Atome - Binary Translation for Accurate Simulation

Mathieu Brethes

September 16, 2004

Abstract

An in-depth study of the field of Virtual Machine optimization is presented. It covers dynamic binary translation, simulation accuracy, and intermediate code representations. It is aimed at readers that have a good knowledge of computer architecture and wish to learn about creating virtual machines.

Contents

Acknowledgements

Many people have helped me through this project. I would like to thank Ola Ågren, my Computer Architecture teacher at Umeå University, and Per Lindström, director of studies of the Computer Science department. Many thanks will also go to my colleagues, Fabien Lagriffoul and Daniel Wibberg for their moral support and kindness.

Introduction

Computer simulation is a concept almost as old as computer science itself. That all computers can simulate each other is an immediate consequence of the theoretical work of Alan Turing and Alonzo Church [MCE+02].

Those virtual machines, as they are called, are these days used in all the fields of computer science and software engineering, from operating system to software development, and from embedded system debugging to leisure and amusement.

The research has focused on ways to optimize this simulation, which is, when done naively, very computationally-intensive. In a general way, the more accurate a simulator is, the more computer resources it consumes — there is a tradeoff between accuracy and speed.

Several projects try to reduce this gap between accuracy and speed, and Atome[1] is one of those projects. Atome aimed at simulating accurately the cycle timings of a pipelined CPU by using binary translation. This project could then be used as a framework to simulate various devices based on the same architecture. However Atome will probably never be completed — but what has been done might be interesting for other researchers.

This report presents the concepts behind computer simulation as well as the state of the art in virtual machines, describes in detail the issues that are to be addressed, and discusses the investigated solutions.

[1] A Translator Optimized for Machine Emulation.

Chapter 1

Background Information

This chapter assumes that the reader already has a good knowledge of computer science in general, and specifically a bit of knowledge in computer architecture. The fundamental notions behind the concept of "virtual machine" are going to be covered in this part of the report.

1.1 What is a virtual machine?

What is a virtual machine? Unluckily for us, there is no general definition. Virtual machines have very different aims and are used for completely different things: think about the popular Java and more recent .Net environments, IBM's virtual-machine based VM operating system, and the gaming-oriented game-console emulators. The concept of virtual machine covers a lot of applications.

The general goal of a virtual machine is to execute a program in a virtual environment which "makes the program believe" that it is running in the real environment.

Java programs do not actually execute the same way on each different computer, but from the program's view the interface is the same. On the other hand, simulators aim at providing a virtual execution which is the same as the execution on the original hardware.

In fact, there are two ways of sorting the different types of virtual machines: one way is based on the virtualization itself, and the other is based on the accuracy regarding simulation. The two classifications complement each other.

1.2 Classification by virtualization degree

A computer is basically made of two parts: a Central Processing Unit (CPU), and a set of peripherals. On modern computers software never access peripherals directly, but uses operating system (OS) calls which prove more convenient for portability. And the CPU itself is nothing but an interpreter for a language (machine language) that is different for each family of CPUs[1]. This language is called an ISA, which stands for Instruction Set Architecture.

Hence a computer is basically an ISA and an OS. We'll come back later to the exceptions.

We can then define four different categories of virtual machines, regarding the degree of similarity between the virtual ISA (vISA) and the host ISA (hISA), and between the virtual OS (vOS) and the host OS (hOS).

1.2.1 Same ISA, same OS

The virtual machine uses the same architecture and the same operating system calls as the host machine. This kind of virtual machine is called "multi-program system", as its objective is to make the virtual programs believe they run on different computers (but there is in fact only one computer running the VM). This is useful to simulate networking and cluster computing. It can be also an interesting principle of organization for an hypothetical operating system: instead of doing a multitask system, it is possible to build a virtual machine simulating several mono-task systems.

1.2.2 Same ISA, different OS

When $vISA = hISA$, but $vOS \neq hOS$, the virtual machine becomes a kind of operating-system simulator. The virtual program can be executed directly on the host CPU, but all its input and output has to be redirected and/or modified to fit the host operating system. Those virtual machines are called "OS virtual machines".

[1] CPUs of the same family can usually understand each other: take today's Pentium IV, which is backwards-compatible with the very old 80286... However, upwards compatibility usually requires tricks, like using software traps to detect unknown instructions and simulate them accordingly.

Wine is an example of such a virtual machine, enabling linux users to execute Windows programs on their Intel x86 computers. Since Windows binaries run on the 80x86 architecture, they can easily run from a 80x86 Linux too. However every program call to the Microsoft Windows system APIs (Application Procedure Interface) has to be translated to the equivalent Linux call(s) by Wine.

1.2.3 Different ISA, same OS

Some operating systems (like Windows NT 4 or Linux) have been written for several different processor architectures. Sometimes it can be nice to have a program, which is usually running on one architecture, to run on another one. A virtual machine of this category, like FX!32, will enable this behavior. This kind of virtual machine is called "Application Binary Interface (ABI) virtual machine".

1.2.4 Different ISA, different OS

This last category regroups all the other virtual-machines — simulators, emulators, co-designed virtual machines, and so on. When $vISA \neq hISA$ and $vOS \neq hOS$, the virtual machine has to interpret the machine instructions of the virtual program to be executed as well as to provide a virtual environment that simulates its operating system.

1.2.5 Exceptions

Some virtual machines are aimed at simulating, not the operating system, but the underlying hardware. They are used mostly to help OS developers with their work, but also to simulate machines that do not have an OS (like video game consoles, for example).

For example, Bochs is a 80x86 PC emulator on which the user can have any Intel-compatible system running (from DOS to Windows XP, OS/2, or Linux).

1.3 Classification by accuracy

Another way to classify virtual machines is by the accuracy in which they simulate the "real" machine which they are built after. The closer to the

original machine, the more accurate — but also the more computationally-intensive...

There are five main categories [Tij00] which are the following:

- Datapath accuracy,
- Cycle accuracy,
- Instruction level accuracy,
- Basic block accuracy,
- High-Level Emulation.

Emulators such as video-game console emulators, or in a more general way embedded devices emulators, are located between 2 and 3 (cycle accuracy and instruction-level accuracy).

1.3.1 Datapath accuracy

Virtual machines which fall in this category simulate the internal workings of hardware devices, such as a CPU, a bus, or peripherals. They also simulate the hardware working of the memories (with latencies, for example) and so on. It's just like simulating the electronics of the machine, and it consumes a lot of resources (around 10000 instructions per CPU cycle). They are generally only used by hardware designers to experiment with new design ideas.

> "[If] we wish to simulate an entire system and to do so with total accuracy [we need] a perfect model. There are obvious problems with seeking perfection, including cost, time to completion, specification inaccuracies, and implementation errors. Most important is the problem of workload realism." [MCE+02]

1.3.2 Cycle accuracy

Cycle-accurate virtual machines do not simulate the inner workings of the electronics, however they simulate the correct timings of the target peripherals (timers, disk subsystems, and so on). The CPU and its pipeline timings must be respected too, as well as the memory hierarchy (memory management unit, cache, memory buses, etc.). Simulators belonging to this category are called complete system simulators [AM00].

1.3.3 Instruction level accuracy

Virtual machines of that level do not accurately simulate CPU timings — however, they interpret each instruction separately. That means they can be used for debugging at the assembly language level. The CPU pipeline is usually not simulated anymore.

> "Instruction-level simulators are a crucial component in developing and analyzing computer architectures and system software. [...] Such simulators allow analysis of program behavior in a manner impossible or impractical on real hardware." [Mag93]

1.3.4 Basic block accuracy

The virtual machines falling in this category do not even interpret each instruction separately. They take groups of instructions, an accurate input, and produce an accurate output. The machine can reorganize the instructions in the group (basic block) as it wants, or recognize the block as some compiler-generated block and produce an according output without having to compute the CPU state for each instruction in the block.

