A Choices Hypervisor on the ARM Architecture

Rishi Bhardwaj, Phillip Reames, Russell Greenspan,
Vijay Srinivas Nori, Ercan Ucan
University of Illinois
April, 2006

ABSTRACT

We describe our experiments building a hypervisor using Choices, an object-oriented operating system, on the ARM architecture. Our approach uses VMX-style hardware support to ensure that execution of any sensitive instruction transfers control to the hypervisor. We provide such an architecture by making modifications to QEMU, an open source ARM emulator. Thus our work is divided into three parts. We first surveyed the ARM7 architecture and identified the sensitive instructions. Next we altered QEMU so that it traps to our hypervisor whenever a sensitive instruction is executed in a low privileged user mode. Finally we added a hyper call handler to Choices to validate and emulate the sensitive instructions QEMU traps on, providing a virtual machine environment to guest operating systems.

1. INTRODUCTION

1.1 Motivation

Advances in computer architecture, semiconductor technology, and system software have enabled computer users access to significantly more computing resources than they need to run their applications. As a result, performance is no longer the overriding concern when designing computer hardware and software, and instead new challenges have emerged such as “How do we efficiently utilize fast hardware and abundant computing resources to securely run our applications?” For example, it should be possible for a user to run an untrusted application without potentially crashing the operating system and causing damage to hard disk data if the application were to perform malicious operations.

A related issue is of reliability, which can be provided by running multiple instances of machines (mirrors). Each machine has the same functionality such that crashing one machine does not bring the whole system down or corrupt user data. Additionally, a computer user might like to run multiple operating systems on a single machine to use two applications which run only on their respective OSes. Furthermore, with the advent of multi-core processors, the ability to run multiple concurrent operating systems will become much easier, and systems that take advantage of this feature will be highly desired.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1.2 Overview

The technique of virtualization [11], proposed in the 1960’s by IBM, provides an elegant solution to the problems presented above. Virtualization provides us the ability to have multiple operating systems run in a fast, secure and seamless manner on a single machine. The layer of underlying software, built to achieve virtualization and protection, gives each operating system the illusion of running on bare hardware and solves the problems presented above.

In this paper we describe the hypervisor we have built for the ARM7 architecture. We have chosen the ARM7 architecture because we did not find any open source hypervisor for the ARM architecture in our literature survey, and because the stable version of QEMU that we are using emulates the ARM7 architecture. Our hypervisor is based on the Choices [2] operating system, which is an object oriented operating system running on ARM developed at the University of Illinois. We have chosen Choices to investigate the process of building an object-oriented hypervisor.

For performance reasons we would like to have as much of the guest OS code executed directly on real hardware as possible, with minimum intervention by the hypervisor. Every intervention by the hypervisor means more executed instructions, which directly translates to more overhead. However, for virtualization to take place, certain sensitive instructions cannot be allowed to be executed on the hardware directly by the guest OS [1]. Thus the hypervisor needs to support the emulation of these sensitive instructions and must provide a mechanism which makes sure that control is transferred from the guest OS to the hypervisor whenever a sensitive instruction needs to be executed by the guest OS.

For our implementation, we identified three different approaches for control transfer from the guest OS into the hypervisor, namely software pure virtualization, paravirtualization, and hardware supported pure virtualization (these methods are described in detail in

Section 2). The third approach, hardware supported virtualization, has the advantages of not having to modify the guest OS binary, as we do in paravirtualization, and remaining less complex in implementation than software pure virtualization. For these reasons we have adopted this approach in designing our system.

2. RELATED WORK

Existing techniques of Hypervisor implementation include pure virtualization, paravirtualization and finally the VMX-style hardware supported approach. Popular systems employing these approaches to virtualization are discussed below.

