Using the LLVM Compiler Infrastructure for Optimised, Asynchronous Dynamic Translation in Qemu

By

Andrew Jeffery

A thesis submitted to University of Adelaide
in partial fulfilment of the Degree of Bachelor of Computer Science
with Honours
Department of Computer Science
November 2009

Examiner's Copy

Typeset in $\LaTeX\,2_\varepsilon$.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Andrew Jeffery

Acknowledgements

First and foremost I'd like to thank Dr Bradley Alexander, Mr David Knight and ASTC Pty. Ltd. for presenting me with the opportunity to take on this project and the guidance they have provided. The gratitude for their patience and willingness to debate ideas throughout the year cannot be overstated.

Following this I'd like to thank the people who have informally supported me over the course of the year: This includes my family and partner, Kate, who have had to cope with my often absent nature whilst studying. Special thanks also goes to Joel Stanley and Karl Goetz for their steadfast friendship in the face of having to constantly listen to all of my disparate thoughts and opinions.

Abstract

Instruction set simulation, while convenient for embedded development, introduces overheads into application execution by having to simulate hardware in software. These overheads range from immediately apparent through to impractical depending on the style of simulation and the software being simulated. In this paper we look at an implementation aimed at increasing simulator efficiency through asynchronous optimisation of code blocks, employing the LLVM Compiler Infrastructure as an alternative JIT backend for the Qemu ARM user-space simulator.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Dynamic Translation

Dynamic translation refers to the practice of just-in-time compilation of a source language into an intermediate or native language that can then be executed. Dynamic translation has quite a history; first references to the concepts seemed to have appeared throughout the 1960s[1]. Over time the idea has evolved, moving in and out of popularity until recent times, where languages such as Sun's Java have thrust the term into prominent focus.

1.2 Where are Dynamic Translators used?

Dynamic translators are used for all manner of purposes, indeed several classification systems have been proposed on how to identify them[2][1]. Mostly they are associated with dynamic languages, where type information isn't determined until runtime. By compiling the code as needed, the availability of the type information potentially allows for more intelligent decisions to be made with regards to code optimisation.

Examples of current popular dynamic languages include those based on Microsoft's .NET and Java, though these are not the only consumers of the Just-In-Time (JIT) compilation concept. The abstract notion of source and destination languages allows us to push dynamic translation into practically any computing environment; one such area is dynamic binary translation, the subject of this paper.

1.3 Dynamic Binary Translation

Dynamic binary translation is commonly used for simulating one processor architecture on another, the source language being the instruction set for the original architecture and the destination language the native instruction set. A popular motivation for dynamic binary translation is developing software for embedded platforms, where developers may not have access to the device, or testing the software on the device involves a tedious process that gets in the way of application development.

This project looks at the potential for reducing the wall-clock execution time of the simulation process, as there are many costs involved in instruction set simulation across different architectures. Issues inherent to instruction set simulation can be categorised into roughly four different areas:

- Dispatch and Execution
- Dynamic Translation / Optimisation
- Memory Management
- Device IO

Each of these impose a time penalty on the simulation, as instead of being handled in hardware the points listed must be handled in software (by the simulator).

1.4 Instruction Set Architectures

Different areas of computing are driven by different requirements, and as such instruction set and processor design is an important aspect of machine development.

Embedded computing often has tight requirements as to the amount of resources it can consume, such as power. Typical desktop processor architectures such as Intel's x86 family are designed around performance, and as such are often unsuitable for embedded environments due to their level of resource consumption.

The ARM line of processor architectures have been designed from the ground up with these resource limitations in mind, and through its well-designed, simple, constant length instruction set the processor designs are compact and energy efficient while still offering good performance.

1.5 Project Tasks and Aims

The initial task of the project was to identify which of these areas imposed the greatest time penalty for an ARM instruction set simulator running on an x86-64 machine. The second stage consisted of following this initial work up with a design aimed at reducing the bottlenecks in the process and increasing simulation efficiency.

2 Literature Review

Dynamic translation, as outlined in the introduction, has quite a history behind it already; failing to pay attention to this would be somewhat of a fallacy. We will now survey some of the work already been done in the field in order to gain insight into areas that have potential for improvements

Starting with A Brief History of Just-In-Time Compilation[1] we find that ideas common-place in todays translators have evolved over quite some time. Different language implementations introduce new concepts or ideas on how to deal with problems encountered: for example the original research on LISP seeds the idea of compiling functions fast enough that they don't needn't be stored.

Significant properties of languages outlined by [1] will be discussed briefly here, as their influence has reached through to the tools used throughout the paper. The concepts can be split into roughly two categories: Optimising the code generation and optimisation of the generated code.

With regard to the former, one of the first significant concepts is from the LC^2 language, that of storing instruction *traces* from interpretation. A modern version of this technique has gained popularity recently in the web-space, where optimisations operate on hot traces to improve performance in a Javascript execution engine[3].

Following this, the BASIC language landed the idea of *throw-away compilation*, which at its core is the notion that code that is dynamically generated code can always be regenerated. In the case of memory pressure the code buffer can be either completely or in part discarded, and the dynamic translation performed again as needed.

Finally, the *templating* strategy as implemented by some ML and C dynamic translators makes use of both static and dynamic environments. Prior to execution of code a static compiler is used to generate relocatable stubs, which are filled in at run-time by the JIT compiler. This removes some of the time pressure on the dynamic translator, enabling the simulation to continue quicker than it would otherwise.

The latter, optimising the generated code, is reported to first be considered by FORTRAN. FORTRAN's hotspot algorithm incrementally applies more expensive optimisations as the number of executions of a code block increase. Taking this concept to the extreme was Oberon, introducing *continuous run-time optimisation.* Throughout the life of the application (during idle-time) the optimiser works on rearranging the program for increased cache performance based on the recent history of memory accesses. This strategy was chosen due to the potential for semantic optimisation having an effect being drastically reduced across long running applications; once an optimisation has been applied further applications are typically redundant. Cache misses on the other hand can be adjusted for as the input varies, potentially providing a close to optimal performance.

The final concept explored here and by the paper is *continuous compilation*, introduced by the Java Virtual Machine. When the virtual machine encounters an untranslated section of code, two processes are launched in separate threads: Application execution continues through interpretation of the bytecode, while in the background on the second thread the code is compiled to the native architecture for faster future execution.

Having looked at an abridged version of the history of JIT systems, we will now explore the current space of instruction set simulators and their claims of performance. Simulation style, and indirectly, performance, is divided by the intended use case: Cycle accurate or functional simulation:

- Functional simulation is typically used where strict hardware accuracy is not required, or where the penalty incurred by cycle-accurate simulation is too high due to the increased constraints on emulated hardware timings.
- Cycle accurate simulation is motivated by hardware development or instruction set functionality testing. A cycle accurate simulator ensures that all actions take proportional time to the actual hardware, a process that can be many times slower than a functional simulation.

Having said that, [4] details a method in which functional simulations can be used to approximate cycle accurate simulations through linear regression. While mentioned for completeness, cycle accurate simulators are not considered throughout the rest of this paper.

The first functional simulator studied is outlined in [5], which claims simulation times of between 1.1 and 2.5 times the native execution time. This is a somewhat fascinating claim until the implementation is discussed[1]. To achieve this level of speed the simulator leverages a static compilation technique via a RISC-like virtual machine. This has the unfortunate side-effect of removing support for self-modifying code, ruling out simulating any binary that dynamically loads libraries. The claim made in the paper is that dynamic loading is somewhat avoided in the embedded world, and as such doesn't have the impact it might seem to initially.

The second functional simulator is Simit-ARM[6]. Unlike the previous simulator it doesn't make use of prior static compilation and does support self-modifying code. The primary claim of Simit-ARM is that it was one of the first to attempt concurrent

[1]Benchmarking ARM binaries in Qemu shows it is in the range of seven to 10 times slower than native performance, making this claim suprising

dynamic translation of hot pages, making use of the multi-core architectures that are becoming only more readily available. The simulation strategy is to interpret instructions whilst the associated compiled page is unavailable[2]. Hot host code pages are modelled as functions in C++ (used as an intermediate representation) and compiled by GCC into a shared object. Once the shared object becomes available it is dynamically linked into the simulator; all future executions of a PC value held in a compiled page are dispatched to the appropriate dynamically linked function.

Simit-ARM's approach requires instruction interpretation while there is no compiled code available for the instruction. This leads to a somewhat slow simulation in some circumstances; situations for which Qemu[7] is much better suited. Like Simit-ARM Qemu is an open source machine emulator, known for its speed and wide variety of supported architectures.

As described in its technical paper[8], Qemu's simulation strategy is purely dynamic translation of basic-blocks from the emulated binary. No interpretation of instructions means that the simulation is temporarily halted whilst the translation is taking place, but as the sections of code being translated are small this pause is typically unnoticeable to the user. Once translated the native code remains stored in a static buffer in memory, negating the need for future translations so long as the buffer is not emptied[3].

To improve the execution speed further, Qemu's *direct block chaining* strategy allows native code to jump directly to the following native block[4]. This avoids the need to return to the dispatch loop, where intensive tasks such as interrupt and exception processing take place.

Due to its flexibility and speed Qemu has been the subject of a number of modifications; it is used in the PTLsim cycle accurate simulator[9], llvm-qemu[10] and as the basis for qemu-mips[11] (now the Open Virtual Platform).

[2]Which will always be the case for cold pages

[3]Qemu implements the throw-away compilation strategy discussed in [1]

[4]Representing the subsequent block in the emulated binary

3 Initial Experiments

3.1 The Baseline

3.1.1 Qemu: A Fast Instruction Set Simulator

As discussed in the Literature Review, Qemu is open source, fast and adaptable to different architectures, substantiated by the number that it supports[12]. As it claims to be one of the faster functional simulators in the open source arena[13], it was chosen as the basis for the project's exploratory work.

In order to reduce the complexity of the project, only user-space emulation will be considered throughout the paper. This does not restrict the implementation from being applied to full system-emulation however; Qemu's user-space emulation is a subset of the functionality of full system emulation. To further restrict complexity, the Thumb and Thumb2 variants of the ARM instruction set were disregarded, though the work presented is applicable to these and other architectures.

3.1.2 GNU GCC

The GNU Compiler Collection[14] is the venerable open source compiler maintained by the Free Software Foundation. All software used throughout the project was compiled using the GCC suite, due to its flexibility in terms of architectures and languages supported, as for its stability.

Two versions of the GCC collection were used throughout the project, one for compiling native applications, and a second, cross-compiling version for generating ARM binaries. Qemu and all other native supporting software was compiled with GCC `x86_64-pc-linux-gnu-4.4.2`, while the SPEC 2006 benchmarks, discussed next, used a GCC ARM cross-compile tool chain: `arm-softfloat-linux-gnueabi-4.3.3`.

