# ANA: A Method for ARM-on-ARM Execution

by

Calvin On

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

Author ........................................................
Department of Electrical Engineering and Computer Science
August 15, 2007

Certified by ........................................................
Larry Rudolph
Principle Research Scientist
Thesis Supervisor

Accepted by ........................................................
Arthur C. Smith
Chairman, Department Committee on Graduate Students

# ANA: A Method for ARM-on-ARM Execution

by

Calvin On

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2007, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science and Engineering

## Abstract

This thesis proposes and implements ANA, a new method for the simulation of ARM programs on the ARM platform. ANA is a lightweight ARM instruction interpreter that uses the hardware to do a lot of the work for the read-decode-execute piece of simulation. We compare this method to the two existing methods of full simulation and direct execution that have been traditionally used to achieve this. We demonstrate that despite some setbacks caused by the prefetching and caching behaviors of the ARM, ANA continues to be a very useful tool for prototyping and for increasing simulator performance. Finally, we identify the important role that ANA can play in our current efforts to virtualize the ARM.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist

## Acknowledgments

I would like to thank Larry for being the inspiration of all my research efforts and successes over the past four years at MIT. In this time, he has taught me more about computer science and about life than all my courses here ever did. This thesis, along with many of my other achievements at MIT, would not have been possible without his ideas and enthusiasm.

Many thanks to Scott for his expertise and unending patience with teaching me more than I could have ever imagined about virtualization. Much of the credit for our implementation successes goes to him.

Thanks to Weng-Fai for sharing his knowledge about the ARM and providing help at times most needed.

Thanks to Julia, Alex, Prashanth, Gerald, and the rest of the folks at VMWare for their continual support and advice, and for being an extreme pleasure to work with this summer.

Thanks to Albert, who taught me the basics at the start and is the main source of inspiration for my efforts to become a better systems programmer.

And above all, thanks to my family (Mom, Dad, Gloria) for their unwavering support and for being the reason behind everything that I am today.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

We propose ANA, a method for performing ARM code execution on ARM platforms that seeks to combine the advantages of two traditional approaches to the problem: full simulation and direct execution. We analyze the tradeoffs between these two extremes and describe where our hybrid approach fits in and how it performs. Through experimentation, we show that ANA offers a simple and efficient solution for executing ARM code that takes a different set of tradeoffs from its predecessors. Such a method will allow an entire computer to be simulated under a controlled environment, and achieving this brings us closer to the vision of virtualization onto the ARM.

## 1.1 Problem Statement

The main goal of a simulator is to be able to execute a program (usually distinguished as a "guest program") inside a controlled environment without the guest knowing that it is being simulated rather than executing on real hardware. As such, there needs to be a way to simulate the hardware architecture in a detailed enough manner to serve as a substitute for the real thing [10].

A lot of the work in creating a simulator lies in building the proper infrastructure. For example, memory needs to be simulated in a way to allow the guest code to use any part of the virtual space of its choosing. Interrupts and exceptions must be handled correctly somehow, since guest code will sometimes expect to trap to get the right behavior [1]. Most of these tasks have previously been done for simulating next-generation hardware, or to do performance profiling of new techniques (i.e. different ways of caching) on existing machines.

We are concerned with a different problem, however, of simulating the same architecture as the hardware it's running on, since our ultimate goal is to create a simulator capable of running a virtual machine. While this has also been done by many others, it has been on high performance computers that have plenty of resources to work with. On low performance computers such as the ARM, processor speed and memory constraints become a major concern. We must seek to minimize both CPU and memory overheads if we hope to have an ARM on ARM execution model that can be used in applications like virtualization.

## 1.2 Full Simulation vs. Direct Execution

The traditional approaches of full simulation and direct execution offer two diametric views of the solving same problem. Full simulation interprets ARM instructions as they come, which has the advantage of runtime flexibility and control. To get this, however, full simulation pays a significant price in terms of number of instructions needed to execute a single ARM instruction. Direct execution, however, pays for this up front via a method called binary translation (more detail on how binary translation works are in the following chapters). This initial cost is then amortized over the course of execution because the translated instructions can run directly on the hardware with no additional overhead. The effect of binary translation on improving performance is further emphasized by the fact that most execution consists of loops, amortizing the one-time cost of binary translation. This makes direct execution perform a lot better than full simulation in almost all cases. However, direct execution does not benefit from the same amount of flexibility and control that full simulation offers.

## 1.3 ANA: Our Solution

We propose ANA as a method for achieving reasonable performance while retaining the flexibility of full simulation. Our simulation starts off with a full simulation model but utilizes direct execution to carry out the interpretation and execution on a per instruction basis. For straight-line code, per instruction direct execution has a lower overhead than the traditional binary translation approach. In this thesis, we show the effects of such a design in terms of the tradeoffs in performance relative to the two traditional approaches.

We then report our efforts to implement this technique and list some of the challenges we encountered. We discuss possible solutions to these problems and the tradeoffs made by each of these. Though it becomes apparent that, given the limitations of the architecture, ANA does not achieve the performance we hope it would, it still possesses much value and may have some very good use cases.

We then demonstrate the importance of this solution in the context of applications such as virtualization. We discuss the aspects of virtualization where ANA may be useful.

## 1.4 Thesis Outline

The rest of this thesis is outlined as follows:

Chapter 2 describes and analyzes the two currently used approaches for ARM on ARM. It also discusses previous work and related work in those areas.

Chapter 3 provides some relevant background information on the ARM architecture and instruction set.

Chapter 4 describes ANA, presents our implementation of ANA, and analyzes its tradeoffs and benefits.

Chapter 5 describes our implementation challenges and their solutions, as well as their implications on the applications of ANA. We also discuss the impact of ANA on our ongoing efforts to virtualize the ARM.

Chapter 6 concludes, discusses future work on ANA, and summarizes our contributions.

# Chapter 2

# Previous and Related Work

In this chapter, we describe more fully the two traditional approaches of full simulation and direct execution. We also discuss previous work that has been done on these respective subjects.

## 2.1 Full Simulation

Full simulation is essentially a high-level interpretation and execution of guest code. Guest instructions are decoded directly by the implementation, and the effects of a particular instruction are carried out via updating the state of a high-level data structure. This data structure essentially serves as the "virtual machine state" of the guest program, making the guest think that it is running on a real machine. Figure 2-1 shows a simplistic view of what a simulator might do for a given instruction. In this figure, we can see that the virtual machine state consists of all the register values in the CPU running the guest program in combination with a representation of memory.

Examples of a typical full simulator include Bochs [8] and Skyeye [15]. Created to simulate the x86 architecture, Bochs has a large loop that models the fetch-decode-execute behavior of a CPU as described above and in Figure 2-1. Main memory is represented in Bochs by a large array of memory within the simulator. Because the implementation resides on such a high-level, full simulators like Bochs have an advan-

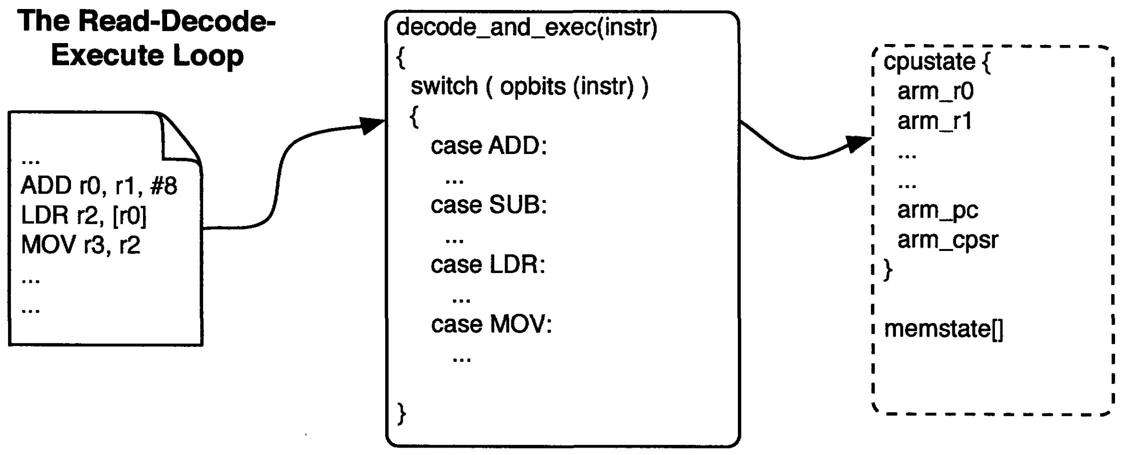

Figure 2-1: Instructions are decoded by a switch statement and then executed by updating the appropriate registers in the virtual machine state.

tage of portability across different host machines. SkyEye is similarly implemented on the CPU emulation side but for the ARM instead.