1.3.5 (Very/ultra) high level emulation accuracy

HLE (High-Level Emulation) is the last level. Virtual machines fall into that category when the input-output they use and produce isn't exactly the same as on the "real" machine they simulate. For example, a machine that would take a Windows program, and simulate it using the Macintosh's GUI (Graphical User Interface) to replace Windows' GUI, is a HLE.

1.4 Well-known virtual machines

There are many, many different virtual machines alive out here, and it is good to know at least some of them if you are interested in the field. Of course, this list is far from complete.

When one talks about virtual machines, the audience usually thinks about Java — but Java is not a virtual machine, it is a programming language with a few specificities. Java programs are compiled to something called "bytecode", which you can think as assembly language instructions for a virtual CPU[2]. This bytecode is then interpreted by one of the various JVMs

[2]This CPU is called the JPU (Java Processing Unit), and it is a stack machine.

(Java Virtual Machines) available for almost every existing platform. The JVM is, here, the virtual machine, not Java itself! Some hardware companies produce CPUs that are able to understand Java bytecode "in hardware".

But Java is not alone here. Microsoft's ".Net" is also considered to be another "Virtual Machine", and one more time it is not — the virtual machine is the program interpreting the .Net programs. Even if this platform is available only for Windows systems, the bytecode produced by the ".Net" compiler suite is hardware-independent.

The advantage of designing a bytecode system is that you do not have to define a "real" CPU to work upon — instead, you tailor the virtual CPU you created to fit your needs. Bytecodes are designed to be interpreted swiftly in order to compensate for the latencies of a simulation.

Well known is also VMWare, a virtual system which enables you to run several operating systems at the same time on the same computer. Digital's FX!32 enables x86 Windows NT programs to run on an Alpha computer (running a native version of Windows NT). And of course, Wine (Wine Is Not an Emulator) provides a convenient way of having Windows programs running on a Linux machine.

CPU manufacturers often bundle virtual machines along with the software development kits they sell. Those machines are used as a cheap alternative to development boards. ARM has built the ARMulator, a customizable simulator which is able to reproduce the behavior of the entire ARM CPU family.

> "[The ARMulator] can operate at various levels of accuracy: Instruction-accurate modeling [...], cycle-accurate modeling [...], timing-accurate modeling." [Fur96]

One can also find simulators and emulators related to gaming — the most known is probably "Mame", a virtual machine able to simulate a broad range of arcade boards.

And of course, all modern Intel and AMD CPUs are, in fact, co-designed virtual machines: they take 80x86 instructions which they translate in hardware to sequences of other instructions that have the same effect (but are more efficient regarding the CPU architecture).

In the next chapter, we are going to consider in more depth virtual machines built for research purposes.

1.5 Concluding remarks

There is something we didn't cover in the previous sections, but which might be worth saying now. The question is to know whether an operating system, by itself, is a virtual machine or not.

Well, all modern operating systems prevent programs from accessing directly the hardware of the computer on which they run. Instead they offer an interface (called API for Application Programming Interface) that all applications use, thus building a virtual environment for the programs.

Moreover, systems like Unix are designed in such a way that when several programs are running at the same time, each program "thinks" it is alone using the CPU, memory and so on. Such a system is also a virtual machine.

So, yes, operating systems are virtual machines. They can even be thought of as virtual machines stacked together, abstracting each time more and more the hardware, each time providing more and more convenient access to the various programs. IBM has created an operating system which is, as a matter of fact, based on virtual machines — it is called VMOS.

Chapter 2

Literature review

A lot of research has been made on virtual machines, which means the literature covers a broad range of topics. In order to understand how research evolved over the years, we will start by covering the history of the field. Then we shall look at various simulators built by researchers and the corresponding literature, which will provide us with more technical insights. All this will prepare the next chapter, where we discuss what we discovered.

2.1 Some history

When and why did the virtual machines appear over the field of computer science?

As we saw earlier, the notion of virtual machine is broad, and the concept of virtual machine has proven to be a powerful tool in shaping today's computer [KMM99]. We saw that modern operating systems are built as stacks of virtual machines, that even a running program is often referred to as a virtual machine — a machine that doesn't exist as a matter of actual physical reality. The virtual machine idea is itself one of the most elegant in the history of technology and is a crucial step in the evolution of ideas about software [Gel97].

However, this idea was not always clear to everyone and it was not until the mid 1960's that it was put into practice [KMM99].

As a matter of fact, the first virtual machine was designed at a time where computers were extremely expensive, so it was necessary to share a single

computer with many different people. The virtual machine was then a program running on the single computer, partitioning it into several independent virtual "sub-computers", each one which could be used by a different user. Each virtual machine was running on the main unit using a partition of the CPUs execution time.

Papers discussing the idea of a time-sharing system began being published about 1959, then followed a period of experimentation at MIT and other institutions. The first time-sharing system, called CTSS (for "Compatible Time-Sharing System"), which was demonstrated in November 1961 can be considered as a kind of virtual machine. This system was running on an IBM 709 machine offered by IBM to the MIT [Var97].

The system evolved along with IBM mainframes, as IBM teams were cooperating with researchers to address hardware issues preventing the creation of more efficient virtual machines.

By that time, a virtual machine was taken to be an efficient, isolated duplicate of the real machine, a notion that was explained through the idea of virtual machine monitors (VMM). As a piece of software a VMM had three essential characteristics. First, it provided an environment for programs which is essentially identical with the original machine; second, programs run in this environment showed at worst only minor decreases in speed; and last, it was in complete control of system resources [PG74].

Theorem 1 *"For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions." [PG74]*

CPUs working with a supervisor and user mode, memory protection, address relocation, trap mechanisms, and the respect of the conditions above led to the first real virtual machines: virtual machines that were able to run themselves recursively.

IBM's TSS (Time Sharing System), released in 1965, was the first "commercial" implementation of a time and resource sharing system based on this idea of virtual machines. It was followed, five years later, by the more successful VM/370 (running on IBM's System/370 mainframe) which took a lot

from previous work, like CTSS[1].

If we now consider virtual machines as they are more though of these days (as platforms for truly portable computer programs) we can see the idea is not quite new, either.

> "Such approaches date back at least to the 1970s, [and] have achieved new impetus based on the current popularity of the programming language Java." [Gou01]

One particular representation, P-Code, which was invented as an intermediate form for the ETH Pascal Compilers, became pervasive as the machine code for the UCSD Pascal System. Pascal programs were compiled in an intermediate form (the "p-Code"), which could be run on any machine owning the p-Code interpreter or compiler [Gou01].

Simulators, virtual machines making a program from one architecture work on another, are also quite old. It is, however, quite hard to know which appeared first or who came up with the idea. We know that in 1964, IBM's new System/360 provided emulation for IBM's older 1401 computers. Microsoft also pioneered some work by creating a CP/M emulator in the late 70-ties [Tij00]. The first simulators were made to ease application porting from one platform to another, or sometimes to create virtual old systems which were not available anymore. Thus the simulator, contrarily to its current reputation of "pirate tool"[2], has a capital interest in the field of computer science: it enables software to survive the death of its associated hardware platform.

The main problem with computer simulation is that it is slow. A typical simulator is mostly a program looping on the listing below:

- Instruction fetching,
- Byte grouping,
- Disassembling,
- virtual machine state updating.

[1]This is summarized quite a bit. For a complete history of IBM virtual machines, please read the 70-page article referenced as [Var97]. It is very interesting.

[2]This is mostly due to the recent game console emulators against which gaming businesses fight — however it is interesting to note that modern game consoles integrate simulators themselves: legacy game boy games can be played on the game boy advance, legacy playstation games can be played on the Playstation II, and so on.