3.1.3 SPEC INT 2006

In order to profile Qemu's execution some benchmark binaries are required. The SPEC benchmark suite[15] is produced by a number of industry parties including Intel, AMD, Microsoft, Sun Microsystems and Redhat, the aim of which is to provide a suite that enables fair comparison of different hardware, compilers or other technologies involved in the hardware/software development stack.

Utilising SPEC benchmarks according to the guidelines provided, parties can produce results that compare their hardware or software against their competitors. As such, this benchmark suite provides a solid basis for testing what this project aims to achieve: A speed up of the emulation process through augmenting of Qemu, attempting to optimise it for increased performance / decreased wall-clock execution time of simulations.

Throughout this paper, all SPEC benchmarks were cross-compiled with GCC toolchain outlined above and with no optimisations enabled.

3.2 Instrumenting Qemu

Having chosen Qemu as the basis for the project, the aim was to find where it was spending the bulk of its execution time and to try optimise that section. Given that Qemu is a dynamic translator this is a much harder task than for regular applications, as a sizable amount of the code being executed is generated on the fly and is devoid of symbol information. Results devoid of symbol information at the known location of Qemu's static code buffer identified the anonymous regions of execution reported by the OProfile tool as the dynamically translated code. The measurements gathered using OProfile, discussed below, showed that a substantial amount of the wall-clock execution time was spent running code that had been dynamically translated.

Several techniques are common for instrumenting code externally - the first being a Monte Carlo style systems that sample the application's execution at regular points in time. A second style is through instruction set simulation - essentially virtualising the application in order to track what it is executing. This second style provides complete coverage of the instrumented code - we can gather exact data on call graphs, memory allocations, usage and cache profiling. A third option is instrumentation applications similar to the venerable `gprof`, which utilise the symbol table and debugging information to track function calls and execution times.

In addition to the techniques outlined above, some manual profiling of TCG, Qemu's JIT subsystem, was performed. This was at a stage where gprof and Valgrind were both failing to give results, and under the assumption that Qemu would be spending a reasonable amount of time translating basic blocks. This assumption turned out to be incorrect, with various SPEC benchmarks and the DietPC ARM release of Debian showing that it in fact spent very little time translating code. According to the profiling information, approximately two seconds of the 70 seconds it took to boot Debian inside Qemu were spent translating the ARM to x86-64.

Applications implementing each of the techniques outlined above were trialled on Qemu: OProfile for the Monte-Carlo style sampling, Valgrind[16] for the Instruction Set Simulation approach, gprof for execution tracking; each with varying success. As hinted in the opening paragraph instruction set simulators are a special breed of application that have requirements outside of what might normally be expected. Due to this, instrumentation with gprof and Valgrind produced little usable information; gprof caused Qemu to immediately exit with an uncaught signal exception, while Valgrind was unable to identify symbols in the `qemu-arm` image, rendering its results unusable. Turning to OProfile, results were finally obtained that allowed the project to move forwards.

3.2.1 OProfile

OProfile[17] fits into the Monte Carlo profiler style outlined earlier; its core feature is to sample the instruction pointer of the executing application at given intervals and resolve it to the parent function name. In addition it makes use of hardware performance counters present on modern-day chips such as Intel and AMD processors to enhance its accuracy. Other statistics such as callgraphs can also be gathered for further in-depth analysis of the application. OProfile consists of a Linux kernel module along with several user-space support utilities to control the profiling daemon and output reporting, including emulating `gprof` output formats.

OProfile was the tool of choice for instrumentation due to its simplicity and lack of impact on the wall-clock execution time of applications being profiled. The graph presented in Figure 3.1 demonstrates approximate proportional time spent executing dynamically translated code, sitting between 60% and 80% across the SPEC benchmarks trialled. This demonstrates the need for perhaps better optimisation of the JITed code, though in a manner that will cause minimal impact on the simulation itself.

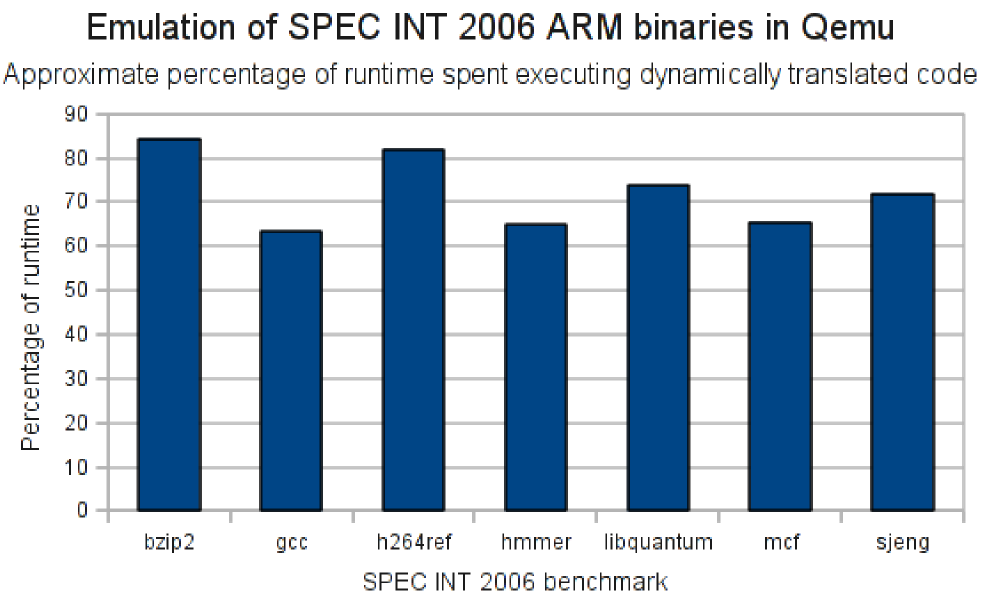


Figure 3.1: Profiling Qemu emulating different SPEC INT 2006 benchmarks. The bars represent the percentage of time Qemu spent executing dynamically translated code over the entire simulation for each benchmark

4 Project Architecture

4.1 Consumer Hardware and Simulator Design

Optimising the wall-clock execution time essentially comes down to ensuring the following:

1. The hardware framework is emulated as efficiently as possible
2. IO is performed as efficiently as possible
3. Target code is generated as efficiently as possibly
4. The generated target code is optimised for minimal use of cycles

Taking into account that the aim is to optimise the generated code, points 3 and 4 are relevant here. These are somewhat conflicting goals on single core machines, an environment that Qemu is optimised for. By performing optimisations on the target code we spend longer in the target code generation process, blocking all other execution while doing so. Introducing this overhead may still be of benefit in the case that the block is hot, i.e. it forms part of a loop, or a frequently called function.

In the case of a single core machine, augmentation of Qemu in the manner proposed below will very likely not have a positive effect unless the benchmark is small in size and quite long running. In order to obtain the optimised code we give the machine more work to do - in this case we'll be generating the code twice as well as optimising it! On a single core machine the implication is in order to do the second, more optimised JIT compilation we have to interrupt the simulation, increasing the time it takes to execute.

In this instance if more optimised code was to be generated, it should be performed in the first JIT compilation. While this delays the execution of the block[1] we do get the more optimised code, which, if it turns out to be a hot block, should give a performance boost. However this is at the cost of cold blocks also being optimised; depending entirely on the type of application this may turn out to be a large disadvantage. Optimising code blocks in Qemu in this way has been attempted; the llvm-qemu project is the end result of these efforts.

Due to the increasing pervasiveness of multi-CPU, and more frequently, multi-core machines, along with the potential benefits of being able to chose the blocks we wish to spend time optimising[2] the structure outlined below was arrived at.

4.2 Architecture Description

The central idea of the project is similar to concepts that characterised Java and Simit-ARM in the literature review: the utilisation of separate threads and cores to optimise code asynchronously, and to push the optimised code into Qemu as it became available. As TCG only has small support for optimising its own intermediate representation (IR),

[1]As we have to wait for it to be generated before we execute it

[2]llvm-qemu, due to its design, pushes all blocks through LLVM's JIT, which depending on the number of optimisations applied is quite heavyweight

it was decided that an external optimising compiler should be used for simplicity[3]. Given there are several open source Just-In-Time compiler libraries available (libJIT and GNU Lightning for example), the question was which should be used?

4.2.1 The LLVM Compiler Infrastructure

The choice was the LLVM Compiler Infrastructure[18], which is designed as a set of libraries on top of which compiler front-ends can be implemented. At the core of the LLVM project is the single static assignment IR, on which may different local and global optimisations operate. Included as part of the suite is a JIT compiler for various architectures, most importantly for this project including x86-64. Projects such as Clang[19] (a new C-family front-end starting to compete with GCC), the Google-sponsored Unladen Swallow[20] (a Python implementation) and Rubinius[21] (a Ruby implementation) all make use of the LLVM infrastructure for target platform independence and for producing fast and efficient machine code. This combination of features and high-profile projects made it the suite of choice for implementing an optimising front-end for Qemu.

Back to the architecture, the second choice was where to intercept the translation process and to begin producing LLVM's IR.

1. Directly translate ARM blocks to LLVM IR
2. Wrap LLVM IR API calls up in TCG IR API calls
3. Take the TCG generated target code and translate to LLVM IR

Item 2 was the subject of an initial attempt to integrate the LLVM JIT into Qemu with the llvm-qemu project. The llvm-qemu project is distinct from the this investigation in several ways; firstly it uses an older release of Qemu containing the `dyngen` translation engine, which is heavily tied to the `GCC3` suite. Secondly, wrapping LLVM

[3]Rather than developing optimisations to work on TCG IR

IR up in TCG IR API calls effectively replaced the TCG backends with LLVM's JIT, removing the some of the speed of the original engine[22].

Items 1 and 3 are perhaps extreme options, though naturally have distinct and interesting properties. Translating directly from ARM ensures we capture the semantics of the block as succinctly as possible. The line of reasoning was that calling into the LLVM IR API from inside the TCG IR API abstracts what the ARM instructions might have originally intended, and doubly so with translating from the TCG generated x86.

Transforming the TCG generated x86 into LLVM IR, optimising then emitting the result back into Qemu's code cache was a little considered idea for a variety of reasons. As discussed above we are already abstracted away from what the original intent of the ARM block was, and additionally have the problem of variable length instructions with a multitude of *addressing modes*. Somewhat bringing this back into consideration is that LLVM has a x86 front-end already written for it, essentially giving the opportunity to simply glue the parts together and be done with it.

Essentially the decision came down to the level of abstraction from the original ARM block, and as such translating directly from ARM into LLVM IR was the chosen path. This seemed like a reasonable metric at the time, but over the course of the project it became clear that generating LLVM IR from the TCG IR would also be quite a reasonable choice.

Having decided that multi-core machines were only going to grow in popularity[4], that LLVM was a good candidate for highly optimised code generation and that translating ARM to LLVM IR would produce the best results, the architecture of the project was laid out so: Qemu and LLVM would operate in separate threads, allowing each to be scheduled simultaneously on separate cores. Qemu, with its basic-block translation units, would enqueue candidate blocks in the external compilation system as it encountered them, while the LLVM thread would fetch queued jobs as they became available. On dequeuing a job the LLVM front-end would perform the translation and

[4]a view shared by the authors of Simit-ARM[6]

optimisation, placing the resulting target code in an output queue. At an appropriate time Qemu would fetch completed jobs from the output queue and replace TCG's target code with LLVM's.