Full simulation also has the advantage of allowing the simulator to always have full control over the hardware at all times. This effectively gives the simulator a great deal of control and flexibility for handling each instruction as it arrives during runtime. For example, the simulation may pause in between guest instructions or sometimes even in the middle of executing a guest instruction. Alternatively, a simulator may decide to handle the same instruction differently depending on the state of the virtual machine.

This, however, comes with a fairly large price, both in terms of implementation and runtime costs. Full simulators pay a hefty implementation cost because they need to interface to every piece of critical hardware that the guest code may need to interact with. As shown in Figure 2-2, a simulator needs to effectively simulate different kinds of hardware in its machine state for certain types of programs. But perhaps the largest factor in the overhead for creating a full simulator lies in the decode-and-execute module. This piece needs to essentially have an implementation for every single instruction. Typical simulators as a result have thousands of lines of code to achieve this level of instruction support.

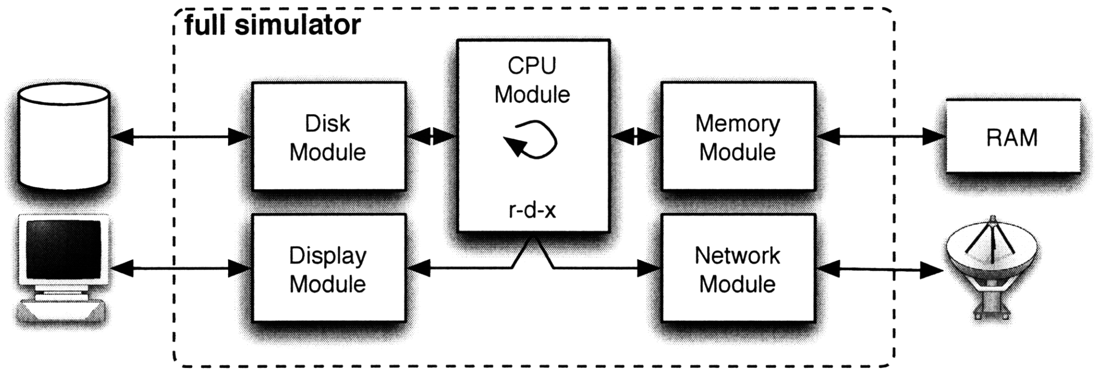

Figure 2-2: A full simulation implementation will typically need to have its own interfaces to individual hardware devices.

In terms of runtime performance, because everything is handled at such a high-level, the ratio of actual machine instructions needed to carry out a single guest instruction is very high. On a typical simulator, this ratio can range from 10:1 to 100:1 or even more. The result is a considerably large slowdown for simulating guest code, and this is undesirable for performance-critical applications. The ratio is especially pronounced for ARM data processing instructions, where two levels of interpretation are necessary to correctly simulate them.

Despite this, full simulation is still an attractive option for those who would like control. As a result, a fair amount of work has been placed into optimizing the performance of full simulation implementations.

SimOS [10] is a full simulation system that optimizes performance by switching hardware simulation modules on the fly depending on the current task being run. These modules had varying levels of detail with respect to the actual hardware they were supposed to be emulating. Thus, by choosing shallow implementations for hardware that a particular task did not need, SimOS is able to achieve better performance. With these techniques and others, SimOS performs at about a 200:1 ratio to full simulation performance.

Simics [7], another full system simulator based off SimOS, overcomes the implementation hurdle of the interpreter by developing a specification language, SimGen, to encode various aspects of their target instruction-set architecture. The creators also optimize their interpreter using partial evaluations whenever they can, depending on their target specifications.

## 2.2 Direct Execution

Direct execution is a solution designed to achieve the best performance possible by using the hardware to do most of the work. The basic idea behind direct execution is to allow the guest code run on the hardware itself. In order to do this, an implementation must first overcome some potential problems before it can safely execute guest code directly:

1. Guest code may not necessarily be able to access all the memory it thinks it has access to. The simulator has a limited address space available to it, but the guest code running inside the simulator has no knowledge of this.

2. Guest code may generate faults or traps. This is bad for the simulator because if a trap occurs during guest code execution, the simulator has essentially lost control of the program.

3. Guest code may contain privileged instructions, while the simulator is running in User mode. This may result in certain instructions trapping or failing silently.

Binary translation, the process of translating one set of instructions to another so that it may be safely run directly on the hardware, is commonly used to address some of these issues. It may, for example, translate the guest code such that the translated code will only access areas of memory known to the simulator. Another common method to address (2) is modifying the trap vectors to return control to the simulator [10].[1]

There are two types of binary translation: *static translation* and *dynamic translation* [1]. Static translation performs the entire translation prior to execution. While

[1]Because of these requirements that direct execution must fulfill in order to even work in most cases, any use of the term "direct execution" in the rest of this thesis refers to this kind of protected execution.

static translation offers more opportunities for optimization, it is more difficult to do and typically requires end-user involvement. Dynamic translation, on the other hand, performs this translation at runtime and caches the result. Code is therefore translated on demand. The translation step in either approach is not cheap, however, since it must at some point or another be done over the entirety of the guest code that is executed. The performance gain comes from the fact that this cost is amortized over the number of instructions executed on the hardware. This number is usually very large compared to the actual size of the translated code because of frequent loops in code execution, resulting in a significantly reduced amortized cost of translation.

Another advantage to this approach is the relatively lower implementation cost from not having to write interfaces to different pieces of hardware. The guest code, when translated properly, is free to directly interact with memory, disk, and other i/o. The need for a large switch block like the one used in full simulation is also eliminated here, since the instruction decoding is left to the hardware. We keep in mind, however, that this gain in implementation cost is offset by the code needed to perform the binary translation step.

Direct execution pays another price by sacrificing control over the code it is executing for performance. Unlike full simulation, a direct execution implementation can only stop in between blocks of translated code rather than at arbitrary locations. While the translated guest code is running, the simulator has no means of regaining control of the hardware; it must simply wait until the program counter has reached the preset endpoint for that particular block of code, or it must insert checkpoints in the middle of that block. Additionally, to optimize performance, the blocks often account for multiple instructions at a time, which means that the simulator will only have control in between multiple instructions rather than individual ones. Direct execution also has no straightforward way to flexibly handle instructions. It cannot, for example, treat the same instruction differently based on the virtual machine state at the time it is executed. Another weakness in binary translation's flexibility is that it is fairly incompatible with any other architecture besides the source and target architectures of the translation, which contrasts starkly to the high-level of compatibility

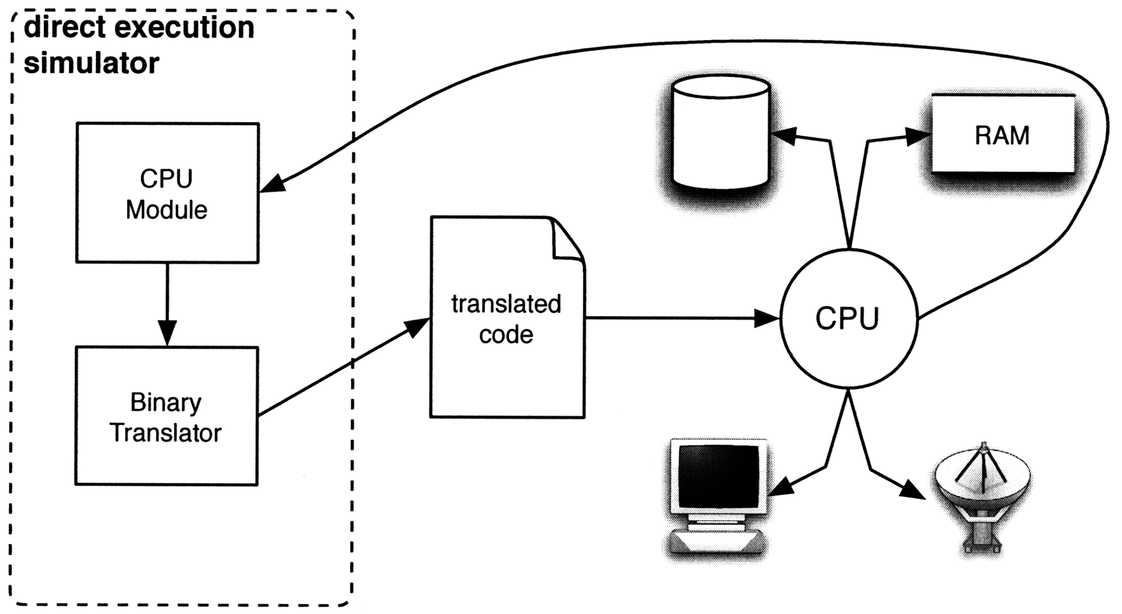

Figure 2-3: A direct execution implementation allows the CPU to directly interact with other hardware devices, thus reducing the need for modules that interface to them.

full simulators typically have [6].

