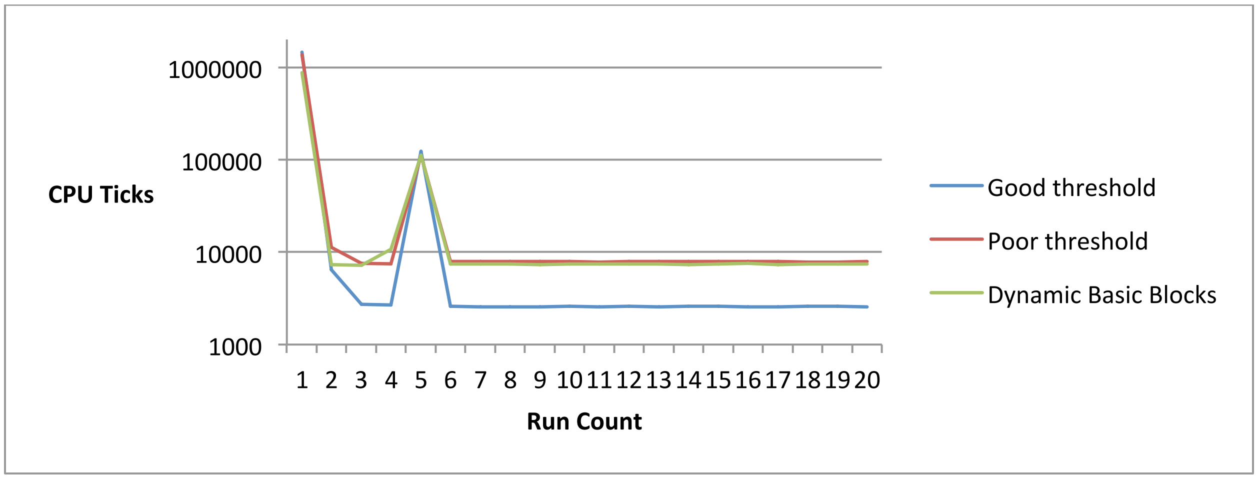

Figure 32 Performance effects of different translation threshold values for translating Generate E program

Figure 32 shows the effects of making the threshold for the number of hot edges required to form an LTU too small for the generate E benchmark program. When the threshold is set too low the control flow graph of the program does not include edges that subsequently become hot. The LTU formed from the control flow graph at this point fails to reduce the number of returns to the emulation manager and is more than twice as slow as the good threshold and slower than the dynamic basic block translation.

If the number of hot edges required to form a region is set too high the translator is identical to the DBB translator with some additional overhead of checking whether the condition to build an LTU have been met. If the number of traversals required to denote an edge as hot is set too high then forming the LTU for the hot region is delayed. Raising either of these thresholds excessively prevents the conditions for creating LTUs from being met.

Using a fixed threshold for determining when to translate blocks and regions is not a very satisfactory approach, since the threshold(s) may need to be reconfigured for each program that is run. In the Edinburgh High Speed simulator [**12**], execution of a program is divided into epochs. The end of an epoch is triggered every time the count of interpreted blocks exceeds some threshold. This means that as the hot sections of the program are translated and fewer blocks are interpreted the epochs grow longer. If the program execution moves to a different region of code the number of blocks being interpreted increases and triggers the end of an epoch. At the end of an epoch the control flow graph is analyzed to determine if there are any new regions to translate.

6.7 What to Translate – Hot versus All

As mentioned in sections 6.2 and 6.3 the first several runs of both the DBB and LTU translators have varying execution times. This variation is caused by translation of instructions and translation of blocks continuing after the first execution of the benchmark program.

Running the Generate E benchmark program (the other benchmark programs exhibit similar but not identical behavior) the second run discovers instructions that have not been translated and translates them. The third and fourth runs are executing a mixture of interpreted and translated code. On run five the blocks that were being interpreted have been flagged as requiring translation. The translation of these blocks is of questionable value because these are instructions that have taken 4 or 5 executions of the program to reach the translation threshold. Depending on the translation threshold these blocks may only be executed once in any given execution of the benchmark program.

The performance difference between translating all blocks and only those that are found to be hot during the first execution can be seen in Figure 33. In the 5th execution of the Generate E benchmark, three additional blocks are marked as hot and translated. This translation is an artefact of the block execution count being used as the threshold to determine when to translate a block and the block execution count continuing to be incremented during each execution of the benchmark program. By not checking for blocks to translate after the first execution of the benchmark program the translation of these cold blocks can be avoided.

