Virtual machines for distributed real-time systems☆☆

Marco Cereia 1, Ivan Cibrario Bertolotti *

IEIIT-CNR c/o Politecnico di Torino, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy

Received 30 November 2006; received in revised form 10 September 2007; accepted 10 October 2007

Available online 22 October 2007

Abstract

The steady increase in raw computing power of the processors commonly adopted for distributed real-time systems leads to the opportunity of hosting diverse classes of tasks on the same hardware, for example process control tasks, network protocol stacks and man–machine interfaces.

This paper describes how virtualization techniques can be used to concurrently run multiple operating systems on the same physical machine, although they are kept fully separated from the security and execution timing points of view, and still have them exhibit acceptable real-time execution characteristics.

With respect to competing approaches, the main advantages of this method are that it requires little or no modifications to the operating systems it hosts, along with a better modularity and clarity of design.

© 2007 Elsevier B.V. All rights reserved.

Keywords: Virtual machines; Real-time operating systems; Embedded systems

1. Introduction and related work

The raw computing power of the processors commonly used to implement distributed real-time systems, especially for industrial applications, has increased steadily.

For example, Programmable Logic Controllers (PLCs) at first relied on special-purpose hardware with limited processing and networking capabilities, but soon evolved into sophisticated, networked control engines – sometimes leveraging COTS hardware – that run either on a special-purpose real-time operating system [1], or on a real-time kernel that, in turn, supports the execution of a general-purpose operating system and its applications [2,3].

Among these approaches, the latter is especially appealing because it supports the orderly coexistence – on the same hardware – of both time-critical industrial control functions and application software that, for example, connects the system to the higher layers of the factory automation hierarchy, gives the system a friendly man–machine interface, and provides for online browsing of the system documentation. In any case, when pursuing this approach, it is necessary to modify the inner components of the operating system itself, hence the extension inherently depends on the operating system itself, hence the extension will work on any other.

Instead, in the approach known as virtualization, hardware resources are partitioned to create a system composed of two or more virtual machines, by means of a software component known as Virtual Machine Monitor (VMM). In this way, it becomes possible to support, for example, the concurrent execution of both a real-time and a general-purpose operating system, each one hosted by its own virtual machine.

The most interesting property of virtual machines is that, at least accordingly to their original definition, also known as full or perfect virtualization [4–7], they are identical in all respects to the physical machine they are implemented on, barring instruction timings. As a consequence, no modifications to the operating systems hosted by the virtual machines are required, except the addition of a special-purpose device driver to handle inter-machine communication if required.

More recently, in order to achieve improved efficiency, the guest operating systems and the VMM were made able to communicate and cooperate, to provide the VMM with a better notion of the
ongoing operating system activities. This approach is referred to as **paravirtualization** [8–10] and its implementation implies modifying and recomposing parts of the guest operating system.

Even if no standard for this kind of communication has been adopted yet, several major providers of commercial VMMs have started to prepare and publish the preliminary specifications of an Application Programming Interface (API) between the guest operating system and the VMM specifically aimed at paravirtualization [11]. The open-source community went along the same path and the so-called paravirt-ops, a paravirtualization interface for the Linux operating system, is scheduled for introduction in version 2.6.21 of the kernel.

Alongside the paravirtualization API development, standardization efforts are also being directed to the virtual machine management interface, that allows management software to deploy, control, and monitor virtual machines running in different virtualization environments. For example, this problem is being addressed within the virtualization, partitioning, and clustering workgroup of the Distributed Management Task Force (DMTF) [12], again with the active participation of the major providers of commercial VMMs.

On the hardware side, several processors families nowadays include support for a more efficient processor virtualization [13,14]. In some cases, like [15], hardware support streamlines the virtualization of I/O devices as well.

However, both the historical and the contemporary implementations of virtualization are mainly oriented to general-purpose systems in large data centers [16], hence their achievements are not directly applicable to distributed real-time systems.

In this paper, we build on an earlier work [17] to show that implementing virtualization on version 5 of the well-known ARM architecture [18] is feasible and requires only marginal modifications to the hardware in the general case. Moreover, we also show that a recent enhancement made in version 6 of the ARM architecture [14] — known as **TrustZone security extensions** and already available in commercial products [19] — albeit mainly conceived to support a secure software execution environment is also really useful to reduce the computational overhead of virtualization, in the special case of two virtual machines.

In both cases, the implementation of a prototype VMM was successfully carried out following the perfect virtualization approach as much as possible, and the performance measurements carried out on the prototypes show results which are both encouraging and compatible with real-time software execution.

The only departure from the full virtualization approach is that the custom architectural extension (in the first case) and the TrustZone extension (in the second case) are no longer available within the virtual machines themselves. As a consequence, for example, a VMM would be unable to run within a virtual machine, but this restriction is unlikely to be of any practical significance for the class of applications being considered.