VMware [10], employing the pure virtualization approach, has developed both Server (ESX) and Workstation virtualization components; both exploit the benefits and drawbacks of full virtualization and require no guest OS modification. In ESX, a Virtual Machine Monitor (VMM) intercepts all I/O calls and forwards them to the VMkernel, which handles communication with the physical hardware [6]. ESX employs direct execution (running the virtual machine directly on the underlying processor) and shadow page tables to improve performance. In VMWorkstation, a driver (VMDriver) is loaded in the OS on which Workstation runs; in each OS application, a special process (VMApp) runs and establishes communication to the hypervisor via this driver [7]. The software-based pure virtualization approach has the advantage of not having to modify the guest operating system at all. However, it has the disadvantage of incurring a performance penalty since it has to scan and validate binary instructions and insert traps prior to execution of these instructions. It is for this reason that we have decided not to adopt this approach.

Xen-style paravirtualization [4] requires slight modifications to host OS instructions. Xen uses the upper 64MB of each host OS's address space to avoid TLB flushes on address space switches, and also requires modifications to the host OS page fault handler. Unlike pure virtualization [5], paravirtualization does not require expensive dynamic machine code rewrites for sensitive instructions and permits the host OS any interaction with the physical hardware, which at times can have performance benefits. Xen's approach, though efficient, requires the guest OS to be modified to be able to run on the hypervisor. Therefore, we have chosen not to pursue this approach in our project since it is unlikely in general to assume access to the guest OS's source code.

In Intel's VMX approach [8], hardware support is added to facilitate virtualization. In general, the hypervisor runs in VMX root operation mode and guest OSes run in VMX non-root operation mode. Processor behavior in VMX root operation is very much as it is outside VMX operation. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited. Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain sensitive instructions (including the new VMCALL instruction) and events cause VM traps to the hypervisor. Because these VM traps replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the hypervisor to retain control of processor resources. We adopt this approach in our project as it is a tradeoff of the performance of the paravirtualization approach with the benefit of not requiring changes to the guest OS as in pure virtualization approach.

QEMU is an ARM7 emulator running on the *x*86 architecture [9]. In our approach, modeled on the VMX approach, we modify QEMU to trap on sensitive instructions when executing in non-privileged mode. This is a simulation of processor-level support for virtualization. Within the scope of the literature we surveyed, no such open source hardware-based virtualization has been attempted on the ARM processor.

3. SYSTEM DESIGN

3.1 System Design Overview

In our virtual machine system we have the guest operating system execute in ARM user mode (less privileged mode), and all privileged instructions are considered "illegal" by the original ARM instruction set architecture. The modifications we have made to QEMU allow the privileged (and sensitive) instructions to trap to the Choices handler. Therefore, the critical design requirements for this system are that the guest OS must not be aware that it is running in a less privileged mode, i.e. its execution model must be preserved with the hypervisor acting as the bookkeeper for VM state, and that sensitive instructions issued by the guest OS which potentially may damage or detect system state beyond access privileges must be emulated and not executed directly on the hardware.

Achieving these goals required the completion of the following milestones:

- Studying the complete ARM7 architecture as emulated by QEMU and identifying the sensitive instructions in it.
- Designing and implementing the trap mechanism from QEMU to the Choices hypervisor by modification of the QEMU source code (for each of the sensitive instructions identified in phase 1). This allows control to be transferred to the Choices hypervisor after a sensitive instruction is executed. Note that if the CPU is running in hypervisor-privilege mode, all instructions are executed unaltered.
- Implementing the trap handler to validate and emulate the sensitive instructions by modification of the Choices source code. Figure 1 depicts a block diagram of the main system components in relation to each other.

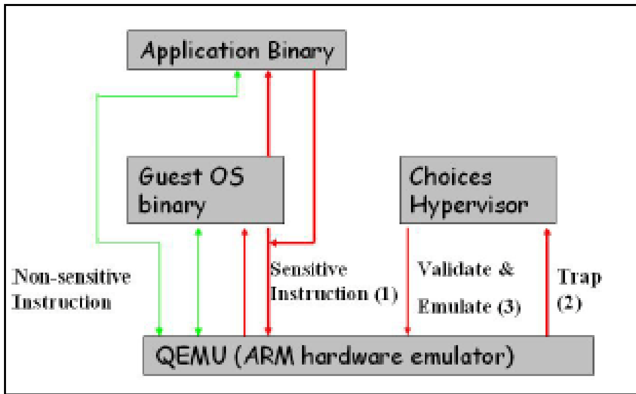

Figure 1: System Block Diagram showing Choices Hypervisor running on QEMU

As shown, a sensitive instruction is trapped to the hypervisor by QEMU for validation followed by emulation. All other instructions are executed and results made visible to the executing guest operating system (or application binary) immediately.