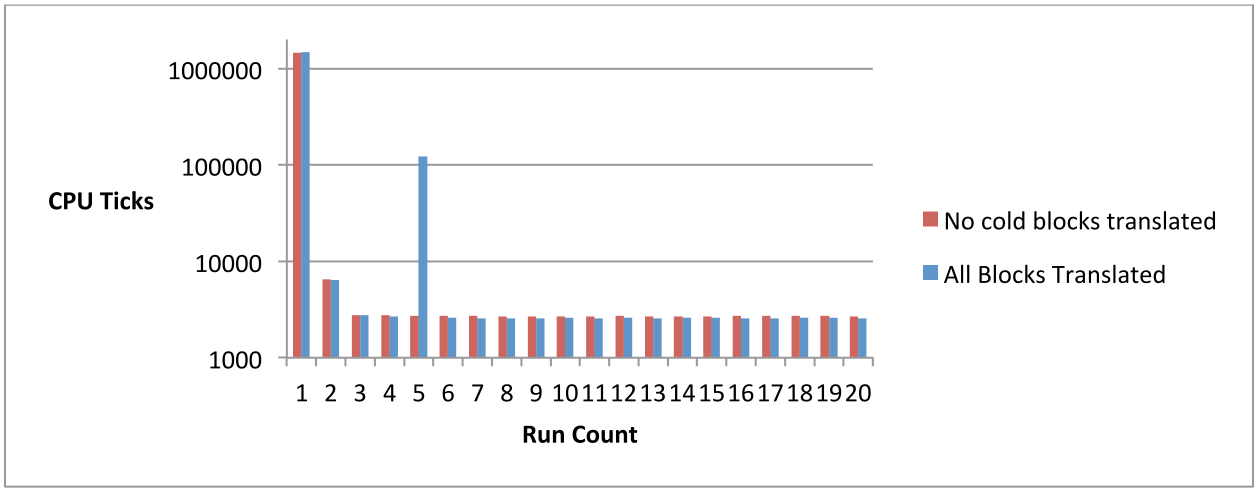

Figure 33 **Translating all blocks versus only hot blocks**

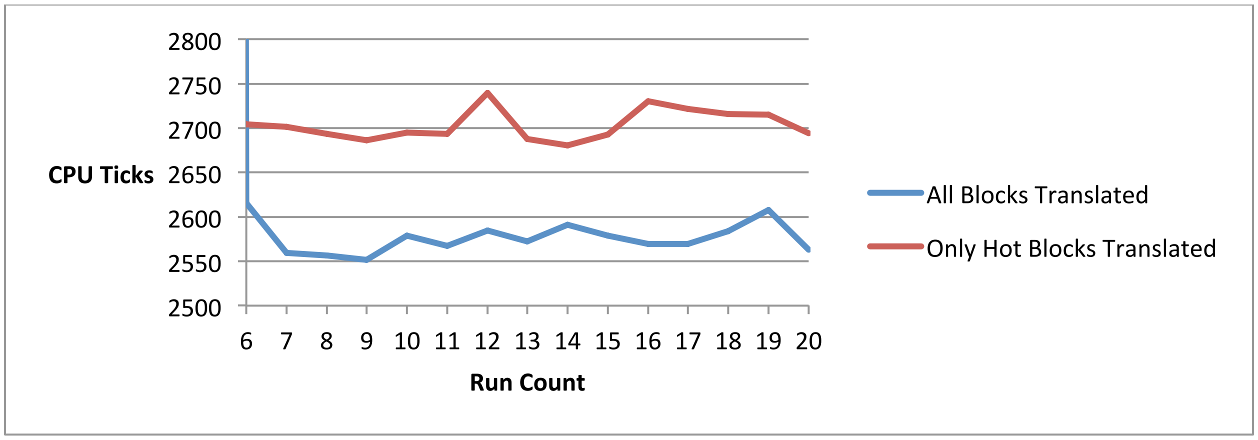

Figure 34 **Hot blocks only versus all blocks translated, after final translation**

Figure 34 shows the improvement in execution times for the benchmark program after the additional translation in run five. In this context there is an approximately 5% improvement in performance. However the cost of this increase is around 120, 000 ticks, so approximately 1000 executions of the benchmark program are required to recover the cost of the translation.

This behaviour is observed in both the DBB and LTU translators because the LTU translator only forms hot regions during the first execution of the benchmark program.

6.8 Garbage Collection

Looking at the raw timing values from the individual program runs occasionally the emulator takes significantly more than the usual number of ticks to complete a run. These occasional slowdowns are present for each of the benchmark programs and regardless of the emulation mechanism.

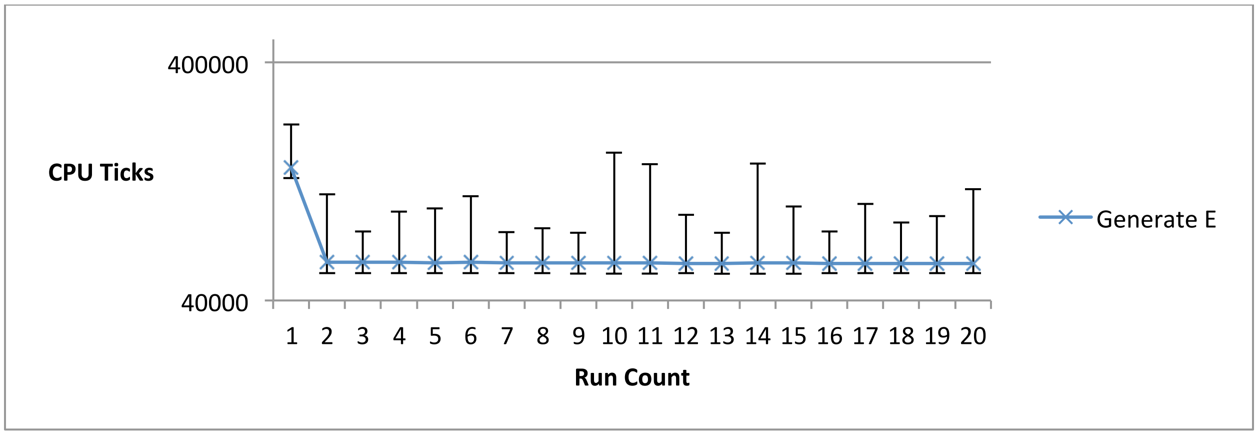

Figure 35 Minimum, maximum and average execution times for the interpreter running the Generate E benchmark

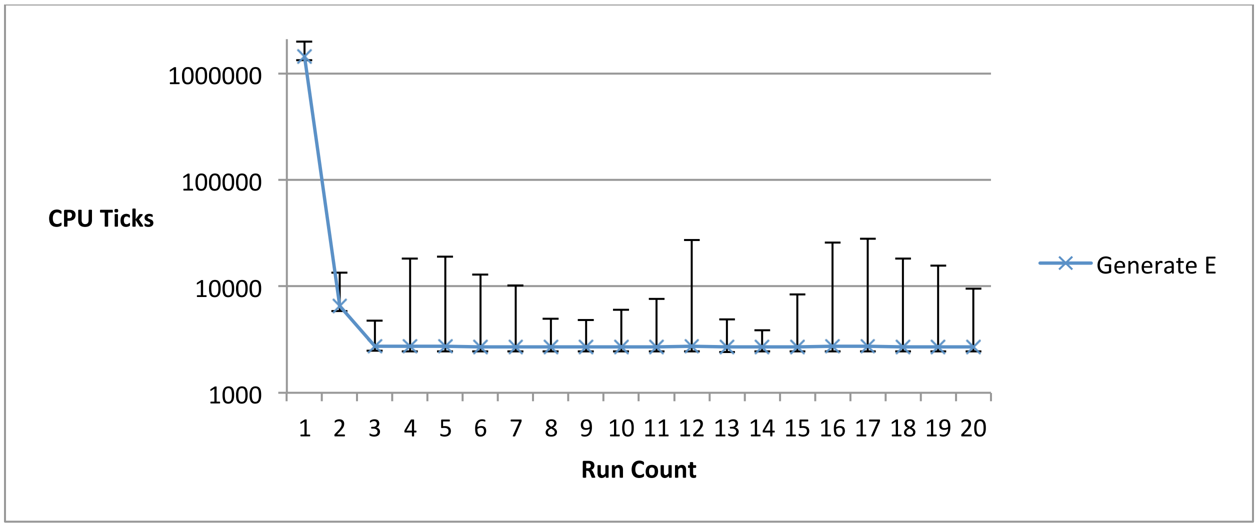

Figure 36 Minimum, maximum and average execution times for the LTU translator running the Generate E benchmark

Figure 35 and Figure 36 show the variation in run times for the first 20 executions of the Generate E benchmark across 1000 runs of the simulator. The cause of these periodic performance glitches was not immediately clear.