Despite its weaknesses, plenty of applications rely heavily on binary translation for its performance benefits. QEMU [2] is an emulator that demonstrates the power of binary translation. Using dynamic translation along with a handful of other optimizations, QEMU suffers only from a slowdown factor of 2 in full system simulation, which is 30 times faster than Bochs. Even SkyEye [15], as a full simulator, has started to use binary translation in an effort to improve performance.

Other work on binary translation include UQBT [6] and DIGITAL FX!32 [5]. UQBT is an effort to allow cheap re-sourcing and re-targetting of binary translation implementations. DIGITAL FX!32 seeks to optimize binary translation performance by first executing and profiling its target application. It then performs static translation on the application, using the profiling results to apply the best set of optimizations.

# Chapter 3

# Background

In this section we provide some background information on the ARM architecture and instruction set. Since this thesis deals mostly with executing ARM instructions, we focus our overview of the architecture on the registers, processor modes, and the instruction set.

## 3.1 Modes of Operation

The ARM has 7 modes of operation: User, FIQ (Fast Interrupts), IRQ, Supervisor, Abort, Undefined, and System mode. All modes but User mode operate at a privileged level - that is, the full ARM instruction set is available in those modes. User mode is restricted from using a subset of instructions, which are referred to as privileged instructions.

## 3.2 Processor Registers

The ARM has 31 general-purpose registers, including a program counter, and 6 status registers. These registers are all 32 bits wide, although for the status registers, only 12 of the 32 bits are actually used.

The primarily used general-purpose registers are R0-R15. These are used to run user-level programs. Typical ARM implementations use R11 to store the frame pointer (fp), R12 for the base pointer (ip), R13 as the stack pointer (sp), R14 as the link register (lr), and R15 to hold the program counter (pc). This is reflected in the instruction set.

With the exception of FIQ and System modes, each different privilege level has a separate set of banked registers for R13-R14. FIQ has an additional set of banked registers from R8-R12. System mode does not have any banked registers; it shares the same set as the normal, User mode registers.

In addition to these general-purpose registers, the ARM also has a Current Program Status Register (CPSR). The CPSR contains bits for condition codes, interrupt disabling, Thumb mode execution, and the current privilege level of the CPU. Each privileged mode (except for System) also has its own Saved Program Status Register (SPSR), which is used to preserve the User CPSR when entering into one of those modes.

## 3.3 The ARM Instruction Set

The ARM instruction set has 55 instructions which can be classified into three major categories: data processing instructions, memory access instructions, and miscellaneous. Data processing instructions include ALU instructions and other instructions that directly modify register data. Memory access instructions are single loads and stores various sizes, while miscellaneous instructions include multiple-register loads and stores and privileged instructions. Data processing and memory access instructions each have multiple addressing modes specific to them. The particular category an instruction belongs to can be determined by looking at the opcode bits plus a few other bits of the instruction.

The instruction set is characterized by many irregularities, making it difficult to decode. Instructions of different categories and subcategories will have different encodings. For example, the opcode field is only present in data processing operations, and a separate set of bits are used to determine if an instruction is a data processing instruction in the first place. In other cases, an interpreter has to check combinations

| **syntax** | **operation** |
|---|---|
| `#<immediate>` | Immediate operand |
| `<Rm>` | Register operand |
| `<Rm>, LSL #<shift_imm>` | Logical shift left by immediate |
| `<Rm>, LSL #<Rs>` | Logical shift left by register |
| `<Rm>, LSR #<shift_imm>` | Logical right left by immediate |
| `<Rm>, LSR #<Rs>` | Logical shift right by register |
| `<Rm>, ASR #<shift_imm>` | Arithmetic shift right by immediate |
| `<Rm>, ASR #<Rs>` | Arithmetic shift right by register |
| `<Rm>, ROR #<shift_imm>` | Rotate right by immediate |
| `<Rm>, ROR <Rs>` | Rotate right by register |
| `<Rm>, RRX` | Rotate right with extend |

Table 3.1: All the possible shifter operand computations for data processing instructions..

of various single bits across the instruction in order to correctly decode it. We suspect that these irregularities resulted from numerous extensions over the generations, and old instruction encodings were left to ensure backward compatibility.

Even after decoding an instruction, an ARM interpreter may have more work to do. For example, data processing operations have shifter operand field. This shifter operand can be handled in 11 different ways depending on the kind of data operation specified by the instruction. Table 3.1 lists of all the processing modes that are possible for a shifter operand.

This additional level of operand processing effectively adds another addressing mode for data processing instructions, making it difficult to efficiently decode and uniformly process data processing instructions in one single step. What this ultimately means is that any implementation that decodes the ARM instructions requires an additional, nested switch block to determine the appropriate shifter operation for data processing instructions. The alternative to this is the equally unattractive choice of a one-level switch statement that has all the combinatorial possibilities of each data instruction with their different shifter operand modes. This complexity causes any attempt to create a full simulator for the ARM to have a very large interpreter. For example, SkyEye's [15] ARM interpreter is implemented with approximately 5600 lines of C code.

The one fortunate feature of the ARM instruction set is that it has uniform register argument locations. If a particular register argument is present in an instruction, we can be guaranteed to find it in the same place regardless of the instruction. Each instruction can have a maximum of 4 register arguments: a base register `Rn`, a destination register `Rd`, a shift register `Rs`, and an operand register `Rm`. Checking which of these registers are present in an instruction is also a relatively straightforward task.

# Chapter 4

# ANA

With an understanding of the tradeoffs between full simulation and direct execution, we now propose a model for ARM on ARM execution that tries to gain the performance benefits of direct execution while retaining the control and flexibility of full simulation.

## 4.1 Concept

The basic idea behind ANA is to exploit the hardware to decode and execute each instruction as it comes without handing over control of the CPU to the guest. The steps for doing this is as follows:

For each instruction:

1. Rewrite the instruction to use 4 preset registers.
2. Setup an execution CPU state with the correct values for those 4 registers, where these values are taken from the virtual CPU state.
3. Save the real CPU version of these 4 registers.
4. Load the execution CPU state onto the real CPU.
5. Load the rewritten instruction into the PC and execute.