The paper is organized as follows: Section 2 recalls the basic VMM design principles, while Section 3 describes their application to both the plain and the TrustZone-enhanced ARM architecture. Then, Section 4 presents the results of the performance evaluation carried out on the VMM prototypes by means of suitable instruction-level simulators. Last, Section 5 draws some concluding remarks.

### 2. VMM design principles

According to its most general definition, a **virtual machine** is an abstract system, composed of one or more **virtual processors** and zero or more **virtual devices**. The implementation of a virtual machine can be carried out in a number of different ways like, for example, interpretation, (partial) just-in-time compilation, and hardware-assisted instruction-level emulation. Moreover, those techniques can be, and usually are, combined to obtain the best compromise between complexity of implementation and performance for a given class of applications.

In this paper, we mostly focus on **perfect** virtualization, that is, the implementation of virtual machines whose processors and I/O devices are identical in all respects to their counterpart in a physical machine, that is also the machine on which the virtualization software runs. The implementation of virtual machines is carried out by means of a peculiar kind of operating system kernel and hardware assistance is used to keep overheads to an acceptable level.

The internal architecture of operating systems based on virtual machines revolves around a simple observation. An operating system must implement two essential but distinct functions, that is:

1. **multiprogramming**;
2. **system services**.

Accordingly, these operating systems fully separate the two functions and implement them as two distinct components:

1. A **virtual machine monitor** runs in privileged mode; it does multiprogramming, and provides many virtual processors **identical** in all respects to the real processor it runs on. In addition, it implements basic synchronization and communication mechanisms between virtual machines and partitions system resources among them, thus giving to each virtual machine its own set of virtual I/O devices.
2. A **guest operating system** runs on each virtual machine, implements system services, and supports the execution of a set of applications within the virtual machine itself. Different virtual machines can run different operating systems, and they must not necessarily be aware of being run in a virtual machine.

In the traditional approach to virtual machine implementation, based on privileged instruction emulation, when an application program running within a virtual machine performs a system call instruction, the virtual machine monitor intercepts it and redirects it to the guest operating system hosted by the same virtual machine.

The guest operating system itself is given the illusion of running in privileged mode, whereas the physical processor actually continues to operate in user mode; in this way, the virtual machine monitor is able to intercept all privileged
instructions issued by the guest operating system, check them against the security policy of the system and then perform them on behalf of the guest operating system, that is, emulate them.

Interrupt handling is implemented in a similar way: the virtual machine monitor catches all interrupt requests and redirects them to the appropriate guest operating system handler, reverting to user mode in the process; thus, the virtual machine monitor can intercept all privileged instructions issued by the guest interrupt handler, and again emulate them as appropriate.

As a consequence, the key difference of the VMM approach with respect to traditional multiprogramming is that most traditional operating systems confine user-written programs to run in unprivileged, user mode only, and to access operating system services through a system call primitive. Thus, each service request made by an application enters the operating system and switches the processor to privileged mode in the process; the operating system then performs the requested service and returns control to the user program, simultaneously bringing the processor back into user mode.

A VMM system, instead, supports many virtual processors that can run in either user or privileged mode but, while doing this, it also makes the real processor execute the software of each virtual machine – also called the virtual machine guest code and encompassing both the application and the operating system code – always in user mode, to keep control of the system as a whole.

Hence, it is important to make a distinction between:

- the real processor mode, that is the mode the physical processor is running in, and
- the virtual processor mode, one for each virtual processor that is the mode each virtual processor is running in.

At any instant, each virtual processor is characterized by its current virtual processor mode, and it does not necessarily coincide with the physical processor mode.

2.1. Views of the processor state

The view that machine code has of the processor state at a given instant is a subset of the whole processor state and depends on the mode the processor is running in. We can define two classes of views:

**User-mode view.** It is the portion of the processor state that is accessible, with either read-only or read-write access rights, when the processor is in user mode.

**Privileged-mode view.** It is the portion of the processor state that is accessible, with either read-only or read-write access rights, when the processor is in privileged mode.

On processors supporting multiple, independent privileged modes, like the ARM version 5 processor, the privileged-mode view involves a collection of independent views, one for each privileged mode. It should be noted that the intersection among views can be, and usually is, not empty. For example, in the ARM processor architecture [18], an unbanked register is accessible from, and common to, all unprivileged and privileged processor modes. By contrast, a banked register exists in multiple instances, one for each processor mode.

It should also be noted that only the union of all views gives full access to the processor state: in general, no individual view can do the same, not even the view corresponding to the “most privileged” privileged mode, even if the processor specification contains such a hierarchical classification of privileged modes.

At the same time, the execution of each machine instruction both depends on and affects the internal processor state, and the portion of state it affects in turn depends on the processor mode.

For example, an instruction that increments a banked register by one will update the instance of the register that belongs to the current processor view, and will not affect any other instance of the same register.

