Formal virtualization requirements for the ARM architecture

Niels Penneman[a,b,1,], Danielius Kudinskas[c], Alasdair Rawsthorne[c], Bjorn De Sutter[a], Koen De Bosschere[a]

[a]*Computer Systems Lab, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium*
[b]*Vrije Universiteit Brussel (VUB), Dept. of Electronics and Informatics (ETRO), Pleinlaan 2, B-1050 Brussels, Belgium*
[c]*School of Computer Science, The University of Manchester, Oxford Road, Manchester M13 9PL, UK*

Abstract

We present an analysis of the virtualizability of the ARMv7-A architecture carried out in the context of the seminal paper published by Popek and Goldberg 38 years ago. Because their definitions are dated, we first extend their machine model to modern architectures with paged virtual memory, IO and interrupts. We then use our new model to show that ARMv7-A is not classically virtualizable.

Insights such as binary translation enable efficient virtualization beyond the original criteria. Companies are also making their architectures virtualizable through extensions. We analyse both approaches for ARM and conclude that both have their use in future systems.

Keywords: Binary translation, Hypervisor, Instruction set architecture, Virtualization, Virtual machine monitor

1. Introduction

The term virtualization is used in many different contexts, ranging from storage or network technologies to execution environments and even to virtual realities. This paper refers to system virtualization—technology which allows multiple operating systems to be executed on the same physical machine simultaneously. It achieves this by decoupling the underlying hardware from the operating systems, which are executed deprivileged as shown in Figure 1. These operating systems are now called guest operating systems. In general, a guest or virtual machine (VM) refers to all software that is executed deprivileged and in an isolated environment under the control of a virtual machine monitor (VMM).

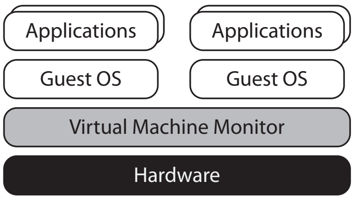

Figure 1: Overview of a virtualized system.

In 1974, Popek and Goldberg [44] wrote a classic paper on "Formal requirements for virtualizable third generation architectures". It defines strict system virtualization criteria for computer architectures, and proves that if they are met, an "efficient" VMM can be constructed for that architecture. Since its publication, the paper has been used as a solid reference point for designing hardware platforms capable of supporting an efficient VMM and it has become the groundwork of virtualization technology.

The criteria defined by Popek and Goldberg are now known as the conditions for *classic virtualizability*. Architectures that meet these criteria are particularly suited for full virtualization. *Full virtualization* (also known as pure or faithful virtualization) is a technique in which the VMM presents a virtual platform that is an exact replica of the real hardware to each of its guests. Hence, applications and operating systems do not require any modification to support virtualization.

Popek and Goldberg based their model on computers available at the time, such as DEC PDP-10 and IBM 360/67. Due to advances in microprocessor architecture, their model no longer fits current architectures. Furthermore, new approaches in the construction of efficient VMMs that do not fit Popek and Goldberg's model and criteria [24], have enabled virtualization of many more contemporary architectures.

Virtualization in the server and desktop world has already matured, with both software and hardware solutions available for several years [1, 14, 16, 45, 48, 50, 53]. However, virtualization on embedded systems has only been explored for the past seven years, and is an area of ongoing research [22, 29]. Solutions for data centres and desktop computing cannot be readily applied to embedded systems, because of differences in requirements, use cases, and computer architecture. We provide a new perspective on the Popek and Goldberg virtualizability requirements. We develop a model that focuses on modern

Email addresses: `Niels.Penneman@elis.UGent.be` (Niels Penneman), `kudinsd6@cs.man.ac.uk` (Danielius Kudinskas), `Alasdair.Rawsthorne@manchester.ac.uk` (Alasdair Rawsthorne), `Bjorn.DeSutter@elis.UGent.be` (Bjorn De Sutter), `Koen.DeBosschere@elis.UGent.be` (Koen De Bosschere)

[1] Niels Penneman is supported by the agency for Innovation by Science and Technology (IWT).

microprocessors and use it to analyse the ARM architecture. We illustrate that our model can also help in the construction of VMMs based on dynamic binary translation (DBT) by using our analysis to show the pitfalls for constructing a DBT VMM on ARM.

ARM is by far the leading architecture in the embedded and mobile market [47], and recently multiple software virtualization solutions have been developed for it, such as Codezero, OKL, Red Bend VLX, VIRTUS, VMware MVP and Xen-ARM [11, 12, 15, 28, 31, 34, 37]. Due to architectural problems none of the currently deployed solutions offers full virtualization [51].

ARM has already published a preliminary specification of extensions to its architecture, which will facilitate full virtualization [8]. The appearance of hardware platforms that implement these extensions is imminent. It is therefore important to understand the problems faced by virtualization technology on ARM today, and evaluate the possible solutions against the formal requirements derived by Popek and Goldberg [44] nearly 40 years ago. Popek and Goldberg also proposed techniques to virtualize architectures which did not meet their criteria. VMMs based on DBT, which do not require changes to the architecture, can be regarded as an evolution of their techniques. Their analysis therefore remains useful for the construction of such VMMs and for understanding the advantages and disadvantages of both the HW supported and DBT supported approaches. On the one hand, hardware extensions enable simple VMM implementations but are tied to specific architectures and platforms. On the other hand, DBT is notorious for complicating VMM implementations, but is known to be versatile and usable in situations where hardware extensions would be infeasible or impractical.

This paper analyses the application profile of the current ARM architecture, ARMv7-A, because it serves as the base for the upcoming virtualization extensions [8]. Our model excludes micro-controllers for deeply embedded systems. Those architectures often lack advanced memory management features and hence they are incapable of running full-blown operating systems.

Our contributions include:

- an update to the model of Popek and Goldberg with paged virtual memory, IO, and interrupts, and the application of such model to analyse the ARMv7-A architecture both without and with the virtualization extensions;
- an example of how an analysis according to our model helps in the construction of a DBT VMM;
- a discussion on the trade-offs between the use of architectural extensions for virtualization and DBT, in which we argue that both have their use in future systems.

The rest of this paper is organized as follows: Section 2 discusses our motivation and gives a quick overview of the model introduced by Popek and Goldberg. In Section 3 we extend this model to include paged virtual memory, IO, and interrupts. We use the updated model to show that the ARM architecture is not classically virtualizable in Section 4. In Section 5 we analyse ARM's upcoming hardware support and discuss how ARM can be fully virtualized using DBT. We show how our analysis from Section 4 helps in the construction of a DBT VMM. We then compare the use of hardware extensions with DBT-based approaches on a theoretical level.

2. Background and motivation

2.1. Existing virtualization solutions

Paravirtualization is the only virtualization technique deployed in today's ARM-based embedded systems. In paravirtualization, a VMM presents a custom interface to its VMs which is similar, but not identical to the underlying hardware [54].

Despite the popularity of paravirtualization, none of the efforts to standardize the required VMM interfaces has gained sufficient momentum to spread to more than a few operating systems or VMMs [3, 38, 46]. As a consequence, the majority of contemporary embedded operating systems do not support such interfaces out of the box. Instead, VMM vendors and third parties must provide source code patch sets to support specific VMM interfaces for specific operating system versions. This situation comes with four major drawbacks:

1. Developing, maintaining, and testing patch sets for each and every combination of a specific operating system version and a VMM interface is an expensive process. Although semantic patches may offer a solution to simplify patch management [10], the effort required for testing remains.
2. Patched operating systems may exhibit unexpected behaviour because their reliability is not guaranteed and patches may introduce new security issues.
3. Licenses may prevent or restrict modifications to operating systems source code, and often impose rules on the distribution of patch sets or patched code.
4. Previously certified software stacks will need to be recertified after patching. The recertification process is expensive and always specific to a particular VMM interface, thereby stimulating vendor lock-in.