3.1.1 Identifying Sensitive Instructions

A classification of sensitive ARM instructions is described in Section 4.1.

3.1.2 Trapping Mechanism in QEMU

QEMU translates each ARM instruction into a set of corresponding *x*86 instructions in order to emulate ARM instructions on an *x*86 system. The translation, done at runtime, is batched with a "block" of ARM instructions translated together in one go. A branch instruction or a page boundary or a mode changing instruction (writing into `CPSR`) marks the end of an ARM instruction block. The translation process is stopped upon encountering a block boundary after which the translated *x*86 code is executed on the host machine. After execution the translated block is cached so that the dynamic translation does not have to be done every time the same ARM instructions are to be run. The whole process is then repeated for the next block of ARM code.

The important point to note is that the block of ARM instructions being translated will be executed either in ARM privilege mode or the ARM user mode. When translating a sensitive instruction in QEMU, we need to check if it will be executed in user mode or privilege mode. When translating a sensitive instruction in privilege mode, we do not modify the translation sequence and have the original QEMU code take care of the translation as before. But when the sensitive instruction needs to be executed in user mode, we translate the instruction like a SWI. Thus sensitive instructions in user mode are treated as SWI.

3.1.3 Designing the Choices Hypervisor

The third part of our work was to build a hyper call handler in Choices. This handler is invoked by the modified QEMU emulator hardware whenever a sensitive instruction is to be executed in a virtual machine. The handler must figure out what instruction was tried on in the virtual machine environment, validate and emulate it. By emulation, we mean that the handler must change the CPU context in a manner such that when the handler returns the control, the guest operating system is not able to tell whether the instruction is being executed in the virtual machine or on actual hardware. For example, suppose the guest operating system running in virtual machine system mode tries to copy (read) the contents of the current program status register (`CPSR`, containing information about the mode bits) into register 1. Then the hyper call handler must copy a value in register 1 which the guest operating system expects to see (i.e. mode bits copied into register 1 must indicate to be in the system mode and not in user mode).

Additionally, the hyper call handler must validate the instruction which need to be emulated. For example, we can not have a privileged instruction emulated when the virtual machine is in user mode. In such a case the guest operating system's *illegal instruction* handler must be called.

As described above, we must have a mechanism to validate and emulate sensitive instructions whenever the hypervisor receives a hyper call. Since Choices is an object oriented operating system, it allows for a very modular design of the hypervisor. To accommodate our changes, we added a new class to Choices (`ArmVMContext`) which represents the context of the CPU for a virtual machine, i.e. the context which the guest operating system thinks the CPU is in. Thus for each virtual machine the hypervisor maintains a shadow of the CPU context for that machine. Emulation of a sensitive instruction changes the `ArmVMContext` object for the virtual machine as it would for the real hardware. The guest operating system in short would never be able to tell the difference between running on the virtual machine or the real machine.

Upon receiving a hyper call from QEMU, the hyper call handler checks the opcode and parameters of the sensitive instruction for violation of privileges. For example, an instruction executing in virtual machine user mode should not attempt to modify the `CPSR` mode bits (or for that matter try any privileged instruction). After validating the instruction the hypervisor changes the context of real hardware CPU such that when control returns back to the virtual machine process the execution may continue as if the instruction was executed directly on the machine.

Some of the other functionalities that a hypervisor must provide are I/O forwarding and virtual memory support. A hypervisor must efficiently arbitrate between the I/O devices and guest operating systems in a seamless manner. Also it must implement a mechanism by which a guest

operating system's memory management works correctly in a virtual machine environment without, conflicting with the memory management of the hypervisor itself. In the current system, we did not implement the above two functionalities. We think that I/O forwarding can be added to the system without much difficulty; memory management, though, is an altogether different ball game. We provide more details about the difficulties in emulating the ARM memory management system in Section 5.