4.3 Further Experiments: Instruction Popularity

As outlined above, the planned design would take ARM instructions and transform them into LLVM IR through the LLVM IR API. The various ARM architecture revisions contain approximately 114 instructions and as such an important question, given the temporal limitations of the project, was *which* instructions should it implement?

The metric used to define which instructions would be implemented was relatively simple: A popularity contest run over the SPEC INT 2006 benchmarks as cross compiled to ARM by GCC 4.3.3 (arm-softfloat-linux-gnueabi). Taking the most popular instructions ensures we have the greatest chance of making a difference whilst minimising the amount of work to be done. Using Qemu's debug flag with the `in_asm` option logs instructions in ARM blocks as they are encountered. This output was analysed with the final results outlined in Table 4.1.

Due to implementation details (discussed further on), we can discount the branch instructions, leaving 10 popular instructions from the list. In the end this list mainly served as a guide, as further implementation issues dictated the difficulty of teaching the translator about different instructions.

Instruction	**Sum**
ldr	325064
mov	154253
str	112354
cmp	72717
sub	69163
add	52213
bl	49092
beq	25126
bne	21113
bx	16097
ldm	15263
push	12080
b	11150
lsl	9043
ldrb	1178

Table 4.1: ARM instruction popularity across 11 SPEC INT 2006 benchmarks compiled by GCC4.3.3

5 Implementation

Based on the architecture description outlined in the previous chapter, we begin to explore the implementation of the system and how the design of the tools affected it.

5.1 Exploring Qemu

As discussed in Chapter 4, the core integration strategy was to have the LLVM front-end/JIT on a separate thread, enabling the underlying operating system to schedule it on a separate core as necessary. The naive view here is that we are effectively getting the optimised code for free[1], though issues like resource and lock checking/contention plus code injection overhead make this not entirely true.

Here we inspect the operational aspects of Qemu, specifically looking at structures

[1]as we're not interrupting the simulation to generate it

involved in the translation process and how the involved functions effect and are affected by them. Qemu carries a lot of context information around about under what conditions a block was translated and how it fits back into the overall system.

As mention previously, Qemu's modus operandi is to translate basic-blocks worth of instructions at a time, caching the result for future executions. Optimisations like *direct block chaining* allow it to continue execution of translated code without requiring a jump out to the dispatch loop; this is handled by encoding the next block to be executed as part of the current block. Blocks can be chained and unchained arbitrarily in time as the emulator sees fit (though according to certain criteria), and as such the functionality needs to be either emulated exactly by the LLVM-based system or left alone. Understanding these small details is essential to ensuring the system works as it should while allowing the LLVM translation process invisibly replaces blocks in the background.

5.1.1 Generating TCG IR in Qemu

Here we will take a brief tour through the internal workings of Qemu and its JIT subsystem, TCG (the Tiny Code Generator) in order to gain an understanding of its operation prior to discussing the implementation of the proposed system. As we are designing the system primarily with userspace emulations in mind, this tour will start in the Linux userspace emulation code then move through the dispatch and translation subsystems. The list below outlines the call sequence from `main` through to TCG IR generation and code emission:

1. `main`: Performs option processing, loading the image to be emulated and initialising the soft CPU state.

2. `cpu_loop`: Initiates execution of basic block sequences. Following the execution of the basic blocks, it handles exceptions and signals.

3. `cpu_exec`: Prior to executing individual translated blocks, `cpu_exec` checks for outstanding exceptions and jumps out to `cpu_loop` if any have occurred. Following the exception handling, any pending interrupts are dealt before seeking out

the next block to be executed.

4. `tb_find_fast` / `tb_find_slow`: Each of these locate the next block to be executed, the latter being called in the case that the former fails to find the appropriate TranslatedBlock struct. `tb_find_slow` always returns the block to be executed, as on failing to locate it, the translation process is triggered.

5. `tb_gen_code` / `cpu_gen_code`: Both initialise the TranslationBlock struct and co-ordinate the translation process respectively.`cpu_gen_code` dispatches individual instructions from the host block to the appropriate disassembly front-end, which in our case is the ARM front-end.

At this point the ARM block being translated has been transformed into TCG's IR (Intermediate Representation), where it is subjected to some quick optimisations before the final transformation into semantically equivalent native code. This second transformation from an intermediate representation to the native code, whilst interesting, doesn't concern this project. We are primarily interested in the process outlined above, and how to perform it asynchronously with the LLVM compiler infrastructure to generate better host code.

5.1.2 Thread Management

From the architecture defined above, where the LLVM front-end is residing on a separate thread to avoid introducing unnecessary overhead into the simulation, we need some way to communicate jobs between Qemu and the LLVM front-end. The data structure chosen for this was a ring buffer; operations on the queue are then constant time, the buffer takes up a constant amount of space and as a datastructure is quite simple to implement.

A ring buffer structure is used at each "end" of the LLVM front-end, one for Qemu to insert jobs into and for the LLVM front-end to pull jobs from, along with another for the LLVM front-end to push finished jobs into and for Qemu to pull the finished jobs from. As these data structures are shared between threads some form of locking needs to take place in order to preserve sanity and correctness. Discussion of the locking techniques used follows a look at what threading technologies are available and which one was chosen to work with.

Several different threading or workload distribution implementations are available. Pthreads is a POSIX threading interface available across most operating system platforms. Qemu already makes use of Pthreads for farming off IO jobs, and as such is already well entrenched in the code base. The cross-platform nature, well worn interface and the fact that it was already integrated into Qemu quickly made it the implementation of choice. Other implementations are available such as OpenMP, though this is somewhat newer and requires a compiler enabled with OpenMP support before it can be used. Having said that, it seems that OpenMP has taken a relatively nice approach to parallelism, abstracting the sometimes difficult lock management issues away from the programmer and embedding the smarts in the compiler.

In order for the simulation to have high potential for operating quicker with the addition of the optimising JIT, we want to hold it up as little as possible when trying to insert or retrieve jobs from the LLVM front-end queues. As such, if a ring buffer is locked when the simulation thread attempts to access it, Qemu should not block until the lock becomes free but instead continue with the simulation. This applies to both queues in question: It's likely that one optimised block won't make so much of a difference if the simulation thread can't obtain the lock to the job insertion queue. Likewise, not obtaining a lock on the finished jobs queue simply means that the simulation thread will try next time it comes around, emptying the queue when it does acquire the lock.

Pthreads offers access to three different styles of lock - *mutex*, *reader-writer* and *spinlocks*. Mutexes were chosen for locking the new/completed job queues, as reader-writer and spinlocks aren't appropriate for the problem. Spinlocks aren't appropriate because we are unable to signal the LLVM thread when work becomes available. The reader-writer lock paradigm doesn't match the situation at hand as both threads will always change values in the ring buffer whenever they want to obtain a lock[2].

[2]They will always enqueue or dequeue objects from the queues

The LLVM front-end, being a separate thread to the simulation, can block as long as it likes. While we'd like to have the optimised blocks available as soon as possible, there isn't an immediate requirement on them being available[3].

Queuing and Dequeuing LLVM Jobs

Having looked at how we interact with the two threads, we will discuss where we introduce the threading into Qemu. The thread should only be instantiated once, and needs to be available to accept jobs once the simulation begins. This indicates that the instantiation should be near the beginning of the Qemu's execution, somewhere in the area of the `main` or `cpu_loop` functions outlined in Section 5.1.1. The entry point of `cpu_loop` prior to its main loop was the place chosen.

While still specific to the `linux-user` front-end, it's removed from the option processing focus of the `main` function. Pushing the thread's instantiation deeper into Qemu's architecture would reduce the boilerplate code in each front-end[4], though checks would be required in each loop iteration[5] to ensure a thread is not instantiated if one is already running.

Given that the thread is now instantiated and waiting for jobs, the location of the job insertion and retrieval code should be discussed. While the job processing is asynchronous, the position of this code is not entirely arbitrary. Certainly the job insertion code should at least reside in a place that is logical with respect to the Qemu/TCG code; from the description of Qemu/TCG's translation process outlined above a relevant place seemed to be in `tb_gen_code`, after processing of the ARM block by `cpu_gen_code`. Figure 5.1 demonstrates the small modification required to insert and retrieve jobs.

It's important that the enqueuing of jobs does not take place prior to this point, as the TranslationBlock structure, described next, will not have all the appropriate metadata initialised. In order to keep changes as local as possible, immediately after

[3]Unlike with TCG, where the simulation is held up until the translated code becomes available

[4]i.e. in the other user-space front ends, `bsd-user` and `darwin-user`

[5]There are several main-loop style loops distributed through different functions.

```
TranslationBlock *tb_gen_code(CPUState *env,
                              target_ulong pc, target_ulong cs_base,
                              int flags, int cflags)
{
    TranslationBlock *tb;
    ...
    /* Generate TCG IR and native code. 'env' is a pointer to the
    CPUARMState struct, 'tb' is the metadata struct for the current
    block and 'code_gen_size' contains the length of the generated
    native code once it has been emitted. */
    cpu_gen_code(env, tb, &code_gen_size);
    ...
    /* If the LLVM front-end can translate the block, enqueue it */
    if(tb->llvmable) {
        ext_blk_cmpl_new_job(&lts, tb);
    }
    /* Check if the LLVM front end has finished any jobs, and if so,
    replace the TCG-generated code for the associated block. */
    ext_blk_cmpl_replace_block(&lts);
    return tb;
}
```

FIGURE 5.1: Qemu's `tb_gen_code` function, augmented with the LLVM front-end's enqueue/dequeue code. The functions of the structs shown are described in later sections

calling `ext_blk_cmpl_new_job` to enqueue jobs the output queue is tested for finished jobs. If any are found, the jobs are dequeued, with `ext_blk_cmpl_replace_block` injecting the LLVM generated code into Qemu's code cache until either the output ring buffer is empty, or the simulation thread cannot immediately obtain a lock on the output ring buffer.

The astute reader might raise the question about checking whether a block can be replaced in `cpu_exec`, prior to locating the block to be executed using `tb_find_fast` or `tb_find_slow`. In this circumstance it's possible that the block could be replaced relatively soon after the LLVM-generated code becomes available, as the output buffer would be checked frequently. In the current position, the block injection code only runs when other blocks require translation; this not necessarily the optimal case as the optimised block might be replaced quite late, potentially after it's required. In our experiments the former in introduces a considerable slowdown, as the output ring buffer is checked throughout the entire simulation; injecting the LLVM translated code from the position outlined in the code above reduces the frequency at which the output ring buffer is checked by forcing it to happen only when calls are made into TCG.