This analysis is shared by major players in industry including ARM, Nokia and STMicroelectronics [19].

2.2. Classic virtualizability

The drawbacks of paravirtualization can be avoided by using full virtualization. With full virtualization, guest operating systems run unmodified on top of a VMM. This also enables virtualization of a priori unknown software. Popek and Goldberg [44] formally derived sufficient (but not necessary) criteria that determine whether an architecture is suitable for full virtualization. In this section, we briefly explain their model and results.

The machine model used by Popek and Goldberg is deliberately simplified. It includes a processor and a linearly addressable memory, but does not consider interaction with IO devices and interrupts.

The processor operates in either supervisor mode or user mode. Supervisor mode is a privileged mode, meant for the operating system, while user mode is unprivileged. The processor also features a program counter and a relocation-bounds register, used for relative memory addressing. An instruction set architecture (ISA) for such a processor can move, look up or process data and alter program control flow. Based on this description, they defined the concept of machine state.

Definition 1. The machine state S is defined by the contents of the memory E, the processor mode M, the program counter P, and the relocation-bounds register R:

$$S \equiv \langle E, M, P, R \rangle .$$

Since all variables that determine the machine state are finite, the set of all machine states Σ is also finite.

Definition 2. An instruction i is a function on the set of machine states Σ that maps one state to another:

$$\begin{array}{rlll} i: & \Sigma & \longrightarrow & \Sigma \\ & S_x & \longmapsto & i(S_x) = S_y . \end{array}$$

All instructions are classified in three categories:

- *Privileged* instructions execute correctly in privileged mode, and always trap in unprivileged mode.
- *Sensitive* instructions are further classified as *control-sensitive* and *behaviour-sensitive* instructions. Control-sensitive instructions attempt to modify the processor execution mode or the amount of available memory resources. Behaviour sensitivity manifests itself in two ways. The result of behaviour-sensitive instructions either depends on their location in physical memory (location sensitivity) or on the processor execution mode (mode-sensitivity).
- *Innocuous* instructions are those instructions that are not sensitive.

Upon a trap, control is transferred to a privileged mode, P will point to a pre-defined trap handler, and the system state before the trap is saved in the memory E, such that it can be restored when the instruction that caused the trap has been dealt with.

Popek and Goldberg also defined three fundamental properties for virtualized systems:

1. **Efficiency**: all innocuous instructions are executed natively without VMM intervention;
2. **Resource control**: guest software is forbidden access to physical state and resources;
3. **Equivalence**: guest software behaves identical[2] to when it is run on a system natively.

Using the above definitions, Popek and Goldberg proved the following theorem:

Theorem 1 (Popek and Goldberg [44])**.** *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.*

On an architecture on which all sensitive instructions are also privileged, a trap is generated and caught by the VMM whenever a guest attempts to execute a sensitive instructions. Such instructions must then be "interpreted" by the VMM. All other instructions (innocuous instructions) must be executed natively. This kind of VMM is also known as an *execute-to-trap* VMM. It is this ability to execute most of the code directly that enable "efficient" virtualization.

Popek and Goldberg also showed that if the efficiency property is loosened, allowing interpretation of all privileged guest code, a so-called *hybrid* VMM can be constructed for otherwise non-virtualizable architectures.

To specify the conditions for the construction of a hybrid VMM, Popek and Goldberg defined a new subset of sensitive instructions: user-sensitive instructions. An instruction is user-sensitive if there exists a state in unprivileged mode for which the instruction is sensitive. Using the categories defined earlier, the user-sensitive instructions can further be classified as user-control-sensitive and user-location-sensitive instructions[3]. Popek and Goldberg then proved the following theorem:

Theorem 2 (Popek and Goldberg [44])**.** *A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user-sensitive instructions are a subset of the set of privileged instructions.*

Alternatively, DBT can be used to rewrite instructions at run time at an acceptable performance cost—some researchers have proposed dynamic optimization solutions that achieve a performance benefit by overcoming fundamental limitations of static compilation [13]. If the conditions for the construction of a hybrid VMM are met, only privileged guest code must be rewritten. Otherwise, full virtualization can still be achieved by rewriting all instructions, i.e., including unprivileged guest code. Using hybrid VMMs and DBT, a much wider range of computer architectures can be virtualized.

[2] The equivalence property only holds under the assumption that guest software is free of timing dependencies.

[3] There are no user-mode-sensitive instructions as the processor mode is limited to the one (and only) unprivileged mode.

2.3. Prior updates to the model

Dong and Hao [20] have attempted to extend the model by Popek and Goldberg [44] to include interrupts and memory-mapped IO. Because they treat the subject from the view of user-space applications rather than the operating system or the underlying computer architecture, they exclude IO operations initiated by operating systems. Their model introduces a set of possible IO states Γ, similar to the set of machine states Σ, and implicitely redefines instructions as functions with domain and range $\Sigma \times \Gamma$. However, they do not revise the original definitions from Popek and Goldberg at all even though they alter the model on which those definitions are based. It is also unclear whether the proposed model can be applied in practice. Interrupts and exceptions are vital for today's computer software, not only because of IO, but also because they enable communication between operating systems and their applications through software interrupts (system calls), and because they are key to modern memory management techniques such as swapping. However, Dong and Hao only consider interrupts and exceptions in the context of IO. Their treatment of the subject is therefore incomplete.

At the time of writing, we are not aware of any other extensions to the model.

2.4. Advances in computing practice

Over the past 40 years, computer architectures have been significantly extended, use cases have changed, and users' expectations have evolved accordingly. Hence, the model by Popek and Goldberg [44] no longer fits current computing practice.

2.4.1. Memory relocation and protection

The original model allows for a minimal form of memory relocation and protection, using a single combined base location and bounds register. Such a register is capable of relocating non-privileged applications under the control of an operating system running outside a virtualized environment, and can relocate operating systems themselves (and their applications) when running under the control of a VMM. Although computer systems with paged virtual memory were available commercially in 1974 [25], virtual memory was still seen primarily as a technique for optimizing the utilization of expensive physical memory. Most applications were written without directly depending on the virtual memory facilities of the operating system, so a simple relocation scheme, typically found on systems without virtual memory, was adequate to model the behaviour of a real system.

Today's operating system designers, however, exploit virtual memory to offer applications many key facilities such as dynamic linking, shared libraries, and dynamically expandable heap and stack areas. It is no longer realistic to hide the details of these facilities in a discussion of virtualization. Therefore, we will introduce the concept of a *memory map* in the definition of machine state. An operating system will use such maps to define the memory space accessible to applications. A VMM will also use such maps to define the memory space accessible to its guests. Correctness conditions for the latter are derived in Section 3.2.

2.4.2. Timing

A second difference that has arisen is the importance of timing in our use of computers. At the time Popek and Goldberg [44] published their results, much of the computing workload was still processed in batches, with input prepared off-line and output printed for later usage. Modern computing is significantly more time-sensitive—much of our interaction with personal computers and portable electronics is characterised as "soft real-time", in which failure to deliver a response in a timely manner is perceived as a vexatious malfunction.

The model by Popek and Goldberg does not aim for analysing timing-dependent behaviour. It also does not lend itself to study the influence multiple VMs running under the same VMM exert on each other. In this paper, we will also limit the discussion to whether an architecture lends itself to virtualization in general, for which studying the case of a single VM is sufficient, and to how a VMM can be constructed for that architecture. As we have stated earlier, we do not consider deeply embedded systems. This rules out hard real-time systems.

The underlying idea of efficiency in the original paper was that in an efficient system relatively few instructions would trap. This implies that if one wants to retain soft real-time behaviour, VMM interceptions must be bounded. Whether or not a VMM of which all interceptions are bounded can be constructed on a specific architecture, is out of the scope of this paper.