Since each virtual machine shall have its own set of virtual processor state views, the VMM shall maintain a Virtual Machine Control Block (VMCB) for each virtual machine that holds its full virtual processor state, both unprivileged and privileged; therefore, it contains state information belonging to, and accessible from, distinct views of the virtual processor state, with different levels of privilege.

Then, given a set of VMCBs, the VMM performs the following functions, likely with hardware assistance:

- It gives to a set of machine code programs, running either in user or privileged mode, their own, independent collection of processor state views; this gives rise to a set of independent sequential processes. Each collection of views is indistinguishable from the view the program would have when run on a bare machine, without a VMM. This gives the illusion that each process actually runs on its own processor, identical to the physical processor underneath the VMM.
- It is able to switch the physical processor among the processes mentioned above by switching it back and forth among their VMCBs; in this way the VMM implements multiprogramming. Switching the processor involves both a switch among possibly different program texts, and among distinct collections of processor views.

2.2. Privileged instruction emulation

The classic approach to processor virtualization [4–7] is based on privileged instruction emulation. With this approach, virtual machines are always executed in physical user mode, also when the virtual processor is in any virtual privileged mode.

When a virtual machine is being executed by a physical processor, the VMM transfers part of its VMCB into the physical processor state; when the VMM assigns the physical processor to another virtual machine, the physical processor state is transferred back into the VMCB.

Most virtual machine instructions are executed directly by the physical processor, with no overhead; other instructions must instead be emulated by the VMM with an overhead due to trap handling.
Unprivileged instructions act on the current view of the processor state only, and are executed directly by the physical processor. Two cases are possible, depending on the current virtual processor mode:

1. Both the virtual and the physical processor are in user mode. In this case, the virtual and the physical instruction execution (and their corresponding processor state views) fully coincide; no further manipulation of the processor state is necessary.

2. The virtual processor is running in a privileged mode and the physical processor is in user mode. Instruction execution acts on the user mode view of the physical processor state, but the intended effect is to act on one of the virtual privileged views. To give this illusion, when the mode of the virtual processor changes, the VMM loads the user-mode view of the processor state with the portion of the VMCB related to the privileged mode into which the virtual processor is moving, after saving it into the portion of the VMCB related to the mode the virtual processor is coming from. Even in this case, the overhead incurred during actual instruction execution is zero.

On the other hand, privileged instructions always act on one of the privileged views of the processor state. When the execution of a privileged instruction is attempted in physical user mode, the processor takes a trap that must be managed by the VMM. The VMM trap handler must emulate either the trap or the trapped instruction, depending on the current virtual processor mode, and update the virtual processor state stored in the VMCB with the outcome of the emulation. Three different cases are possible:

1. If the virtual processor was in user mode when the trap occurred, the VMM must emulate the trap, because the guest code did not have the right of executing the instruction in the first place: the virtual processor mode is switched to the appropriate privileged mode and the actual trap handling is performed by the guest code of the virtual machine. In turn, this will typically lead the guest operating system to take control of the virtual machine.

2. If the virtual processor was in privileged mode, and the trap was triggered by lack of the required physical processor privilege level, the VMM must emulate the privileged instruction, because the guest code had the right of executing the instruction but was hampered by the interposition of the VMM. In this case the VMM performs trap handling by itself and the privileged guest code of the virtual machine does not receive the trap at all. Instead, it sees the outcome of the emulated execution of the privileged instruction.

3. If the virtual processor was in privileged mode, but the trap occurred for some other reason, the VMM must emulate the trap, and the handling will be performed by the guest code of the virtual machine, in virtual privileged mode. Also in this case, the guest operating system will typically be in charge of handling this kind of trap.

In the first and third case above, the behavior of the virtual processor exactly matches the behavior of a physical processor in the same situation, except for the trap enter mechanism, that is emulated in software instead of being performed in either hardware or microcode.

In the second case, the overall behavior of the virtual processor still matches the behavior of a physical processor in the same situation, but the trap is kept invisible to the virtual machine guest software because, in this case, the trap is instrumental for the VMM to properly catch and emulate the privileged instruction.

Above all, the emulation of all privileged instructions issued by the guest code, performed by the VMM, has the effect of confining the guest code to a specific, and unprivileged, processor state view. On the other hand, it also keeps the guest code unaware of being run inside such a confined environment.

Unfortunately, the instruction classification just set forth does not cover the whole instruction set of many contemporary processors, including the ARM processor. In fact, on, these processors a third class of instructions exists, and encompasses unprivileged instructions whose behavior and outcome depend on a physical processor state item belonging only to one or more privileged processor state views.

This class of instructions is anomalous and problematic in nature from the point of view of processor virtualization, because these instructions allow a program to infer something about a processor state item that would not be accessible from its current view of the processor state itself.

As a consequence, the presence of instructions of this kind hampers the privileged instruction emulation approach to processor virtualization just discussed, because it is based on the separation between the physical and virtual processor states, and enforces this separation by trapping (and then emulating in the virtual processor context) all privileged instructions that try to access privileged processor state views.