The last step is the most computationally-intensive, especially if the virtual machine is aimed to be very accurate regarding the original system being simulated. Now this is okay as long as the system to be simulated is order of magnitudes slower than the system on which the simulator is running[3]. But challenges arise when this condition is not met[4].

Since computer science is the science of solving problems, a lot of research has been made on how to optimize simulation. We are going to cover below what has been done.

2.2 Static binary translation

Static binary translation is an idea which appeared along with the research in portable computer programs.

A program encoded for an abstract virtual machine[5] may be used in two ways: a compiler back-end may compile the code down to the machine language of the actual target machine, or an interpreter may be written which emulates the abstract machine on the target [Gou01].

If that compilation was possible for abstract virtual machines, it could also be possible for virtual machines simulating "real" devices. The first binary translators, called "static translators", were working in an "offline" fashion, translating the input program first, and then executing it using a form of monitoring as we will see.

We can differentiate between 7 global passes a static translator can take:

- Byte grouping: using knowledge about the format of an executable and some heuristics, groups of bytes are retrieved from the executable which make up the programs code section(s). Precise code discovery is generally an unsolvable problem, but various techniques have been very successful in practical situations.

[3]For example, simulating a 4MHz Game Boy on a Pentium II at 400 MHz does not require any special technique.

[4]For example, simulating a 100 MHz ARM board on a 400 MHz Pentium II CPU will be quite hard.

[5]Here, like Java, .Net or p-Code.

- Disassembling: the found bytes are disassembled/decoded and transformed to instruction-operand(s) pairs. For this pass a description of the binary representation and possible operands of each instruction is required.

- Intermediate code generation: the assembly instructions of the previous pass are transformed into a machine independent representation. At this level the language is still assembly-like (e.g. three address code) but abstracting from any machine specifics. This is sometimes called the semantic specification.

- De-compilation: this is a special optional pass. Here we transform the intermediate code from the previous pass into high-level language statements (like C or Pascal code etc.). This is the highest level that we can reach. From hereon we can use traditional and well-understood compiler technology to carry out the remaining steps of the transformation[6].

- Compilation

- Assembling

- Byte storing

However, it is not always possible to translate the whole program. Specifically, self-modifying code cannot be statically translated. A static translator has then to be combined with some form of emulation in order to support handling undiscovered code, address calculation for data sections (which need the original program counter), and so on [Tij00].

A good example of this is the Digital FX!32 software. This piece of software was created to execute high-end Intel x86 applications on Alpha machines, both systems using Windows NT4.

Digital FX!32 does not do any translation when the program is being executed for the first time: it is not, then, a dynamic translator. Instead, it captures an execution profile that is later used by a binary translator to translate into native Alpha code those parts of the application that have been executed. As this translation takes place in the background, the software

[6]If we omit this pass, we have to generate the target code (at least the assembly code) ourselves from the intermediate representation.

can use computationally intensive algorithms to improve the quality of the generated code [HH97].

This is probably a big advantage of static translation: a translator-based virtual machine needs only translate each program once — moreover it can take as long as it wants to do so, since the program is not supposed to be executed when it is translated! Therefore it is a lot easier to obtain good performance with such a simulator.

Digital FX!32 is made of 7 parts, each having a special role in the virtual machine's execution.

- *The transparency agent* provides transparent launching of x86 applications [HH97]. Since the virtual machine is supposed to make Intel Windows NT programs work on Alpha Windows NT computers, transparency is very important. The user should be able to start both kinds of programs the same way — by double-clicking on them.

- *The runtime* is invoked by the transparency agent whenever an attempt is made to execute an x86 image [HH97]. It is responsible for the set-up of the translator, it can for example look whether the image had already been translated or not, and act accordingly.

- *The emulator* has the fundamental job of running x86 applications before they are translated [HH97]. The major drawback of the Digital FX!32 approach, of course, is that the first time a program is executed, it is only emulated — hence its execution is quite slow. Moreover, the emulator has to prepare the field for the translator, by generating execution profiles, which means it is quite a heavyweight process. This is unavoidable.

- *The translator* is invoked by the server to translate x86 images for which a profile exists in the database [HH97]. As it is executed "offline", it can take more time to create an efficient translation.

- *The database* contains profile files, translated images, and control information. Translators in general need a database like this, even if it is only temporary, to organize the information they use.

- *The server* is a Windows NT service that normally starts whenever the system is booted [HH97]. It handles the general organization of the virtual machine as well as the database.

- *The manager* is here to provide an interface to the configuration information present in the database, so that the user can control various aspects of FX!32 operation [HH97]. Even if the virtual machine runs transparently on the system, it is still possible to observe what it is doing by using that part, which plays the very important role of user interaction agent.

Static translators are most useful when there are only a few applications that need to be executed often on the host system. If we take the example of Alpha machines and FX!32, the usual Intel x86 applications to be executed are probably the Microsoft Office suite and Netcape/Mozilla browser, which do not exist as native applications. As those applications are executed frequently by the user, the initial cost of translation is covered by the use of the translated versions over time.

Problems then arise when there are a lot of different applications that are not executed often — take for example a gaming emulator running old games, where each game might be used one or two times... Moreover, this idea of static translation is not adapted to situations where the application to be executed is being developed — its source code changes often, and the resulting binaries always need new translations[7]. A static translator is not, then, a developer tool.

2.3 Dynamic binary translation

Dynamic binary translation is here to deal with this issue, and has been the main focus of research over the last decade.

For developing applications (specially for embedded devices) it is a lot easier to use a simulator to test and debug than to work directly on the real hardware. Simulators provide the developer with a lot of things that hardware cannot achieve — think about breakpoints, step-by-step execution, source-level debugging, and so on. They also have the advantage of being virtual: there is no need to actually own the hardware in order to develop something for it!

[7]But why would someone need to develop a program on a simulator? Well, think about Java: if your JVM (Java Virtual Machine) is a static translator, and that the binary you develop has changed every time you run it, it will always be emulated and never executed natively.

However, those simulators aimed at developers need to be quite accurate. This implies that they use a lot of computing power as we saw earlier. Therefore a lot of effort has been put into translation techniques fitting the needs of those simulators: this is the birth of dynamic binary translation.

Dynamic translation in general works because a computer program is more or less a construction made of programming loops. As in static translation, the first time an instruction is met, it is emulated and its translation stored. If the instruction is executed again, the translated version is used, thus speeding up the simulation[8]. The challenges of dynamic binary translation deal with how to organize translated code to optimize simulation, how to detect which parts of the code are executed most frequently in order to perform even more optimizations, and so on.

In the rest of this report, we will talk about the *target machine* for the device to be simulated, and the *host machine* for the device on which the simulator is running.

2.3.1 Interpretive translation and partial translation

Interpretive translation is not a binary translation per se, but is a quite easy way to start optimizing a simulator. Target code is translated into a format that is easier to interpret, and this format is then interpreted [Mag93]. This approach has the advantage of simplicity, portability (the intermediate format can be machine-independent), and accuracy. As the simulator interprets each instruction, events like asynchronous interrupts, breakpoints and so on can be generated exactly when they are supposed to happen.

Since translating to an intermediate format is somewhat faster than translating to binary, dynamic binary translation and interpretive translation can be combined into something new called partial translation. The choice of which and how many instructions to translate can be done according to several heuristics, like detecting an inner loop [Mag93].

This enables the host system to get rid of possible unnecessary work that interpretive translation has to do: instruction flow (the control program has to run through each interpreted instruction in sequence, whereas fully translated instructions use the standard processor flow), event handling (if we

[8]Of course, if the program to be translated does not contain any loop at all, the translation process is useless.

assume that all events occur at a fixed ratio of the CPU clock frequency, we can build blocks of instructions that can "fit" into this ratio), updating the state of the virtual machine (which can be done at the end of the block instead of after every instruction).