4. CONTRIBUTIONS

4.1 Sensitive Instruction Study on ARM

An operating system runs in a kernel-mode, which allows higher privileges compared to the user applications. This higher privileged mode allows the OS access to certain privileged machine instructions and resources not available to user-mode applications. This creates a problem in building a hypervisor, since the OS will now run in a semi-privileged hardware mode, with traps on sensitive instructions going to the hypervisor. It is the responsibility of the hypervisor to keep track of the OS's state, allow the safe operations to execute, and block the illegal sensitive operations from executing. Analysis of virtualization on the Intel Pentium architecture [1] shows that the instructions following instructions must be considered as sensitive:

- An instruction represented by the same bit pattern or format executes differently in modes of different privileges. This means that on execution, the instruction must trap to the hypervisor which has the mode information to execute the instruction properly.
- A instruction whose execution requires greater privileges than the privilege in which the guest OS is being run.

Using this as a baseline, we classify the following instructions in the ARM7 architecture [12] as sensitive and present our reasons for the classification. As a result of our study we identified six categories for sensitive ARM instructions.

- `MRS` and `MSR` instructions: instructions that are related to the `CPSR` (Current Program Status Register). `MRS` (Move `PSR` to General-purpose Register) instruction moves the value of the `CPSR` or the `SPSR` of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions. `MSR` (Move to Status Register from ARM Register) instruction transfers the value of a general-purpose register or immediate constant to the `CPSR` or the `SPSR` (Saved Program Status Register).
- Common ALU (Arithmetic Logic Unit) instructions: instructions like `ADD`, `AND`, etc. that have their `S` bit set to 1 and also the `Rd` (Destination register) = 15 (`Rd` is the Program Counter.) If this is the case for these group of instructions, then the `CPSR` is updated (Value in `SPSR` is written into `CPSR`).
- Thumb Related instructions: instructions that have the capability to switch to the Thumb mode for ARM. The Thumb instruction set is re-encoded subset of the ARM instruction set and is designed to increase the performance using a 16-bit or narrower memory data bus and to allow better code density than regular ARM. There are branch instructions (e.g. `BX`, `BLX`) which can switch instruction set, so that execution continues at the branch target using the Thumb instruction set of ARM.
- Privileged Coprocessor instructions: instructions that make use of the coprocessors in the system. There are three types of coprocessor instructions: Data-processing (e.g. `CDP`), data transfer (e.g. `LDC`), register transfer (e.g. `MRC`) and all of them are considered to be sensitive.
- MMU (Memory Management Unit) related instructions: instructions that make use of the MMU are sensitive instructions as well. Load and store type of instructions are sensitive since they go through the memory management unit.
- Exceptions: instructions such as `SWI` (software interrupt). Since these instructions cause mode change, they are considered as sensitive instructions.

5. CHALLENGES IN DESIGN AND IMPLEMENTATION

The first part of the project was to read, understand and make changes to the QEMU source code to force QEMU to trap when trying to execute a sensitive instruction from low privileged mode, thus allowing our Choices Hypervisor to emulate the sensitive instruction. The whole mechanism becomes a bit confusing, where the ARM instruction are being emulated by Choices which itself is executing over QEMU ARM CPU emulator. The various system components are shown in Figure 2.

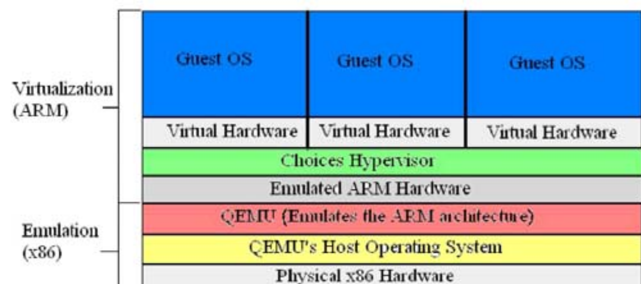

Figure 2. System Block Diagram showing Choices Hypervisor running on QEMU

5.1 Privilege Modes

A design choice we made was to decide upon the number of privilege modes we could provide in the hardware (by making changes in QEMU). Including the various interrupt modes, there are in total 7 privilege modes in the ARM

CPU. By providing a VMX-style architecture we can in total add 7 more modes (virtual machine modes), one each for the current privilege modes. Each of the new virtual machine modes would be a mirror of its corresponding current privilege mode but would trap on executing a sensitive instruction. We decided to maintain the additional virtual machine modes in software than in hardware. This decision was made to keep the system simple and easy to implement. All state information is stored and maintained by the hypervisor and not by the hardware.