5.1.3 The `TranslationBlock` Structure and Code Caching

The `TranslationBlock` struct inside Qemu carries the block metadata for translation and execution. Integration into the front-end is not required for block semantics to be correct, but in order to place the externally generated code in the correct position for execution it is vital. As such we don't need to emulate this structure inside the LLVM translator - the information is purely used as glue for the two systems.

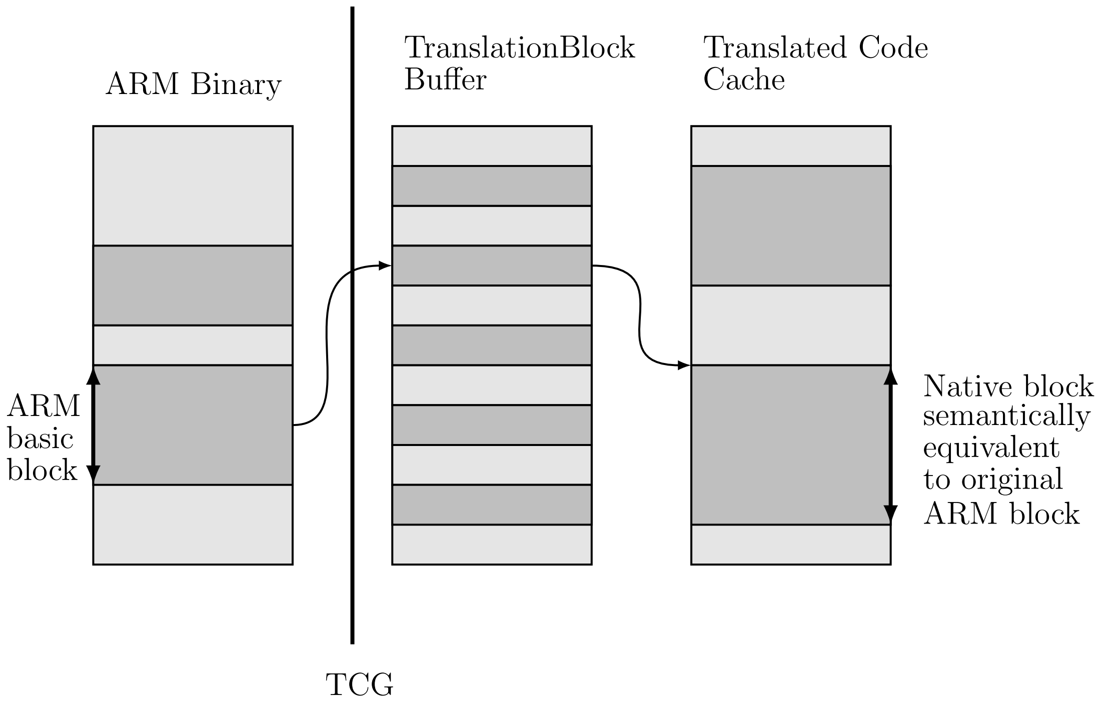
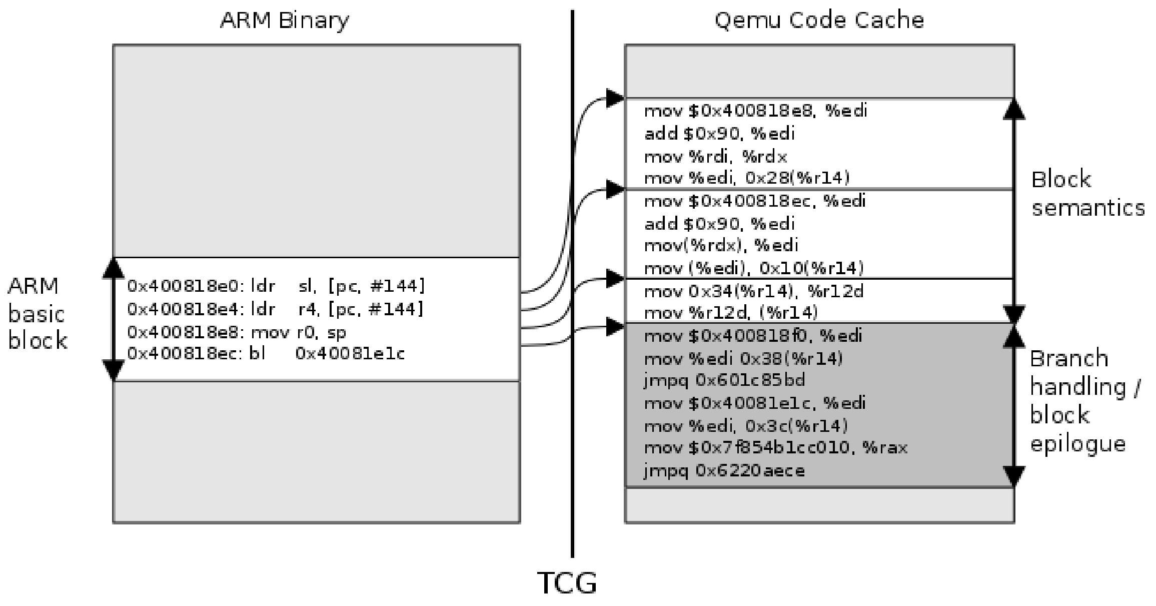


Figure 5.2: An abstract view of some structures involved in Qemu's dynamic translation/execution framework. An ARM basic block is mapped to a TranslationBlock, which in turn points into the translated code cache at the place where the native code for the ARM basic block was emitted



```
0x400818e0: ldr   sl, [pc, #144]
0x400818e4: ldr   r4, [pc, #144]
0x400818e8: mov r0, sp
0x400818ec: bl    0x40081e1c
```

```
mov $0x400818e8, %edi
add $0x90, %edi
mov %rdi, %rdx
mov %edi, 0x28(%r14)
mov $0x400818ec, %edi
add $0x90, %edi
mov(%rdx), %edi
mov (%edi), 0x10(%r14)
mov 0x34(%r14), %r12d
mov %r12d, (%r14)
mov $0x400818f0, %edi
mov %edi 0x38(%r14)
jmpq 0x601c85bd
mov $0x40081e1c, %edi
mov %edi, 0x3c(%r14)
mov $0x7f854b1cc010, %rax
jmpq 0x6220aece
```

FIGURE 5.3: An ARM block and the equivalent x86-64 instructions as generated by TCG. As can be seen each ARM instruction decomposes into several x86-64 instructions. The branch handling assembler is also highlighted, used by Qemu to jump either back to the dispatch loop or to the next block.

The generated code that lands in the cache has a certain structure to it in order to handle branches at the end of the ARM basic blocks. Translating the original ARM branch instruction's location into the equivalent target instruction will break the simulation - we'll wind up in a location containing ARM code, not native code. For the branch, several cases are possible in Qemu: it either jumps out to the dispatch loop to find the next block, or, if the block to be executed next has been generated, Qemu may chain them together to avoid the dispatch loop[13].

The chaining code is is shaded dark grey in Figure 5.3. Depending on the route taken with regards to integrating the LLVM-generated code back into Qemu, two approaches can be taken. The first is to generate the epilogue in the LLVM front-end and to calculate the correct jumps as would be performed by the original TCG epilogue. The jump values would necessarily be different, relative to where the JITed block of code was placed by LLVM. This is most likely suboptimal: We would be reimplementing the epilogue logic for no real gain as it has to have the form Qemu expects[6]. The second option was to replace the TCG-generated code with the LLVM-generated code

[6]It cannot be optimised

at the location pointed to by the `pc` in the corresponding `TranslationBlock` (see Figure 5.4), leaving the TCG-generated epilogue in tact and in use. The implications of this second option are discussed in the following section.

5.1.4 Block Injection Techniques

Block injection concerns how we integrate the code generated by LLVM back into Qemu. Whatever approach is taken, it has to integrate transparently and in a way such that it doesn't cause obvious overhead to the simulation. This can be achieved in a variety of ways:

1. Create a new TB for the translated code's associated ARM instruction pointer, and insert this into Qemu's block list
2. Overwrite the instructions at the translated block's code cache pointer with a call into LLVM's JIT
3. Overwrite the instructions at the translated block's code cache pointer with a jump out to the LLVM JIT's code cache
4. Overwrite the instructions at the translated block's code cache pointer with the code from LLVM's JIT

It was the latter case that was chosen as a solution; the front-end was designed in such a way as to avoid generating instructions that might contain relative or absolute references that may break when the code is shifted in memory[7]. The first option listed was the least discussed for several reasons. Using this method we would have to regenerate the block epilogue[8], which as discussed above was decided to be a task to be avoided.

[7]In experiments with GCC at least, absolute references were compiled down to relative jumps. As such avoiding conditional execution in output blocks may or may not have been necessary

[8]Due to relative jumps to locations being moved in memory

```
struct TranslationBlock {
    /* simulated PC corresponding to this block (EIP + CS base) */
    target_ulong pc;
    target_ulong cs_base; /* CS base for this block */
    /* flags defining in which context the code was generated */
    uint64_t flags;
    uint16_t size;       /* size of target code for this block (1 <=
                            size <= TARGET_PAGE_SIZE) */
    ...
    uint8_t *tc_ptr;     /* pointer to the translated code */
    ...
    /* the following data are used to directly call another TB from
       the code of this one. */
    uint16_t tb_next_offset[2]; /* offset of original jump target */
#ifdef USE_DIRECT_JUMP
    uint16_t tb_jmp_offset[4]; /* offset of jump instruction */
#else
    unsigned long tb_next[2]; /* address of jump generated code */
#endif
    ...
    uint32_t icount;
    /* Fields added for LLVM front-end integration */
    target_ulong epilogue_pc; /* pointer to the br/jmp insn at end of ARM BB */
    uint16_t *opc_epilogue_ptr; /* pointer to br/jmp handling IR at end of TB */
    uint8_t *tc_epilogue_ptr; /* pointer to br/jmp handling code at end of TB */
    int llvmable;
};
```

FIGURE 5.4: Qemu's TranslationBlock struct from exec-all.h, showing some core fields used in the dynamic translation process. The final four members are initialised throughout TCG's translation, and are used by the LLVM code injection function (`ext_blk_cmpl_replace_block`) to overwrite the TCG-generated code correctly

The main motivation for this choice was that it introduced the least overhead at run time. The cost of calling out to instructions in the LLVM JIT was not ascertained, but having the code inline would arguably cause less overhead.

With respect to the second item outlined above, this may have a prohibitive cost on the first call to the function if not managed correctly. This cost would be the time taken to generate the function if it had not happened already, effectively making the design of putting the LLVM front-end on an asynchronous thread void as the emulator would block waiting for the code to be generated. This could be avoided by also asynchronously generating the function, which is what the method chose also does; this approach is necessary in order to inject the code into the Qemu code cache to begin with.