Since instructions of this kind are able to bypass this mechanism as a whole, because they take information directly from the physical processor state and never generate a trap, they make the VMM unable to detect and emulate them and shall be refined by any architectural extension aimed at virtualization.

3. Enhancements to the ARM architecture

This section outlines version 5 of the ARM architecture [18], discusses the architectural elements that pose some difficulties for virtualization as they are currently defined, and then presents a minimum set of enhancements that must be implemented in order to make virtual machines feasible and reasonably efficient.

The enhancements affect the exception handling mechanism and the instruction set; moreover, they add some registers to the system control coprocessor.

It should also be noted that only the 32-bit architecture and the ARM instruction set was taken into consideration for the instruction set extension. The extension of the (now obsolete) 26-bit architecture and of the Thumb instruction set for the purpose of virtualization is beyond the scope of this paper.
3.1. ARM architecture overview

The ARM architecture supports seven processor modes mapped onto two privilege levels. When the processor is in user mode the level of privilege is unprivileged whereas, for all other modes, the level of privilege is privileged.

The architecture provides a total of 37 registers, each one 32 bits wide, arranged in partially overlapping banks, with a different register bank for each processor mode:

- User and system modes share the same view of the processor state, as defined in Section 2.1, and differ only by their privilege level.
- General registers R0 through R12, the program counter (PC), and the Current Processor Status Register (CPSR) are shared among, and accessible from, all processor state views. As an exception, the fast interrupt mode has its own copy of registers R8 through R12, in order to reduce the number of registers to be saved on entry into the corresponding handler and make this mode more suitable for fast interrupt handling.
- Each processor mode (except for user and system modes, which share the same instance of these registers) has its own copy of register R13, conventionally used as a stack pointer, and R14, the link register.
- Each processor mode, except for user and system modes, has its own copy of the SPSR register, used by the hardware to save the CPSR when the processor enters that mode. This happens, for example, when an exception is caught.

It should be noted that the CPSR contains condition code flags, interrupt control bits, an indication of the current processor mode, and other control information, so it contains both unprivileged and privileged information, and these portions belong to distinct views of the processor state.

The ARM version 5 instruction set can be partitioned into seven classes of instructions, summarized in Table 1:

Branch instructions. Branch instructions allow either a conditional or an unconditional branch with a PC-relative 24-bit offset. The branch and link instruction, used to perform a subroutine call, also saves the address of the instruction following the branch into the link register. In either case, no mode change can occur by effect of a branch instruction, unless it triggers an exception.

Data-processing instruction. The behavior of a data-processing instruction mainly depends on the value of the S modifier in the instruction opcode. In their basic form, when the S modifier is reset, all data-processing instructions operate on the current view of the processor state and, when executed, do not change the processor mode unless they trigger an exception. Moreover, they do not change any processor state item that does not belong to the current view. Instead, when the S modifier is set, data-processing instructions can have two distinct behaviors, depending on the destination register:
- When the destination register is not the PC, they update the CPSR flags according to the result of the instruction. When the destination register is the PC, they copy the SPSR into the CPSR and their execution is unpredictable when attempted in user or system mode, because they do not have a SPSR.

In the second case, the hardware is unable to trap an attempt to access a privileged processor state item, the SPSR, from user mode.

Other arithmetic instructions. This class of instructions includes the specialized CLZ instruction, to count the leading zeros in a word, as well as several multiply and multiply/accumulate instructions. They support the S modifier like data-processing instruction, but using the PC as destination register is not a special case; instead, any attempt to use the PC as either source or destination register leads to unpredictable results, regardless of the processor mode. Moreover, these instructions never access the SPSR.

Status register access instructions. There are two instructions to operate on the status register: one to move the contents of either the CPSR or the SPSR into a general-purpose register, and one to perform the opposite. The accesses to the privileged portion of the status registers are of most interest from the virtualization point of view. The architecture specification states that:
- Read access is granted to the privileged portion of the CPSR in user mode.
- Any write access to the privileged portion of the CPSR in user mode is silently ignored.
- Read and write accesses to the SPSR are unpredictable in user and system modes.

In all situations described above, albeit an unprivileged instruction is trying to access some processor state items outside its view, no trap is generated, hence there is no way for the VMM to be informed of the attempt.

Load and store instructions. There are three types of instructions that perform memory access cycles: regular load and store instructions, swap instructions and load and store multiple instructions.

Regular load and store instructions operate only on the current processor state view. Swap instructions are atomic and can be used to implement process synchronization based on lock

<table>
<thead>
<tr>
<th>Table 1: ARM instruction classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
</tr>
<tr>
<td>Branch</td>
</tr>
<tr>
<td>Data processing</td>
</tr>
<tr>
<td>Other arithmetic</td>
</tr>
<tr>
<td>Status register access</td>
</tr>
<tr>
<td>Load and store</td>
</tr>
<tr>
<td>Exception-generating</td>
</tr>
<tr>
<td>Coprocessor</td>
</tr>
</tbody>
</table>
variables and busy wait, hence care must be taken to preserve this atomicity if they are emulated by the VMM.