This approach can work well because target code tends to have a very high degree in locality [Mag93]. 90% of the execution time is spent in 10% of the code [CLU02]: the critical loops. To check for loops, a system could be counting how many times the program flow jumps to certain addresses.

2.3.2 Optimization of the critical paths

Instead of just optimizing translation for critical loops, one can also consider that there also exist something called a critical path in a program's execution flow. There might be one, or several, critical paths in a target binary program. If we can manage to collect the various scattered parts of translated code and reorganize them (or even merge them) we might be able to further optimize the simulation.

Critical paths, also called hot paths [UC01], have to be detected (this is also true for critical loops). However, it is important to notice that since the optimization is done in real-time, we have to be careful on how much resources it consumes — there is a tradeoff between simulation speed and optimization degree. We can spend more time to translate a block that is executed a lot of times than to translate a block used only a few times during program execution.

To identify hot paths, we can use an algorithm working on basic blocks. A basic block is a sequence of instructions, which ends with a control transfer, i.e. a branch, call or jump instruction. An edge is a directed pair that denotes the control flow from one basic block to another, and each edge has a weight, which measure the number of times it has been traversed by the flow [UC01].

Once an edge has been reached a certain number of times, it is said to be a *candidate*. When an edge has been reached a bigger number of times, it activates a trigger which suspends the standard translation/execution process and starts the special optimization code. This first edge is marked as "hot".

All the edges starting from this "hot" edge and contained in the candidates pool are then considered to be hot, too; and for each new "hot" edge, we

repeat the process, until the candidates pool is empty, or until only hot edges can be reached.

Afterwards, we can reorganize the program's control flow to get a more efficient result, for example by improving code locality (moving and merging basic blocks), using path prediction feedback to change order of branches, or other possibilities like inlining code (copying basic blocks several times, to remove calls and returns) [UC01].

Since the program's behavior can change over time (following what is called stages of execution), there can be several hot paths to be discovered and optimized. The virtual machine must also deal with that fact.

2.3.3 Virtual memory management

Modern hardware often includes MMUs (Memory Management Units). A simulator that has to simulate a MMU must be designed carefully to provide good optimization. The problem gets even more complicated when caches must also be simulated. For many applications of simulators, the simulation of memory is a significant if not dominant portion of the workload [MW95].

Memory simulation may ask for as much as the eight steps [MW95]:

- calculate the logical address which is requested by the target binary program,
- calculate the physical address corresponding to this logical address, simulating the TLB (Translation Look-aside Buffer),
- check for TLB misses, if the page is not present,
- simulate hardware table walks,
- check protection,
- check for alignment violation[9],
- perform the actual read/write operation,
- update the CPU state.

[9]For example requesting a byte at an odd address on an ARM board.

The general idea of memory simulation is to create yet another cache specially designed for it, the Simulator Translation Cache (STC) [MW95]. It will store translated memory addresses, using a hash table designed after the page size. If this cache yields a miss when a memory access is tried, then we have to use a more standard memory management unit. It will perform the complete calculation as stated before, but will also update the STC.

Physical memory itself can be simulated by a module that allocates it based on the target systems' page size.

When developing a simulator, one can consider that they are two kinds of memory accesses: data accesses (read or write bytes, half-words, words and so on) and instruction accesses. The latter have the advantage of sequentiality: usually, instructions follow each other in memory (as long as there is no page borderline). The simulator can exploit this fact to optimize even more memory addresses calculations for code. Regarding branches, we can distinguish between on-page branches and off-page branches. On-page branches require no TLB look-up, and thus execute faster [MW95].

Chaining, the patching of translated blocks to bypass the translation cache lookup for the current PC, is an important optimization for high performance simulation. By default, all translations end with a jump back to the dispatch loop. Embra[10] implements chaining by overwriting this jump with a jump to the next translated block in the program's execution order. Embra chains both the taken and not taken side of conditional branches [WR96].

2.4 Organization of various simulators

It might be worthwhile to study how those virtual machines, those dynamic binary translators, are organized, in order to understand better where to start if we want to develop one ourselves.

All the dynamic binary translators seem to share some common points, a common architecture that derives from dynamic compilation systems, that we will cover in the next chapter when we'll come to the design itself.

[10]Embra is a simulator developed to provide efficient machine-level simulation.

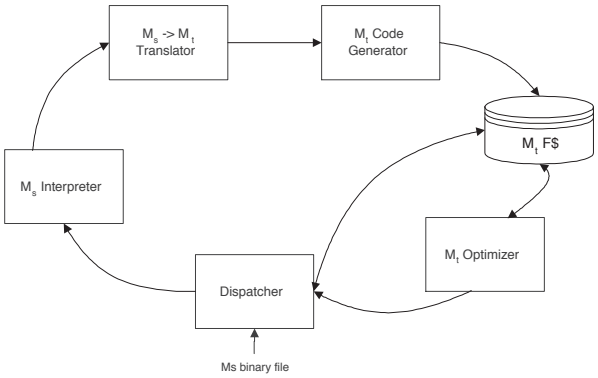

Figure 2.1: **Simulator organization (Walkabout).**

2.4.1 Walkabout, an approach of retargetability

Walkabout is a dynamic binary translator that can be customized to deal with several different architectures. It is thought of as a framework for quick experimentation with dynamic binary-manipulation techniques, and it limits binary translation to user-level code (this means it doesn't have to deal with MMUs, caches, CPU timings, and so on) [CLU02].

Even if the architecture of the Walkabout framework borrows from the architecture of most existing dynamic compilation systems, it has been designed with retargetability in mind. This retargetability is supported through the use of specifications:

- Machine descriptions specify the syntax and semantics of machine instructions. These specifications are precise enough so that an emulator could be automatically generated out of them.

- Hot path selection methods, defined using a scripting language, can also be used to customize functionality of the virtual machine.

2.4.2 Luxdbg, varying simulation accuracy

Luxdbg is a framework and debugger for Lucent's embedded processors [PBGS99]. It is used by processor architects, designers, system developers, and software engineers. Those different categories of workers have different needs regarding accuracy and performance of simulation.

In Luxdbg, virtual processors are simulated using nested machine definitions.

- At the innermost level, the CPU is defined as a circuit machine, each circuit being a collection of signals, memory cells, and transitions. They all interact to extremely accurately simulate a CPU.
- This circuit machine is contained in something called a machine code processor — which fetches, decodes and executes binary instructions (without caring about how it was done at the electronics level).
- This machine code processor is contained in something else, called a procedural code processor — which is able to work with symbolic information (generated for example for a debugger).

It is possible to switch between the various levels of precision at runtime. The first level is dealing with datapath accuracy, the second with instruction accuracy — however we could imagine a sub-level that would deal with cycle accuracy. The last level is more of an extension to support debugging features, that we can put in the high-level emulation category.

Chapter 3

Discussion

3.1 Designing a virtual machine

Now, if we want to code a simulator by ourselves, where shall we start? And what challenges are we going to face? After the initial statement of what the simulator should be and what it should achieve, there are a few steps to be careful about.

3.1.1 Preliminary questions

The first thing is to define what we need, and this requires to pass through the (non-exhaustive) list of questions below:

- What are we going to simulate (CPU, peripherals...)?
- What kind of users are going to use our virtual machine?
- On what kinds of computers is our virtual machine going to run (architectures, operating systems)?
- Is portability an important aspect of the project?
- What functionalities are we going to provide (debugging, step by step execution...)?
- Do the users need to customize the simulation themselves, for example by adding I/O (Input/Output) plugins or dynamically selecting accuracy?
- What level of accuracy does our project require?

- Is performance an important aspect of the project?
- What prior art exists to help us in the realization?
- Does any specific challenge appear after those thoughts?