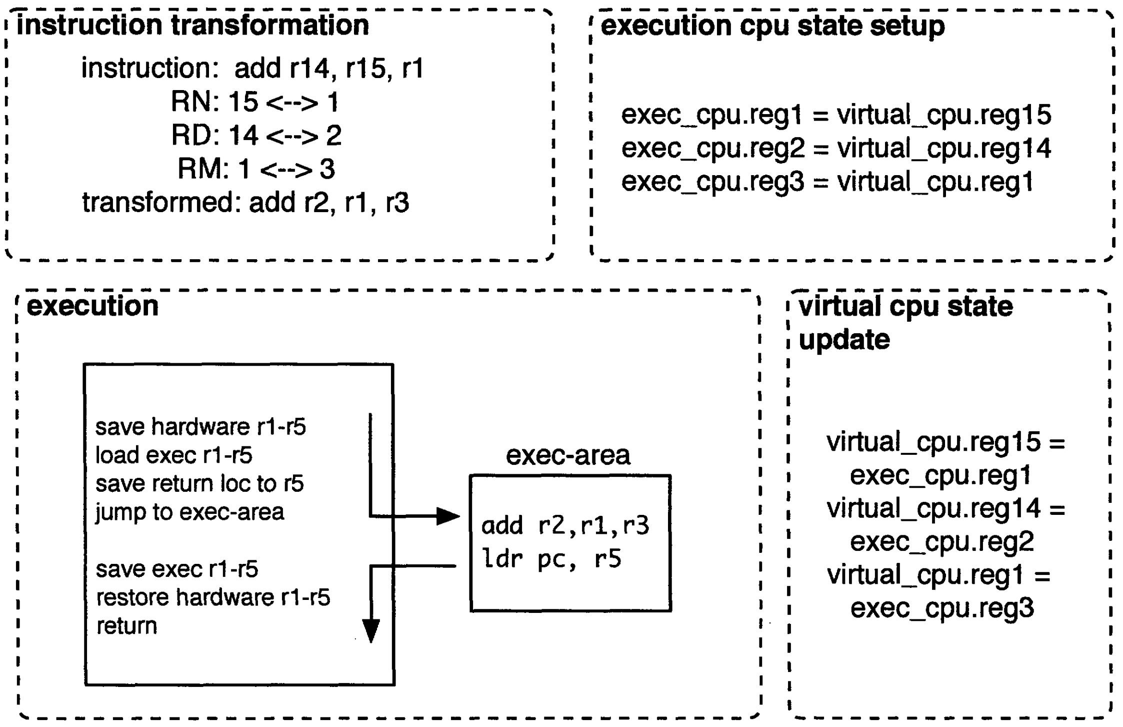
instruction transformation
instruction: add r14, r15, r1
RN: 15 <--> 1
RD: 14 <--> 2
RM: 1 <--> 3
transformed: add r2, r1, r3
execution cpu state setup
exec_cpu.reg1 = virtual_cpu.reg15
exec_cpu.reg2 = virtual_cpu.reg14
exec_cpu.reg3 = virtual_cpu.reg1
execution
save hardware r1-r5
load exec r1-r5
save return loc to r5
jump to exec-area
save exec r1-r5
restore hardware r1-r5
return
exec-area
add r2,r1,r3
ldr pc, r5
virtual cpu state update
virtual_cpu.reg15 = exec_cpu.reg1
virtual_cpu.reg14 = exec_cpu.reg2
virtual_cpu.reg1 = exec_cpu.reg3

Figure 4-1: An instruction is first transformed to use our preset registers. We then execute the transformed instruction by setting the PC to its location in memory followed by an instruction that immediately returns the PC back to the simulator.

6. Write back out the 4 registers from the real CPU to the execution CPU.

7. Restore the original 4 registers from their backup location.

8. Update the virtual CPU state by reverse-mapping the execution CPU state.

Figure 4-1 illustrates the transformation and execution steps more clearly.

There are several reasons why the instruction transformation process is used. First of all, by limiting the set of registers being used by all incoming instructions, we only need to save and restore 5 registers for executing the instruction as well as for preserving the actual CPU state (which contains the state of the simulator program itself). It should be noted, however, that this savings in overhead is offset by the routines needed for properly transforming an instruction. A second, more important reason is that a full register set switch is simply not possible to do without the danger of losing complete control of the processor. For example, if the guest instruction modifies the PC or the frame pointer, there would be no effective way for the simulator to regain control of the program. By transforming the instructions and carefully choosing the 5 registers we use, we can fully control which registers get modified. (It is worth noting here that we could check to see if the one of the registers was the PC or the FP, but this requires a lot of conditionals because they could be in one of two possibly-affected registers. One could perhaps optimize by doing this check and only remap registers if necessary, assuming that the cost of this check is less than the cost of the remapping process.)

## 4.2 Implementation Details

To obtain actual results on the performance of our proposed method, we set out to implement a simple user-level version of ANA that can execute user level programs in an ARM Linux environment. Execution of a user level program involves setting up a virtual CPU data structure and then interpreting and executing the instructions of the program on that virtual state.

Our ANA simulator performs all data processing instructions and all single-register loads and stores using our method. Multiple loads and stores, branches, and other miscellaneous instructions were implemented via full simulation. Since we expect user level programs to only touch legal areas of memory, we simply pass memory accesses through to the hardware. The only complication to this approach arises from having to set up a stack for the guest so that it would not conflict with the stack of simulator itself. To do this, we allocate a large static array in the program and then initialize the virtual CPU state's stack pointer register the array's location.

The simulator itself runs a loop that executes guest instructions. After each instruction, it updates the virtual CPU state and increments the guest PC until it reaches an end address specified by the user. We have a function, `go()`, that initializes the guest state and makes a call to the function that does the actual work, `single_step()`. `single_step()` performs a read-decode-execute for a single instruction. This function does a basic check on the instruction to decide whether it is a data or memory processing instruction or if it is an instruction that needs to be fully simulated. If it is a data or memory processing instruction, it is then passed to a `direct_exec()` function that carries out the direct execution of the instruction. Figure 4-2 shows excerpts of the `direct_exec()` function.

The `direct_exec()` function transforms the guest instruction to use 4 preset registers (registers 1-4) and stores it into `run[]`. It sets up the appropriate hardware state (preserving the simulators hardware state) and then jumps to `run[]`, where it executes the single transformed guest instruction before returning to `direct_exec()`. At this point, the remainder of the function simply cleans up and updates the guests virtual CPU state before returning.

## 4.3 Initial Analysis of ANA Overhead

An initial estimate on the performance of ANA can be done by looking at the number of instructions a typical path through a `direct_exec()` sequence will take:

- 20 instructions to transform the instruction to use the preset registers
- 12 instructions to save and restore the real CPU register state
- 12 instructions to swap the fake_cpu registers before and
- 16 instructions to set up the preset registers and extract the results from them
- 6 instructions to load and execute the rewritten instruction with a proper return instruction afterwards

If, on average, a full simulation implementation requires more than 66 instructions to execute a particular ARM instruction, then our approach gives us a performance boost across the board in addition to having a much smaller memory footprint. While this may appear to be a big number, we can expect, at least for certain data processing instructions, to outperform full simulation because of the extra overhead incurred by the need to calculate the shifter operand.

```
uint32 run[4];
void direct_exec(uint32 instr, union cpustate &vcpu)
{
  struct smallcpustate fake_cpu, real_cpu;
  // find out what type of instruction instr is
  // modify the instruction so that the registers are remapped
  rep_intsr = xform_instr(instr);

  // setup memory area where the cpu will jump to
  run[0] = rep_instr;
  run[1] = 0xe596f00c;
  run[2] = &run[0];  // we set a register here so ldr pc, register will
                     //  bring the pc to run[0]

  // swap registers 1-4 with RN, RD, RS, RM
  fake_cpu.regs[1] = vcpu->regs[RN(instr)];
  ...
  fake_cpu.cpsr = vcpu->cpsr;

  asm volatile(
      - read hardware cpu regs 0-6 and cpsr into real_cpu
      - load fake_cpu into the hardware cpu
      - set r6 to point to run[2]
      - set run[3] to point to pc+4
      - ldr pc, r6
      - save hardware cpu back into fake_cpu
      - restore hardware cpu from real_cpu
      )

  // update RD (do not support LDR/STR with writeback)
  vcpu->regs[RD(instr)] = fake_cpu.regs[3];
  vcpu->cpsr = fake_cpu.cpsr
  return;
}
```

Figure 4-2: Partial code and pseudocode for the direct_exec() function.

We could also perform further optimizations of this process to reduce the overhead of instruction rewriting and execution. For example, as mentioned in the previous section, we could get rid of the instruction transformation and register swapping process if we could devise a way to check for the PC and FP as as modifiable register arguments. Another approach may be to keep the register switching, but to execute more than one instruction at a time using the same four registers. If we can force the compiler to not use registers R1-R5, then we don't need to save and reload those registers after each instruction. Instead, we would only need to save the destination (and possibly the base) register each time.

Another benefit of this implementation is that it essentially covers almost all of the ARM instruction set without a large implementation cost or a large space requirement. This makes it an ideal way for prototyping applications that require an interpreter, especially if optimal performance is not extremely important.

# Chapter 5

# Results and Analysis

In this chapter we will describe our attempts to benchmark our ANA implementation on two Nokia devices, the problems we encountered, and an analysis of the results of our attempts.