2.4.3. IO, interrupts and exceptions

Most IO in contemporary computer architectures is either memory-mapped (MMIO) or port-mapped (PMIO). IO operations may result in interrupt generation, but generally this is not necessary, unlike what is suggested by Dong and Hao [20].

In an architecture that supports MMIO, IO device registers are mapped into the same address space as RAM. We put forward that resource control is retained if all accesses to IO device registers trap or can be configured to trap a priori. The instructions that perform these accesses will be generic load and store instructions. However, we cannot require all load and store instructions to be privileged, because their sensitivity depends on the address they act upon. Load and store instructions that operate on RAM are clearly innocuous, given that they operate on virtual addresses. Hence, treating all load and store instructions as sensitive violates the efficiency property. Instead, virtual memory can be used to protect device memory, and make all accesses to such memory trap. The efficiency property is retained if memory protection is possible at such granularity that accesses to RAM are not affected.

PMIO is typically used in systems where the memory address space is too small to accommodate IO devices. On other architectures, it may be a relic which has been preserved for backwards compatibility. Communication with devices is carried out using dedicated IO instructions. Such instructions can be seen as controlling resources; they are hence similar to control-sensitive instructions: it is sufficient that all dedicated IO instructions are also privileged to retain classic virtualizability.

3. An updated model

3.1. Machine state

We redefine the machine state concept from Definition 1 based on the features of modern computer architectures. We add the complete set of general-purpose registers and configuration registers (also known as system registers). We introduce paged virtual memory by substituting the relocation-bounds register with a fine-grained memory map. We also extend the machine state with the state of IO devices.

Definition 3. The machine state S is defined by the memory E, the processor mode M, the program counter P, the general-purpose registers G, the configuration registers C, the memory map A, the MMIO device state D^M and the PMIO device state D^P:

$$S \equiv \left\langle E, M, P, G, C, A, D^M, D^P \right\rangle .$$

The first three parameters, namely the memory E, the current processor mode M, and the program counter P, retain their meaning. To generalize the concept of processor mode, we define M_P to be the set of privileged modes[4], and m_U to be the unprivileged mode, such that $M = (M_P \cup \{m_U\})$ and $m_U \notin M_P$. We also introduce four new parameters. G contains the value of all general-purpose registers excluding the program counter[5]. Similarly, C contains the value of all configuration registers. We let E refer to the contents of the entire physical address space. Physical memory is now accessed using the virtual to physical address translation map A. Last but not least, D^M and D^P refer to the state of IO devices. We omit all IO device registers from E.

In our model, instructions that communicate with devices always alter either D^M or D^P. We also exclude any external influences from modifying D^M and D^P during the execution of any instruction. This limitation of the model is required to model instruction behaviour accurately and independently of timing behaviour of devices. Device state may change in between the execution of instructions.

3.2. Address mapping

The address map A is a many-to-one map, meaning that many virtual addresses can correspond to the same physical address. Each entry in A also contains an access permission specifier, which may be used to restrict access to read-only, or can forbid access altogether. Without loss of generality, we can assume that all memory accesses are virtual. When virtual addressing is turned off, no page tables are active and A would be a one-to-one identity map.

Definition 4. An address map A is a set of 3-tuples (v, p, x) in which $v \in V$ represents a virtual address, $p \in P$ represents the physical address v is mapped to, and $x \in X$ represents the access permission specifier. For any $v \in V$, there is at most one 3-tuple in A that contains v.

Definition 5. Let V_A be the set of virtual addresses mapped by A:

$$V_A = \{ v \mid v \in V \land \exists (p, x) \in (P, X) : (v, p, x) \in A \} ;$$

the translation function T_A for the address map A is then defined as:

$$\begin{array}{rccl} T_A : & V_A & \longrightarrow & (P, X) \\ & v & \longmapsto & T_A(v) = (p, x). \end{array}$$

Upon every memory access, the address translation function $T_A(v)$ takes the virtual address v of the access and finds its corresponding physical address p using the address map A. If A does not contain an entry for v or if the access permission specifier x indicates that the requested kind of access is not allowed, a *memory trap* occurs.

An operating system may create its own set of page tables, mapping a set of virtual addresses V_g to a set of physical addresses P_g. When this operating system is virtualized, its physical addresses become virtual in the context of the VMM. The VMM is responsible for mapping guest physical addresses to host physical addresses P_t. Let A_g and A_h denote the sets of all valid mappings in the operating system's page table and the VMM's page table, respectively:

$$\begin{aligned} \forall v_g \in V_g : \exists (p_g, x_g) \in (P_g, X), (p_t, x_t) \in (P_t, X) : \\ T_{A_g}(v_g) = (p_g, x_g) \land T_{A_h}(p_g) = (p_t, x_t). \end{aligned}$$

A typical memory management unit (MMU) can only perform a single address translation in hardware. A double translation, as described above, would require translation by software which in turn would require all guest memory accesses to trap, thereby sacrificing the efficiency property. This problem can be solved by composing the guest and VMM mappings into a set of direct mappings A_s from guest virtual to host physical addresses. Such address map is also known as a *shadow address map* [1, 14, 48].

Creation and maintenance of shadow address maps is one of the most complicated tasks of a VMM. The shadow

[4] Some architectures provide more than one privileged mode. We assume that all such modes are equally privileged.

[5] The ARM architecture offers a set of 16 "general-purpose" registers, which contains the program counter as R15 [23].

address map must contain all virtual addresses mapped by the guest. Since it is common for guests to be relocated in the physical address space by the VMM, a new formal requirement has to be imposed on the allocator. Let E_n be the contents of the physical address space, and let A_g be the address map that translates virtual addresses $v \in V_g$ to physical addresses in E_n on a machine without a VMM present. Let E_v and A_s denote the physical address space and the address map with a VMM installed. The formal requirement is that at any time, every virtual address v_g is mapped by A_s and A_g to the same physical data, or memory traps otherwise:

$$\begin{aligned}\forall v_g \in V_g : \exists (p_g, x_g) \in (P_g, X), (p_t, x_s) \in (P_t, X) :\\ \Big(T_{A_g}(v_g) = (p_g, x_g) \wedge T_{A_s}(v_g) = (p_t, x_s)\\ \Rightarrow E_v[p_t] = E_n[p_g]\Big) \vee x_s \text{ causes a memory trap.} \quad (1)\end{aligned}$$

In addition to the mappings from A_g, a VMM will add its own set of system mappings to A_s. These system mappings comprise the memory space occupied by the hypervisor and MMIO devices. Because the hypervisor and its guests have to coexist in the same address space, mappings may overlap. This is certainly the case for MMIO devices. It is the job of the VMM to maintain separation of guest and VMM resources by setting appropriate access permissions on the overlapping regions in the map A_s so that unprivileged code is denied access.

3.3. *Instruction behaviour*

This section redefines the notions of privileged, sensitive and innocuous instructions using the new machine state model. We adopt Definition 2 that states an instruction is a function that maps one machine state to another. We write $\mathcal{I}$ for the set of all instruction functions, and $\mathcal{S}$ for the set of all possible machine state mapping functions. Because typically not every machine state can be reached through instructions, $\mathcal{I} \subset \mathcal{S}$.

Definition 6. An instruction i is privileged if for any two states $S_1\langle e, m_U, p, g, c, a, d^M, d^P\rangle$ and $S_2\langle e, m_2, p, g, c, a, d^M, d^P\rangle$, where $m_2 \in M_P$ and both $i(S_1)$ and $i(S_2)$ do not memory trap, $i(S_1)$ causes a trap and $i(S_2)$ does not. This trap is referred to as a privileged instruction trap.

All instructions which change the processor mode, modify the system registers, modify the address map or communicate with PMIO devices are control-sensitive. We exclude communication with MMIO devices to prevent treating generic memory instructions as sensitive (see Section 2.4.3).

