Hardware-Supported Virtualization on ARM

Prashant Varanasi
prashant.varanasi@gmail.com

Gernot Heiser
gernot@nicta.com.au

NICTA, University of New South Wales and Open Kernel Labs
Sydney, Australia

ABSTRACT

ARM is the dominant processor architecture for mobile devices and many other high-end embedded systems. Late last year ARM announced architectural support for virtualization, which will allow execution of unmodified guest operating system binaries. We have designed and implemented what we believe is the first hypervisor supporting pure virtualization using those hardware extensions and evaluated it on simulated hardware. We describe our approach and report our initial experience with the architecture.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design

Keywords

Hypervisors, virtual machines, architecture, hardware support, ARM

1. INTRODUCTION

Virtualization, formerly mostly at home in data centres and enterprise computing infrastructure, is now spreading to embedded systems, driven by cost and security concerns [Hei08, Kro09]. ARM, the dominant architecture of high-end processors for mobile devices, is not trap-and-emulate virtualizable. This means that virtualization of ARM processors requires binary translation or para-virtualization. Binary translation is generally too resource intensive for mobile devices, which is why all known commercial and research hypervisors for ARM use this approach. Para-virtualization has the drawback of high engineering cost for having to adapt each supported operating system (OS) to the hypervisor-specific platform interface.

In the server world, which is dominated by the (also not trap-and-emulate virtualizable) x86 architecture, the virtualization boom lead hardware extensions to support virtualization [NSL$^+$06]. These allow running an unmodified, native OS binary in a virtual machine (VM) with minimal performance degradation, and greatly simplify the implementation of hypervisors and reduce run-time overheads.

The same is now happening with embedded processors: last year, ARM announced virtualization extensions for their architecture [ARM10], along similar lines as the manufacturers of x86 processors.

In this paper we present the first hypervisor which uses these extensions to support pure virtualization on ARM, and is able to run multiple concurrent unmodified Linux guests. We report on our experience with using the new extensions. Unfortunately, only extremely limited performance evaluation is possible, as the hardware extensions are presently only available in a simulator which is not timing-accurate.

The rest of the paper is structured as follows. Section 2 outlines existing work. Section 3 presents an overview of the ARM architecture, the virtualization extensions, and a comparison to x86 approach. Section 4 outlines the design and Section 5 presents the implementation of our hypervisor. We show TCB size and indicative performance numbers for our hypervisor in Section 6. We discuss our experience with the extensions in Section 7, and draw our conclusions in Section 8.

2. RELATED WORK

Commercial virtualization solutions for ARM platforms are provided by Open Kernel Labs [OKL11], VMware [VMwa10] and Red Bend Software [RedB10], these all use para-virtualization. Green Hills Software's Integrity product [Gree10] uses the TrustZone features of the ARM architecture to run a native guest binary, but architecture limitations restrict this to a single guest.

A port of Xen to ARM was performed by Samsung [HSH$^+$08], but performance is poor: a Linux guest runs at about half of native speed. In contrast, the OKL4 microvisor from OK Labs, which is the only commercial product for which performance data is available, exhibits overheads which are about an order of magnitude lower [HL10]. NOVA [SK10] is a hypervisor for x86 which, like the OKL4 microvisor and our current design, uses a microkernel architecture aimed at minimising the *trusted computing base* TCB of virtual machines (VMs).

Fisher-Ogden presented a thorough analysis of virtualization extensions for x86 from Intel and AMD [FO06]. Adams and Agesen [AA06] found that binary translation outperformed pure virtualization, but this evaluation was completed before the hardware extensions included MMU virtualization. A later evaluation [Bha09] found MMU virtualization significantly reduced overheads, especially when using large pages. The ARM virtualization extensions already include MMU virtualization.

3. ARM ARCHITECTURE

The ARM architecture has evolved over the decades. Here we focus on the latest version, v7, which is the one for which the virtualization extensions are specified.

3.1 Overview

ARM is a 32-bit RISC architecture, featuring 16 general-purpose (GP) registers (which includes the program counter). There is one unprivileged processor mode (user) and six privileged kernel modes. All kernel modes have the same level of privilege, they differ in the kind of exception which forces their entry, the exceptions allowed while executing in them, and the number of banked registers.

The architecture supports a feature called *TrustZone*, which provides an orthogonal processor mode, called *secure mode*. Hardware resources (memory and on-chip devices) are configured to be accessible always or only in secure mode. A super-privileged mode, called *monitor mode*, is used to switch between secure and insecure mode. On power-up the processor enters secure kernel mode. TrustZone can be used to run an unmodified native OS binary next to other native code, similar to virtualization. This is achieved by running the guest in non-secure mode and all other code in secure mode. Unlike real virtualization, this only supports a single guest.