Initial investigation of the periodic performance decline employed the Perfmon.exe tool to examine .NET memory performance counters while the simulator was running. The Perfmon.exe tool showed that the garbage collector is active while the simulator is running, but provided no way to correlate the activity of the simulator with the activity of the garbage collector. The .NET garbage collector has been shown to have deleterious effects on simulator performance [**11**] so this issue deserved further investigation.

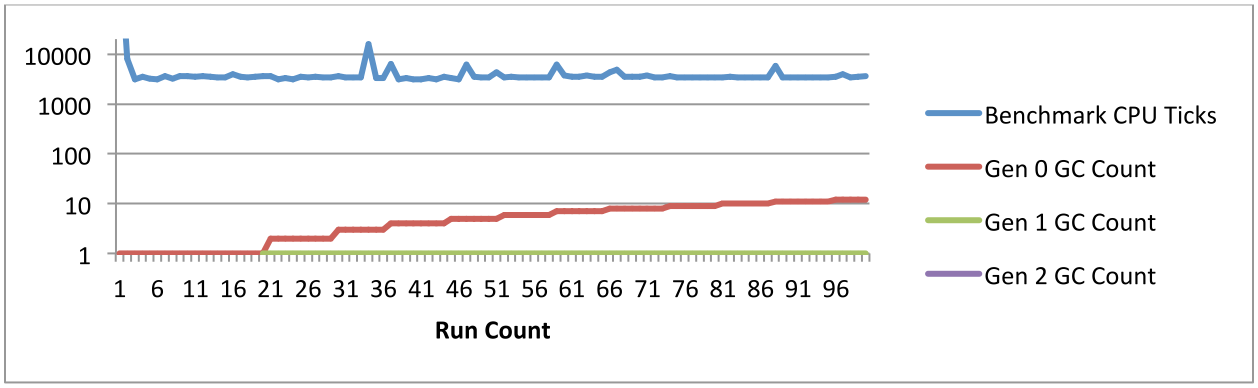

Figure 37 CPU Ticks and Garbage Collection

In an attempt to correlate the behaviour of the garbage collector with the performance glitches in the simulator the simulator source code was instrumented using the PerformanceCounter class provided by the .NET framework. Figure 37shows the first 100 runs from a 10000 run sequence. The glitches in the benchmark CPU Ticks can be seen clearly but there is no obvious correlation with the garbage collector generation count increasing. At the end of 10000 runs of the program the generation 2 collection had run once and the generation 1 collector had run twice. The frequency of simulator glitches was significantly higher than the frequency of generation 1 and generation 2 garbage collection events. However since the performance counter for the garbage collector is only incremented when the collection is completed it may not be possible to observe a correlation with the point in time at which the garbage collector affects the running simulation.

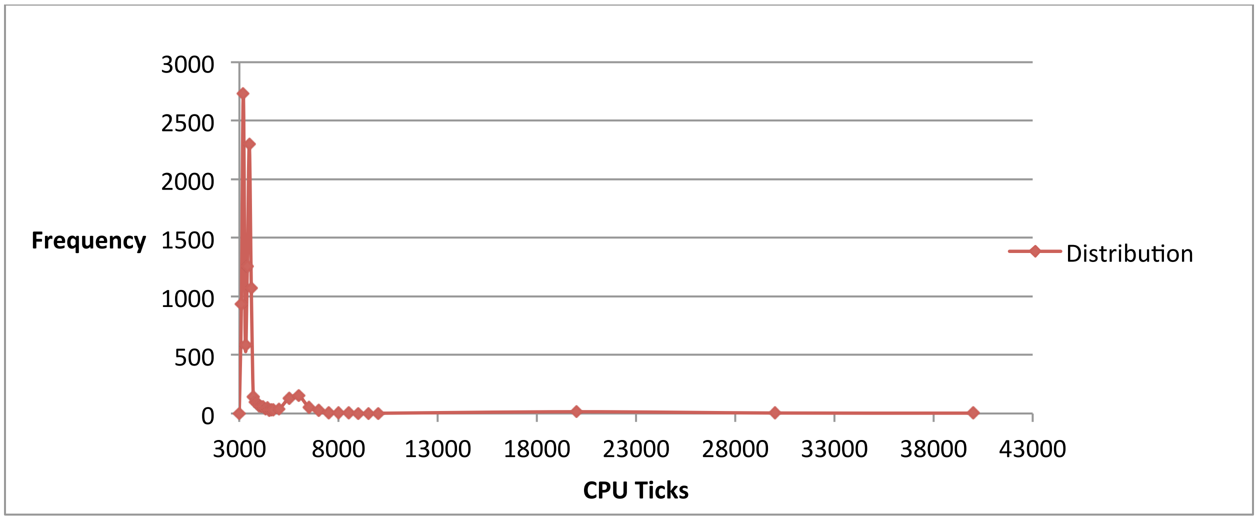

Figure 38 Execution CPU ticks distribution

Figure 38 shows that effect of garbage collection on the execution time is less significant than it might first seem. Figure 35 and Figure 36 reinforce this position since the average and minimum execution times for each run remain close, indicating that values between the maximum and the average occur infrequently.

7 Conclusions and Future Work

A high performance, portable dynamic binary translator for the ARM v4 architecture was developed. The translator, developed in two stages, showed dramatic performance improvements at each stage. Overall, it demonstrated performance improvements of between 15 and 25 times that of a simple decode and dispatch interpreter.

During the first stage, a dynamic basic block (DBB) translator was implemented, which discovered DBBs as they were executed. An execution count was maintained for each DBB, and when the count reached a threshold value, the DBB became ‘hot’ and was translated.

For the second stage, DBBs were coalesced into large translation units (LTU) by translating control flow instructions. A count was maintained for each edge in the control flow graph (CFG). When the number of ‘hot’ edges in the CFG exceeded some threshold the CFG was analysed using Tarjan’s algorithm. Each strongly connected component (SCC) discovered by the algorithm was checked. If it contained any hot edges, it was used to form an LTU. Additionally, within each LTU the code layout was modified to reduce branching on the ‘hot’ path.