The load and store multiple instructions, LDM and STM, are more complex, and come in three forms:

- The first form loads or stores a group of general-purpose registers belonging to the current processor state view from/to a block of contiguous locations. This form does not access the CPSR, the SPSR, or any other register outside the current processor state view.
- The second form loads or stores a group of user mode general-purpose registers, excluding the PC, from/to a block of contiguous memory locations. This form does access general purpose registers outside the current processor state view, and gives unpredictable results when executed either in user or system mode.
- The third form can be used by the LDM instruction only, and loads a group of general-purpose registers, including the PC, from a block of contiguous memory locations, and also copies the SPSR of the current processor mode into the CPSR, thus possibly leading to a mode change. Moreover, it gives unpredictable results when executed in user or system mode.

In all situations classified as unpredictable above, no trap is generated. So, an unprivileged instruction can try to access some processor state items outside its view without this attempt being trapped.

**Exception-generating instructions.** There are two instructions that unconditionally trigger an exception: the SWI instruction, often used as the operating system call instruction, and the BKPT instruction, that inserts a software breakpoint in the instruction stream. None of them has access to information outside the current processor state view.

**Coprocessor instructions.** These instructions perform data transfers between the processor and its coprocessors, the initiation of a coprocessor data processing operation, and address generation on behalf of the coprocessor for load and store operations. The ARM architecture never specifies a privileged instruction exception to occur when any attempt is made to either access or modify a privileged processor state item when in user mode; instead, the offending instruction is simply ignored, in part or as a whole.

The ARM architecture adopts a unified handling of interrupts and traps, commonly referred to as exceptions. Each exception type has its own code, that is used as an index to point to an instruction, inside an exception table, to be executed. In turn, the instruction will usually be an unconditional jump instruction to the address of the actual exception handling routine).

The exception table can be located at either one of two fixed addresses in physical memory, namely 0x00000000 or 0xFFFF0000. All vectors used by the physical processor when handling an exception reside in the privileged address space, and are accessed after the processor has been switched into an appropriate, privileged mode, which depends on the kind of exception about to be processed.

### 3.2. Architecture extension enable register

For backward compatibility, the enhancements to the ARM architecture described in this section must be disabled by default, and enabled only when a VMM is present. To this purpose, the system control coprocessor register 12, previously marked as reserved, is now defined as the Extension Enable Register, XER.

Writing any non-zero value into the XER register enables the architectural enhancements, a zero value disables them; the default value of the XER register is zero. The XER register is privileged and can be accessed only when the CPU is in a privileged execution mode.

### 3.3. Exception table base register

In order to intercept all traps and keep full control of the system, the VMM must be able to relocate the exception table obeyed by the hardware anywhere in the address space and install its own trap handlers there, while giving to the guest code the illusion of being able to set its own exception table at one of its default locations.

To this purpose, the system control coprocessor register 11, previously marked as reserved, is now defined as the Exception Table Base Register ETBR, and contains the base address of the exception vector table. The ETBR register is privileged and can be accessed only when the CPU is in a privileged execution mode.

The CPU uses the contents of this register to locate the exception vector table whenever the extensions to the architecture are enabled, otherwise it reverts to the standard table locations.

### 3.4. Instruction set refinement

As discussed in Section 3.1, the ARM architecture does not take a trap when an instruction attempts to access a processor state item outside its own view. Since, by contrast, virtualization has its grounds on the confinement of the guest code inside a specific processor view, all these situations are inherently problematic. As a consequence, the exception handling mechanism should be extended to include a new kind of exception, namely a “privileged instruction” exception.

The ARM processor should take a privileged instruction trap, and switch to supervisor mode in the process, whenever the execution of a problematic instruction is attempted in user mode, provided that the architectural extensions described here are enabled. In particular, as discussed in Section 3.1, the instructions affected by the extension are:

- Data processing instructions with the S modifier set and the PC as destination register.
- Status register access instructions. Both the instructions that read (MRS) and write (MSR) a status register must be modified.
- Load/Store Multiple, when either loading/storing user mode registers.
- Load Multiple, when loading the PC and the CPSR.
Coprocessor access instructions, when accessing a privileged coprocessor register. For example, all registers of the system control coprocessor are privileged and triggered an undefined instruction trap when accessed in user mode in the original architecture specification; in the enhanced architecture, they trigger a privileged instruction trap instead.

It should also be noted, as a possible performance enhancement, that it is not strictly necessary to take a privileged instruction trap on any attempt to execute the MSR instruction on the CPSR, because only a portion of it contains privileged information and belongs to the privileged state view, and the actual portion of the CPSR being accessed can be readily determined by examining the field mask within the MSR instruction.