The standard "ARM" instruction set uses 32-bit instructions. Notable features are predication of all instructions, and a barrel shifter which supports complex indexing and address-register updates. A system co-processor contains the MMU, cache control and the performance-monitoring unit (PMU). Further (up to 15) co-processors can be used to implement optional functionality, such as the FPU. There are two further instruction sets: Thumb-2, which uses variable-width (16- and 32-bit) instructions, achieves higher code density at a performance close to that of the ARM instruction set. A third instruction set, Jazelle, is designed for efficient execution of Java bytecode.

There are two levels of on-chip cache: a virtually indexed, physically tagged split I/D-L1, and a unified physically-addressed L2. The TLB is hardware-loaded from a two-level page table (PT) with 32-bit entries. TLB entries are tagged with an 8-bit *address-space ID* (ASID). Supported page sizes are 4KiB, 64KiB and 1 MiB. I/O is memory-mapped.

The architecture defines a *generic interrupt controller* (GIC) which contains two parts. The per-CPU GIC *CPU interface* performs interrupt priority masking and preemption handling, while the global *distributor* receives interrupts and controls interrupt priorities, enabling and the CPU interface to which it is routed. Software acknowledges an interrupt to the interface.

A number of unprivileged instructions reveal privileged information, thus preventing ARM from being trap-and-emulate virtualizable. For example, the `cps` (change processor state) instruction is silently ignored in user mode. In addition, while traps on ARM are relatively cheap (kernel entry- and exit costs are about a dozen cycles on ARM, compared to hundreds of cycles on x86), pure virtualization, if it was possible, would still have a high overhead. The reason is that modern OSes tend to perform frequent manipulations of privileged state, such as processor status and page table.

3.2 Virtualization extensions

The virtualization extensions [ARM10] to the ARM architecture are superficially similar to those for x86, in that they provide a new processor mode and a number features to improve performance.

The extensions only apply to non-secure mode. They introduce a new processor mode, *hyp* mode, which is more privileged than the existing non-secure kernel modes. This leaves the existing kernel and user modes for unmodified guest OSes and applications, as shown in Figure 1.

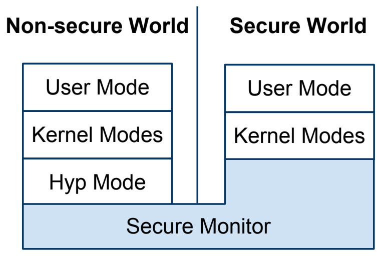

Figure 1: ARM processor modes

Hyp mode is entered from kernel mode via a new instruction (`hvc`), and optionally on a configurable set of exceptions from user or kernel mode. It has banked registers, as well as additional hyp-only registers for system configuration and information on the event which caused entry of hyp mode. There is a hyp-only *virtual machine identifier (VMID)* register. TLB entries are tagged with the VMID, which supports co-existence of mappings from multiple guests and thus eliminates the need to flush the TLB on a world switch.

The hypervisor's own PTs (for translating guest-physical as well as hypervisor-virtual addresses) use a new, wider format which supports 64-bit physical addresses (and has the potential to support 64-bit virtual addresses in the future). It increases the largest page size from 1 MiB to 2 MiB. The new format can optionally be used by the guest as well.

ARM supports a number of mechanisms for reducing the cost of pure virtualization, which we discuss now.

3.2.1 *Configurable traps*

Many exceptions can be configured to trap either into the hypervisor or the guest kernel. This drastically reduces the frequency of hypervisor entries, e.g. by configuring the syscall trap to be handled directly by the guest. Interrupt traps are configured globally, so either all interrupts invoke the hypervisor, or all are delivered directly to the guest of the presently active VM.

3.2.2 *Emulation support*

Load and store instructions are not inherently virtualization-sensitive, but become sensitive when operating on privileged data (eg. device registers). The hypervisor must decode such an instruction and emulate it. The overhead of emulation is not just the extra instructions executed (which include translating guest physical to physical addresses) but also the D-cache miss generated when loading the offending instruction (despite the fact that it has already been fetched into the instruction register and the I-cache). ARM's emulation support in most cases eliminates both the load and the software decode, by keeping the relevant information in hypervisor registers (source or target registers, whether it was a load or a store, the size of the data item to be transferred etc.)

3.2.3 *Second-stage translation*

Similar to extended PTs on x86, ARM supports two-stage address translation: guest-virtual to guest-physical (called *intermediate physical* by ARM) followed by guest-physical to physical. On a TLB miss in non-hyp mode, the hardware page-table walker traverses first the guest and then the hypervisor PT, and constructs a TLB entry representing the guest-virtual to physical translation. While this requires traversal of four levels of page tables, this can be reduced to three if the hypervisor uses 2 MiB superpages. Obviously, only a single translation stage is used when running in hyp mode.