While Tarjan’s algorithm provided an efficient way of discovering SCCs in the CFG, not all of the blocks in a SCC were necessarily hot. This resulted in the translation of blocks that were not hot, which is an expensive operation with little payback. Other ways of forming LTUs might provide better overall performance.

The use of a fixed threshold to control when to form large translation units was also problematic. A threshold value that was too low reduced the performance of the LTU translator to that of the DBB translator. A threshold that was too high delayed the formation of the hot region and increased the cost of translation. Using an adaptive approach to decide when to create a hot region could overcome this issue.

Translation is an expensive operation and the first execution of the target program running in the translator was significantly slower than the first execution of the same program running in the interpreter. The DBB translator adds the cost of discovering and translating DBBs to the cost of interpreting the program. The LTU translator increases the time taken for the first execution further by adding the cost of region detection and control flow translation. For the benchmark programs used in this thesis the translation overhead increased in proportion with the number of translated blocks.

The translated Bubble Sort benchmark program showed larger performance gains when compared to the translated Sieve of Eratosthenes benchmark. The Sieve of Eratosthenes benchmark is more compute intensive than the Bubble Sort benchmark so the translator may offer superior performance when the source code is biased towards input/output operations rather than data operations.

Combining translation to an intermediate form for execution by a virtual machine, and forming large translation units does not seem to be described in the literature. Translating hot traces to intermediate form for a compiler is described [**31**], as is creating hot regions in the virtual machine [**32**]. This may be due to the use of a virtual machine with its stack architecture preventing the use of many common dynamic binary translation optimizations that are available when translating to a native ISA. However the optimizations provided by .NET CLR might offset the need to implement these optimizations to some extent. For example the dead code elimination provided by the CLR is useful in eliminating redundant flag setting code which is a significant contributor to the code expansion issues experienced in translating from ARM v4 assembler to CIL.

7.1 Future Work

While this thesis proved the feasibility of implementing a dynamic binary translator with good performance on the .NET platform there are several avenues that deserve further exploration:

- Performance
- Integration with ARMSim#
- Measuring Performance
- Thumb instruction set support

7.1.1 Performance

- One approach to improving performance is to perform some tasks in parallel [**31**], such as generating translations. In this implementation the translation of an instruction happens sequentially after it has been executed by the interpreter. The translation of the instruction is not used until it is incorporated into a dynamic basic block that is being translated. By performing the translation on a separate thread the interpreter thread would be free to interpret the next instruction.

- The fixed translation threshold is both a performance and usability issue. To make the translator truly useful and enhance its performance some form of adaptive mechanism is required to ensure that translation happens when it is required. One possible mechanism that could be implemented is the “epoch” based system of the Edinburgh High Speed simulator [**12**].
- Translating all blocks in a strongly connected component of the CFG of a program may not offer the best performance as work is being done to add blocks that are not hot to the hot region. Reducing this work may significantly reduce the work done in the first execution of any source program.
- One of the goals of this thesis was to avoid performing large amounts of code analysis. However, some common structures that make use of the s bit version of an instruction could potentially be easily optimized. If we know the flags are dead after they are checked by the branch instruction in the following for loop [**33**] then the SUBS instruction need only set the Z flag:

```
        MOV    R0, #loopcount
loop
        …loop body…
        …
        SUBS   R0, R0, #1
        BNE    loop
```

  This would greatly reduce the size of any flag setting instructions.
- Implementing hot regions as single compiled functions may not be optimal for the CLR JIT compiler. It may be better if each block formed a function that was called by the region code. This would allow the JIT compiler to optimize and compile code as it chooses as opposed to forcing the translation to be a single, possibly large, function. Implementing this would require further investigation into the functionality provided by the Expression class and the Reflection.Emit class in the .NET framework.

7.1.2 Integration with ARMSim#

Integrating the translator with the ARMSim# simulator could provide the simulator with significantly improved performance while maintaining the ability to step and set breakpoints.

The DBT would need to be extended with a mechanism to switch it in and out of interpreter mode. Switching modes would be quite straightforward for the DBB translator, but require some way to break out of a hot region in the LTU translator that did not adversely affect the performance.

Fully integrating the translator would also require implementing the instructions and instruction variations that are not currently implemented.

7.1.3 Measuring Performance

The benchmark programs used in this thesis were small simple programs. It would be more useful to use benchmarks that have been used previously such as EEMBC [**12**], [**20**], SpecInt95 [**34**] SPEC2000 [**3**], or MiBench [**18**]. However, using these benchmarks would require a more complete implementation of the ARM ISA than is presented in this thesis.

7.1.4 Thumb Instruction Support

The T in the ARM7TDMI name specifies that the CPU supports the Thumb instruction set, which is a 16 bit subset of the 32 bit instruction set. The Thumb instruction set is used to reduce the memory footprint of a program. While the translator presented in this thesis does not currently support the Thumb instruction set, extending it to do so should be relatively straight forward.

8 Bibliography

[1] J. Smith and R. Nair, *Virtual Machines, Versatile Platforms for Systems and Processes*. San Francisco: Morgan Kaufmann, 2005.

[2] Wikipedia. (2014, Mar.) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Binary_translation

[3] Edson Borin and Youfeng Wu, "Characterization of DBT overhead.," in *IEEE International Symposium on Workload Characterization IISWC*, Austin, 2009, pp. 178-187.

[4] Haibing Guan et al., "A Dynamic-Static Combined Code Layout Reorganization Approach for Dynamic Binary Translation.," *Journal of Software*, vol. 6, no. 12, pp. 2341-2349, 2011.

[5] Achim Nohl et al., "A universal technique for fast and flexible instruction-set architecture simulation," in *Proceedings of the 39th annual Design Automation Conference (DAC '02)*, New York, 2002, pp. 22-27.

[6] Nigel Topham and Daniel Jones, "High speed CPU Simulation Using LTU Dynamic Binary Translation," in *HiPEAC '09 Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, Berlin, 2008, pp. 50-64.

[7] Marco Kaufmann, Matthias Häsing, Thomas Preußer, and Rainer Spallek, "The Java Virtual Machine in retargetable, high-performance instruction set simulation," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*, New York, 2011, pp. 21-30.

[8] Guilherme Ottoni et al., "Harmonia: a transparent, efficient, and harmonious dynamic binary translator targeting the Intel® architecture," in *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF '11)*, New York, 2011, p. Article No. 26.