3.1.2 Gathering documentation

The second stage of the process is to gather documentation about the target machine. It is important to be familiar with the hardware we simulate, and specially the CPU instruction encoding and the inner workings of the I/O subsystem. Depending on what we want to achieve, a closer look at the CPU hardware and memory timings might be worthwhile[1].

Depending on the level of accuracy and performance necessary, one might also have to dive into the host machine(s) documentation(s). For example, if you decide to code a dynamic binary translator, you will need to know the host machine assembly syntax. It gets even worse if you plan to program your binary translator on several different host machines. It is also important to know how system calls and interrupts are handled by the host operating system(s) on which you are working — we assume here you plan on developing an user-space simulator, and not something like VMWare.

Still at the operating system level, one also has to determine how do procedures and functions calls work. This is important in order to generate semantically correct binary translated code as well as for optimizing the target programs. Knowledge about memory management (stack and heap handling, sharing memory between processes), exception handling, and system calls will be a plus.

And if you want to program some special I/O devices, for example for a gaming console, you also need to dive into those graphic libraries known as Open GL, DirectX or SDL. And you need some knowledge about signal generation and transformation (for sound devices).

[1]Documentation is not always available, specially when one wants to simulate an old system. In that case, a process of trial and error based on reverse-engineering of binaries for that system will be necessary (but this is out of the scope of this paper).

All this knowledge covers almost all the field of computer science, and all the field of computer architecture, as a matter of fact. But this is understandable, since what you are trying to do is to create a (virtual) computer. Gathering and learning is probably the part of the work that takes most of the time, and the most rewarding in terms of experience gained.

3.1.3 Designing intermediate representations

When doing binary translation (especially when creating a translator supposed to run on several host architectures) portability is an important part of the work. It is then imperative to start planning it far ahead in the project timeline.

As we saw in the previous chapter, a lot of research was made on the CPU simulation and representation of its opcodes. From intermediate representations to high-level languages representations like C, there are a lot of possibilities. For a dynamic binary translator however, going too "high-level" might not be worth the effort, and a more straightforward intermediate translation is probably better. C has the advantage of summarizing a lot of information in a single instruction (even if C is considered as almost a low-level language these days). However this advantage is also a disadvantage if the accuracy level requires that you keep an exact execution order.

We will see in the next section an example of intermediate opcode representation for an ARM7TDMI simulator. The ARM7TDMI is a RISC[2] processor with a simple design that is well-suited for simulator studies. The idea of an intermediate representation is to provide a straightforward translation of the target opcode into a shape that will easily fit the translation into the host opcodes. Since in general several host instructions are required to simulate one target instruction, the idea was to divide the target instruction into several virtual sub-instructions, each translatable in a very straightforward way.

You can think of the registers of the target CPU as a set of variables (or an array in some cases, like RISC processors, but not CISC ones), and define sub-instructions that will suit the host instruction set well.

[2]Reduced Instruction Set Computer, as opposed to CISC, Complex Instruction Set Computer.

Imagine for example you have to simulate an `add a,b,c` on a machine that is only able to add one value at a time. You will then design an intermediate representation which might look something like `add a,b ; add a,c` that is way more easily translated.

As this intermediate form must be interpreted quickly by the translator, it has to be created in an efficient form — in the example above, a C structure like the one below is at the same time simple and efficient if used correctly.

```
#define OP_PLUS   0
#define OP_MINUS 1
// ...

struct Int_Rep{
int Operator; // The operator can be defined as a set of
// integer values
int regDestination; // The destination register.
int regOperand; // The operand register.
};
```

Of course, the more complex the CPU is, the more complex the intermediate representation will be. It is then possible to define various translators, depending on the host CPU. And if you do not want to dive into the host CPU instruction sets, then you might use a C-like representation that can be compiled and assembled by a compiler you may link with your simulator.

3.2 A semi-practical example

As part of this project, I began to develop a dynamic binary translator for an ARM CPU, for which I knew the architecture quite well: the ARM7TDMI. This CPU is used in a lot of embedded devices, cell phones, handheld organizers and something I used to program for as an amateur: the Game Boy Advance[3]. The idea was to create a customizable simulator (with the initial goal of simulating this device) that could run on lower-end computers by using the technique known as dynamic binary translation, that we studied in the previous chapter, while keeping a good level of accuracy.

[3]Game Boy Advance is a trademark of the Nintendo Corporation.

ARM provides a lot of technical documentation on its CPUs and bus systems, which will prove very useful through the project [Adv95, Adv99].

3.2.1 Answering the questions

The first thing, then, was to answer the list of questions stated earlier:

What are we going to simulate?

The most interesting part of the project is the CPU simulation. The ARM7TDMI is the simplest CPU of the ARM7 family, consisting of only an integer core which doesn't provide any division (nor floating-point arithmetics). The CPU has a 3-stage pipeline to optimize execution speed, and we plan to simulate this pipeline. This CPU also has an almost unique feature: it is able to run with two different instruction sets, and this is also to be simulated.

We also plan to simulate the memory system of a board designed for this CPU. There is no advanced memory management provided with this core (since there is no memory management unit available) but there is a simple memory protection scheme; and memory chipsets might have varying latencies (depending on their hardware and the way data is fetched from them) and varying data transfer sizes (byte, half-word, word).

And last, we plan to provide an interface for simulated peripheral devices. The ARM7 CPU has no input-output instructions, so the peripherals are connected on the main data and address bus and are memory-based. We plan to simulate synchronous and asynchronous peripherals, as well as DMA (Direct Memory Access).

Since this simulator is going to be an hardware simulator, no operating system calls are going to be simulated.

What kind of users are going to use our virtual machine?

This simulator is intended to be a tool used by embedded software developers and programmers, as well as testers.

On what kinds of computers is our virtual machine going to run?

As the simulator is intended to be used by developers, it has to run in environments used by developers.

Since it is easier to create a dynamic binary translator for one kind of host processor, the choice of the hardware machine was quickly narrowed down to the Intel x86 family of computers, which currently are the most widespread processors on the desktop computer market. Including PowerPC architecture would be interesting too, but it is a lot more work.

Regarding the host operating system, a wise choice would have been to target the Microsoft Windows family of systems. However, since this is a research simulator, since our funds are limited, since the target audience is technical, and since we need a complete documentation of the host operating system and its inner workings (specially regarding the assembly language system calls and such), a Posix-compliant system was chosen — Linux. This might also ease a potential port to another Unix system.

Is portability an important aspect of the project?

We partly answered this question at the previous stage — portability is not a main objective, and is almost impossible to obtain with a dynamic binary translator. However, it is possible to approach it by taking care in designing the architecture and making a clear separation between platform-independent code and x86-Linux code.

Instead of portability, we should here talk about reusability — this is always good to achieve.

What functionalities are we going to provide?

We are going to provide accurate and mostly real-time simulation of ARM7 boards, with several features: on-screen disassembly of the target code, memory dumping, basic breakpoints, step-by-step execution, and dumping of the state of the target machine.

Do the users need to customize the simulation themselves?

Users will not be able to change the simulation accuracy by themselves. On the other hand, the idea of a plugin system for custom I/O devices is interesting. The goal would be to be able to completely customize the "board" while the simulator is not running, for example by allocating virtual memory chipsets at locations in the target RAM space. Peripherals would be defined by configuration files (telling what kind of memory is used, where it is located, and so on).

What level of accuracy does our project require?

We are definitely aiming for the "Cycle accuracy" level, which includes target CPU pipeline simulation and memory timings simulation.

Is performance an important aspect of the project?

Performance is an important aspect — the accuracy level needs more computing power than ordinary emulators usually do, and we have to balance this by optimizing the simulation itself.

What prior art exists to help us in the realization?

ARM sells its own simulator, for debugging and developing ARM programs. It is part of their development toolchain. The ARMulator, as it is named, can operate at various levels of accuracy, from timing accuracy to instruction accuracy.