3.2.4 *Virtual interrupts*

In order to avoid emulation of the interrupt controller (which would add significant complexity and require frequent traps into hyp mode), ARM introduced the concept of virtual interrupts. It is supported by a new hardware component, the *virtual CPU (VCPU) interface*. This can be mapped into the guest as the GIC CPU interface, and can be used by the guest to acknowledge and clear interrupts without trapping into the hypervisor. The hypervisor must still emulate the interrupt distributor; all guest accesses to it trap. This is not expected to cause performance issues, as the distributor is normally only accessed at boot time (or module load time) to register drivers for particular interrupts and route them to specific (virtual) CPUs.

If interrupts are configured to be handled by the hypervisor, the hypervisor can explicitly forward the interrupt to the current guest by raising the appropriate virtual interrupt on the guest's VCPU interface. A virtual interrupt can be linked to the physical interrupt, in which case the physical interrupt will be cleared without hypervisor intervention when the guest clears its virtual interrupt. The sequence of events is shown in Figure 2. The hardware drops the current interrupt priority during virtual interrupt processing, in order to allow further interrupts of the same priority, which may be destined for other VMs.

3.3 Comparison to x86

Unlike x86, where VT-x root mode is orthogonal to the existing protection rings, ARM's hyp mode is strictly more privileged than the ex-

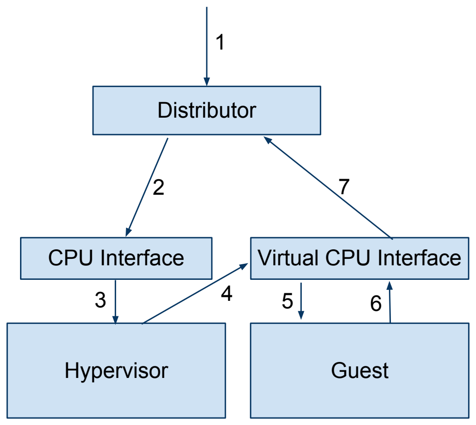

Figure 2: Virtual interrupt processing (actions 4 and 6 are initiated by software).

isting kernel modes. ARM requires the hypervisor to save guest register state, while on x86 this is done automatically by hardware. Second-stage translation and TLB tagging with a VMID are similar. While both architectures allow the hypervisor to inject interrupts into a running guest, ARM allows the guest to acknowledge, clear and mask interrupts without trapping; on x86 these trap into the hypervisor.

ARM's emulation support mostly frees the hypervisor from having to load and decode sensitive instructions. In contrast, the CISC nature of the x86 ISA does not support such a simple and elegant solution, and requires a complete ISA emulator in the hypervisor, adding significant run-time overhead as well as tens of thousands of lines of code (LOC) to the hypervisor [SK10].

ARM will support I/O virtualization via a *system MMU* (SMMU) [ARM11] similar to the IOMMU on x86, but this is a platform feature, and ARM has to date only released their extensions to the core ISA.

4. HYPERVISOR DESIGN

4.1 General

Our hypervisor supports a fixed, statically-configured number of VMs. Dynamic creation and destruction of VMs is a complication which reveals nothing of interest about the architecture, and therefore has no benefit for the prototype. Furthermore, virtualization use cases in the embedded-systems world typically do not require dynamic VMs, which is why they are not even supported by the commercial OKL4 microvisor. Hence even a productised successor of our prototype is likely to have this restriction.

We also decided against the use of virtual memory inside the hypervisor, as it would provide no benefit (especially when not supporting dynamic VMs). In fact, not using virtual memory has performance benefits, by avoiding PT walks inside the hypervisor and preventing the hypervisor from competing for TLB entries. Of course, virtual memory is available for use by guest OSes without limitation, and running the hypervisor in physical memory does not eliminate the need to set up PTs for the translation of guest-physical addresses.

As the OKL4 microvisor [HL10], and unlike other microkernel-like hypervisors (e.g. Nova [SK10]), we do not split the hypervisor into a privileged and a user-mode part. Nova, like other x86 hypervisors, require large amounts of code for instruction emulation, device emulation and BIOS emulation; separating these into user-level modules enhances robustness. In contrast, instruction emulation on ARM is almost trivial (thanks to the RISC architecture and architectural support), and there is no BIOS to emulate. Furthermore, thanks to virtual interrupts, the bulk of functionality required for device emulation is emulation of the distributor component of the GIC, which needs to be done in hyp mode anyway. Hence there is little functionality which could sensibly be moved to user-mode, and the hypervisor can be kept small nevertheless.

These design decisions helped to keep the hypervisor simple (without limiting its use in embedded systems). Beyond that we made one design decision which would be an unacceptable shortcut for a production system: not to support multiple cores. This is acceptable for the prototype as it avoids unnecessary complexity for the purpose of our evaluation, as multicore support would be mostly orthogonal to the use of the virtualization architecture. The structure of our kernel should make adding multicore support relatively straightforward.