[9] Claude Helmstetter, Vania Joloboff, Xinlei Zhou, and Xiaopeng Gao, "Fast Instruction Set Simulation Using LLVM-based Dynamic Translation.," in *International MultiConference of*

Engineers and Computer Scientists, vol. 2188, Hong Kong, 2011, pp. 212-216.

[10] Bob Cmelik and David Keppel, "Shade: A fast instruction-set simulator for execution profiling.," in *Fast Simulation of Computer Architectures.*: Springer, 1995, pp. 5-46.

[11] Antoine Trouvé and Kazuaki Murakami. (2010, January) Laboratoire d'Informatique Fondamentale de Lille. [Online]. http://www2.lifl.fr/rapido/RAPIDO_2010/Rapido/rapido2010Proc.pdf

[12] Daniel Jones and Nigel Topham, "High Speed CPU Simulation Using LTU Dynamic Binary Translation," *High Performance Embedded Architectures and Compilers*, pp. 50-64, 2009.

[13] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler, "Fast and accurate simulation using the llvm compiler framework.," in *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, vol. 9, Paphos, Cyprus, 2009, pp. 1-6. [Online]. http://www.complang.tuwien.ac.at/cd/brandner/papers/rapido09-brandner.pdf

[14] Fabrice Bellard, "QEMU, AFastand Portable Dynamic Translator," in *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, 2005, pp. 41-46.

[15] Emmett Witchel and Mendel Rosenblum., "Embra: Fast and flexible machine simulation.," *ACM SIGMETRICS Performance Evaluation Review.*, vol. 24, no. 1, pp. 68-79, 1996.

[16] Chun-Chen Hsu et al., ""LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends"," in *International Conference on Parallel Processing (ICPP)*, 2011, pp. 226-234.

[17] Ryan W. Moore, Jose A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser, "Addressing the challenges of DBT for the ARM architecture.," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '09)*, New York, 2009, pp. 147-156.

[18] Xuhao Chen, Zhong Zheng, Li Shen, Wei Chen, and Zhiying Wang, "GSM: An Efficient

Code Generation Algorithm for Dynamic Binary Translator.," in *Fourth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, Tianjin, 2011, pp. 231-235.

[19] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt, "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation," *ACM Transactions on Embedded Computer Systems*, vol. 8, no. 3, p. 27, 2009.

[20] Jiunn-Yeu Chen, Wuu Yang, Tzu-Han Hung, Hong-Men Su, and Wei-Chung Hsu, "“A Static Binary Translator for Efficient Migration of ARM-based Applications”," in *6th Workshop on Optimizations for DSP and Embedded Systems (ODES)*, Boston, 2008, pp. 55-64. [Online]. http://odes-workshop.weebly.com/uploads/6/5/6/0/6560023/odes-8_proceedings.pdf

[21] David Seal, Ed., *ARM Architecture Reference Manual*, 2nd ed. Harlow: Addison Wesley, 2000.

[22] ARM Ltd. (2012, Nov.) ARM Information Center. [Online]. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

[23] Chris Sells and Shawn Van Ness. (2002, August) Dr Dobb's. [Online]. http://www.drdobbs.com/generating-code-at-run-time-with-reflect/184416570

[24] Microsoft. (2014) Microsoft Developer Network. [Online]. http://msdn.microsoft.com/en-us/library/dd233052(v=vs.110).aspx

[25] Jeff Hardy. (2012) The Architecture of Open Source Applications. [Online]. http://www.aosabook.org/en/ironlang.html

[26] S. Biswas. (2009, April) MSDN Magazine. [Online]. http://msdn.microsoft.com/en-us/magazine/dd569747.aspx#id0400010

[27] Ashwin Kamath, "An Overview of Performance Improvements in.Net 4.5," *MSDN Magazine*, vol. 27, no. 4, pp. 76-81, April 2012.

[28] David Broman. (2007, Jun) David Broman's CLR Profiling API Blog. [Online]. http://blogs.msdn.com/b/davbr/archive/2007/06/20/enter-leave-tailcall-hooks-part-2-tall-tales-of-tail-calls.aspx

[29] Fei Chen. (2007, Nov.) MSDN Blogs. [Online]. http://blogs.msdn.com/b/clrcodegeneration/archive/2007/11/02/how-are-value-types-implemented-in-the-32-bit-clr-what-has-been-done-to-improve-their-performance.aspx

[30] R.E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing Vol.1, Iss.2*, pp. 146-160, 1972.

[31] Edler von Koch T., Kyle S., Franke B., Topham N. Böhm I., "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation(PLDI '11)*, New York, 2011, pp. 74-85.

[32] Ian Rodgers. (2001, April) School of Computer Science, Manchester University. [Online]. ftp://ftp.cs.man.ac.uk/pub/apt/papers/irogers_prep01.pdf

[33] William Hohl, *ARM Assembly Language*. Boca Raton, FL, USA: CRC Press, 2009.

[34] Mark Probst, Andreas Krall, and Bernhard Scholz, "Register Liveness Analysis for Optimizing Dynamic Binary Translation," in *Ninth Working Conference on Reverse Engineering (WCRE'02}*, Richmond, 2002, pp. 35-44.

[35] A. H. J. Sale, "The Calculation of e to Many Significant Digits," *The Computer Journal*, vol. 11, no. 2, pp. 229-230, 1968.

[36] Nigel Topham, Björn Franke, Daniel Jones, and Daniel Powell, "Adaptive High-speed Processor Simulation.," in *Processor and System-On-Chip Simulation*. New York: Springer Verlag, 2010, pp. 145-160.

[37] Jeremy Singer, "JVM versus CLR: a comparative study.," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java (PPPJ '03)*,

New York, 2003, pp. 167-169.

[38] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg, "Virtual machine showdown: Stack versus registers.," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 4, no. 4, p. 36, 2008.

[39] Jeffrey Richter, *C# via CLR*.: Microsoft Press, 2012.