## 5.1 Methodology

Our benchmarking methodology was a measurement of the time it took to complete the following Fibonacci calculator program:

```
int fib(int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n - 1) + fib(n - 2);
}
```

Our ANA implementation was developed on an ARM Linux environment called Scratchbox [11]. Scratchbox is a toolkit that with the capability to cross-compile code for the ARM on an x86 platform. Additionally, it can integrate with QEMU to run ARM code on the same x86 host. Our initial tests were performed on Scratchbox before the binaries were run on the N770 and the N800 Internet Tablets.

## 5.2 Challenges

Unfortunately, we were unable to successfully execute the code on either the N770 or the N800 to obtain any measurements on the devices themselves. The program ran perfectly on the emulator on an x86 host, but that behavior could not be replicated on the hardware.

On the N800, `direct_exec()` seemed to fail on executing the transformed instruction. In an attempt to debug this, we compiled and installed GDB on the N800 to step through the code. Unfortunately, the program would work just fine when we stepped through it in GDB, thus making it nearly impossible to debug. We also found that the program would work unreliably if enough debug statements were inserted in between calls to `direct_exec()`.

This led us to the conclusion that the most likely cause for this problem is instruction caching on the ARM. Our ANA implementation is essentially a small piece of self-modifying code (1 instruction of modified code, to be exact). We discovered too late that self-modifying code is handled differently per each ARM implementation. In our particular case, the data seems to indicate that the instruction cache simply is not being updated during each call to `direct_exec()`, so when the CPU jumps to the location of the rewritten instruction, it will still be executing the instruction thats in the cache.

## 5.3 Solutions

We have come up with several solutions to this problem, but each of them requires a reasonably large tradeoff in performance. They are:

1. *Flush the instruction cache line each time* `direct_exec()` *is called.*
   Each time `direct_exec()` is called and before the PC jumps to the written instruction, invalidate the instruction cache line that the instruction resides in. This ensures that the CPU will fetch the page from memory. The problem, however, is that the instruction used to flush the cache is a privileged instruction.

This means that the only way this approach is even possible is to have a kernel level module do it for the simulator. This kernel call incurs a pretty large overhead, especially considering that it has to happen per guest instruction that goes through ANA.

2. *Mark the page that* `run[]` *resides on as uncacheable.*
   On ARMs that have an MMU, each page table entry has a cacheable bit (the C bit) that marks a page as cacheable or uncacheable. Clearing the C bit for the page where `run[]` resides ensures that the CPU will go to memory to fetch the rewritten instruction each time. Since this approach requires access to the page tables, an implementation would have to have access to a privileged level (i.e. via a kernel module). This approach will also require walking the page tables of the host machine to correctly find the page for `run[]`. As a result, a high initial overhead must be paid in order for this to be effective. Last but not least, any other data that happens to be on the same page will suffer runtime access performance because they are now uncacheable.

3. *Use more than X number of* `run[]` *arrays in a round-robin fashion, where X is the number of cache lines on the target CPU.*
   The idea here is to simply make sure that the instruction cache can never be used to fetch any instruction in `run[]`. The idea here is to crowd out the cache enough so that by the time the PC returns to a particular `run[]` array, that array will have already been evicted from the instruction cache and need to be fetched from memory again. This approach sacrifices a lot less in performance and requires no additional privilege levels to work. It does, however, suffer very much from being implementation-specific (not all ARM processors have the same caching behaviors or the same number of cache lines) and from having a larger memory overhead from the extra `run[]` arrays.

At the moment it remains unclear which solution is the best. It may ultimately be that each has its own use cases. If the simulator can be run directly in a privileged mode, (1) will likely be the best approach. If the simulator must run in user mode but can have access to a kernel module capable of properly modifying bits in the page table, then (2) will outperform (1). However, if the ultimate concern is to have a program that works solely in user mode, then (3) is the way to go.

## 5.4 Analysis

The ultimate reality that we have learned from this experience is that there is no current way to implement ANA in a way that can outperform binary translation or, most of the time, even full simulation. The cost incurred from ensuring cache coherency is simply too large to be overcome at the moment. But ANA is far from useless and still stands to have many useful applications.

One of the biggest advantages of ANA is its low implementation cost. With very little programming effort, one can use ANA to build up a decent ARM on ARM emulator. This makes ANA the ideal candidate for prototyping applications that need an ARM interpreter. We certainly made good use of it in our ongoing project with ARM virtualization: our ANA implementation was used to handle guest data processing instructions on the virtual machine monitor. (We did not suffer problems from the caching issue because our virtualization prototype flushed the cache after every instruction for other reasons.) This saved us a lot of the time and effort that writing a full interpreter for ARM data processing instructions would have taken.

Another, equally large advantage of ANA is its low space cost. This is important especially in the context of ARM devices and applications because memory is still an extremely scarce and valuable resource on today's ARM devices. Most simulators suffer from large memory overheads because of the size of their interpreters. With ANA, one can achieve the same functionality as a large, clunky simulator in a compact piece of code - all at the cost of a slight performance decrease.

Last but not least, there may be ARM implementations that are significantly friendlier to self-modifying code. In these cases, ANA has the potential to outperform most full simulators and perhaps even in some cases such as straight-line code execution, binary translation implementations.

## 5.5 Virtualization

Another area where ANA continues to have promise beyond being a simple prototyping tool is the realm of virtualization. All machine virtualization techniques require a well-featured (if not fully-featured) interpreter that is capable of handling a large number of guest instructions. This interpreter lives in the virtual machine monitor, where guest instructions are being executed. ANA can still be of use here for several reasons:

- A kernel module is accessible to the virtual machine monitor and thus to the interpreter. Either solutions (1) or (2) would be viable here.
- World switching generally results in the flushing of all caches because of a page table switch. If these happen often enough in between instructions, the ANA code may always be running on a clean cache.
- The VMM usually does other things after instructions are executed. If there are enough instructions in between calls to `direct_exec()`, the cache will evict the entry with the previous, stale instruction each time and fetch from memory the latest rewritten instruction.

If any or all of these conditions are true for a given virtualization implementation, then ANA will be a worthwhile approach for efficiently executing guest instructions that would otherwise have been fully simulated.

More detail about our efforts toward virtualization can be found in Appendix A.

# Chapter 6

# Conclusions and Contributions

## 6.1 Conclusions

We originally set out to find a method that allowed us to perform cheap and efficient ARM on ARM execution. One of our main goals for doing this was to be able to create a simulator that could run on inherently resource-constrained ARM devices, so that this simulator could be used as a major piece in ARM virtualization. We examined the two traditional approaches of full simulation and direct execution with binary translation and found a hybrid approach between these two extremes that had the potential to outperform full simulation while retaining the same amount of control. This hybrid approach yielded a framework that offers a lot of scope for optimizations depending on the particular application and platform. These optimizations can also be added independently of each other, giving the implementor a considerable amount of flexibility for which ones they want to include for their particular use case.

Our discovery that the caching mechanisms of most ARM devices prevents a self-modifying code approach such as ANA has led us to believe that while it may not be a universal solution, ANA can still be ideal for a variety of different problems. We proposed three possible workarounds for the issue of the ARM instruction cache, analyzed the tradeoffs of each, and identified the instances where each one could be used. Finally, we showed that, despite its shortcomings, ANA remains to be a very viable candidate for prototyping a virtual machine monitor because it fulfills the role of the VMMs instruction interpreter very well.

Ultimately, all these different tradeoffs for different approaches lead us to the following conclusions about efficient simulation:

- Code sections that are frequently executed should be binary translated and then directly executed.
- Short-straight line code sections should be treated on a line by line basis: if the instruction is easy to fully simulate with relatively little overhead, it should just be simulated. If, however, the instruction is complex, it can be directly executed via ANA.

## 6.2 Future Work

Much work remains to be done on ANA and in the area of ARM-on-ARM execution in general. For one thing, it would be interesting to see how our proposed workarounds to the caching problem actually perform in practice. It is also unclear right now which exact cases will ANA outperform the traditional two methods of full simulation and direct execution in, so measurements of the system's performance still needs to be done. A lot of the optimizations that we've mentioned in the course of describing ANA's implementation have also yet to be tried, so a lot of work remains there as well.