It was however not possible to work with this simulator for the project.

Foreseen challenges

As a matter of fact, a lot of issues arise when we start to think about the design of such a virtual machine. There is the challenge of extensibility, with this "I/O plugin system" that came to our minds earlier; there is also the challenge of performance; and so on. How are we going to handle CPU simulation in real-time? And what about peripheral management? The next sections are going to present some insights on those issues.

3.2.2 Real-time simulation

It seems interesting to start with this specific issue. As a matter of fact, we are trying to simulate a device in "real-time" on a personal computer using a multi-task operating system which is not a real-time system.

Depending on the host computer and its workload, two problems are to be discussed:

- Assuming the host system is powerful enough to simulate the hardware at full speed, how will we synchronize it?

- If the host system is not powerful enough to simulate the target hardware at full-speed, the problem is even more critical: should the emulation be "proportional"; meaning running synchronized and as fast as possible? Maybe some peripherals that will take a lot of CPU power (especially graphics card emulation) can be "skipped" every once in a while to retain a constant speed? Maybe some other peripherals need to be refreshed at the very same rate, whatever happens (sound)?

The host system is powerful enough

The idea here is to use a system timer (which is not a real time timer, but a good approximation) to "halt" the simulation accordingly, and start it again at the next tick. The problem is how to define this "tick" — well, it is the shortest time interval during which no peripheral interrupt shall occur. This yields the problem of peripheral simulation, that is to be treated in the next section.

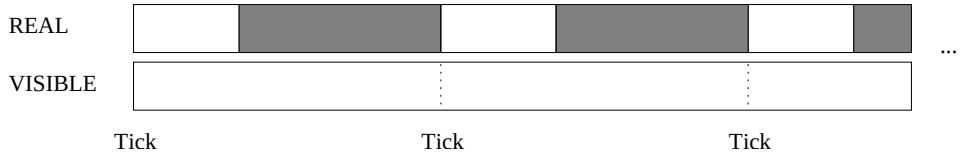

Figure 3.1: **The host is powerful enough: simulation has some spare time.**

The host system is not powerful enough

In this case, we have less solutions. Skipping some peripheral updates (a technique known as "frame dropping" in video chipset simulation) might save some CPU time. If it is not enough, we have no choice but to scale down the simulation speed, since we need to keep the cycle accuracy.

We do not use the host timer anymore, but instead we refresh peripherals when the "tick" is finished on the virtual CPU.

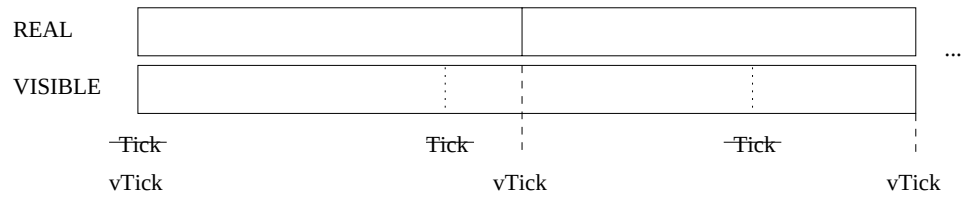

Figure 3.2: **The host is not powerful enough: we slow everything down.**

3.2.3 Peripheral simulation

As we saw in the previous subsection, the real-time simulation depends on one thing, and that thing is the "tick" made by peripherals when they need to be refreshed.

Peripheral devices can be divided into two main categories: synchronous devices and asynchronous devices.

Synchronous devices

The first devices are always refreshed at a "constant" rate[4]. If a computer was based only on such devices, it would be easy to calculate this "tick", the virtual moment of simulation during which no peripheral event (i.e. interrupt) can occur — it is the highest frequency of peripheral refresh rate.

For example, if you have a video board producing 60 pictures per second, and a sound chipset creating a wave at 22050hz (22050 times per second), then the "tick" duration is the highest frequency — here, 22050hz or 1/22050 of a second. During those 1/22050 of a second, the virtual CPU can run as fast as it can in order to simulate how many instructions it would need to run for 1/22050 of a second on the target system. Then the simulation stops for what is left of this "tick", and then we start a new "tick", the video board and the sound chipset are simulated. Then the virtual CPU starts again, and so on.

This is what is called synchronous sampling.

[4]This "constant" rate can of course vary: think about a video board able to draw things on screen at 60hz *or* 70hz. However, it does not vary randomly.

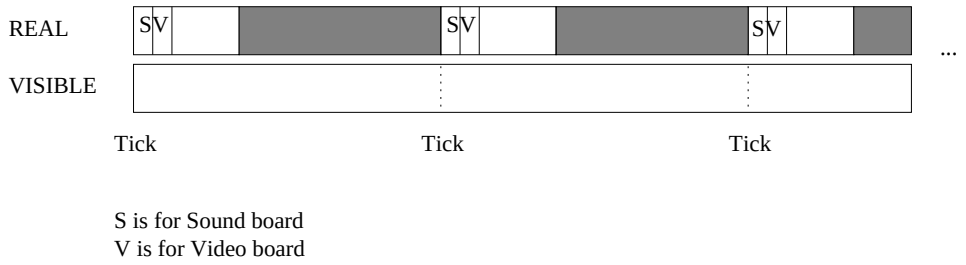

Figure 3.3: **Driver routine calls during simulation.**

Asynchronous devices

Not all peripheral devices are synchronous — most of them can yield an interrupt at any time, and those interrupts are supposed to be serviced without delay by the CPU. This basically means checking for every peripheral after each simulated cycle, and it would take too much time as we are planning to build an optimized simulator.

How should we deal with that problem?

The ARM7TDMI-S solves partly this issue by providing a synced interrupt (i.e. interrupts are fired always on rising edge of the system clock). Moreover, a board based on that CPU has only one system clock, which means that all the peripherals "refresh" at a proportional frequency of that clock.

We can then figure out a solution, which is transforming our asynchronous peripherals into synchronous peripherals. This doesn't exactly respect our goal of accuracy, but it has the advantage of speed — and the difference between real asynchronous and simulation is minor, as we will see.

Each asynchronous peripheral has to be studied in detail. If we take the example of the keyboard, it is quite easy to see where we are going: a keyboard is an asynchronous device, since the user can press a key anytime. However, your average human being never presses more than, say, 20 keys per second; which means that there is basically no difference between a virtual keyboard refreshed 20 times per second and a virtual keyboard refreshed at every virtual cycle.

Some peripherals also have varying access rates, implying varying interrupt frequencies. For example a disk drive, where accesses are dependent on the

current disk rotation. However, the interrupt frequency can be considered, in most of the cases, as a multiple of a "base frequency" for that peripheral (in a disk driver, the shortest access time of the drive).

The next question is then how to integrate the host operating system functionalities in the drivers for the virtual machine.

Integrating host and target machines

A peripheral driver is then made of two sections: one function that is called by the simulator whenever the peripheral is supposed to be "refreshed" by the virtual machine, and another function that is called from the host system when it gets incoming signals and events so that the virtual machine can be updated.

Basically, the first function is mainly used for output devices, and the second function is mainly used for input devices.

We have the simple solution of calling both functions from the main execution loop, but another idea might be more interesting.

First, this solution of calling works fine only when the driver routines are very short (in cycles). What happens when we have to refresh a "big" peripheral like a video board, which has to recalculate the output of the whole screen?

Moreover, signals and events can be signaled asynchronously to the virtual machine by the host operating system. We can benefit of this only if the main simulation loop and the "peripheral simulation loop" are somehow distinct.

We start to see here how to organize our virtual machine. The idea of using threads comes to our mind, with a high priority thread for the main simulation, and a lower priority thread for the peripheral simulation. Synchronization is then handled by message passing using the functions above to a special "simulator function" which runs apart from the main simulation (or maybe one thread for every peripheral?). And the same thread handles incoming events.