5.2 Virtual Memory

A tricky implementation issue is providing memory management support for virtual machines. For virtualization to take place we cannot allow a guest OS to read/write directly from/to system page tables. The hypervisor needs to act as an interface for any access to the page tables by the guest OS. Since the page tables are stored in memory, we will have to keep tab of all memory reads/writes the guest OS make to see if the memory area accessed stores a page table or not.

The performance overhead for trapping on all memory accesses by the guest OS makes the above solution infeasible. The solution then is to have the page frame storing the page tables (for a virtual machine) be inaccessible for the guest OS (i.e. the page table entry for the logical address of the page frame storing the page tables be invalid). This will ensure that a trap to the hypervisor is generated only when attempts to access the page frame storing the page tables are made. Due to the intricacy of this implementation, we for now assume that the ARM memory management will never be activated by the binary program running in the virtual machine.

6. EXPERIMENTAL RESULTS

6.1 Motivation

Our experiments sought to demonstrate that our modified system preserves correctness and efficiency when executing sensitive instructions, that the RISC-nature of ARM provides an excellent instruction set from which to implement VMX-style virtualization primitives, and that the Choices OS is a worthwhile candidate in which to implement a hypervisor.

6.2 Procedure Overview

To test that our implementation properly traps to the hypervisor when executing sensitive instructions in low privileged user-mode, we run an ARM7 binary application that executes the supported sensitive instructions both in an unchanged QEMU environment (executing in privileged system mode) and in a virtual environment (running in low privileged user-mode) using our Choices hypervisor on the modified QEMU. We then test that our modified implementation correctly emulates the sensitive instructions by checking that the output of the binary program is the same when run in both environments, and measure the efficiency of our system by analyzing the number of instructions executed at runtime when running the test binary in each environment.

6.3 Equipment and Experimental Environment

Our environment consists of a single x86 host running our modified version of the ARM emulator QEMU, which hosts our modified version of the Choices OS with hypervisor support running in kernel-mode, which hosts guest OSes running in user-mode. We are using the CSIL-LINUX machines, running QEMU 0.8.0 on Linux 2.6.15 and compiling with gcc 3.2. Note that an attempt was made to compile and run QEMU on Windows using the Linux API emulator CYGWIN, but this effort was abandoned due to lack of CYGWIN support.

6.4 Metrics and Results

6.4.1 Correctness

We would like to have assurance that every targeted ARM7 instruction emulated by QEMU is handled correctly by our virtualization approach. Moreover, the virtualized instruction should not have any additional side effects on the VM or the real host other than it supposed to have. To study this, we tested several valid binaries in both an unmodified system and in our modified environment, and ensured that they execute their operations correctly and return the expected results.

For example, a sample ARM7 binary might include the following instructions:

```
CMP R1, R3
MSR CPSR, #0x10
```

with the following output:

```
We got a hypercall for MSR
context->CPSR before MSR = 0x60000010
context->CPSR after MSR = 0x10
```

The first non-sensitive `CMP` instruction sets the `CPSR` bits to `#0x60000010`, which is the expected comparison output stored in the `CPSR`. Then the `MSR` instruction changes the `CPSR` to `#0x10` representing a change from the VM system mode to the VM user mode. Since, this is a legal operation in system mode it is allowed by the hypervisor without any errors.

6.4.2 Completeness

As described above, with the exception of instructions pertaining to memory management, sensitive ARM7 instructions that are emulated by QEMU have been implemented in our virtualization approach. These instructions show exactly the same behavior before and after being virtualized using our approach.