Definition 7. An instruction i is control-sensitive if there exists a state $S\langle e_1, m_1, p_1, g_1, c_1, a_1, d^M, d_1^P\rangle$ and for $i(S) = \langle e_2, m_2, p_2, g_2, c_2, a_2, d^M, d_2^P\rangle$ we have:

$$(m_1 \neq m_2 \vee c_1 \neq c_2 \vee a_1 \neq a_2 \vee d_1^P \neq d_2^P) \\ \wedge\, i(S) \text{ does not memory trap.}$$

Mode-sensitive instructions behave differently when executed in machine states which differ solely in their mode.

Definition 8. An instruction i is mode-sensitive if, given two states $S_1\langle e, m_1, p, g, c, a, d^M, d^P\rangle$ and $S_2\langle e, m_2, p, g, c, a, d^M, d^P\rangle$ such that for some $m_1 \neq m_2$, and $i(S_1)$ and $i(S_2)$ do not memory trap, for $i(S_1) = \langle e_1, m_1^*, p_1, g_1, c_1, a_1, d_1^M, d_1^P\rangle$ and $i(S_2) = \langle e_2, m_2^*, p_2, g_2, c_2, a_2, d_2^M, d_2^P\rangle$ we have:

$$e_1 \neq e_2 \vee m_1^* \neq m_2^* \vee p_1 \neq p_2 \vee g_1 \neq g_2 \\ \vee c_1 \neq c_2 \vee a_1 \neq a_2 \vee d_1^M \neq d_2^M \vee d_1^P \neq d_2^P.$$

We introduce a new class of sensitive instructions, similar to mode-sensitive instructions. Their behaviour depends on the set of system registers S.

Definition 9. An instruction i is configuration-sensitive if, given two states $S_1\langle e, m, p, g, c_1, a, d^M, d^P\rangle$ and $S_2\langle e, m, p, g, c_2, a, d^M, d^P\rangle$ such that for some $c_1 \neq c_2$, and $i(S_1)$ and $i(S_2)$ do not memory trap, for $i(S_1) = \langle e_1, m_1, p_1, g_1, c_1^*, a_1, d_1^M, d_1^P\rangle$ and $i(S_2) = \langle e_2, m_2, p_2, g_2, c_2^*, a_2, d_2^M, d_2^P\rangle$ we have:

$$e_1 \neq e_2 \vee m_1 \neq m_2 \vee p_1 \neq p_2 \vee g_1 \neq g_2 \\ \vee c_1^* \neq c_2^* \vee a_1 \neq a_2 \vee d_1^M \neq d_2^M \vee d_1^P \neq d_2^P.$$

Popek and Goldberg also introduced the notion of location-sensitive instructions, which were able to bypass memory relocation. An example of a such sensitive instruction is the *Load Real Address* (LRA) instruction from the IBM 360/67 instruction set. In a modern architecture, such instructions would bypass address translation and expose absolute physical addresses. In virtualized systems it is common for guests to be relocated in physical memory by the VMM. Location-sensitive instructions would therefore break the equivalence property. However, we are not aware of any such instructions in modern processor architectures including ARM, MIPS, PowerPC, SuperH and even x86 [4, 32, 35, 40, 41, 42, 49].

At first sight, a new difficulty arises on architectures that have an explicitly visible program counter. Use of the program counter as an operand in any instruction may break the equivalence property if a guest is relocated. However, current memory management techniques enable a VMM to map guests to the same virtual address as they would see when executed natively. Given that requirement (1) in Section 3.2 is met, this will be the case, and instructions that operate on the program counter will have no influence on the virtualizability of a system.

3.4. *Events*

Multiple definitions exist for the terms *interrupt* and *exception*. On ARM, for example, interrupts are seen as a particular subclass of exceptions [4], while the x86 manual describes them as different and unrelated types of *events* [36].

From this point on, we refer to the whole set of interrupts and exceptions as *events*.

Events cause the processor to save its current state and enter an event handler in a privileged mode. Events are either synchronous or asynchronous. A *synchronous* event directly results from the execution of an instruction. Synchronous events are already included in our model as traps. In modern architectures, several kinds of traps exist. They can usually be classified in one of the following categories:

- *Privileged* instruction traps are the result of executing a privileged instruction.
- *Memory* traps are caused by instructions or instruction fetches attempting to access an address that is either invalid or inaccessible with respect to the current machine state S and access bits X.
- *Arithmetic* traps may occur when attempting to perform invalid arithmetic operations (such as division by zero), or when a hardware FPU lacks an implementation for a requested floating point operation, in which case software must emulate the operation.
- *Undefined instruction* traps may occur when executing an instruction which is not recognized by the processor.

An *asynchronous* event may happen at any time, unrelated to the instruction being executed. In practice, all interrupts generated by IO devices are asynchronous. When an asynchronous event happens while the processor is executing an instruction, it will cause the processor to revert or to complete that instruction. If not, the processor would be left in an inconsistent state. Hence, all instruction functions i are independent of such events.

Definition 10. An asynchronous event is a function e that brings a processor from one machine state into another of which the mode is privileged, without and unrelated to the execution of an instruction:

$$\begin{array}{rrcl} e: & \Sigma & \longrightarrow & \{S\langle e, m, p, g, c, a, d^M, d^P\rangle \\ & & & \qquad\qquad |S \in \Sigma \wedge m \in M_P\} \\ & S_x & \longmapsto & e(S_x) = S_y \end{array} .$$

Hence, asynchronous events can be expressed in the same way as instruction execution. We write $\mathcal{E}$ for the set of asynchronous event functions. Because of the constraint on M, $\mathcal{E}$ is a proper subset of $\mathcal{S}$.

3.5. Result

Based on our updated model and its definitions, we conclude that Theorem 1 remains unmodified. However, we need to impose a new formal constraint on the allocator of the VMM (see (1) in Section 3.2) to support paged virtual memory.

The proof of the theorem also requires a subtle update. The consequence of adding asynchronous events to the model is that for any instruction sequence $(i_1, i_2, \ldots i_n)$, in which neither instruction generates a synchronous event, we can no longer guarantee that $S_{k+1} = i_k(S_k)$ $(1 \leq k < n)$, because asynchronous events may occur at any time. The same observation applies to changes in device state due to timing effects or external influences. Although both the model developed by Popek and Goldberg and our extended model do not rely on sequences of instructions, the proof of the theorem that builds upon this model does. More precisely, they are used in the formalisation of the equivalence property as a homomorphism on the set of machine states Σ. However, as we have shown, asynchronous events may be modelled as functions that map one machine state on another similar to instructions. The same reasoning can be applied to asynchronous changes in IO device state. It suffices to generalize the notion of instruction functions $i \in \mathcal{I}$ to functions of the set $\mathcal{S}$ to formalise the equivalence property under the new model.

4. Analysis of the ARM architecture

In this section, we present an analysis of the bare ARMv7-A architecture [4], i.e., without the upcoming virtualization extensions, based on our updated model. An analysis of ARMv7-A including the new extensions follows in section 5.1. ARM provides four different instruction sets: 32-bit (fixed-width) ARM, Thumb-2, Thumb Execution Environment (ThumbEE) and Jazelle. We will omit ThumbEE and Jazelle from our discussion. ThumbEE closely resembles Thumb-2, with few additions and modifications, and is deprecated with the upcoming virtualization extensions. Jazelle's specification is not publicly available, and it cannot be combined with the upcoming virtualization extensions [8].

4.1. Machine state

We recall the definition of machine state in our new model:

$$S \equiv \left\langle E, M, P, G, C, A, D^M, D^P \right\rangle .$$

On ARM [4], multiple equally privileged modes exist; they are system mode (SYS), supervisor mode (SVC), interrupt mode (IRQ), fast interrupt mode (FIQ), abort mode (ABT) and undefined mode (UND). There is only one unprivileged mode: user mode (USR). M is always one of these modes. The mode is altered explicitly by instructions or implicitly when an event arrives.