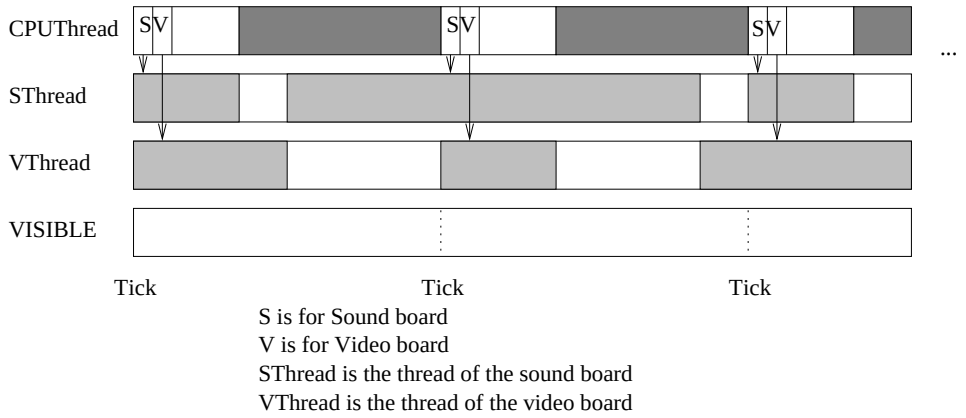

Figure 3.4: **Two threads run in the time free of CPU calculations.**

The threads have to be synchronized, however, to the virtual CPU refresh frequency, but this can be solved by storing information about the "ticks" above...

Memory-driven output and the problem of DMA.

DMA is a fancy acronym standing for "Direct Memory Access". It has nothing to do with Microsoft DirectX, but it is a simple hardware mechanism to hasten hardware memory accesses and copy operations (in the graphics card world, we talk about *blitting*).

DMA works by "halting" temporarily CPU execution, performing directly the transfer from one memory area to another one, and restarting the execution again.

To start a DMA transfer, the programmer writes certain values at certain locations in memory which correspond to the DMA control areas — and the transfer starts at the very moment where memory is updated.

DMA is probably not the only peripheral working this way, but it is the most common. What can we do when we need an *immediate* response to an I/O action? Here, we cannot wait for our "tick" to occur, do the transfer and start again — all the instructions after the transfer instruction might need to access the updated memory area.

A solution would be to effectively halt the CPU simulation when one of those "special addresses" is written — since the peripheral list is known at runtime, those addresses are also known. Then we can perform our transfer, update the "ticks", and start again.

However, it might not work so easily.

The DMA halts the CPU while it finishes its memory transfer, but the DMA might also be stopped by another interrupt, so this has to be simulated as well. An interesting idea would be to code the DMA as a small piece of ARM code and to make it translated by the VM, but maybe it is possible to do otherwise.

This special peripheral device has to calculate itself how long it will take for the transfer, do part of it, and call other peripherals if needed. We do not use a thread here, but the function started from the main CPU code.

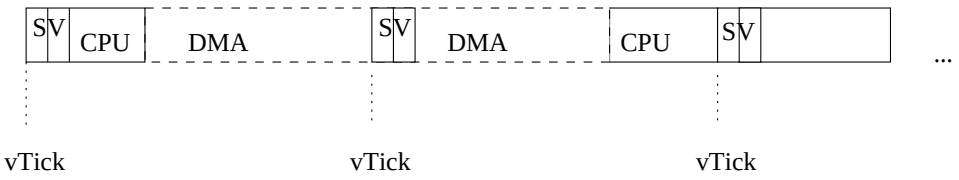

Figure 3.5: **DMA simply replaces CPU simulation for a while.**

General memory management

Memory is not exactly a peripheral, since it doesn't produce any output and doesn't get any input. We just have to know that the ARM7TDMI bus has been made to enable faster accesses of sequential memory locations, and that for most memory chipsets the latency for sequential access and non-access, expressed in number of cycles, is different.

3.2.4 CPU simulation

CPU simulation is the realm of the binary translator. Programming a binary translator is not an easy task by itself, and regarding the specifications above we have some special issues to take into account.

Instruction translation

A single target instruction can be translated by four main blocks of host instructions:

- Load block (load virtual register values)
- Instruction execution block (perform calculation)
- Store block (store virtual register values, increase virtual Program Counter)
- Timer block (decrease CPU timer, jump to peripheral process code if necessary)

However, since we know that we have those "ticks" during which no interrupt can take place (with the exception of some special system traps that we will cover later), we can imagine translating blocks of instructions that correspond to that "tick".

The size of such a block would be determined by the number of cycles it takes to execute it. Since we cannot know how many times a loop may execute, we have to define several conditions for block creation:

- First, a block may start anywhere in the target memory space,
- A block may be ended when: either the cycle counter exceeds the "tick" duration, or when an instruction that interrupts the normal program flow is found (i.e. a branch, a conditional jump, a software interrupt call),
- Special code is appended to the translation of an instruction writing to a memory location in order to check for DMA — but this does not end the block.

When the translator is done with a block, the block can be executed. The instruction breaking the program flow either "jumps" to the translation routine, or to the corresponding block.

The pipeline problem

As we saw in the previous section, block size is determined by the number of cycles taken by the target instructions to execute. Moreover, to accurately simulate the CPU, we have to precisely compute the number of cycles that it takes. Everything would be simple if the CPU had no pipeline, i.e. if it was possible to know in advance how many cycles it will take to execute a given instruction.

However, the ARM7TDMI provides a 3-stage pipeline, organized around the three operations of fetch-decode-execute.

The number of cycles for instruction fetching can be determined when we know the memory location of the current instruction — this depends on the kind of memory (with latencies and so on). Since a block is only made of a non-interrupted sequence of instructions, we can assume that this value will be the same for each instruction of the block (and that it corresponds to the "sequential" latency of the memory).

The number of cycles for instruction decoding is constant.

The number of cycles for instruction execution will vary depending on the type of instruction, and whether it needs to access memory (data fetches and stores). It is not always possible to compute this delay at translation-time, since data fetches and stores can depend on a varying memory location (defined by a register). In that case, we have to leave in the translated code a block of instructions to compute this during execution — and for block creation purposes, we shall assume it takes the longest delay possible.

The virtual pipeline can be implemented by storing the fetch / decode / execute cycle timings of the two previous instructions, and for each instruction computing the number of cycles it passes in each stage.

On the diagram below, i1 takes 11 cycles, i2 then takes only 4 more cycles, and i3 will take 8 more cycles. The total cycle count of the block is 23.

	Fetch	Decode	Execute
i1	4	2	5
i2	2	2	4
i3	4	2	8

Figure 3.6: **Three instructions and their default cycle count.**

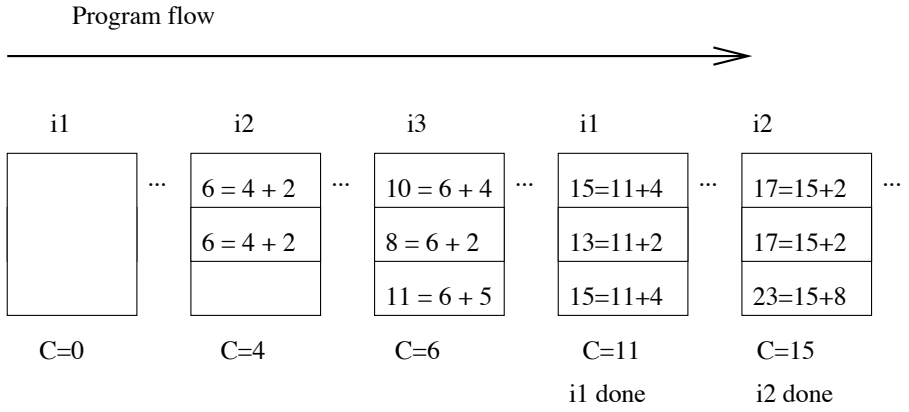

Figure 3.7: **The pipeline algorithm.**

Interrupts and traps

When an interrupt occurs on the virtual hardware or when an exception arises, the virtual program execution must be stopped and another path taken — namely the IO routine.