6.4.3 Performance

Another key metric for evaluation is the performance of the system. Trapping into the hypervisor for the interpretation

of sensitive instructions and validation checks for these instructions provides significant overhead, but it is highly undesirable for the hypervisor to excessively slow valid operations. We did not implement the means by which to analyze the number of extra ARM instruction being executed by the Choices handler to emulate a sensitive instruction, but instead observed the number of lines of ARM instructions in the compiled binary code of our handler function.

On average, we observed 320 additional ARM instructions in the compiled handler code for emulating a sensitive instruction. This figure tells the total number of ARM instructions in the hyper call handler binary for emulating one sensitive instruction; note that this is a pessimistic bound since there are a number of branch instructions (corresponding to if statements in the C++ code) that are executed as well.

6.4.4 *Security*

One of the main motivations for a VM implementation is to keep VMs isolated from each other and the hypervisor. Therefore, a key test metric is the system's security: does the hypervisor achieve its intended goal of keeping VM accesses in their own space and allow valid VM operations only? To measure this we tested the system with several invalid binaries in different VMs that execute privileged instructions in VM user-mode or access out-of-bounds registers. Our system prevents invalid binaries from executing invalid instructions or accessing protected registers, and from each VM's perspective, there is no interfere and each will function like a separate operating system on a separate host.

For example, expanding on the example above, if we now execute an ARM binary with the following instructions:

```
CMP R1, R3
MSR CPSR, #0x10
MSR SPSR, R0
```

we get the following output:

```
We got a hypercall for MSR
context->CPSR before MSR = 0x60000010
context->CPSR after MSR = 0x10
We got a hypercall for MSR
ERROR: Tried to do SPSR = reg from VM
User_mode
```

In this case, since we executed an instruction with SPSR access in VM user mode, the hypervisor flags an error immediately and halts the execution of the erroneous code.

7. CONCLUSION

Our current design and implementation supports a simple ARM binary file (not employing memory management) running in a virtual machine environment. The virtual machine environment is supported by our Choices Hypervisor, executing in our modified QEMU, which provides emulated VMX style ARM architecture. The design and implementation of the system is such that it is can be easily extended to provide support for full fledged virtual machine environment with memory management turned on.

We hope that our work will influence future directions of virtualization research by showing the baseline requirements for a system that virtualizes ARM via a hypervisor. Acceptable performance shows that we have created a proof-of-concept for future hardware-virtualization support in ARM processors, in addition to the design and implementation of an object-oriented hypervisor that future Choices designers can exploit.

8. REFERENCES

[1] J.Robin, C.Irvine, "Analysis of the Intel Pentium's ability to Support a Secure virtual machine Monitor", *In Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA, pages 129-144, Aug 2000.*

[2] Campbell R., Johnston G. and Russo V. 1987., "Choices (class hierarchical open interface for custom embedded systems)", *SIGOPS Oper. Syst. Rev. 21, 3 (Jul. 1987), 9-17.*

[3] ARM7 TDMI data sheet. http://www.e-lab.de/ARM7/ARM-instructionset.pdf.

[4] P.T.Barham, B.Dragovic, K.Fraser, S.Hand, T.L.Harris, A.Ho, R.Neugebauer, I.Pratt, A. Warfield., "Xen and the art of virtualization", *SOSP 2003:164-177.*

[5] Popek, G. J. and Goldberg, R. P. 1974. "Formal requirements for virtualizable third generation architectures", *Commun. ACM 17, 7 (Jul. 1974), 412-421.*

[6] "ESX Server Architecture and Performance Implications", VMware Technical Papers.

[7] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted virtual machine Monitor", *USENIX Annual Technical Conference, pages 1--14. USENIX Association, 2001.*

[8] VMX Manual. ftp://download.intel.com/technology/computing/vptech/C97063-002.pdf.

[9] QEMU technical documentation. http://fabrice.bellard.free.fr/qemu/qemutech.html\#SEC1.

[10] Presented by Jack Lo, *VMWare and CPU Virtualization Technology*, www.vmware.com/vmworld/2005/pac346.pdf.

[11] An Introduction to Virtualization, http://www.kernelthread.com/publications/virtualization.

[12] David Seal, "ARM Architecture Reference Manual", Addison-Wesley, 2001.