Therefore, if the MSR instruction only acts on the unprivileged portion of the CPSR, it is possible to execute it directly, without generating any trap, thus reducing the probability of taking a trap and making the overall emulation overhead smaller.

3.5. Privilege violation reason register

Along with the privileged instruction trap just described, a new privileged register, the Privilege Violation Reason Register PVRR has been added to the ARM architecture. This register conveys the exact reason of a privileged instruction trap by means of a reason code.

At any time, its contents are related to the last privileged instruction trap taken by the processor and allow the VMM to perform a proper emulation of the trapped instruction, when appropriate.

The PVRR register can be accessed through the system control coprocessor register 4, previously marked as reserved. Being privileged, the PVRR can be accessed only when the CPU is in a privileged mode. Moreover, the PVRR register is implicitly reset to zero on CPU reset and after each read operation.

3.6. The TrustZone approach

The TrustZone security extensions enhance the sixth version of the ARM architecture and enable a secure software execution environment by introducing several modifications to the basic processor architecture.

These modifications are much more pervasive than those described in Sections 3.2–3.5 and can be summarized as follows:

- At each instant, the processor can be in one of two possible states: the secure and the non-secure state. The current state of the processor is mainly controlled by a bit of a newly-introduced Secure Configuration Register in the system control coprocessor, the non-secure bit or NS bit.
- Any undue modification to the Secure Configuration Register is forbidden, by making it accessible only when the processor is in a secure, privileged execution mode.
- In order to support two distinct and isolated execution environment – or worlds – depending on the processor state, the concept of secure versus non-secure state is also applied to, and enforced at, all levels of the system’s memory hierarchy.

Accordingly, most system coprocessor registers related to virtual memory management, for example the Translation Table Base Register, now come in two instances (one for each state) and each memory (or memory-mapped peripheral) access requested by the processor is validated against the security attribute set for that memory region.

Also other kinds of hardware resources can be designated as either secure or non-secure. Secure resources can only be accessed by the processor core when it is in the secure state.

- A new privileged processor mode, in addition to the seven already foreseen by the basic architecture, links the two worlds and switches the processor between them in a carefully controlled way.

This new mode, the monitor mode, is always considered secure, regardless of the state of the NS bit. Like most other modes, the monitor mode has two banked general purpose registers, R13_mon and R14_mon, and a banked SPSR.

- In order to have a clear-cut interface and preserve the integrity of the secure system the processor always enters the monitor mode from the non-secure state through fixed entry points. In particular, several exception sources (IRQ, FIQ, and external abort) can be configured to enter the secure monitor mode, and a new instruction, the Software Monitor Call or SMC, triggers an exception which brings the processor into the monitor mode regardless of its current state.

- Since each world must have the ability to handle its own exceptions, the base of the exception table can now be set independently for the secure and the non-secure states. Yet another exception table is used for the exceptions which bring the processor into the monitor mode.

Albeit security is undoubtedly the main focus of TrustZone, the hardware enhancements it introduces are useful for virtualization too, in the special case of two virtual machines, because they fulfill most of the requirements set forth in Section 2 when one virtual machine is run in secure mode, and the other one in non-secure mode. Several key differences remain, namely:

- TrustZone is tailored to support exactly two worlds, instead of an arbitrary number of virtual machines.
- The isolation between worlds enforced by TrustZone is “unidirectional”, that is, the non-secure world is prevented from accessing and interfering with the secure world, but the converse is not true.
- The virtual architecture is not identical to the physical one, because the guest operating system can no longer use the TrustZone features by itself.

Nevertheless, if one is willing to accept these differences, TrustZone eliminates completely one of the most important sources of overhead related to virtualization, namely privileged instruction emulation. This is because the hardware itself ensures that, even if the processor is operating in a privileged
but non-secure mode, it can neither access nor modify any state
information belonging to the secure world, and this is exactly the
reason why privileged instruction emulation was introduced in the
first place.
Moreover, TrustZone also helps to speed up context switches
between virtual machines, because a significant portion of the
memory management context is duplicated, hence it is possible to
switch from one context to the other quickly, by toggling the NS bit.
Since the monitor mode:

• is considered to be secure regardless of the NS bit,
• has the ability to freely switch the processor between the
secure and the non-secure states,
• can access both the secure and the non-secure execution
contexts at the same time,
• intercepts the classes of exceptions (IRQ, FIQ, and external
abort) which are of interest for virtualization,

it is also the most appropriate execution mode for the VMM.
As a consequence, the only virtualization overhead when
using TrustZone is due to the necessity of saving and restoring
the processor registers during a switch from one virtual machine
to the other because these registers, unlike memory manage-
ment registers, come in a single instance. In the general case,
during a switch, the full view of the processor registers of the
current virtual machine must be saved into its VMCB, and then
the view of the virtual machine being scheduled must be
restored from its VMCB.