[40] N. Horspool, M. Serra, and W. Lyons. (2008, Dec) The ARMSim# User Guide. [Online]. http://armsim.cs.uvic.ca/AttachedFiles/ARMSim_UserGuide4Plus.pdf

9 Appendices

9.1 Appendix A: Benchmark ARM Assembler Programs

9.1.1 Bubble Sort

Adapted from ARM: Assembly Language Programming by Peter Knaggs, 2006.

http://www.rigwit.co.uk/ARMBook/ARMBook.pdf.

```
.text
.global _start
_start:
      LDR    R6, =Start          @ pointer to start of list
      MOV    R0, #0      @ clear register
      LDRB   R0, [R6]            @ get the length of the list
      MOV    R8, R6      @ make a copy of the start of the list
SORT:
      ADD    R7, R6, R0          @ get address of last element
      MOV    R1, #0      @ zero flag for changes
      ADD    R8, R8, #1          @ move 1 byte up the list each iteration
NEXT:
      LDRB   R2, [R7]            @, #-1 @ load the 1st byte
      SUB    R7, R7, #1
      LDRB   R3, [R7]            @ and the second
      CMP    R2, R3      @ compare them
      BCC    NOSWITCH            @ branch if R2 less than R3
      STRB   R2, [R7], #1 @otherwise swap the bytes
      STRB   R3, [R7]            @like this
      ADD    R1, R1, #1          @ flag that changes made
      SUB    R7, R7, #1          @ decrement address to check
NOSWITCH:
      CMP    R7, R8      @ have we checked enough bytes
      BHI    NEXT                @ if not do innner loop
      CMP    R1, #0      @ did we make changes
      BNE    SORT                @ if so check again outer loop
DONE:
      SWI          0x11
.data
.align
Start: .byte 0x80, 0x2a, 0x5b, 0x60, 0x3f, 0xd1, 0x19, 0x30, 0xc5, 0x66, 0xEF, 0x19,
0x11, 0x9B, 0xC7, 0x1C, 0x63, 0x94, 0x82, 0x5F, 0xA9, 0xD7, 0x62, 0x27, 0xB5, 0x78,
0x7B, 0x7D, 0xCB, 0x57, 0xFF, 0x16, 0xF9, 0xB5, 0x9C, 0x4F, 0xCF, 0x04, 0xC1, 0x2E,
0xC0, 0x6B, 0x49, 0x4D, 0x07, 0xFE, 0x97, 0x67, 0xA0, 0x60, 0xA6, 0x2D, 0xC5, 0x91,
0x62, 0x28, 0xAA, 0x9B, 0xEA, 0x04, 0x15, 0x4A, 0x0B, 0x01, 0xA4, 0xDF, 0x12, 0x4D,
0xDF, 0xCE, 0x81, 0x22, 0x6E, 0x5C, 0xDD, 0x57, 0xB3, 0xFA, 0x3B, 0x68, 0x71, 0xEC,
0x50, 0x63, 0x93, 0xA5, 0x57, 0x6E, 0xA3, 0x97, 0x22, 0x63, 0x1B, 0x74, 0x6B, 0x49,
0x64, 0x39, 0x39, 0x28, 0xB8, 0xEE, 0xEA, 0x4F, 0x2A, 0x2B, 0x62, 0x38, 0x28, 0xE5,
0xBF, 0x1D, 0x1F, 0xE6, 0x63, 0x29, 0xBD, 0x13, 0x99, 0x27, 0xC6, 0xDF, 0x90, 0x1D,
0x43, 0xD4, 0xDD, 0x55, 0xE6
List: .word Start
```

9.1.2 Sieve of Eratosthenes

Adapted from http://www.peter-cockerell.net/aalp/html/ch-6.html

```
.equ org, 2000;DIM org 2000
@;REM Register allocations
@;count = 0  ; R0
@;ptr = 1    ; R1
@;i = 2      ; R2
@;mask = 3   ; R3
@;base = 4   ; R4
@;prime = 5  ; R5
@;k = 6      ; R6
@;tmp = 7    ; R7
@;size = 8   ; R8
@;iter = 9   ; R9
@;link = 14  ; R14/LR
.equ SIZE, 8190     @;SIZE = 8190
.equ iterations, 10 @;iterations = 10
@;The array of SIZE flags is stored 32 per word from address 'theArray'.
@;The zeroth element is stored at bit 0 of word 0,the 32nd element at
@;bit 0 of word 1, and so on. 'Base' is word-aligned
@;
@;Registers:
@; count holds the number of primes found
@; mask used as a bit mask to isolate the required flag
@; ptr used as a general pointer/offset into the array
@; i used as a counting register
@; size holds the value SIZE for comparisons
@; base holds the address of the start of the array
@; prime holds the current prime number
@; k holds the current entry being 'crossed out'
@; tmp is a temporary
@; iter holds the count of iterations
@;
sieve:
      MOV R9,#iterations
mainLoop:
      ADR R4,theArray
      MVN R3,#0           @;Get &FFFFFFFF, ie all bits set
      LDR R8,=SIZE        @;Initialise the array to all 'true'. First store the
                          @;complete words (SIZE DIV 32 of them), then the partial
                          @;word at the end
      MOV R2, R8, LSR #5         ;Loop counter = number of words
      MOV R1,R4                  ;Start address for initing array
initLp:
      STR R3,[R1],#4      @;Store a word and update pointer
      SUBS R2,R2,#1       @;Next word
      BNE initLp
      LDR R7,[R1]         @;Get last, incomplete word
      MOV R3,R3,LSR #2    @ 32-SIZE MOD 32 @;Clear top bits
      ORR R7,R7,R3        @;Set the bottom bits
      STR R7,[R1]         @;Store it back
      MOV R2,#0           @;Init count for main loop
      MOV R0,#0
```

```
lp:
      MOV R1,R2,LSR #5    @;Get word offset for this bit
      MOV R3,#1           @;Get mask for this bit
      AND R7,R2,#31       @;Bit no. = i MOD 32
      MOV R3,R3,LSL R7
      LDR R7,[R4,R1,LSL #2] @;Get the word
      ANDS R7,R7,R3       @;See if bit is set
      BEQ nextLp          @;No so skip
      ADD R5,R2,R2        @;Get prime
      ADD R5,R5,#3
      ADD R6,R2,R5        @;Get intial k
      ADD R0,R0,#1        @;Increment count
while:
      CMP R6,R8           @;While k<=size
      BGT nextLp
      MOV R1,R6,LSR #5    @;Get word for flags[k]
      MOV R3,#1
      AND R7,R6,#31
      MOV R3,R3,LSL R7
      LDR R7,[R4,R1,LSL #2]
      BIC R7,R7,R3        @;Clear this bit
      STR R7,[R4,R1,LSL #2]     @;Store it back
      ADD R6,R6,R5        @;Do next one
      B while
nextLp:
      ADD R2,R2,#1        @;Next i
      CMP R2,R8
      BLE lp
      SUBS R9,R9,#1
      BNE mainLoop
      SWI 0x11
theArray:
.space 1023
.end
```