Something further to consider is that the target code block generated by LLVM will (hopefully) be shorter that the TCG generated block. The question in this circumstance is what to do once Qemu has finished executing the shorter code. Certainly it would not be correct to continue to execute the TCG generated instructions following, which might also be invalid due to the property of variable instruction lengths in x86(-64). Two solutions are proposed:

- Insert a *nop-slide*[9] from the end of the LLVM block down to the TCG generated epilogue.
- Perform a relative jump over the bytes in the gap between the end of the LLVM block and the TCG epilogue

Currently the former bullet is implemented, albeit with a little bit of style. Gaps up to 15 bytes are filled with optimised x86 nops[10]; the typical x86 nop is the one byte value `0x90`. Inserting many single byte nops theoretically has a slight negative effect on execution time due to each being an individual instruction and needing to be treated as such. The optimised nops mostly comprise of longer instructions moving values back

[9]A nop-slide is a contiguous set of "no operation" instructions which allows the CPU to "slide" from one location to another without modifying any state

[10]These nops have been taken from the GNU Assembler project - this fact is also noted in the source

into their own registers. This is more efficient in the sense that less instructions are pushed into the pipeline as less instructions are taking up the same space in memory. However, according to benchmarks, this technique does not seem to make a visible difference.

As noted in Future Work (Chapter 7), jumps to epilogues is a technique that is intended to be implemented in the short term. It is expected this would have some visible impact, as it removes the need to fetch and execute any instructions between the end of the LLVM block and the beginning of the epilogue. Gaps of 40 to 50 bytes are not uncommon[11] and it's expected that jumping over these, especially in hot blocks, should show some improvement in runtime.

5.1.5 The `CPUARMState` Structure and Modelling Basic Blocks

The `CPUState` set of structures form the core of Qemu's emulation system. This set of structs, specifically the `CPUARMState` struct in our case, keeps track of the execution environment that would normally be handled by some ARM-based hardware. This includes the user register set[12], banked registers, the CPSR[13] and various modes that ARM CPUs can operate in[14].

In the translated code, TCG turns ARM register references into memory references pointing into a register array, defined in Figure 5.5. Practically speaking, this is the only real strategy available, as we are running software for another architecture inside a piece of software. The emulated application's register values will be clobbered on jumping out to the emulator, and as such they must be preserved in memory in order for the emulated application to execute correctly.

TCG performs some micro-optimisations regarding register allocation at code generation time, storing one or two highly reused values in registers until they are no longer required. This avoids perhaps costly writes to memory and gives a speed increase with

[11]According to observations from the Qemu debug log

[12]The first 16 registers on the ARM architecture, the last 15 registers are for system operation; interrupts and the like

[13]There are multiple CPSR registers in most ARM chips[23]

[14]*Modes* in this context refers to the different instruction set types: *ARM*, *Thumb* and *Thumb2*

```
typedef struct CPUARMState {
    /* Regs for current mode. */
    uint32_t regs[16];
    /* Frequently accessed CPSR bits are stored separately for efficiently
       This contains all the other bits.  Use cpsr_{read,write} to access
       the whole CPSR. */
    uint32_t uncached_cpsr;
    ...
    /* cpsr flag cache for faster execution */
    uint32_t CF; /* 0 or 1 */
    uint32_t VF; /* V is the bit 31. All other bits are undefined */
    uint32_t NF; /* N is bit 31. All other bits are undefined.  */
    uint32_t ZF; /* Z set if zero.  */
    uint32_t QF; /* 0 or 1 */
    uint32_t GE; /* cpsr[19:16] */
    uint32_t thumb; /* cpsr[5]. 0 = arm mode, 1 = thumb mode. */
    uint32_t condexec_bits; /* IT bits.  cpsr[15:10,26:25].  */
    ...
} CPUARMState;
```

Figure 5.5: Qemu's `CPUARMState` struct from `target-arm/cpu.h`, showing some core fields used in emulation, particularly the `regs` array.

the reduction in the need to go to cache or main memory in order to retrieve values that might have been stored only the previous instruction.

Naturally, we want to retain compatibility with Qemu in the blocks we are generating externally through LLVM, and as such compatibility with this core structure is required for transparent integration. We will see later on that the front-end is mainly concerned with the very first field shown in Figure 5.5, `regs`. Creating transparent access to this was enough for the instructions implemented, but if further instructions are to be realised other members of the `CPUARMState` struct should also be implemented. Probably the most important would be those members also shown in Figure 5.5: `uncached_cpsr`, `CF`, `VF`, etc.

The `CPUARMState` struct is defined and initialised by Qemu, however we need access

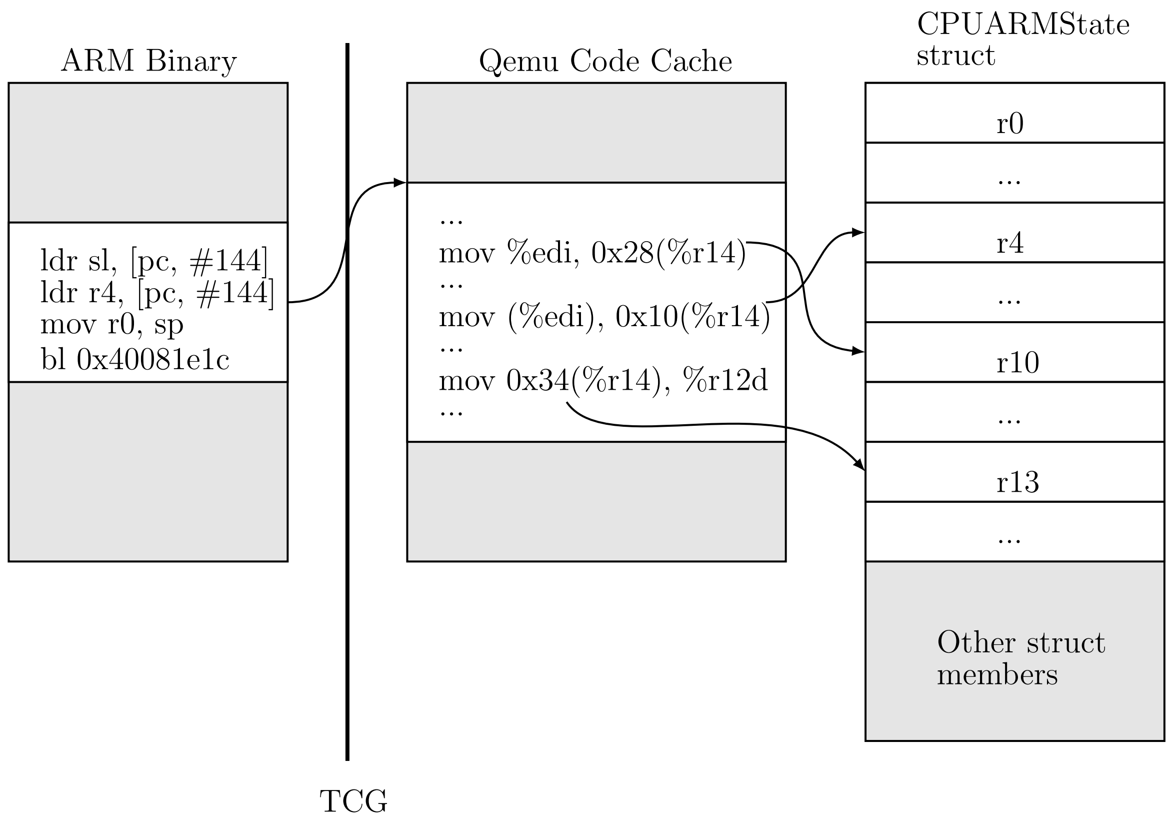


FIGURE 5.6: The diagram demonstrates how the native code emulates accesses to the ARM register set and how it utilises the pinned register value (`r14`, pointing to the CPUARMState struct). Accesses to registers and other members of the struct are simply described as offsets rather than magic memory addresses.

to it inside LLVM. Necessarily, we don't want LLVM to define the structure as part of its generated code, but instead to understand the *view* into the data structure - i.e. how it fits together in memory and where its members lie.

A concern at this point is whether LLVM lays out structures in memory the same way as GCC, as GCC was used to compile Qemu in all instances. In the code show in Figure 5.7, this is controlled by the `LLVM_STRUCT_NOT_PACKED` constant passed to the `LLVMStructType` function. Passing 0 (the value of the constant) instructs LLVM to lay out the members as they appear in the array defining the members, and not in some other, perhaps more optimal, configuration.[15]

[15]The astute observer will note that there is no alternative configuration in this case, as the members are all 32bits wide

```
...
#define QEMU_ARM_NUM_REGS          16
#define LLVM_STRUCT_NOT_PACKED   0
...
/* Sets up dc->function and dc->cpustate */
inline void
initialise_llvm_function(LLVMARMDisasContext *dc, const char *name)
{
    /* Equivalent to 'uint32_t regs[16]' in CPUARMState */
    LLVMTypeRef struct_elements[] =
        { LLVMArrayType(LLVMInt32Type(), QEMU_ARM_NUM_REGS) };
    /* Define the struct containing regs[] (CPUARMState in Qemu) */
    LLVMTypeRef llvm_cpustate =
        LLVMStructType(struct_elements, 1, LLVM_STRUCT_NOT_PACKED);
#define ADDR_SPACE 0
    /* Define a pointer to the CPUARMState-like struct type we
    defined above */
    llvm_cpustate = LLVMPointerType(llvm_cpustate, ADDR_SPACE);
#undef ADDR_SPACE
    /* Define the function we want to build the basic block in */
    LLVMTypeRef args[] = { llvm_cpustate };
    dc->function = LLVMAddFunction(dc->module, name,
            LLVMFunctionType(llvm_cpustate, args, 1, 0));
    dc->cpustate = LLVMGetParam(dc->function, 0);
    /* dc->cpustate now contains a pointer type to the CPUARMState
    struct, passed as a parameter to the function we're defining */
    ...
}
```

Figure 5.7: Describing the `CPUARMState` struct and a single member, `uint32_t regs`, in LLVM. `initialise_llvm_function` also does the ground work for setting up the function representing the ARM basic block to be translated

Referencing `CPUARMState` in LLVM-generated Blocks

At this point another of Qemu's optimisations should be noted. As may be implied from the description above, `CPUARMState` is a frequently accessed struct. It would perhaps be expensive if the pointer to this struct ever got spilled and then refetched from cache/main memory. To avoid this case, Qemu makes use of a GCC feature that enables applications to reserve a register and *pin* a value in it. In effect, this removes the *pinned* register from the allocator's pool and as such the value should not be overwritten.

LLVM lacks support for global registers[24][16], and as we're generating code with LLVM that is aimed at replacing blocks of instructions inside Qemu[17] we must be wary of LLVM clobbering it. The register in question is `r14` on the x86-64 architecture, an interesting and appropriate choice as it turns out.

As defined in the x86-64 ABI[25], `r14` is reserved for the caller function, so for compilers conforming to the x86-64 ABI it can be guaranteed that functions generated will either not use this register, or push it to the stack and restore it before returning. This behaviour is rather favourable, as we know that code LLVM generates must provide that value back to the parent, thus preserving it.