4. Performance evaluation

This section is mainly concerned with measuring the
overhead induced by two distinct prototype implementations
of a VMM following the design guidelines discussed in Section
2. They run on the minimally enhanced ARM architecture and
on the TrustZone-enhanced architecture described in Section 3,
respectively.
For the sake of comparison, in both cases the VMM supports
just two virtual machines, and the real-time capabilities of one
of the two virtual machines have been preserved by giving it
priority on the other one.
Since the enhanced ARM architecture we proposed in
Sections 3.2–3.5 has never been implemented in hardware, the
experiments were carried out using a custom simulator based on
the “ARMulator”, a GPL-licensed ARM instructions simulator
distributed by ARM Ltd. The existing simulator has been
extended in two ways:

• First of all, the support for the architecture extensions
discussed in Sections 3.2–3.5 has been added; the support
also includes the capability of counting how many times
each extension has actually been used during the simulation,
for statistical analysis.
• The simulation environment has been further enhanced by
adding support for the ARM reference memory management
unit and for a small set of devices needed to execute the guest
code.

On the other hand, since TrustZone is already available in
several commercial products [19], the measurements on the
TrustZone-enhanced architecture discussed in Section 3.6 have
been carried out on a commercial, cycle-accurate simulation tool.

4.1. Performance of the minimally enhanced architecture

On the minimally enhanced architecture the main sources of
overhead are the emulation of privileged instructions and the
context switch between virtual machines.
As discussed in [17] and summarized in Table 2, on the
prototype implementation of the VMM for this architecture, the
privileged instruction emulation handler is made up of 39
machine instructions. They can be partitioned into four classes:

• The first-level handler, written in assembler, that sets up the
execution environment for the portion of the VMM written in C.
• The second-level handler, written in C that actually emulates
the trapped instruction.
• The VMCB switch code that saves and restores the full
VMCB of the virtual machine that raised the trap.
• The debugging instructions, used to perform several internal
consistency checks.

On the other hand, for the context switch between virtual
machines, the total overhead is 66 instructions, including 4
instructions to switch between memory management contexts.

4.2. Performance of the TrustZone-enhanced architecture

When a VMM with the same characteristics outlined above
is implemented on a TrustZone-enhanced architecture, the privi-
ileged instruction emulation overheads drop to zero, as motiva-
ted in Section 3.6.
Instead, the context switch requires between 77 and 83
instructions depending on the direction of the switch because,
even if the switch between the memory management contexts is
performed by the hardware in this case and costs no instruc-
tions, more instructions are needed to cope with a more complex
architecture.
In order to check the correctness of the context switch
code, and to better assess its performance, the test system depicted
in Fig. 1 has been set up and executed on a commercial, cycle-
accurate simulator of the TrustZone-enhanced ARM architecture.
It contains a very simple non-secure test application consisting
of a tight loop, the VMM under test, a bare bone secure interrupt

<table>
<thead>
<tr>
<th>Class</th>
<th>Number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>First level handler</td>
<td>11</td>
</tr>
<tr>
<td>Second level handler</td>
<td>8</td>
</tr>
<tr>
<td>VMCB switch</td>
<td>9</td>
</tr>
<tr>
<td>Debugging</td>
<td>11</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>39</strong></td>
</tr>
</tbody>
</table>
handler, as well as a periodic source of secure interrupts (not shown in the picture). During the test, the system works as follows:

- The non-secure application runs continuously until a secure interrupt request arrives (Fig. 1, op. 1); at this point, the hardware starts executing the VMM.
- The VMM saves the execution context of the non-secure application and restores the secure context to execute the secure interrupt handler (Fig. 1, op. 2).
- The secure interrupt handler performs its job and then returns to the VMM (Fig. 1, op. 3).
- At this point, the VMM saves the secure context and switches back to the non-secure application (Fig. 1, op. 4).

By means of a set of simulator breakpoints, the number of cycles needed to perform operations 2 and 4 has been measured and is reported in Table 3; these operations represent the overhead introduced by the VMM to perform a full context switch from the non-secure into the secure world and back.

Operations 1 and 3 have not been taken into account because the overhead incurred by the system when a secure interrupt request arrives (operation 1) is entirely determined by hardware, hence it does not depend on the VMM. Similarly, the number of clock cycles spent in the secure interrupt handler (operation 3) has not been considered because it depends only on the characteristics of the device being handled and not on the presence of a VMM.

4.3. Comparison

As Table 4 summarizes, both approaches to virtualization are suitable for use because they have a limited overhead, which mainly depends on the number of instructions executed by the VMM to perform its critical activities: privileged instruction emulation and context switch.

Regarding context switches, the number of instructions required by both approaches is quite similar and, as Table 3 shows, corresponds to a maximum of 425 clock cycles required to perform a full context switch from one virtual machine to the other, and back to the first.