We are optimistic about the possibility that ANA will eventually evolve into a more general solution to the problem of efficient ARM on ARM execution. While it may not be the right solution for the problem, ANA has many principles in it that could be used to build the right solution. Such a solution will not only allow us to build great simulators on the ARM for the ARM, but also great virtual machines as well.

## 6.3 Contributions

In summary, our contributions in this thesis are:

- Formulated the problem of executing ARM code on the ARM architecture.
- Identified the strengths and weaknesses of traditional approaches to simulation.
- Proposed and implemented a hybrid method for solving this problem, and discovered the challenges to such an implementation in the form of the ARM instruction cache.
- Designed and analyzed solutions for overcoming the caching issues and identified the situations for which each of these solutions could be used.
- Identified the utility of ANA in the process of implementing virtualization for the ARM.

# Appendix A

# ARM Virtualization

For our first attempt in virtualizing the ARM, we chose to implement a hosted model of full virtualization. This means that the user can simply install an application on their existing operating system, and by installing a module in the kernel capable of performing privileged level instructions, this application is able to create a virtual machine monitor that resides in its own, independent machine state.

## A.1 Overview of the Virtualization Framework

The core idea behind a hosted virtualization framework is in the ability to world switch a system into an entirely different memory space that is owned by a virtual machine monitor. Guest programs operate on top of this monitor, whose purpose is to maintain the virtual machine's state. In order to achieve this from a user level applications perspective, several key pieces need to be in place.

1. A user level process to serve as an entry point to the entire framework (the VMX).
2. A kernel module that allows for register save/restore and page table switching.
3. A virtual machine monitor that serves as an interface between the guest level programs and the host.

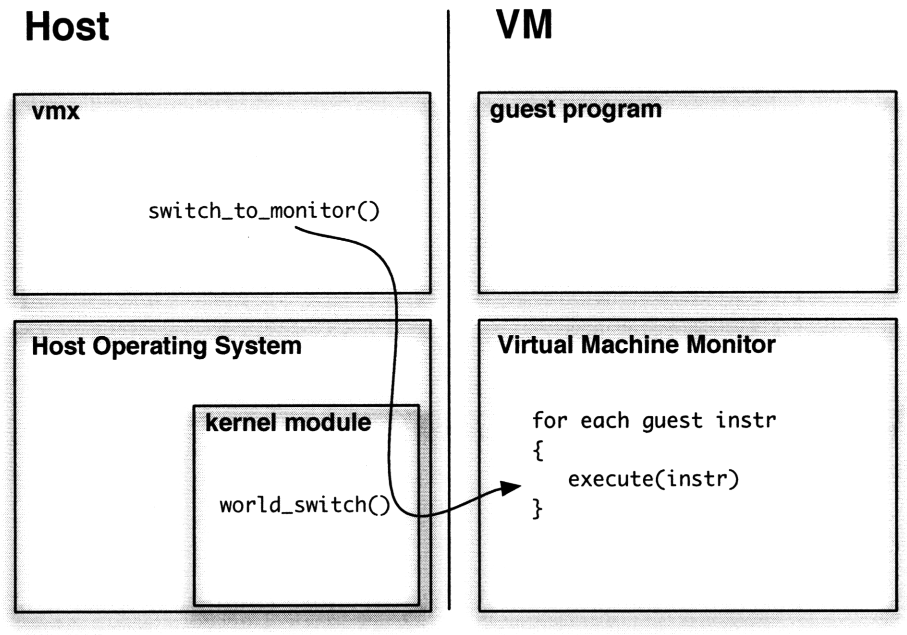

Figure A-1: The VMX sits on the host OS as a user process, utilizing the kernel module to world switch into the virtual machine monitor. Guest programs are then run on top of the virtual machine monitor in an instruction execution loop.

Figure A-1 shows a clearer picture of how these modules relate to one another.

In this setup, a world switch from the host to the virtual machine is essentially a saving of the real CPUs state and loading in a virtual (or guest) CPU state in its place. Along with this, a different page table must be used to present the guest the illusion that it has access to its own full memory address space. All of this can be done by the kernel module, which is called via the VMX as it starts. This process is reversible to allow a switch back to the host from the virtual machine. The VMX is also responsible for initializing the CPU and memory state for the guest.

On the virtual machine side, the VMM needs to be capable of executing guest instructions. This can be achieved via full simulation, binary translation, or something in between like ANA.

| Exception type | Mode | Normal address | High vector address |
|---|---|---|---|
| Reset | Supervisor | 0x00000000 | 0xFFFF0000 |
| Undefined instructions | Undefined | 0x00000004 | 0xFFFF0004 |
| Software interrupt (SWI) | Supervisor | 0x0000008 | 0xFFFF0008 |
| Prefetch Abort | Abort | 0x0000000C | 0xFFFF000C |
| Data Abort | Abort | 0x00000010 | 0xFFFF0010 |
| IRQ (interrupt) | IRQ | 0x00000018 | 0xFFFF0018 |
| FIQ (fast interrupt) | FIQ | 0x0000001C | 0xFFFF001C |

Table A.1: ARM exception and interrupt tables for high and low vectors.

## A.2 Exception and Interrupt Handlers

A virtual machine implementation needs to be able to handle interrupts generated by the software and hardware, as well as any exceptions caused by the guest code, the monitor, or the host. This is our policy regarding interrupts and exceptions:

- Interrupts should always be forwarded to and handled by the host. We are uninterested in all interrupts because they will generally concern the host and not the guest.

- Exceptions (or faults) need to be examined, and it must be determined whether the fault was caused by the guest code, the host, or the monitor. If the guest code generated the exception, we need to handle the exception in the monitor, or in some cases, let the guest code handle it. Otherwise, the exception must be forwarded onto the host so that it can be properly dealt with by the host OS.

On the ARM, the exception and interrupt vectors could be found in two possible locations, as illustrated in Figure A.1.

The N800 uses high exception vectors, so we mapped into the monitors memory an instruction that jumped to our interrupt handler. Our interrupt handler would then simply make a `UserCall(NOP)`, a routine that is used to request execution of some code on the VMX, requiring a world switch to the user and back. In this case, the `NOP` specifies a non-operation, so it will effectively trigger only the world switches with no other effects. The idea behind this is to leave the interrupt unserviced, switch to the host where the interrupt line is still raised, let the host deal with the interrupt on its own, and then return back to the monitor as if nothing had happened.

Handling exceptions is a significantly more complex issue, since it requires careful examination of the machine state combined with a fair amount of deduction to decide whether the exception needs to be handled by the monitor, the host, or the guest. How this is exactly done is beyond the scope of this thesis. For now, our current implementation simply exits upon an exception (whereas previously, the N800 would simply crash because of the lack of an exception handler).

## A.3 A GDB Stub for the Virtual Machine Monitor

We decided that implementing a GDB stub to debug guest code would be very useful. In this section, I will briefly discuss the benefits of having a GDB stub for the VMM, the differences between implementing a normal stub and one for a virtual machine, and the way we integrated it into our framework.

As we continued to add and extend modules on the VMM to support a wider variety of guest code, we needed to find a way to reliably stop the guest program at arbitrary locations and inspect the state of the virtual machine. Having a GDB stub on the VMM to debug the guest code would allow us to do exactly this and more. In addition to breakpoint and memory read/write operations, the stub would provide us with single-stepping functionality for debugging guest code, or in some cases, the VMM itself.

Traditionally, there is only one way to actually implement a GDB stub for a typical program. You need to:

1. Implement the basic read/write methods used by the stub to communicate with the server.

2. Implement several key functions that points the stub to the exception vectors of the host machine.

3. Modify the guest code to make the actual call to modify the hosts exception vectors with GDBs vectors.

(1) completes the stubs functionality as a GDB server that can be connected to by a remote debugger (via serial port or TCP/IP). (2) and (3) are necessary so that GDB can gain control of the CPU by modifying the hosts breakpoint exception vectors to point to GDBs own handler. This handler can then pause the guest program and respond to the commands from the debugger.

However, since we have full control of the virtual machines CPU via the monitor, it is possible to implement a GDB stub on the VMM without touching the exception vectors. We can choose to simply invoke the stub from within the VMM and allow the stub to control the read-decode-execute loop of the interpreter. Reading machine state is simply a matter of looking at the virtual CPU struct and performing virtual memory reads. Breakpoints could be implemented simply by keeping a list of addresses that the monitor should stop at and hand control over to the stub. This allows for a relatively straightforward implementation of GDB stub support on the monitor side.