There are 16 general purpose registers, labelled R0 to R15. We exclude R15 from G, because it represents the program counter P. Some registers are duplicated for different modes; this technique is called *banking*. Unbanked registers are shared between all processor modes. The stack pointer register (R13) and the link register (R14) are banked for all privileged modes; the instances for SYS are shared with USR. Furthermore, FIQ has its own banked set of registers R8 to R12. The set G includes all of the mentioned registers, excluding P.

The currently active processor state is represented in the current program status register (CPSR). It stores the processor mode M, data memory endianness and interrupt mask bits among other fields. There are also five saved program status registers (SPSRs). SPSRs have the same layout: they are copies of the CPSR register used as backup upon entry of an event handler.

Although it sounds logical to include the entire set of program status registers into the set of configuration registers C, there are two pitfalls. Firstly, because the mode field of the CPSR always reflects the current processor mode M, including M in C would render the definition of mode-sensitive instructions (Definition 8) useless. Secondly, part of the CPSR reflects global state, while the other part represents a state specific to the current mode M. The latter part is also exposed to user mode. For example, data memory endianness can be set by software executing in user mode, and will only affect user mode. However, data memory endianness is part of the CPSR.

We clearly cannot add the entire CPSR to the set of configuration registers C, as it would make our analysis overly restrictive. Altering the mode-specific state for execution in one mode from another is always done through the SPSRs. Hence, we can safely omit any mode-specific state from the CPSR in C, but we must include the SPSRs completely.

The set C also contains all of the system control coprocessor (CP15) registers. This coprocessor is used for cache, memory and whole system configuration, in addition to debugging and monitoring. Its registers can be accessed using dedicated coprocessor register read or write instructions.

We also model an event register in the set C. This register is part of the mechanism of halting and resuming instruction execution based on machine specific system events and should not be confused with our usage of the term *event* in this paper for the whole of interrupts and exceptions.

The ARM architecture provides an MMU with support for paged virtual memory[6]. Hence, address translation maps can be implemented using page tables.

All IO on ARM is memory-mapped—PMIO is not supported. Hence, D^P is empty. Interactions with coprocessors are not seen as IO operations.

4.2. *32-bit ARM instruction behaviour*

There are many sensitive instructions on ARM. Below, we provide a detailed analysis of all sensitive instructions, grouped by purpose. The results of our analysis are summarized in Table 1.

Table 1: Sensitive and privileged 32-bit ARM instructions

Instruction	Control	Mode	Conf.	Priv.
CPS	●	●	○	○
LDC	○	●	●	○
LDM (exception return)	●	●	●	○
LDM (user registers)	○	●	○	○
MCR	●	●	○	○
MRC	○	●	●	○
MRS (SPSR)	○	●	●	○
MSR	●	●	○	○
RFE	●	●	○	○
SEV	●	○	○	○
SRS	○	●	●	○
STC	●	●	○	○
STM (user registers)	○	●	○	○
SVC	●	○	○	●
SUBS, ... (exception return)	●	●	●	○
WFE	●	○	●	○
WFI	●	○	●	○

4.2.1. *Coprocessor instructions*

The ARM instruction set contains a number of instructions to interact with coprocessors. In this paper, we limit our discussion to a basic implementation of the ARM architecture without extensions. Hence, only two coprocessors are available in the system: CP14, which is used for debugging and tracing, and CP15, the system control processor. All their registers are part of the set C.

Out of all instructions there are only four that can operate on these coprocessors:

- MCR and STC are control-sensitive because they write data to a coprocessor's registers or memory;

- LDC and MRC are configuration-sensitive because they load data from a coprocessor's registers or memory.

A coprocessor may also deny access from USR mode. Hence, all of the above instructions are also mode-sensitive.

In systems with extra coprocessors, these coprocessors have to be carefully analysed to determine whether any of their registers belong to C. The specification of the coprocessor also determines the set of instructions that can operate on it. This set may not be limited to the instructions discussed above; it may also include any of the following: CDP, CDP2, LDCL, LDC2, LDC2L, MCR2, MCRR, MCRR2, MRC2, MRRC, MRRC2, STCL, STC2 and STC2L. The effect of each of these instructions, including the ones mentioned above, must be studied for each coprocessor individually to determine which instructions are sensitive.

4.2.2. *Event handling*

ARM provides a plethora of instructions for handling events:

- LDM, SUBS, MOVS, MVNS, ADCS, ADDS, ANDS, BICS, EORS, ORRS, RSBS, RSCS and SBCS can all be used to return

[6] The application profile is the only architecture that provides an MMU. Other ARM architecture profiles provide a simpler memory protection scheme that does not support page tables.

from event handlers. Since this updates the processor mode by definition, all these instructions are control-sensitive. These instructions are also configuration-sensitive, because they copy the SPSR into the CPSR.

- Another variant of LDM can be used to load values into USR mode registers from memory. A similar form of STM provides the inverse operation.

- RFE is control-sensitive because it loads values into the program counter (R15) and the CPSR from memory.

- SRS is configuration-sensitive because it stores the value of the link register (R14) and the current SPSR to memory.

All of the above instructions are also mode-sensitive, because their behaviour in USR mode is unspecified.

4.2.3. *Direct modification of system registers*

Some instructions directly read or write to system registers such as the CPSR or one of its banked versions:

- MRS reads from the SPSR, hence it is both configuration-sensitive and mode-sensitive.

- CPS and MSR write to system registers. The former acts as a NOP while the latter is unpredictable when executed in USR mode. Hence, they are both control-sensitive and mode-sensitive.

- SVC is used as system call (or software interrupt) by applications to call the operating system. It unconditionally changes the current mode to supervisor mode (SVC). Hence, it is control-sensitive.

4.2.4. *Sleep and wake up*

In a multiprocessor system, software executing on different processors can communicate using events. The architecture provides two instructions for this purpose: send event (SEV) and wait for event (WFE). Software can also hint the processor that it is waiting for an interrupt or external event through the WFI instruction. While waiting, the processor may go to a low-power state. After an external event or interrupt occurs, the one-bit abstract event register will be asserted. The processor wakes up and this bit is cleared. Since the abstract event register is part of the system state, the SEV, WFE and WFI instructions are control sensitive.

4.3. *Thumb-2 instruction behaviour*

The Thumb-2 instruction set provides more or less the same instructions as the 32-bit ARM instruction set. Moreover, the set of sensitive Thumb-2 instructions is a proper subset of the set of sensitive ARM instructions, so no elaborate discussion is required. The results of our analysis are summarized in Table 2.

Table 2: Sensitive and privileged Thumb-2 instructions

Instruction	Control	Mode	Conf.	Priv.
CPS	●	●	○	○
LDC	○	●	●	○
MCR	●	●	○	○
MRC	○	●	●	○
MRS (SPSR)	○	●	●	○
MSR	●	●	○	○
RFE	●	●	○	○
SEV	●	○	○	○
SRS	○	●	●	○
STC	●	●	○	○
SVC	●	○	○	●
SUBS (exception return)	●	●	●	○
WFE	●	○	●	○
WFI	●	○	●	○

4.4. *Conclusion*

Investigating the set of sensitive instructions in the ARM and Thumb-2 instruction sets in tables 1 and 2, it is clear that both instruction sets contain sensitive instructions that are not privileged. Based on our findings in Section 3.5, we conclude that the ARM architecture is not classically virtualizable.

5. Full virtualization in practice

The criteria for virtualizability introduced by Popek and Goldberg [44] and extended in this paper are sufficient but not necessary to construct an efficient VMM for a particular architecture. Advances in the construction of VMMs have enabled full virtualization on architectures that fail these criteria. However, pure software solutions using binary rewriting or emulation are typically deemed to introduce too much overhead or to be too complex.