Given the pointer to `CPUARMState` is pinned in a register, it would be nice if we could make use of it in the code generated by LLVM. It isn't immediately obvious how this might work however - we cannot easily tell LLVM this value is in `r14` because it's hard to influence the register allocation choices. There are some workarounds however:

1. Define a new calling convention that modifies all function calls to pass the pointer as the first value

2. Define LLVM functions to represent ARM blocks and pass the pointer as the first parameter

Point 1 is entirely possible; LLVM allows for new calling conventions to be defined as necessary, though this isn't a job to be taken lightly. Point 2 ended up being the chosen approach; while perhaps hack-ish, minimal effort is required to enable it. As it

[16]In practice it may be hard to guarantee the value hasn't been changed, and this appears to be the reason for the lack of support in LLVM for this feature

[17]That also know about the pinned value

turns out the effort of implementing a custom calling convention would only save one instruction overall, severely reducing its chances of being implemented.

The pointer, as far as the function is concerned, is now usable in the function; LLVM knows what register the value is in due to it being passed as a parameter. As it turns out on x86-64 the first parameter is passed in the `rdi` register given that it's a pointer to a struct. Our value is still in `r14` however, and the magic method of getting it to `rdi` isn't entirely pretty. Prior to injecting the LLVM-generated code into Qemu, `mov %r14, %rdi` is manually injected into the buffer, which the LLVM code then follows. As `rdi` is not expected to be preserved by the callee, no more has to be done with regards to looking after Qemu's environment.

5.2 Generating the LLVM Intermediate Representation

We now begin to explore the ARM architecture in a little depth, as this knowledge will be crucial to understanding how to approach implementing an ARM front-end for LLVM and integrating it back into Qemu. In essence, ARM is a *load/store* architecture consisting of 32bit fixed length instructions[18]. The design is purposefully simple in order to provide for extremely low power budgets, though it has some interesting tweaks to increase throughput and instruction density. All information presented here on the ARM architecture is outlined in the ARM Architecture Reference Manual[23].

While reading through the following sections, it pays to keep in mind the following: The ARM-LLVM front-end is split into roughly three different phases, mostly implemented as `switch` statements. The first phase is concerned with fetching raw operands common to most instructions from either the instruction itself (immediate values) or the `CPUARMState` register array. The second phase involves addressing-mode calculations on the operands, such as shifts, to reach the actual operand value. The final

[18]For the original ARM instruction set. The *Thumb* instruction set is 16 bits wide, giving greater code density, but is less expressive. Because of this, *Thumb2* was designed, which stretches the instruction width back out to 32 bits

phase is to perform the operation on the operands and store the result as necessary. The narrative will follow only the implementation of the `LDR` and `STR` instructions, as most other instructions are implemented in a similar fashion.

In the ARM-LLVM front end the following instructions were implemented: `LDR` and `STR` from the *Load/Store Immediate/Register* instruction categories and `ADD`, `AND`, `MOV`, `ORR` and `SUB` from the *Data Processing* category. With regard to the `LDR` and `STR` instructions, only the word size variants were implemented. We'll now look briefly at how ARM instructions are composed, in order to understand how the ARM-LLVM front end disassembles them.

As mentioned above all ARM instructions are 32 bits in length, making them easy to fetch randomly from memory as no knowledge of prior instructions is required[19]. These 32bits are broken down into several different subsets, commonly the `condition`, `category`, `opcode` and `addressing mode` bits, each of which are explained briefly below. The type of instruction and whether any, all or other bit subsets are present is usually determined by the `category` bit-field.

The `condition` bit-field (bits 31 to 28 inclusive), common to all instructions defined by ARM, is one of the tweaks hinted at earlier. The status register, named the Current Program Status Register (CPSR) on ARM platforms, is a critical part of most modern processors. Its purpose is for determining whether certain events have taken place during the execution of prior instructions. ARM instructions are conditionally executed based on whether the `condition` bits match part of the CPSR. This technique allows for greater code density through the removal of conditional branch instructions determining whether a block should be executed.

To keep things simple, the code judging block suitability for translation by the ARM-LLVM front-end only queues blocks where the `condition` field of all contained instructions is set to the `AL` pattern (`0xE`). These instructions are executed under all circumstances; they do not depend on the value of the CPSR.

Prior to inspecting the bit-fields of the relevant instructions, it should be pointed out that there are a further two fields which are common to the instructions implemented in

[19]Unlike with x86, where prior instructions must be disassembled in order to determine their length

addition to the `condition` and `category` fields. These are the `Rn` and `Rd` fields, which describe the base and destination registers for the operation respectively[20]. Because of this, the first operation (phase) performed in the front-end is to extract the `Rn` and `Rd` values, and fetch the associated register values from the `CPUARMState` struct. The base register, `Rn`, sometimes has corner cases attached to it; these are mainly centred around `r15`, the Program Counter register on ARM. In some cases the `PC` cannot be used as a base register due to side-effects introduced by the particular instruction type[21].

As the `category` bit-field goes a long way towards determining the type of instruction currently being disassembled, the front-end begins by working with it to determine what fields it should initially extract; as seen in Figure 5.8 we extract the `Rn` and `Rd` values as necessary.

By determining the instruction's category as described above, we now have a good idea about the bit-fields that are contained in the remaining part of the instruction. We will now look at the patterns present in the *Load/Store* categories, shown in Figure 5.9. There are three different categories of *Load/Store* instructions, which are *Immediate*, *Register* and *Multiple.* The *Load/Store Immediate* and all but one specific style of *Load/Store Register* instruction were implemented. The *Load/Store Multiple* category was omitted as it isn't as commonly used as instructions from the former two categories.

The `P`, `U`, `B`, `W` and `L` bits detailed in Figure 5.9 determine the type of *Load/Store* operation the instruction is to be. Bits `B` and `L` control the size of the data to be manipulated and the manipulation to be performed, *load* or *store*, respectively. `U` controls whether the offset is added or subtracted from the base register (`Rn`), while `P` and `W` control the offset application and side-effects of the instruction.

Side-effects of the *Load/Store* category include *none*, *pre-* and *post-instruction* increments applied to the base register. These side-effects are intended as optimisations for loop constructs, allowing the hardware to increment the loop counter and perform a separate operation all in the one instruction. To make these variants more sane to deal with, macros were defined that represented each instruction type: load/store,

[20]The exception is the `MOV` instruction, which does not require `Rn`

[21]This is the case with the *Load/Store* instructions in *pre-* or *post-increment* mode.

```
uint32_t category    = insn & INSN_CAT;
uint32_t opcode      = insn & DATA_PROC_OP;
uint32_t reg_rd, reg_rn;
LLVMValueRef rd, rn, rn_value;

reg_rd = INSN_Rd(insn);
rd = qemu_get_reg32(dc, reg_rd);
/* Don't try extract Rn for MOV, it doesn't use it */
if(!((DATA_PROC == category || DATA_PROC_IMM == category) && MOV == opcode)) {
    reg_rn = INSN_Rn(insn);
    rn = qemu_get_reg32(dc, reg_rn);
    rn_value = LLVMBuildLoad(dc->builder, rn, "rn_value");
    if(15 == reg_rn) {
        int error;
        rn_value = insn_helper_rn_r15(dc, insn, rn_value, &error);
        if(!rn_value) {
            switch(error) {
                case EUNPREDICTABLE:
                    fprintf(stderr, "%x: Instruction has UNPREDICTABLE"
                                    "results!\n", insn);
                    break;
                case EUNIMPLEMENTED:
                    fprintf(stderr, "%x: Unimplemented instruction\n", insn);
                    break;
                default:
                    fprintf(stderr, "%x: Ummm...\n", insn);
                    break;
            }
            return -1;
        }
    }
}
```

FIGURE 5.8: The initial phase: Determining the instruction's category, loading values ready for addressing-mode calculations (second phase) and instruction operations (final phase).

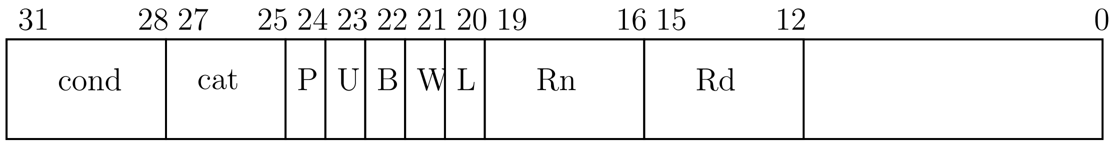


Figure 5.9: The ARM instruction bit-fields common to the *Load/Store* categories. Bits 11 to 0 are determined by the category (`cat`), and form the operands for the addressing mode.

immediate/register, word/byte, pre and post increment.

The second phase of disassembly is introduced for the following reason: The immediate and register-based variants of the instructions differ only on how the operands' final values are calculated. The final phase applies the instruction's actual operation, be it a `LDR`, `STR`, `ADD` or `MOV` to the values calculated in the second phase. The `LDR` and `STR` second phase operations are described in Figure 5.10. By separating out the calculation of the operand's final value from the instruction operation we reduce duplicate code and have a cleaner implementation of each of the instructions.

Following the second phase as outlined in Figure 5.10, the instruction operation is performed, along with the operations to the loop counters in the case of `LDR` and `STR` instructions. The final phase is handled in much the same manner as the previous phases first switching on the `category` and then on `flags` or `opcode` as appropriate. The final phase for the `LDR` and `STR` instructions is demonstrated in Figure 5.11.

5.3 Optimising and Compiling the LLVM Intermediate Representation

Above we explored the process of disassembling the ARM instructions and generating the LLVM intermediate representation by calling into the LLVM IR API via the C bindings. In this section we will look at the result of these calls, the intermediate representation, the optimisation passes and finally generating the native code via LLVM's JIT compiler.

```
LLVMValueRef address;
switch(category) {
    case LDR_STR_IMM:    /* Load/Store Immediate */
        /* Calculate the address from the immediate value and addressing mode */
        address = insn_helper_imm_addr(dc, insn, rn_value);
        /* Set up the value as a pointer */
        address = LLVMBuildIntToPtr(dc->builder, address,
            LLVMPointerType(LLVMInt32Type(), 0), "");
        break;
    case LDR_STR_REG:    /* Load/Store Register Offset */
        /* Calculate the registers position in CPUARMState */
        rm = qemu_get_reg32(dc, LDR_STR_REG_Rm(insn));
        /* Load the register's value */
        rm_value = LLVMBuildLoad(dc->builder, rm, "");
        /* Calculate the address from the register value and addressing mode */
        address = insn_helper_reg_addr(dc, insn, rn_value, rm_value);
        /* If there's an error, abort disassembly */
        if(!address) { return -1; }
        /* Set up the value as a pointer */
        address = LLVMBuildIntToPtr(dc->builder, address,
            LLVMPointerType(LLVMInt32Type(), 0), "");
        break;
    ...
}
```

FIGURE 5.10: The second phase: Determining the value, through initial operands and the addressing mode, to work with in the final phase. For *Load/Store* instructions this will be an address, for *Data Processing* instructions this will be an integer value. The *Load/Store* immediate and register semantics are displayed above.