The ARM7TDMI CPU has seven possible exceptions. Two of them, FIQ and IRQ, are here to handle interrupts. As we said the interrupts were sampled, this is easy to deal with. When the peripherals routines have run, if one generated an interrupt, we have to perform the context-switching operations, and restart the vCPU translation/execution at the appropriate location.

Another exception is raised by the software executed: the system call. The equivalent of INT in the PC world is called SWI (for SoftWare Interrupt). This instruction is not to be simulated by software, instead it will be an "end-of-block", calling the switching context procedure.

Another exception is raised when the software tries to execute an unknown instruction. This one is quite easy to catch: when the translator is unable to translate an instruction, well, it will "just" halt. Then the sequence of instructions translated is executed up to the halt point, and this point jumps to the VM exception handler.

The "reset" exception is trivial to manage — jump back to vector contained at address 0x0000000 of the virtual machine whenever a "reset" signal is received from the host system.

The two last possible exceptions are more tricky.

The first one occurs when the program tries to read, or write, to a location either out of memory bounds, or undefined in memory. This check cannot be performed at translation time, since memory fetches can be vectorized (pointers of pointers of...). Hence the test should be done at runtime in assembly code. If a memory location cannot be accessed, the program may jump to the VM exception handler.

The second one occurs when the program tries to execute an instruction which is at an address out of memory bounds or unreadable. This can be detected in most cases by the translator, and is solved in the same way as

the “unknown instruction” exception. The ARM CPU is also subject to misalignment.

3.2.5 An intermediate representation

We defined an intermediate representation for the ARM7TDMI instruction sets. This CPU works with two distinct sets, however the Thumb instruction set is a subset (a compressed form) of the ARM instruction set. It seems easier to handle both sets by transforming them into the same intermediate representation.

We decided to program the simulator using C++, and the concept of class inheritance can be used here.

We defined an abstract class called “Instruction”, which contains methods that can be used to access commonly to ARM and Thumb instructions. Then we define two derived classes, one for each instruction set. The translator only uses the abstract functions to access the fields.

Below is the description of the abstract class Instruction and its methods.

```
Instruction: Base class for an instruction.

This class represents an instruction, and those functions it
defines must be redefined in the derived classes (i.e.
ARM_Instruction as well as Thumb_Instruction). The goal is
to use this base class as a common access handler, once
an instruction has been translated to the internal format we
use.

virtual unsigned int getCCode() = 0
Returns condition code.

virtual unsigned int getType() = 0
Returns instruction type.

virtual unsigned int getOpcode() = 0
Returns instruction operation code.
```

```
virtual unsigned int getDestReg() = 0
Returns destination register.

virtual unsigned int getBaseReg() = 0
Returns base register.

virtual unsigned int getSourceReg() = 0
Returns source register.

virtual unsigned int getFirstOpReg() = 0
Returns first operand register.

virtual unsigned int getSecondOp() = 0
Returns second operand.

virtual unsigned int getThirdOp() = 0
Returns the shift register.

virtual unsigned int getShiftType() = 0
Returns the shift type.

virtual signed int getOffset() = 0
Returns offset.

virtual unsigned int getRegList() = 0
Returns register list.

virtual unsigned int dataTransferSize() = 0
Returns Data transfer size, in bytes.

virtual unsigned int getFieldMask() = 0
Returns Field mask, for MSR instruction.

virtual bool isUsingRegister() = 0
Returns true if the second operand is a register.

virtual bool isTransferSigned() = 0
Returns whether the data transfer is supposed to
be signed or not.
```

```
virtual bool isUsingImmShift() = 0
Returns true if the instruction uses an immediate shift.

virtual bool isUsingRegShift() = 0
Returns true if the instruction uses a register shift.

virtual bool performWriteBack() = 0
True if the register transfer must perform a writeback
on the register involved.

virtual bool preIndex() = 0
True if pre-indexed, false if post-indexed.

virtual bool increment() = 0
True if the base register is to be incremented, false if
it is to be decremented.

virtual bool modifiesCPSR() = 0
True if the operation modifies the CPSR.

virtual bool withLink() = 0
True if the branch is using link mode, false elsewise.
```

Each derived class has its special way of transforming instructions into an internal representation that makes those functions trivial to implement.

Conclusion

In the last two chapters we covered a lot of information on the field of virtual machines, and we progressively focused on simulation of computer hardware, discussing optimization through dynamic binary translation.

We saw that binary translation alone is not enough to make a fast and accurate simulator: a carefully designed architecture is also important. And since we simulate the hardware memory management, we also need an optimized memory translation unit — and of course, for each peripheral that the target machine is bundled with we need an appropriate driver.

We finally focused on real-time simulation, and we saw the challenges awaiting us there. We discussed a possibility of using threads to separate peripheral simulation and core simulation. However other solutions might also work, each having its own drawbacks and benefits — using separate processes communicating over a network, for example.

Virtual machines are a fundamental concept of computer science and are here to stay. As the hardware gets more and more complicated, simulators require more and more engineering to perform optimally, that means the field might attract more people in the years to come.

Bibliography

[Adv95] Advanced Risc Machines Ltd. *ARM7TDMI Data Sheet*, 1995.

[Adv99] Advanced Risc Machines Ltd. *ARM7TDMI-S Technical Reference Manual*, 1999.

[AM00] Lars Albertsson and Peter S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 191–198, August 2000.

[CLU02] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout — a retargetable dynamic binary translation framework. Technical Report TR-2002-106, Sun Microsystems laboratories, 2002.

[Fur96] Steve Furber. *The* ARM *System Architecture*. Addison Wesley Longman, 1996.

[Gel97] David Gelernter. Truth, beauty, and the virtual machine. *Discover*, 18(9), September 1997.

[Gou01] K John Gough. Stacking them up: A comparison of virtual machines. In *Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 55–61. Queensland University of Technology, Brisbane, Australia, 2001.

[HH97] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[KMM99] Eric Kohlbrenner, Dana Morris, and Brett Morris. Core of IT: The Virtual Machine: History. Web page, may 1999. Available at `http://cne.gmu.edu/itcore/virtualmachine/history.htm`.

[Mag93] Peter S. Magnusson. Partial translation. Technical Report 93:5, SICS, October 1993.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[MW95] Peter Magnusson and Bengt Werner. Efficient memory simulation in SIMICS. In *Proceedings of the 28th Annual Simulation Symposium*, pages 62–73, Santa Barbara, California, USA, April 1995. IEEE Computer Society, Washington DC, USA.

[PBGS99] Dale Parson, Paul Beatty, John Glossner, and Bryan Schlieder. A framework for simulating heterogeneous virtual processors. In *Proceedings of the Thirty-Second Annual Simulation Symposium*, pages 58–70. IEEE Computer Society, Washington DC, USA, 1999.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7), July 1974.

[Tij00] Arjan Tijms. Binary translation: Classification of emulators. *Leiden Institute for Advanced Computer Science*, 2000.

[UC01] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. *ACM SIGARCH Computer Architecture News*, 29(1):55–65, 2001.

[Var97] Melinda Varian. VM and the VM community: Past, present, and future. SHARE 89, Office of Computing and Information Technology, Princeton University, aug 1997. Available at `http://pucc.princeton.edu/~ melinda/`.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 68–79, Philadelphia, Pennsylvania, United States, 1996. ACM Press, New York, NY, USA.