9.1.3 Generate E to n Decimal Places

Based on A. J. H. Sale [**35**] and N. Horspool

```
@;GenerateE
@; Based on A. Sale and N. Horspool
@; Stores the decimal part of e in the array defined below
.equ nDigits, 20
.equ n, 32 ;this is calculated in the C# version, but just fix it here

@; R0 is the main loop counter -- dd in cs version
@; R1 is the inner loop counter -- i in c# version
@; R2 -- c in the C# version, also used in initing the coeffs
@; R3 is the starting address of the coefficients array
@; R4 is the value to init the coefficents with, when nDigits == 1000, max coeff is
463, use a 32 bit value to start with
@; R5 is used to hold the value read from the coeffs
@; R6 is used to hold intermediate results
@; R9 is the number of digits to calculate e to
```

```
generatee:
       MOV R9,#nDigits
       MOV R0,#0
       MOV R2,#n

       ADR R3,coeffs;start address for initing coeffs
initcoeffs:
       MOV R4,#1
       STR R4,[R3],#4      ;store 1 in the word and update the
                                                                    ;pointer
       SUBS R2, R2, #1     ;set z if result is 0
       BNE initcoeffs      ;branch if z clear
       ADR R4,result; R4 points to the result
       SUB R3, R3, #4      ; R3 points to the last coeff
mainloop:
       ADD R2, R9, #1             ;set up c
       MOV R2, R2, LSR #1  ;c = (nDigits + 1)/2
       MOV R1, #n                 ;init the inner loop counter
       SUB R1, R1, #1             ;i = n - 1
innerloop:
       LDR          R5, [R3]            ; R3 is pointing at coefs element i
       ADD          R6, R5, R5, LSL#3   ; coefs[i] * 9
       ADD     R6, R6, R5        ; (coefs[i] * 9) + coefs[i]
       ADD    R2, R2, R6                ; c = coefs[i]*10 + c
       ADD    R7, R1, #2                ; i + 2 is the divisor
       MOV    R6, R2                    ; copy c to dividend
       BL           divide                     ;modulo is returned in R6
       STR          R6, [R3], #-4; post decrement the address in R3 and store in
                                 ; the coefs array
       MOV          R2, R8       ; store the quotient back in c
       SUB R1, R1, #1            ; i--
       CMP R1, #0
       BGE innerloop      ; i >= 0
       ADR    R3, result          ; reset r3 to point to the last element
       SUB R3, R3, #4             ; of the coefs array
       STR R2, [R4], #4    ; store c and increment the pointer
       ADD R0, R0, #1                          ; dd++
       CMP R0,R9                               ; dd < nDigits
       BLT mainloop
       SWI 0x11
       @; from ARM Assembly Language, William Hohl
divide:             ; input dividend R6, divisor R7, quotient R8,
             ; output  quotient R8, remainder R6
             ; overwrites R5
       MOV R5, #1
divide1:
       CMP    R7,#0x80000000
       CMPCC R7, R6
       MOVCC R7, R7, LSL#1
       MOVCC R5, R5, LSL#1
       BCC divide1
       MOV R8,#0
divide2:
       CMP          R6, R7
       SUBCS  R6, R6, R7
       ADDCS  R8, R8, R5
       MOVS   R5, R5, LSR#1
       MOVNE  R7, R7, LSR#1
       BNE          divide2
       MOV          pc,lr        ;return form subroutine
coeffs:
.space 128          ; this is the space for the coefficients storage, 32 words
result:
.space 80           ; the decimal part of e 20 words for 20 decimal places
```

9.2 Appendix B: Comparing Translation Targets

Both Reflection.Emit and the Expression classes can be used to generate dynamically executable code. Since performance is a primary goal of binary translation it was necessary to compare their performance. This small program compares the time for making an update to the value of an instance field using the Expression class versus performing the same operation using IL via Reflection.Emit.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq.Expressions;
using System.Reflection;
using System.Reflection.Emit;
using System.Linq;

namespace ExpressionCallTest
{
    public class Program
    {
        private static List<long> _expressionTimes;
        private static List<long> _reflectionTimes;

        private static void Main(string[] args)
        {
            _expressionTimes = new List<long>();
            _reflectionTimes = new List<long>();

            var test = new MethodInvocationTest();
            test.TestMethodCall();

            var ftc = new FieldTestClass();
            ftc.DisplayTestField();
            var sw = new Stopwatch();
            var efs = ftc.GetExpressionFieldSetter();
            for (var i = 0; i < 100; i++)
            {
```

```
            sw.Start();
            efs(); // Set field to 1 using expression
            _expressionTimes.Add(sw.ElapsedTicks);
            sw.Reset();
        }
        Debug.Print("Av time to call expression {0} ticks",
         _expressionTimes.Average());
        Debug.Print("Total for 100 calls - expression {0} ticks",
         _expressionTimes.Sum());

        ftc.DisplayTestField();
        var rfs = ftc.GetReflectionFieldSetter();
        for (var i = 0; i < 100; i++)
        {
            sw.Start();
            rfs();
            _reflectionTimes.Add(sw.ElapsedTicks);
            sw.Reset();
        }

        Debug.Print("Av time to call Reflection field set {0} ticks",
         _reflectionTimes.Average());
        Debug.Print("Total for 100 calls - Reflection field set {0} ticks",
         _reflectionTimes.Sum());
        sw.Stop();
        ftc.DisplayTestField();

        Console.WriteLine("OK - hit enter to exit");
        Console.ReadLine();
    }
}

public class FieldTestClass
{
    private int _testField;
    private readonly FieldInfo _testFieldInfo;

    public FieldTestClass()
    {
        _testFieldInfo = typeof (FieldTestClass).GetField("_testField",
         BindingFlags.NonPublic | BindingFlags.Instance);
    }


   // get a delegate to set the field using the Expression class
   public Action GetExpressionFieldSetter()
   {
         var a = Expression.Assign(Expression.Field(Expression.Constant(this),
         "_testField"), Expression.Constant(1));
         return Expression.Lambda<Action>(a).Compile();
   }

   public delegate void DynMethDelegate();

   // get a delegate to set the fields using Reflection
```

```
        public DynMethDelegate GetReflectionFieldSetter()
        {
                // set arg 0 as an instance of the type
                Type[] methodArgs = {typeof(FieldTestClass)};

                var dm = new DynamicMethod("dynMeth", null, methodArgs, typeof
                (FieldTestClass));
                var il = dm.GetILGenerator();
                il.Emit(OpCodes.Ldarg_0);
                il.Emit(OpCodes.Ldc_I4_2);
                il.Emit(OpCodes.Stfld, _testFieldInfo);
                il.Emit(OpCodes.Ret);
                return (DynMethDelegate)dm.CreateDelegate(typeof(DynMethDelegate),
        this);
        }

        public void DisplayTestField()
        {
                Debug.Print("test field = {0}", _testField);
        }
}
```

9.3 Appendix C Code Expansion

The following small basic block of ARM assembler demonstrates the code expansion issue when translated to CIL. These three lines initialize a block of memory with the value that is stored in R3, the address to initialize is stored in R1 and the number of words to initialize is stored in R2.