5.3.1 JIT Compiling the Intermediate Representation

To keep things short and relevant the result of the translation process for one instruction, "`mov r2, r0`" (GAS syntax)[22] will be shown. Pushing this instruction through the disassembly process outlined in the preceding sections, the LLVM IR API calls

[22]Interpreted as "Move r2 into r0"

```
/* Ignore ldr/str U flag, dealt with in helper functions */
uint32_t flags = insn & INSN_FLAGS & ~LD_ST_U;
LLVMValueRef rd, rd_value, address, address_value;
switch(category) {
    case LDR_STR_IMM:
    case LDR_STR_REG:
        switch(flags) {
            case STRUW: /* Store word */
                /* Grab the value from the register in CPUARMState */
                rd_value = LLVMBuildLoad(dc->builder, rd, "");
                /* Store it out to memory */
                LLVMBuildStore(dc->builder, rd_value, address);
                break;
            case LDRUW: /* Load word */
                /* Load the value from memory */
                address_value = LLVMBuildLoad(dc->builder, address, "");
                /* Store the value in the CPUARMState register set */
                LLVMBuildStore(dc->builder, address_value, rd);
                break;
            ...
        }
        break;
    ...
}
```

Figure 5.11: The final phase: Performing the load or store operation once the address has been calculated

generate the IR outlined in Figure 5.12.

LLVM has generated a function in accordance with our strategy for simulating blocks. In C (with some Qemu context), the function prototype would look like:

```
  CPUARMState *block_test(CPUARMState *);
```

```
define { [16 x i32] }* @block_test({ [16 x i32] }*) {
entry:
    %tmp1 = getelementptr { [16 x i32] }* %0, i32 0, i32 0, i32 0
    %tmp2 = getelementptr { [16 x i32] }* %0, i32 0, i32 0, i32 2
    %tmp3 = load i32* %tmp2
    store i32 %tmp3, i32* %tmp1
    ret { [16 x i32] }* %0
}
```

FIGURE 5.12: The LLVM intermediate representation of the `mov r2, r0` instruction.

The body consists of a basic block of LLVM IR instructions[23] simulating the input instruction. `%tmp1` holds a pointer into `state` pointing to `r0`, while `%tmp2` holds a pointer to `r2`. The value at the address in `%tmp2` is then loaded into `%tmp3`, which is subsequently stored at the location pointed to by `%tmp1`. At this point our `mov` instruction is complete and the function returns.

As this is one instruction, attempting to optimise it yields the IR that we have already - there's nothing in it that's redundant or not optimal. From here, the native code (x86-64) is generated. The result is presented in Figure 5.13, and while reasonably straightforward, does need some explaining.

```
mov %r14, %rdi
mov 0x8(%rdi), %eax
mov %eax, (%rdi)
mov %rdi, %rax
ret
```

FIGURE 5.13: The x86-64 assembly generated by LLVM's JIT compiler from the IR representation of `mov r2, r0`.

[23]While we're simulating one block of ARM, this doesn't necessarily map to one basic block of LLVM or the native architecture

The need for explanation stems from the very first instruction! The `mov` is not actually generated by LLVM's JIT[24], but is essential in order for the block to work correctly if injected into Qemu. The `mov` places Qemu's pinned pointer to the `CPUARMState` struct into `%rdi`, where it becomes accessible to the following instructions. As explained in Section 5.1.5 the `%rdi` register is designated by the x86-64 ABI as the register holding the first parameter to a function. As we're modelling the ARM basic blocks in LLVM as functions taking one parameter, we give the LLVM generated code access to the `CPUARMState` pointer by moving the pointer into `%rdi`. The remaining instructions are all generated by the LLVM JIT compiler.

In terms of the semantics of the code, there are only two instructions that are relevant here - the second and third `mov` instructions. The first operand of the second instruction, `0x8(%rdi)`, points into the `CPUARMState` struct. Using our knowledge of how it is constructed, we can understand why this offset is generated. The first member of the `CPUARMState` struct is a 16 element array of type `uint32_t`. As such, each element is 32 bits, or 4 bytes. Given we have an offset of `0x8` to `%rdi`, this is pointing at the third element in the array: `r2`. The third `mov` inserts the value from `r2` (now in `%eax`) at offset `0x0` into the `CPUARMState` struct: `r0`. The final two instructions are concerned with returning the `CPUARMState` struct from the function. This is in fact a hang-over from early attempts at keeping LLVM from optimising the struct accesses away. Ideally the function should return `void`, though this is left for future changes to the implementation: In terms of injecting the instructions into Qemu, these last two instructions are discarded.

5.3.2 Optimising the Intermediate Representation

Having looked at how the LLVM IR compiles down to machine instructions, we'll now look at what is arguably the core of the project: The optimisations that are applied to the IR prior to the JIT compilation. Optimisations in LLVM can be performed in two different styles, as *module* or *function* passes. In the scope of this project,

[24]Currently it's manually injected into the buffer, but it should be possible to specify inside the LLVM functions using inline ASM

function passes are the appropriate choice - as the name implies they apply only to individual functions. Module passes, by comparison, are designed for applying global and inter-procedural optimisations (i.e. across multiple functions). Applying a module pass in this circumstance introduces a large penalty, as the system will be applying the optimisations every time it generates a function. This leads to an ever increasing workload for the optimiser as the functions are developed inside a single persistent module.

The optimisation passes exposed by LLVM's C bindings include:

- Constant propagation
- Promote memory to register
- Dead instruction elimination
- Dead store elimination
- Instruction combining
- Global variable numbering.

Each of these are applied to all the blocks that are fed through the LLVM subsystem, frequently producing blocks that are smaller in length than their TCG equivalents. One such case is demonstrated below. The ARM block in Figure 5.14 was encountered during a simulation of one of the SPEC benchmarks by Qemu and pushed through both the TCG and LLVM subsystems.

```
0x400818b4: add ip, pc, #0
0x400818b8: add ip, ip, #147456
0x400818bc: ldr pc, [ip, #1796]!
```

FIGURE 5.14: A simple ARM block. The instruction locations are shown here as they play an important role in the optimisations performed by LLVM

Dynamically translated by TCG, the native code seen in Figure 5.15 was generated. The code shown excludes the branch handling code generated by TCG in order to give a fairer visual comparison.

```
xor %r12d,%r12d
mov $0x400818bc,%r15d
add %r12d,%r15d
mov %r15d,0x30(%r14)
mov $0x24000,%r12d
mov 0x30(%r14),%r15d
add %r12d,%r15d
mov %r15d,0x30(%r14)
```

Figure 5.15: The semantically equivalent native block, as dynamically translated by TCG.

LLVM on the otherhand, generates just *one* instruction (ignoring the required `mov`), seen in Figure 5.16.

```
mov %r14, %rdi
mov 0x400a58bc, 0x30(%rdi)
```

Figure 5.16: The semantically equivalent native block, as dynamically translated by LLVM

Such an optimisation of the block warrants some investigation to ensure correctness. Is this optimisation legitimate? Consider the first ARM instruction: `add ip, pc, #0`. This is semantically equivalent to to a `mov`, the value is just shifted between the `pc` and `ip` registers. Unravelling the transitive relationships for the second instruction, `add ip, ip, #147456`, reveals that this could really be reduced down to `add ip, pc, #147456`. The optimisation doesn't stop there though. Since the ARM binary is loaded into memory, the `pc` is now a known static memory location, a constant value. ARM defines references to the `pc` to report the value as eight bytes in front of the current instruction's location; in the case of the first instruction giving the value `0x400818bc`. At the second instruction we're now adding two constants, something that can be done at compile time. Unsurprisingly, the result of the constant addition is `0x400a58bc`, making the optimisation legitimate[25].

[25]`0x30(%rdi)` resolves to `r12`, aliased as `ip` here.

6 Results

The aim of the project was to achieve shorter wall-clock execution times for ARM simulations in Qemu. In the previous chapters the concepts, designs and implementation of an optimising ARM front-end were outlined; the material in this chapter focusses on benchmarking the augmented version of Qemu to test whether it meets the stated goals.

The `sjeng` and `h264ref` benchmarks from the SPEC suite were chosen for their different wall-clock execution times: The sjeng benchmark runs in a relatively short time at just under two minutes, while the h264ref benchmark is longer, taking approximately ten minutes to complete[1].

The aim in choosing these two benchmarks was to show the effect of the optimised

[1]Approximate times were gathered from running the benchmarks inside mainline Qemu, version 0.10.6

blocks over different execution times. Unfortunately comparisons cannot really be made *between* the sjeng and h264ref benchmarks, as they are semantically and structurally different. This has influences on how effective the ARM-LLVM front-end will be, as it can only translate blocks completely composed of instructions it can disassemble. The blocks will naturally have different purposes inside different applications - they may or may not be hot blocks and contribute a lot or very little in improvements in application execution time.

There is also the issue of non-determinism in the system: Separate runs will most likely translate the same blocks, but this isn't guaranteed due to locking constraints. For much the same reason, it cannot be guaranteed that blocks will be translated or replaced at the same point in time (relative to the start of translation), as it depends not only on the locking constraints but also on the scheduling of the LLVM front-end's thread by the underlying operating system.

What we can get a picture of is approximately how much the ARM-LLVM front end is contributing to the individual benchmarks. This can be a negative or positive effect; it must be remembered that we're adding code to the hot path [2] inside Qemu - this will always introduce some overhead into the emulation.

6.1 Profiling Strategy

The most immediate test is whether the augmented version of Qemu is quicker than the mainline version. However, its possible to gather more data than this as there are several feature layers to the augmented version: Threading/job queuing, code generation and code injection. Benchmarks were performed in the following order:

1. Mainline Qemu: Version 0.10.6

2. Augmented Qemu: Threading/Job Queuing enabled, code generation disabled

3. Augmented Qemu: Code generation enabled, code injection disabled

[2] Code generation

4. Augmented Qemu: Code injection enabled

This strategy allows us to measure the overheads and benefits introduced by each stage. In the event that the wall-clock execution time is slower than mainline due to overheads introduced, it's still possible to gather whether the optimised blocks are having an effect by analysing the difference between items 3 and 4.

6.2 Execution Time Profiling with SPEC INT sjeng

Thirty samples were taken of the sjeng benchmark for each of the points outlined above. The statistics in Table 6.1 appear to show that there is only a loss of performance by introducing the ARM-LLVM subsystem. That said the loss isn't large, somewhere in the order of two to four seconds.

As the statistics presented in Table 6.1 show no positive results, an execution trace was obtained justify why this might be the case. Table 6.2 shows a small snippet of the data gathered. In total, 425 blocks were captured and translated, however the number of executions of the optimised code is well inside the realm of insignificant. Some interesting future work is most likely in the space of integrating optimised blocks immediately as they become available, a strategy for which is outlined later.