Assuming that the maximum clock frequency of the processor is 320 MHz, as suggested by ARM for a typical area-optimized core [20], this corresponds to about 1.33 μs. Hence, for example, if the real-time virtual machine requires a periodic interrupt source with a frequency of 1 KHz for timing purposes, the total overhead introduced by virtualization is limited to about 0.13%, with respect to a bare machine in the same situation.

Table 3: Context switch overheads in the TrustZone-enhanced VMM

<table>
<thead>
<tr>
<th>Operation</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Secure int. request</td>
<td>–</td>
</tr>
<tr>
<td>2. Switch to secure world</td>
<td>227</td>
</tr>
<tr>
<td>3. Int. handling and return to VMM</td>
<td>–</td>
</tr>
<tr>
<td>4. Switch to non-secure world</td>
<td>198</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>425</strong></td>
</tr>
</tbody>
</table>

Table 4: VMM overheads as a function of the virtualization approach being used

<table>
<thead>
<tr>
<th>Approach</th>
<th>Number of instructions</th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimalistic approach</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priv. instruction emulation</td>
<td>39</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>Context switch</td>
<td></td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>TrustZone</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priv. instruction emulation</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Context switch</td>
<td></td>
<td>77</td>
<td>83</td>
</tr>
</tbody>
</table>

Regarding privileged instruction emulation, quantifying the overhead is more difficult, because it depends on the relative frequency of privileged instructions in the virtual machine instruction stream and, in turn, on the characteristics of the guest operating systems and applications.

However, [17] shows that this is an important source of overhead when using the minimally extended architecture to host, for example, the Linux operating system. In this case, the total overhead can be up to 21% even when the guest operating system stays idle after the boot sequence.

Hence, the complete removal of this burden in TrustZone more that compensates its slightly bigger context switch overhead, which is a direct consequence of the greater complexity of the TrustZone extensions, and further remarks the strong importance of proper hardware assistance for an efficient virtualization.

5. Conclusions and related work

The raw computing power available in modern distributed real-time systems is steadily increasing, and leads to the possibility of running both time-critical industrial control functions and a general-purpose operating system, with its higher-level applications, on the same hardware.

In order to do this most existing systems, for example [3], require the extension of the inner components of the operating system itself, and this is not always desirable.

Other well-understood approaches, for example [2], are based on a real-time microkernel having full access to, and control of, system resources. This microkernel runs both the real-time portion of the application and a modified version of a general-purpose operating system, as its lowest-priority task.

Albeit these techniques have the invaluable advantage of imposing a very small overhead on the real-time tasks, even in this case the general-purpose operating system cannot be run “as
is”, but needs modifications that can be difficult to implement, especially if its source code is unavailable.

By contrast, the virtualization technique described in this paper, albeit more difficult to implement and possibly characterized by a greater overhead – especially when the underlying hardware has not been designed with virtualization in mind and has been minimally extended to support it – has the advantage of not requiring any modification to the operating systems it hosts.

We have shown that, albeit version 5 of the ARM processor architecture is not directly amenable to virtualization as it is, virtualization can be made feasible by means of a limited number of simple architectural enhancements.

Moreover, our results also show that several additional hardware features recently added to some members of the ARM processor family [19] can be leveraged to assist virtualization and greatly reduce its overhead, at least for the special case of two virtual machines.

The practical feasibility of virtualization has been further remarked by the implementation of two prototype virtual machine monitors, one for each of the extended architectures being analyzed. In both cases, their overheads have been measured with encouraging results.

Further performance gains can be expected by means of thorough code optimizations in the virtual machine context switch path. For example, in order to favor the readability and maintainability of the code, as well as to speed up the development process, the prototype implementations of the virtual machine monitor often perform a full context switch even in situations where it is not strictly necessary.

As a future work, we foresee to better investigate the ability of TrustZone to support an arbitrary number of virtual machines instead of two, and to assess the performance penalty to be incurred in this case.

References


Marco Cereia graduated in Electronic Engineering from the Polytechnic of Turin in 2001. Since then he has cooperated with the Istituto di Elettronica e di Ingegneria dell’Informazione and the Telecommunicazioni of the Italian National Research Council (IEIIT-CNR). Since 2003 he is a Research Fellow at the Istituto di Elettronica e di Ingegneria dell’Informazione and the Telecommunicazioni of the Italian National Research Council (IEIIT-CNR). His research interests are in the area of real-time operating systems and embedded devices, with particular emphasis on embedded systems virtualization.

Ivan Cibrario Bertolotti received his Laurea degree (summa cum laude) in computer science from Università di Torino, Turin, Italy, in 1996. Since then, he has been a Researcher with the Istituto di Elettronica e di Ingegneria dell’Informazione and the Telecommunicazioni of the Italian National Research Council (IEIIT-CNR). His current research interests include real-time operating system design and implementation, industrial communication systems and protocols, and formal methods for cryptographic protocol analysis and verification. Dr. Cibrario also teaches several courses on real-time operating systems at Politecnico di Torino and has served as a reviewer for several international conferences and journals.