```
initLp:
        STR R3,[R1],#4 @;Store a word and update pointer
        SUBS R2,R2,#1 @;Next word
        BNE initLp
```

The following is a CIL translation of the above block, color coded for each instruction. Code expansion due to accessing simulated registers and setting simulated CPSR flags is readily evident. The first block in each CIL translation increments the program counter. The STR instruction is simulated with a block of instructions to update the word in memory and a block of instructions to increment the pointer to the next word in memory

```
// Store a word and update pointer
incPc1:
        Ldarg_0                    // push reference to class instance
        Ldfld, _registerInfo       // push ref to the registers array
        Ldc_I4, (int)pc            // push index of pc
        Ldarg_0                    // push reference to class instance
        Ldfld, _registerInfo       // push ref to the registers array
        Ldc_I4, (int)pc           // push index of pc
        Ldelem_U4
        Ldc_I4, 4
        Add                                       // increment the pc value
        Stelem, typeof(uint)      // store pc
        Ldarg_0                   // push reference to class instance
        Ldfld, _programInfo       // push ref to the program array
generateIndex:
        Ldarg_0                   // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 1                         // push index of R1
        Ldc_I4, 4096
        Sub
        Ldc_I4, 2
        Shr
        Ldarg_0                   // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 3                 // push index of R3
        Ldelem_U4                 // push R3 value
        Stelem, typeof(uint)      // store into progarray[R1]

incrR1:
        Ldarg_0                   // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 1                 // push index of R1
        Ldarg_0                   // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 1                 // push index of R1
        Ldelem_U4
        Ldc_I4, 4
        Add                       // increment the R1 value
        Stelem, typeof(uint)      // store R1
```

The translation of the SUBS instruction is constructed from a block of instructions that load the simulated registers and perform the subtraction, followed by several blocks that set the simulated CPSR flags.

```
decrement counter and update flags
incPc2:
        Ldarg_0               // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, (int)pc       // push index of pc
        Ldarg_0               // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, (int)pc       // push index of pc
        Ldelem_U4
        Ldc_I4, 4
        Add           // increment the pc value
        Stelem, typeof(uint)      // store pc
decR2:
        Ldarg_0               // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2             // push index of R2
        Ldarg_0             // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of R2
        Ldelem_U4           // push the value from R2
        Ldc_I4, 1           // push 1
        Sub         // perform subtraction
        Stelem, typeof(uint)      // store element in R2
cflag:
        Ldarg_0             // push reference to class instance
        Ldfld, _flagsInfo   // push ref to the flags array
        Ldc_I4, (int)cFlagIndex   // push index of c flag
        Ldc_I4, 1
        Ldarg_0             // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of register
        Ldelem_U4
        bgt clearCflag
setCflag:
        Ldc_I4, 1
        Br storeCflag
clearCflag:
        Ldc_I4, 0
storeCflag:
        Stelem, typeof(uint)
zflag:
        Ldarg_0             // push reference to class instance
        Ldfld, _flagsInfo   // push ref to the flags array
        Ldc_I4, (int)zFlagIndex   // push index of z flag
        Ldarg_0             // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of R2
        Ldelem_U4           // push the value from R2
        Brfalse setZflag    // r2 == 0, z = 1
clearZflag:
        Ldc_I4, 0
        Br storeZflag
setZflag:
        Ldc_I4, 1
storeZflag:
        Stelem, typeof(uint)
nflag:
        Ldarg_0             // push reference to class instance
        Ldfld, _flagsInfo   // push ref to the flags array
        Ldc_I4, (int)nFlagIndex   // push index of n flag
        Ldarg_0             // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of R2
        Ldelem_U4           // push the value from R2
        Ldc_I4, 31
        Shr         // set n to the MSB of the result
storeNflag:
        Stelem, typeof(uint)
vflag:
        Ldarg_0             // push reference to class instance
        Ldfld, _flagsInfo   // push ref to the flags array
        Ldc_I4, (int)nFlagIndex   // push index of v flag
        Ldarg_0             // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of R2
        Ldelem_U4           // push the value from R2
        Ldc_I4, 31
        Shr          // get msb from op 1
        Ldc_I4,1
        Ldc_I4, 31
        Shr          // get msb from op2
        Beq clearVflag:
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of R2
        Ldelem_U4           // push the value from R2
        Ldc_I4, 31
        Shr          // get msb from op 1
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, 2           // push index of R2
        Ldelem_U4           // push the value from R2
        Ldc_I4, 31
        Shr          // get msb from result
        Beq clearVflag:
setVflag:
        Ldc_I4, 1
        Br storeVflag:
clearVflag:
        Ldc_I4, 0
storeVflag:
        Stelem, typeof(uint)
```

The final block of CIL corresponds to the branch instruction where the not equal prefix checks if the simulated Z flag is clear.

```
branchInitLp:
        Ldarg_0                         // push reference to class instance
        Ldfld, _flagsInfo         // push ref to the flags array
        Ldc_I4, (int)zFlagIndex   // push index of v flag
        Ldelem_U4
        brfalse fallthrough
        Ldarg_0                         // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, (int)pc           // push index of pc
        Ldc_I4, 4132              // next instruction
        Stelem, typeof(uint)      // store pc
        br done
fallthrough:
        Ldarg_0                         // push reference to class instance
        Ldfld, _registerInfo      // push ref to the registers array
        Ldc_I4, (int)pc           // push index of pc
        Ldc_I4, 4120              // start of loop
        Stelem, typeof(uint)      // store pc
done:
```