6.3 Execution Time Profiling with SPEC INT h264ref

Due to the length of time needed to execute the h264ref benchmark, only 10 samples were collected for each of the profiling points listed above. Analysis of the data is shown in Table 6.3. The immediately apparent result, like the sjeng benchmark, is that the augmented version, even with code injection is likely to perform worse than mainline. While this isn't optimal, there is a result that suggests the overall technique has potential. Looking at the 95% confidence intervals for the code generation and code injection stages, it shows that they are very nearly distinct, suggesting that injecting the generated code may be having a positive effect on the emulation execution time.

Mainline Qemu: 0.10.6	**Seconds**
Mean execution time	119.66
Sampled execution time std. dev.	2.45
Sampled execution time std. error	0.45
95% Confidence Interval for Sampled Mean	[118.62, 120.70]
Augmented Qemu: Threading/Job Queuing	**Seconds**
Mean execution time	121.90
Sampled execution time std. dev.	0.67
Sampled execution time std. error	0.12
95% Confidence Interval for Sampled Mean	[121.61, 122.18]
Augmented Qemu: Code Generation	**Seconds**
Mean execution time	122.01
Sampled execution time std. dev.	0.45
Sampled execution time std. error	0.08
95% Confidence Interval for Sampled Mean	[121.82, 122.20]
Augmented Qemu: Code Injection	**Seconds**
Mean execution time	122.23
Sampled execution time std. dev.	0.25
Sampled execution time std. error	0.05
95% Confidence Interval for Sampled Mean	[122.13, 122.34]

TABLE 6.1: Statistical analysis of the wall-clock emulation times for the SPEC sjeng benchmark

Rank	**Block Identifier**	**Total Executions**	**LLVM-based Executions**
...	...	...	...
47	0x6064e610	2843888	5
60	0x60642530	2244995	10
74	0x606479d0	1869222	8
86	0x6064d7d0	1615497	2
92	0x60659380	1454777	2
98	0x60600e50	1366103	3
106	0x6063e1a0	1366005	5
117	0x60642d30	1151286	5
120	0x60645650	1093709	7
121	0x60644240	1088131	7
122	0x606463e0	1020576	8
132	0x606458c0	959371	7
...	...	...	...

Table 6.2: Block execution data gathered from the sjeng benchmark with Qemu's `exec` debug option. The *Total Executions* column represents the number of times the block was executed over the whole simulation. The *Rank* value is the block's position in the list, as sorted by total executions. The *LLVM-based Executions* value represents the number of times the block was executed after injecting the optimised code.

Mainline Qemu: 0.10.6	**Seconds**
Mean execution time	573.465
Sampled execution time std. dev.	7.391
Sampled execution time std. error	2.337
95% Confidence Interval for Sampled Mean	[567.436, 579.495]
Augmented Qemu: Threading/Job Queuing	**Seconds**
Mean execution time	578.849
Sampled execution time std. dev.	6.819
Sampled execution time std. error	2.156
95% Confidence Interval for Sampled Mean	[573.286, 584.412]
Augmented Qemu: Code Generation	**Seconds**
Mean execution time	589.497
Sampled execution time std. dev.	2.935
Sampled execution time std. error	0.928
95% Confidence Interval for Sampled Mean	[587.103, 591.891]
Augmented Qemu: Code Injection	**Seconds**
Mean execution time	580.728
Sampled execution time std. dev.	7.860
Sampled execution time std. error	2.486
95% Confidence Interval for Sampled Mean	[574.316, 587.141]

Table 6.3: Statistical analysis of the wall-clock emulation times for the SPEC h264ref benchmark

7 Future Work

Here we discuss the potential developments for the project - ideas conceived whilst working with Qemu and the LLVM Comiler Infrastructure. We take into account the results discussed in Chapter 6 to suggest areas that might be worth further exploration. The points listed below are ordered from short to long term projects and by relevance to the approach presented.

7.1 Jump-To-Epilogue Support

It was intended that if the time was available during this project that jumping over large gaps between the end of the LLVM code and the beginning of the TCG epilogue would be implemented. There wasn't time however, and in the case that more instructions are implemented in the front-end this particular optimisation will most likely only become more critical. LLVM has shown the capability to optimise blocks to much shorter lengths than their TCG counter-parts; as the blocks grow longer LLVM's optimised versions often approach half the length of the original block.

Some optimised no-ops are in place for length differences up to 15 bytes, but beyond this the difference is simply accounted for by an un-optimised no-op slide down to the epilogue. A jump in this case would present much better efficiency - it will not require the intervening instructions to be read from memory, and a relative jump won't cause bubbles in the instruction pipeline as it is deterministic.

7.2 Add Further Instructions to the ARM-LLVM Front-end

Currently only the `LDR`, `STR`, `ADD`, `AND`, `MOV`, `ORR` and `SUB` instructions are implemented in the ARM front-end. Debug output from Qemu indicates that even this small amount of instructions captures roughly 10% of blocks in the SPEC INT 2006 `h264ref` and `sjeng` benchmarks. Extrapolating out from the results achieved so far indicates that implementing several more high impact instructions could have a very positive result. A promising instruction category appears to be *Load/Store Multiple*, and some of the test *Data Processing* instructions such as `TST` or `CMP` appear frequently in many different applications and large blocks.

Instructions adjusting the CPSR figas require some work in terms of implementing more of Qemu's `CPUARMState` struct in the LLVM front-end. This is not a lot of work, however needs to be done accurately - Qemu makes some optimisations with regards to the CPSR in order to reduce the number of comparisons it needs to make to determine if flags are enabled or not.

7.3 Dynamic Injection of Optimised Blocks During Execution

As part of the direct block chaining framework, Qemu has tools in place to adjust pointers in block epilogues to control whether execution should return to dispatch or proceed directly to the following block. Leveraging these functions, it should be possible to dynamically patch blocks to jump to the optimised code as it becomes available[1]. This may be in the form of a new TranslationBlock entry in the system, or a jump out to the LLVM JIT's code cache. Evidence presented in Chapter 6 on the SPEC sjeng benchmark suggests this could give significant improvements in performance.

7.4 Multiple LLVM Threads

As more instructions are implemented in the front-end this may require more CPU power than might be available on a given core, due to an increase in the number of blocks that can be successfully translated. The number of cores available on modern machines continues to increase, and so an area of research may be to investigate whether the approach presented in this project can be scaled out to other cores. Some primary concerns in this area would be proper queue management for a greater number of threads, and investigation into whether this is actually worth while for the number of blocks pushed into the queue

7.5 Asynchronously Read TCG IR and generate LLVM IR

The front-end for LLVM implemented by this project translated ARM instructions into LLVM IR for optimisation. While in the beginning it was thought that this might lessen the abstraction and subsequent padding between what was intended by the ARM

[1]Blocks jumping into the now optimised block will also have to be modified to point to the new code

blocks and what might've been output at layers such as TCG IR or x86, it turned out that the front-end to some extent reimplements the idea of TCG micro-ops. As such, it may be advantageous to implement a front-end to LLVM in terms of TCG IR - half the work is already achieved in splitting instructions up into operand calculation, instruction calculation and result storage.

Creating a TCG IR front-end for LLVM has the benefit of enabling supported for each front-end implemented by Qemu; support for MIPS, x86, ARM, PowerPC, M68k and several other architectures.

7.6 Teach TCG about x86 LEA Instruction

This idea stems from observing the LLVM generated x86(-64) instructions - that most of the optimisations come in the form of the `LEA` instruction. Using `LEA` allows the processor to combine a lengthy chain of arithmetic operations into one. This particular instruction shortens the number of bytes needed to represent the overall operation, thus increasing the speed of the code.

This may require some finesse, as TCG is necessarily quite fast at generating target code. Delaying the code generation through what might effectively boil down to false optimisation due to the block not being hot could have a significant effect in some applications. As such it should be judged whether this idea is feasible - at least it would require a reverse traversal of the TCG micro-ops in order to determine which ones could be bundled together in an efficient manner.

8 Conclusion

The work presented profiled an existing dynamic translator, Qemu, to determine where the bulk of the performance degradation was introduced to functional ARM instruction set simulations on x86-64. For Qemu, this was found to be in the dynamically translated code; in the SPEC INT benchmarks profiled its execution accounted for between 60% and 80% of the wall-clock execution time. Armed with this information, the aim became to achieve faster functional simulations through asynchronous optimisation of translated blocks. Asynchronous, or threaded optimisation was chosen due to the increasing number of multi-CPU / multi-core machines entering the market, a trend which is not predicted to decline any time soon.

While a reduction in wall-clock execution time was not achieved, there are strong indications that the implementation has the potential to increase simulation efficiency. Adding further instructions to the ARM-LLVM front end to cover more blocks, along with dynamic patching of the translated blocks as the optimised code becomes available look to hold the most opportunity for increased performance.

References

[1] J. Aycock. *A brief history of just-in-time.* ACM Comput. Surv. **35**(2), 97 (2003).

[2] C. May. *Mimic: a fast system/370 simulator.* SIGPLAN Not. **22**(7), 1 (1987).

[3] A. Gal and M. Franz. *Incremental dynamic code generation with trace trees* (2006).

[4] B. Franke. *Fast cycle-approximate instruction set simulation.* In *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pp. 69–78 (2008).

[5] J. Zhu and D. D. Gajski. *A retargetable, ultra-fast instruction set simulator.* In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, p. 62 (ACM, New York, NY, USA, 1999).

[6] W. Qin, J. D'Errico, and X. Zhu. *A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation.* In *Hardware/-Software Codedesign and System Synthesis* (2006).

[7] `http://www.nongnu.org/qemu/`, accessed March 2009.

[8] F. Bellard. *Qemu, a fast and portable dynamic translator.* In *The Proceedings of the 2005 USENIX Annual Technical Conference* (Anaheim, CA, USA, 2005).

[9] `http://www.ptlsim.org/`, accessed November 2009.

[10] `http://code.google.com/p/llvm-qemu/`, accessed March 2009.

[11] http://qemu-mips.org/, accessed November 2009.

[12] http://www.nongnu.org/qemu/status.html, accessed November 2009.

[13] http://www.qemu.org/qemu-tech.html, accessed November 2009.

[14] http://gcc.gnu.org/, accessed November 2009.

[15] http://www.spec.org/, accessed November 2009.

[16] http://valgrind.org/, accessed October 2009.

[17] http://oprofile.sourceforge.net/, accessed October 2009.

[18] http://llvm.org/, accessed March 2009.

[19] http://clang.llvm.org/, accessed November 2009.

[20] http://code.google.com/p/unladen-swallow/, accessed November 2009.

[21] http://github.com/evanphx/rubinius, accessed November 2009.

[22] http://code.google.com/p/llvm-qemu/wiki/Status, accessed November 2009.

[23] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html, accessed October 2009.

[24] http://markmail.org/message/ko3rrf5zacoocvg5\#query:+page:1+mid:5uygbscyiqmuxqi4+state:results, accessed November 2009.

[25] http://www.x86-64.org/documentation/abi.pdf, accessed October 2009.