Hardware vendors have also adapted to the need for virtualization support, and architectures that were formerly not classically virtualizable such as x86 have already been made virtualizable [2, 50]. ARM is following the same path with its upcoming virtualization and large physical address (LPA) extensions for ARMv7-A and ARMv7-R [5, 6, 7, 8].

In this section, we discuss and compare both software and hardware approaches. For the latter, we analyse ARM's upcoming extensions.

5.1. *Hardware support for full virtualization*

The upcoming virtualization and LPA extensions introduce a new processor mode, HYP, and a number of new instructions. They also impact the behaviour of many existing instructions, and modify the mechanisms for interrupt handling, memory management and performance monitoring.

The new HYP mode is more privileged than the original set of privileged modes (SVC, SYS, ABT, UND, IRQ, FIQ); the latter is now referred to as the set of *kernel modes*.

HYP mode enables a VMM to run below the operating system level without forcing the operating system kernel to run unprivileged. Instead, the operating system kernel can use all kernel modes transparently, as if no VMM was present. Events for the VMM are handled in the HYP mode, instead of the traditional *exception modes* (ABT, UND, IRQ, FIQ), which are a subset of the kernel modes. This significantly reduces the set of sensitive instructions.

In order to make all sensitive instructions executed in any of the kernel modes trap to HYP mode, the virtualization extensions add *configurable traps* to the architecture. A VMM can then make certain instructions trap as required. When running just one operating system without a VMM, the traps will be disabled.

A quick analysis of the original set of sensitive instructions confirms the implications stated above:

- Coprocessor instructions can be configured to trap. These traps provide coarse-grained control over accesses to coprocessors CP0 to CP13. There are more fine-grained controls for the remaining coprocessors—the debug and execution environment support coprocessor (CP14) and the system control coprocessor (CP15).

- Memory access and event handler return instructions are no longer sensitive, since a guest running on a VMM with hardware extensions can use the exception modes in the same way as for the non-virtualized case.

- Instructions that directly modify system state now act on a guest's state, rather than on the entire system state. Hence, they are no longer sensitive.

- Instructions that deal with external events are adapted to work with a per-guest interrupt state. Each guest has its own virtual interrupts. Both `WFE` and `WFI` have independent configurable traps. These traps can be used as hints by the VMM to schedule other guests. However, the `SEV` instruction *cannot* be made to trap.

It may seem surprising that `SEV` cannot be made to trap. It is the only sensitive instruction that remains unprivileged with the new hardware extensions. However, none of the sleep and wake-up instructions can cause functionally incorrect behaviour of the VMM [4]. The configurable traps for `WFE` and `WFI` are provided so that a VMM is able to detect when a guest is idle. The `SEV` instruction cannot provide such useful information to the VMM.

Hence, a VMM can be constructed that executes VMs until they trap to HYP mode. All other additions by the virtualization and LPA extensions are not strictly necessary but speed up common VMM tasks and reduce the code size of the VMM. For example, LPA adds a second stage to the address translation mechanism in the MMU. This mechanism makes the hardware capable of combining a translation table for the guest with a translation table for the VMM, eliminating the need for software-based shadow maps.

5.2. Dynamic binary translation

Full virtualization can also be supported without modifications to the hardware; this is typically achieved through dynamic binary translation (DBT), sometimes also called software dynamic translation. The concept of a DBT VMM evolved from the theory of Popek and Goldberg's hybrid VMM. An analysis according to our model remains useful, because it can be used to determine whether an architecture is suitable for the construction of a DBT VMM. Furthermore, the analysis will reveal which instructions will need to be rewritten.

In a hybrid VMM, all instructions normally executed in a privileged mode are interpreted by the VMM. Instructions normally executed in unprivileged mode are executed natively [44]. In a traditional software stack, a hybrid VMM would interpret the OS but execute applications natively. Popek and Goldberg [44] proved that a hybrid VMM can be constructed if all user-sensitive instructions are also privileged. In our model, user-sensitive instructions are defined as follows:

Definition 11. An instruction i is user-sensitive if there exists a state $S\langle e, m_U, p, g, c, a, d^M, d^P\rangle$ for which $i(S)$ is control-sensitive and/or configuration-sensitive.

A DBT VMM can achieve a performance benefit over interpretation by rewriting instruction sequences at run time [13]. The basic idea is to execute as many rewritten instructions as possible natively, without intervention of the VMM. Sensitive instructions must be rewritten such that they trap to the interpreter of the VMM. Because the VMM needs to keep track of the execution of its VMs, it will also need to rewrite control flow instructions.

5.2.1. DBT for the ARM architecture

DBT has been used successfully to virtualize the Intel x86 architecture by VMware, Microsoft, QEMU, VirtualBox and others [16, 30, 52, 53]. The earliest DBT VMM dates from 1999, several years before the emergence of hardware extensions for virtualization in 2005 [2, 50].

Early x86 virtualization solutions used DBT, as paravirtualization was yet to be invented. Since virtualization for embedded systems only became a topic of interest much later, all efforts on virtualizing the ARM architecture were geared towards paravirtualization. At the time of writing, to the best of our knowledge no VMM with full virtualization using DBT has been constructed for the ARM architecture.

In order to construct a DBT VMM for ARM, it is required to determine how much of the guest code must be rewritten: if all user-sensitive instructions are also privileged, only guest privileged code must be translated, otherwise, all code must be translated. We can extend our analysis from Section 4 to analyse user-sensitivity. As it turns out, there are four user-sensitive instructions, shared by both ARM and Thumb:

- the supervisor call instruction SVC, because it always changes the processor mode to SVC;
- the sleep and wake-up instructions SEV, WFE and WFI.

All other sensitive instructions either act as NOP or exhibit innocuous behaviour when executed in USR mode. Of all user-sensitive instructions, only SVC is privileged. Hence, ARM does not meet Popek and Goldberg's conditions for a hybrid VMM either.

In practice, the sleep and wake-up instructions are merely hints for a processor. Even a masked interrupt will wake up a sleeping processor. Furthermore, if any interrupts are pending, the processor will not sleep either [4]. As such, not intercepting these operations cannot lead to resource control violations. In other words, we can ignore them and still obtain functional correctness. When we effectively do ignore the presence of unprivileged sleep and wake-up instructions, we can construct a DBT VMM that only translates privileged guest code. The downside of this approach is that the VMM cannot intercept sleep instructions in unprivileged guest code, which could otherwise be used by the VM to inform the VMM that the VM is idle. The same problem applies to paravirtualisation solutions for ARM: because only guest privileged code (such as OS kernels) is altered, they have to make the same assumption. They hence suffer from the same deficiency.

Using DBT also requires the VMM to keep track of guest control flow. On ARM, the program counter is explicitly visible as R15 and can be altered by several instructions, including ALU instructions. This complicates the design of an instruction decoder for DBT.

Furthermore, the program counter can be used as source operand for even more instructions [4, 26, 43]. Because the width of instructions is fixed at 16 or 32 bits, absolute address operands cannot be encoded. Therefore, distributed literal pools—pools of data embedded in code—are used which are accessed using PC-relative addressing. Other architectures offer instructions that can encode absolute addresses in full, such as on x86, or use a single *global offset table* instead of distributed embedded data pools, such as on Alpha [39].