Unfortunately, this will only work for instructions that get executed from the monitors interpreter. This model breaks apart when we start directly executing guest code on the hardware. Since breakpoints, single-stepping, and other control flow mechanisms of the GDB stub are implemented at the monitor level, there would be no way for the stub to take control of the hardware like a normal implementation would allow.

A solution to this issue would be to treat the virtual machine as actual hardware and implement the stub directly onto that. This essentially is a reversion to the traditional method where we allow GDB to modify the virtual machines exception vectors. However, a significant level of complexity arises from the fact that we will need to be extra careful about when GDB should get control of the PC as opposed to when the monitor should, since we have already modified the vectors as explained in the previous section.

As of writing, we have implemented the first half of the GDB stub, allowing us to effectively debug guest programs running on the interpreter. Perhaps the most

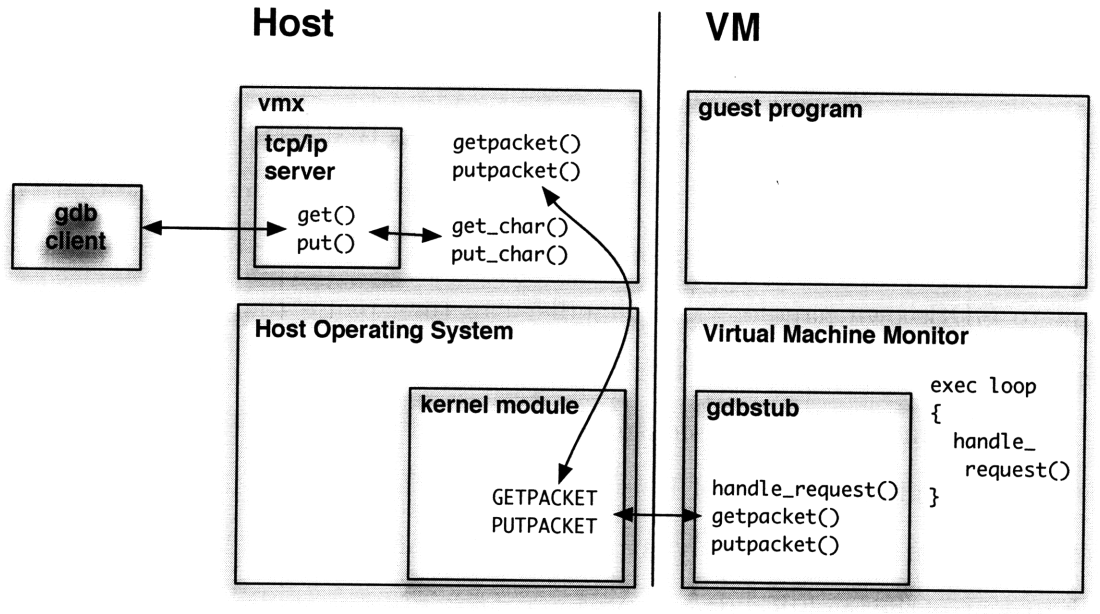

Figure A-2: The GDB debugger client connects via TCP/IP to the server in the VMX. Requests get forwarded through the kernel module over to the stub in the VMM, which actually handles the request and provides information about the guest program state.

important thing to note is that (1) needed to be implemented in a special way to deal with the fact that the monitor has no serial or TCP/IP support by itself because in its world, it has no access to the communication devices on the host. The way we dealt with this was to implement the read and write packet interface as a User Call through the VMX. We ran a server stub on the VMX (the host) that does the actual communication with the remote debugger via TCP/IP, forwarding and responding to packet requests through the UserCall interface. The interaction between the monitor and the debugger is illustrated in Figure A-2.

# Appendix B

# Flash Memory Access Statistics

One of the most often cited reasons for employing virtualization on embedded processors is the lack of spare memory. Indeed, memory has a linear cost and inexpensive devices have just enough memory to keep the cost down. Luckily, many embedded devices have the ability to add flash memory. Flash is better than disk in that one can execute from NOR flash and one can quickly swap because the overheads are low. This appendix provides evidence to support this claim.

## B.1 Types of Flash Memory

There are two most commonly used types of flash memory: NOR flash and NAND flash [3]. NOR flash is capable of high-speed random access but suffers from slow writes, low density, and high cost. Conversely, NAND flash's sequential access architecture achieves reasonable write speeds, has a much higher density, and is very cheap [9]. However, this same architecture makes random access reads on NAND flash comparatively slow.

The diametric properties of these two types of flash means that embedded devices will typically employ both to cover different tasks. NOR flash is especially good for code storage and applications that execute in place, while NAND flash offers cheap data storage good for frequent rewriting [9].

| file size | read speed | write speed |
|---|---|---|
| 500k | 10.65 M/s | 22 M/s |
| 1MB | 10.59 M/s | 24 M/s |
| 2MB | 10.77 M/s | 24 M/s |

Table B.1: Read/write speeds of external flash on the Nokia N800.

| file size | read speed | write speed |
|---|---|---|
| 500k | 11.35 M/s | 2.18 M/s |
| 1MB | 11.25 M/s | 4.4 M/s |
| 2MB | 10.63 M/s | 4.3 M/s |

Table B.2: Read/write speeds of internal flash on the Nokia N800.

## B.2 Results

Tables B.1, B.2, and B.3 display results from testing file reads and writes on both the internal and external flash memory as well as the speed of RAM on the Nokia N800. The speeds displayed here are an average of 10 trials per file size.

## B.3 Analysis

These tables show several surprising results:

- External and internal flash have comparable read speeds, but external flash outperforms internal flash on write speeds by nearly an order of magnitude.
- External flash performs writes faster than reads by nearly a factor of 2.5.
- External flash write speeds suffer a slowdown in comparison to RAM only by a factor of 3.

| file size | read speed | write speed |
|---|---|---|
| 500k | 78 M/s | 78 M/s |
| 1MB | 78 M/s | 78 M/s |
| 2MB | 78 M/s | 78 M/s |

Table B.3: Read/write speeds of RAM on the Nokia N800.

The results also suggest that external flash uses NAND technology and internal flash uses NOR technology. This makes sense given our knowledge of the properties of NAND and NOR memory[1].

But perhaps the most important implication of these results comes from the fact that flash speeds are not that far away from RAM speeds. The 3:1 and 7:1 ratios are orders of magnitude away from the access gap ratios of 5000:1 to 10,000:1 between traditional disk and RAM [14]. This opens up a lot of opportunities for redesigning operating systems and virtual machine monitors for embedded devices.

[1]It should be noted here that the internal NOR and external NAND flash performed similarly for reads most likely because our test methods favored sequential access. If we introduced randomness into our tests, the results would likely favor internal flash.

# Appendix C

# Development Environment Setup

## C.1 Setting up the Nokia N800

### C.1.1 Flashing To R&D Mode

Download the Flasher-3.0 utility from:

*http://tablets-dev.nokia.com/d3.php*

1. With the N800 unplugged and powered off, plug in the USB cable to your PC.
2. Press and hold the Home button on the N800 and power it on while holding this button down. Once you see the little usb symbol on the top right corner of the "NOKIA" screen, you can release the button.
3. On the PC, as ROOT, run:
   ```
   # ./flasher-3.0 --enable-rd-mode
   ```
4. Disconnect the USB from the N800, and it should now boot in R&D mode.

### C.1.2 Red Pill Mode

Taken from:

*http://www.digitalknk.com/2007/05/27/howto-ruby-on-rails-on-your-nokia-n800/*:

1. Open Application Manager

2. Click on the Application Menu(upper left hand corner)
3. Click on Tools and then click on Application catalog
4. Click on the New button
5. In the Web Address delete http:// and type in matrix
6. Click on Cancel
7. This will bring up a prompt with Red Pill or Blue Pill, click on Red Pill.

### C.1.3 Installing xterm

On your N800, navigate your way (yes, ugh) to this page:
*http://downloads.maemo.org/products/osso-xterm-advanced*
and click on the green arrow to install xterm.

### C.1.4 Installing SSH

On your N800, navigate your way (yes, ugh) to this page:
*http://maemo.org/downloads/product/openssh*
and install SSH.
To start, stop, and restart, use the usual `/etc/init.d` methods.