We call instructions that depend on the value of the program counter *virtual-location-sensitive*. Formally, they are defined as follows:

Definition 12. An instruction i is virtual-location-sensitive if, given two states $S_1\langle e, m, p, g, c, a, d^M, d^P\rangle$ and $S_2\langle e, m, p+r, g, c, a, d^M, d^P\rangle$ such that for some offset $r \in \mathbb{Z}^*$, and $i(S_1)$ and $i(S_2)$ do not memory trap, for $i(S_1) = \langle e_1, m_1, p_1, g_1, c_1, a_1, d_1^M, d_1^P\rangle$ and $i(S_2) = \langle e_2, m_2, p_2, g_2, c_2, a_2, d_2^M, d_2^P\rangle$ we have:

$$\begin{aligned} e_1 \neq e_2 \vee m_1 \neq m_2 \vee p_1 \neq (p_2 - r) \vee g_1 \neq g_2 \\ \vee c_1 \neq c_2 \vee a_1 \neq a_2 \vee d_1^M \neq d_2^M \vee d_1^P \neq d_2^P. \end{aligned}$$

Virtual-location-sensitive instructions cannot be relocated in the *virtual* address space; as opposed to Popek and Goldberg's location-sensitive instructions which cannot be relocated in the *physical* address space. Virtual-location-sensitive instructions are innocuous to execute-to-trap VMMs if requirement (1) on the address map is met (see Section 3.2).

Virtual-location-sensitive instructions wreak havoc with DBT VMMs because privileged guest code cannot be executed in place, as opposed to unprivileged guest code. In-place translation and execution is impossible for several reasons: firstly, a VMM cannot prevent a guest from observing the alterations to its code, and secondly, the size of the translated code will not necessarily match the size of the original code. Instead, code is translated to a cache located elsewhere in the virtual address space. The program counter observed by the translated code will be incorrect unless the VMM also intercepts virtual-location-sensitive instructions. Optimization techniques can be used to rewrite virtual-location-sensitive instructions to equivalent non-sensitive instruction sequences, instead of merely replacing them by a trapping instruction [43].

5.2.2. Comparing DBT with hardware extensions

One of the common arguments against the use of DBT in a VMM is that it introduces substantial complexity. In reality, hardware extensions introduce similar complexity, but on the hardware level. Although they make it possible to construct a smaller VMM, thereby decreasing the potential for bugs and vulnerabilities, the design complexity of hardware extensions is illustrated by the fact that early implementations were outperformed by software [1].

An argument that speaks in favour of DBT is its versatility. Virtualization is often considered as a solution to software portability issues across different architectures. Hardware extensions merely recreate this problem at a different level. On the contrary, DBT is more versatile since it *transparently* enables:

- **Optimizations across the border between OS and applications**: DBT can be used to optimize and reduce context switching between an OS and its applications under a VMM, as many operations become redundant when executed on virtual hardware [1].
- **Legacy emulation and optimization**: DBT can be used as a glue layer to emulate legacy hardware platforms and to resolve backward incompatibility between generations of architectures [18]. Such emulation is useful as today's hardware has become more flexible than software [21, 27]: system-on-chips have

an average lifetime of 4 years, while software stacks often have to last much longer. Because of implementation differences between system-on-chips and evolutions in hardware architecture, software must be continuously rewritten and recertified for newer platforms.

- **Full system instrumentation**: entire software stacks may be instrumented from a VMM similar to the approach used in PinOS [17].
- **Load balancing in heterogeneous multi-core systems**: DBT can be used to translate code from one core to another if cores have different ISAs [33, 55]. If cores share the same ISA, such as in ARM's big.LITTLE [9], hardware extensions remain a feasible approach to virtualization. But even in this case, DBT can potentially improve performance by dynamically optimizing code for the micro-architectural features of the target architecture, such as a static branch predictor in little cores.

6. Conclusions

Despite the popularity of paravirtualization in today's embedded systems, full virtualization remains important. The theory introduced by Popek and Goldberg is still useful for determining whether an architecture is suitable for the construction of efficient VMMs for full virtualization, which are based on the execute-to-trap principle. However, their model does not take into account features often found in modern architectures. Therefore we have extended their model with paged virtual memory by introducing the concept of an address map, and we derived a new formal constraint for the correctness of such maps. We also studied the effect of IO and events, and updated the model, definitions and results accordingly.

Our model can be applied to analyse modern computer architectures. We have demonstrated this in a formal analysis of ARMv7-A, which proved to be not classically virtualizable. Nevertheless, modern techniques in the construction of VMMs such as DBT can enable full virtualization on ARM. DBT does not require neither hardware changes nor guest modification and provides a solution to the lack of classic virtualizability on architectures such as ARM. In other architectures, DBT has already been shown to be able to match or even outperform the hardware assisted virtualization approach. However, it comes at a price of software implementation complexity and increased memory footprint. On the other hand, DBT technology provides more further development possibilities like cross architecture virtualization, legacy software stack support and better heterogeneous multicore system utilisation.

In the mean time, industry has started to adapt to the interest in full virtualization. Hardware virtualization extensions attempt to make ARM compatible with the Popek and Goldberg classic virtualization requirements, and implement certain parts of the VMM functionality in hardware for efficiency reasons sacrificing portability. Utilising the hardware functionality can reduce VMM design complexity at the cost of moving the complexity to the design of the hardware layer.

Performance measurements of fully virtualized ARM systems cannot yet be collected, as the market is still awaiting the first devices to implement the upcoming virtualization extensions, and we are not aware of any software solutions at the time of writing. However, we have shown that it is important to evaluate all methods of virtualization not only on grounds of performance, but also functionality, versatility and costs.

7. Future work

Although our model can be used to determine whether modern computer architectures are suitable for the construction of efficient execute-to-trap VMMs, it does not lead to any conclusions on whether or not an architecture can support more than one guest. For example, routing of asynchronous events to VMM and guests may prove to be difficult if such events have priorities, as VMMs have to make sure lower-priority guests cannot block higher-priority guests.

Another issue that remains untouched by our model is multi-threading of privileged code, which may happen on multi-core and multi-threading architectures.

We are currently working on a full virtualization solution for ARM based on DBT, which will serve as a research platform for the opportunities discussed in Section 5.2.

References

[1] K. Adams, O. Agesen, A comparison of software and hardware techniques for x86 virtualization, in: Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XII, ACM, New York, NY, USA, 2006, pp. 2–13.

[2] AMD, AMD64 Virtualization Codenamed "Pacifica" Technology – Secure Virtual Machine Reference Manual, 3.01 edition, 2005.

[3] Z. Amsden, D. Arai, D. Hecht, A. Holler, P. Subrahmanyam, VMI: an interface for paravirtualization, in: Proceedings of the Linux Symposium, volume 2 of *2006 Linux Symposium*, pp. 371–386.

[4] ARM Architecture Group, ARM® Architecture Reference Manual: ARM®v7-A and ARM®v7-R edition – errata markup, ARM Limited, ARM DDI 0406B_ errata_ 2010_ Q3 (ID100710) edition, 2010.

[5] ARM Architecture Group, ARM® Generic Timer Specification, ARM Limited, PRD03-GENC-009660 9.0x edition, 2010.

[6] ARM Architecture Group, ARM® Performance Monitoring Architecture version 2 Virtualization Extensions, ARM Limited, DSA09-PRDC-010447 6.0x edition, 2010.

[7] ARM Architecture Group, Large Physical Address Extensions Specification, ARM Limited, PRD03-GENC-008469 15.0x edition, 2010.

[8] ARM Architecture Group, Virtualization Extensions Architecture Specification, ARM Limited, PRD03-GENC-008353 14.0x edition, 2010.

[9] ARM Limited, big.LITTLE processing, `http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php`, 2011.

[10] F. Armand, J. Berniolles, J.L. Lawall, G. Muller, Automating the port of Linux to the VirtualLogix hypervisor using semantic patches, in: 4th European Congress ERTS EMBEDDED REAL TIME SOFTWARE, ERTS 2008, pp. 1–7.

[11] F. Armand, M. Gien, A practical look at micro-kernels and virtual machine monitors, in: 6th IEEE Consumer Communications and Networking Conference, CCNC 2009, IEEE, Piscataway, New Jersey, USA, 2009, pp. 1–7.