### C.1.5 Becoming Root

Becoming root on the N800 is simple once red pill mode and R&D mode are enabled:

```
$ sudo gainroot
```

Note: you can also directly ssh into the N800 as root using the default password, 'rootme'.

## C.2 Setting Up A VMWare Environment

### C.2.1 Setting Up A Ubuntu VM

1. Create a new VM and install Ubuntu with it. From here on, the instructions are to be executed in your VM.

2. Now you need to install VMWare Tools. But before you can do this you need to (as root):

3. Update your Ubuntu installation by repeating:

   ```
   # apt-get update
   ```

   until you have nothing to update/upgrade. Then do a dist-upgrade:

   ```
   # apt-get dist-upgrade
   ```

   Reboot as necessary.

4. Install build tools:

   ```
   # apt-get install build-essential
   ```

5. Get the linux headers for your kernel. Find out what version of Ubuntu you're running with:

   ```
   # uname -r
   ```

   then call:

   ```
   # apt-get linux-headers-<ubuntu version here>
   ```

6. Install VMWare Tools itself by selecting VM-¿Install VMWare Tools from the VMWare Workstation menu. This will mount a VMWare Tools folder on your Ubuntu desktop. Copy the VMWareTools-5.x.tar.gz file somewhere and cd into that directory.
   Then run:

   ```
   # ./vmware-install.pl
   ```

7. (optional) If after installing VMWare Tools and restarting X, your mouse wheel stops working, edit `/etc/X11/xorg.conf` (as root)

Change the line:

```
Option Protocol "ps/2"
```

to:

```
Option Protocol "IMPS/2"
```

And restart X (with CTRL+ALT+Backspace).

## C.3 Setting Up Scratchbox

### C.3.1 (optional) Installing QEMU

Download and untar into `/.` (as root):

*http://fabrice.bellard.free.fr/qemu/qemu-0.9.0-i386.tar.gz*

(or whatever is the latest version on the site)

### C.3.2 Installing Scratchbox

1. Edit `/etc/apt/sources.list` and add the line:

   ```
   deb http://scratchbox.org/debian/ apophis main
   ```

2. Update apt.

   ```
   # apt-get update
   ```

3. Download scratchbox with all its core utilities and toolchains:

   ```
   # apt-get install scratchbox-core scratchbox-devkit-cputransp \
   scratchbox-devkit-debian scratchbox-devkit-doctools \
   scratchbox-devkit-perl scratchbox-libs \
   scratchbox-toolchain-cs2005q3.2-glibc-arm
   ```

4. Add yourself to the scratchbox group (as root).

   ```
   # /scratchbox/sbin/sbox_adduser <username>
   ```

   You may need to relogin for this add to take effect.

### C.3.3 Setting Up A Scratchbox Target For The N800

1. Download the ARMEL SDK rootstrap for scratchbox:

   *http://tablets-dev.nokia.com/bora/armel/maemo-sdk-rootstrap_3.1_armel.tgz*

   Place the rootstrap tarball into `/scratchbox/packages` (as root).

2. Login to scratchbox as a normal user.

3. Create an ARMEL target.

   ```
   [sbox-SDK_PC: ~] > sb-conf setup \
    SDK_ARMEL --compiler cs2005q3.2-glibc-arm \
    --devkits=debian:perl:doctools:cputransp \
    --cputransp=/scratchbox/devkits/cputransp/bin/qemu-arm-0.8.1-sb2
   ```

   Note that your version of qemu is not the same as what we enter here. This is because scratchbox doesn't have support for higher version numbers, but it still works here. If you opted to not install QEMU, you can omit the cputransp line, I believe... or set it =None or something.

4. Select that target.

   ```
   [sbox-SDK_PC: ~] > sb-conf select SDK_ARMEL
   Restarting Scratchbox shell...
   Hangup
   Shell restarting...
   [sbox-SDK_ARMEL: ~] >
   ```

5. Extract the rootstrap.

   ```
   [sbox-SDK_ARMEL: ~] > sb-conf rootstrap
      /scratchbox/packages/<ROOTSTRAP>.tar.gz
   Unpacking rootstrap...
   [sbox-SDK_ARMEL: ~] >
   ```

6. Setup devkits, libfakeroot environment and /etc directory with following command:

```
[sbox-SDK_ARMEL: ~] > sb-conf install SDK_ARMEL
   --devkits --fakeroot --etc
Installing fakeroot version 1.4.2...
[sbox-SDK_ARMEL: ~] >
```

7. Outside of scratchbox, edit your /etc/apt/sources.list file to include:

   ```
   deb         http://repository.maemo.org/ bora free non-free
   deb-src     http://repository.maemo.org/ bora free non-free
   ```

   Do an `apt-get update`.

8. Download the kernel source into a folder in your scratchbox user's home directory. Here's how I did mine:

   ```
   $ cd /scratchbox/users/calvinon/home/calvinon/
   $ mkdir kernel
   $ cd kernel
   $ apt-get source kernel-source-rx-34
   ```

9. Go back into the scratchbox to config and make the needed kernel modules.

   ```
   $ scratchbox
   [sbox-SDK_ARMEL: ~] > cd kernel/kernel-source-rx-34-2.6.18
   [sbox-SDK_ARMEL: ~] > make n800_defconfig
     lots of output from make
   [sbox-SDK_ARMEL: ~] > make modules
     lots of output from make
   ```

   Your scratchbox environment is now complete and ready to compile ARM programs along with Linux kernel modules.

## C.4 (optional) Setting Up USB Networking

1. PC setup (as root)

   Make sure your kernel has usbnet module installed (if not, you'll need to recompile the kernel). The Ubuntu installation should have it by default.

check if you have it (as root) with:

```
# lsmod | grep usbnet
```

should output something like:

```
usbnet             18888 0
```

2. Edit your `/etc/network/interfaces` file to add the following at the end:

```
allow-hotplug usb0


mapping hotplug
        script grep
        map usb0


iface usb0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        broadcast 192.168.2.255
        up iptables -I INPUT 1 -s 192.168.2.2 -j ACCEPT
```

3. N800 setup (as root)

   Edit `/etc/network/interfaces` file to add the following at the end:

```
 iface usb0 inet static
        address 192.168.2.2
        netmask 255.255.255.0
        broadcast 192.168.2.255
        up route add default gw 192.168.2.1
```

4. Check if g_ether module is up:

```
# lsmod | grep g_ether
```

   Chances are that it isn't. You'll need to load the module. Run:

```
# insmod /mnt/initfs/lib/modules/`uname -r`/g_ether.ko
```

   The first time usually fails because of a busy drive/resource. Ignore this and just do it again and it should work.

5. Connect the device to the PC. On both the N800 and PC, enable the usb0 network interface:

   ```
   # ifup usb0
   ```

   You should now be able to connect to the phone via 192.168.2.2.

# Bibliography

[1] Erik R. Altman, David Kaeli, and Yaron Sheffer. Welcome to the opportunities of binary translation. *Computer*, 33(3), March 2000.

[2] Fabrice Ballard. Qemu, a fast and portable dynamic translator. *USENIX*, 2005.

[3] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4), April 2003.

[4] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *SOSP*, 1997.

[5] Anton Chernoff and Ray Hookway. Digital fx!32 running 32-bit x86 applications on alpha nt. *USENIX*, 1997.

[6] Cristina Cifuentes and Mike Van Emmerik. Uqbt: Adaptable binary translation at low cost. *Computer*, 33(3), March 2000.

[7] Peter S. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2), March 2002.

[8] Kevin Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996.

[9] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim, and Bumsoo Kim. Cost-efficient memory architecture design of nand flash memory embedded systems. *Proceedings of the 21st ICCD*, 2003.

[10] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel & Distributed Technology*, 1995.

[11] Scratchbox. Movial. *http://www.scratchbox.org*, 2007.

[12] David Seal. *ARM Architecture Reference Manual.* The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[13] Andrew N. Sloss, Dominic Symes, and Chris Wright. *ARM System Developer's Guide.* Elsevier, Reading, Massachusetts, first edition, 2004.

[14] Alan Jay Smith. Disk cache - miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[15] Chen Yu, Ren Jie, Zhu Hui, and Shi Yuan Chun. Dynamic binary translation and optimization in a whole-system emulator - skyeye. *ICPPW*, 0:327–336, 2006.