[12] B Labs Ltd., Codezero project overview, `http://www.l4dev.org/codezero_overview`, 2010.

[13] V. Bala, E. Duesterwald, S. Banerjia, Transparent dynamic optimization: the design and implementation of Dynamo, Technical Report, Hewlett Packard Laboratories Technical Report HPL-1999-78, 1999.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, SIGOPS Operating Systems Review 37 (2003) 164–177.

[15] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, B. Zoppis, The VMware mobile virtualization platform: is that a hypervisor in your pocket?, SIGOPS Operating Systems Review 44 (2010) 124–135.

[16] F. Bellard, QEMU, a fast and portable dynamic translator, in: Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track, USENIX '05, USENIX Association, Berkeley, CA, USA, 2005, pp. 41–46.

[17] P.P. Bungale, C.K. Luk, PinOS: a programmable framework for whole-system dynamic instrumentation, in: Proceedings of the 3rd international conference on Virtual Execution Environments, VEE '07, ACM, New York, NY, USA, 2007, pp. 137–147.

[18] C. Cifuentes, V. Malhotra, Binary translation: static, dynamic, retargetable?, in: Proceedings of the 1996 International Conference on Software Maintenance, ICSM '96, IEEE, 1996, pp. 340–349.

[19] CORDIS, Report on the EC Workshop on Virtualisation, Consultation Workshop "Virtualisation in Computing", `ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/computing/report-on-the-ec-workshop-on-virtualisation_en.pdf`, 2009.

[20] H. Dong, Q. Hao, Extension to the model of a virtualizable computer and analysis on the efficiency of a virtual machine, in: Second International Conference on Computer Modeling and Simulation, volume 2 of *ICCMS 2010*, IEEE Computer Society, Los Alamitos, California, USA, 2010, pp. 503–507.

[21] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, M. Valero, The HiPEAC vision, 2010.

[22] D.R. Ferstay, Fast Secure Virtualization for the ARM Platform, Master's thesis, The University of British Columbia, Faculty of Graduate Studies (Computer Science), 2006.

[23] S. Furber, ARM system-on-chip architecture, ARM system-on-chip architecture, Addison-Wesley, Boston, Massachusetts, USA, 2000, second edition, p. 39.

[24] J. Gallard, A. Lèbre, G. Vallée, C. Morin, P. Gallard, S.L. Scott, Refinement proposal of the Goldberg's theory, in: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP '09, Springer-Verlag, Berlin - Heidelberg, Germany, 2009, pp. 853–865.

[25] R.P. Goldberg, Architecture of virtual machines, in: Proceedings of the workshop on virtual computer systems, ACM, New York, NY, USA, 1973, pp. 74–112.

[26] K. Hazelwood, A. Klauser, A dynamic binary instrumentation engine for the ARM architecture, in: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, CASES '06, ACM, New York, NY, USA, 2006, pp. 261–270.

[27] T. Heinz, R. Wilhelm, Towards device emulation code generation, in: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '09, ACM, New York, NY, USA, 2009, pp. 109–118.

[28] G. Heiser, The role of virtualization in embedded systems, in: Proceedings of the 1st workshop on Isolation and Integration in Embedded Systems, IIES '08, ACM, New York, NY, USA, 2008, pp. 11–16.

[29] G. Heiser, The Motorola Evoke QA4—A Case Study in Mobile Virtualization, Technology White Paper, Open Kernel Labs, 2009.

[30] J. Honeycutt, Microsoft Virtual PC 2004 Technical Overview, 2003.

[31] J.Y. Hwang, S.B. Suh, S.K. Heo, C.J. Park, J.M. Ryu, S.Y. Park, C.R. Kim, Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones, in: 5th IEEE Consumer Communications and Networking Conference, CCNC 2008, IEEE, Piscataway, New Jersey, USA, 2008, pp. 257–261.

[32] IBM, PowerPC® Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors, International Business Machines Corporation, 3.0 edition, 2005.

[33] IBM, PowerVM Lx86 for x86 Linux applications, 2011. http://www.ibm.com/developerworks/linux/lx86/index.html.

[34] H. Inoue, A. Ikeno, M. Kondo, J. Sakai, M. Edahiro, VIRTUS: a new processor virtualization architecture for security-oriented next-generation mobile terminals, in: Proceedings of the 43rd annual Design Automation Conference, DAC '06, ACM, New York, NY, USA, 2006, pp. 484–489.

[35] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z, Intel Corporation, 2011.

[36] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide, Intel Corporation, 2011.

[37] S.M. Lee, S.B. Suh, J.D. Choi, Fine-grained I/O access control based on Xen virtualization for 3G/4G mobile devices, in: Proceedings of the 47th Design Automation Conference, DAC '10, ACM, New York, NY, USA, 2010, pp. 108–113.

[38] J. LeVasseur, V. Uhlig, Y. Yang, M. Chapman, P. Chubb, B. Leslie, G. Heiser, Pre-virtualization: soft layering for virtual machines, in: 13th Asia-Pacific Computer Systems Architecture Conference, ACSAC 2008, IEEE Computer Society, Los Alamitos, California, USA, 2008, pp. 1–9.

[39] J.R. Levine, Linkers and Loaders, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[40] MIPS, MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set, MIPS Technologies, Inc., MD00086, 3.02 edition, 2011.

[41] MIPS, MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set, MIPS Technologies, Inc., MD00087, 3.02 edition, 2011.

[42] MIPS, MIPS® Architecture for Programmers Volume II-B: The microMIPS32™ Instruction Set, MIPS Technologies, Inc., MD00582, 3.05 edition, 2011.

[43] R.W. Moore, J.A. Baiocchi, B.R. Childers, J.W. Davidson, J.D. Hiser, Addressing the challenges of DBT for the ARM architecture, in: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '09, ACM, New York, NY, USA, 2009, pp. 147–156.

[44] G.J. Popek, R.P. Goldberg, Formal requirements for virtualizable third generation architectures, Communications of the ACM 17 (1974) 412–421.

[45] M. Rosenblum, VMware's virtual platform: a virtual machine monitor for commodity PCs, in: Hot Chips 11 (1999).

[46] R. Russell, virtio: towards a de-facto standard for virtual I/O devices, SIGOPS Operating Systems Review 42 (2008) 95–103.

[47] B. Smith, ARM and Intel battle over the mobile chip's future, Computer 41 (2008) 15–18.

[48] J. Smith, R. Nair, Virtual Machines: Versatile Platforms for Systems and Processes, The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2005.

[49] SuperH, SuperH™ (SH) 64-Bit RISC Series: SH-5 CPU Core, Volume 1: Architecture, SuperH, Inc., 05-cc-10001, v1.0 edition, 2002.

[50] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, L. Smith, Intel virtualization technology, Computer 38 (2005) 48–56.

[51] P. Varanasi, G. Heiser, Hardware-supported virtualization on ARM, in: Proceedings of the 2nd ACM SIGOPS Asia-Pacific Work-

shop on Systems, APSys 2011, ACM, New York, NY, USA, 2011, pp. 11:1–11:5.

[52] VMware, Understanding full virtualization, paravirtualization, and hardware assist, 2007. White paper.

[53] J. Watson, VirtualBox: bits and bytes masquerading as machines, Linux Journal 2008 (2008).

[54] A. Whitaker, M. Shaw, S.D. Gribble, Denali: Lightweight Virtual Machines for Distributed and Networked Applications, Technical Report 02-02-01, University of Washington, Seattle, Washington, USA, 2002.

[55] Y. Wu, S. Hu, E. Borin, C. Wang, A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing, in: 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2011, IEEE, IEEE Computer Society, Piscataway, New Jersey, USA, 2011, pp. 236–245.