4.2 Inter-VM communication

In addition to standard hypervisor functionality, our prototype provides a high-performance *inter-VM communication* (IVC) mechanism and the ability to set up shared buffers. This is essential to support shared device drivers encapsulated in their own VM, as well as the tight integration of subsystems typical of embedded systems [Hei09]. Such a mechanism is also provided by the OKL4 microvisor [HL10].

As in the microvisor, and unlike most other microkernels, IVC is fully asynchronous: A short (three-word) message is deposited in virtual registers in the receiver VM, and a virtual interrupt is delivered to the receiver. Further sends to the same receiver fail until the receiver acknowledges the interrupt. For the prototype we do not implement access control for IVCs, other than requiring the sender to know the receiver's VMID. Broadcast send to all other VMs is also supported.

A shared buffer needs to be set up between VMs at system-configuration time. A VM may have read-only or read-write access to a shared buffer. Within the bounds of this limited form of protection, VMs are fully responsible to manage access to a shared buffer. They can use virtual interrupts for synchronisation.

Remember that this simple model of IVC is designed for evaluation purposes. For a production system, more sophisticated access-control will be desirable, most likely an adaptation of the capability-based protection system of the OKL4 microvisor [HL10]. Furthermore, the message-passing semantics would need to be refined to avoid creating covert channels.

5. IMPLEMENTATION

The above design choices allowed us to based our implementation on an existing code base, the OK Labs "pico" product, which is an executive for MMU-less microcontrollers. No hardware supporting the virtualization extensions is available yet, so we used the ARM *Fast Models* simulator, which models the *RealView* emulation baseboard plus the virtualization extensions. The simulator is efficient yet functionally very accurate, but it is not timing-accurate. Also, it does not model complex external devices such as Ethernet, so we only support simple devices such as the UART consoles, the LCD controller, and the on-board timers. Here we highlight some implementation details.

5.1 World switch

A context switch performed by an OS needs to switch the GP registers plus the PT pointer and the address-space ID register, a total of 18 registers. The FPU state (16 or 32 double-word registers) is normally switched lazily. A world switch is more heavyweight, as it requires handling of additional state: the banked registers for all kernel modes (21), the system coprocessor registers, including the MMU state (22), all VCPU interface (interrupt controller) state (3 plus virtual interrupt registers, up to 96), all timer state (20), the second-stage PT pointer and the VMID register. In total this is up to 164 additional registers. Furthermore, co-processor registers and device registers are more expensive to access than GP registers.

World switches occur as a result of a timer tick (indicating the end of a time slice for a VM running on a shared core) and on an interrupt of priority higher than that of the presently running VM. The extensions introduce new timers which can be used by the hypervisor without interfering with the guests' use of timers. In our prototype we did not implement VM priorities and hence only perform world switches at the expiry of a (33 ms) time slice.

5.2 Instruction emulation

For most instructions which trap into the hypervisor, the information provided in hypervisor registers is sufficient to emulate them. One exception is an ARM peculiarity known as *write-back*. It allows adding an immediate value to the address register in a load instruction. For example, the instruction

```
    ldr   rd, [rs], #imm
```

is equivalent to

```
    ldr   rd, [rs]
    add   rs, #imm
```

except that it is atomic and executes as fast as a normal load.

Write-back is not supported by the emulation support. Also, the (previously fetched) instruction is not made available to the hypervisor, so it must be loaded explicitly from guest memory, decoded and emulated, and the guest instruction pointer must be incremented before returning to the guest. A very simple (50 LOC) emulator for write-back instructions was sufficient to support all Linux code we tried to run in a VM.

Note that loading the faulting instruction will cause at an L1 D-cache miss, as on fetch the instruction would have been placed in the I-cache.

5.3 Device pass-through

Pass-through allows a device to be exclusively used by a single guest, a situation common in embedded systems, where many devices are owned by individual subsystems. This does not incur any virtualization overhead and usually only requires mapping the correct memory regions, and ensuring that the device interrupt is routed to the guest. We used this for a range of devices, including the SMC flash chip, the network controller, and the audio controller.

Some cases require a bit more work, for example the LCD controller, which requires a physically-contiguous buffer and translating guest physical to physical addresses. In our implementation this is simple, as we map all guest physical memory (other than shared buffers) onto a contiguous memory region and trap the write to the buffer pointer register in the controller.

5.4 Shared devices

So far we have only added support for some very simple shared devices: a console (UART0), the distributor component of the GIC, the SP804 dual timer and the PL031 PrimeCell real-time clock.

The virtual console device runs in a VM in user mode on top of a real-time OS (we use the microvisor for this). It can be accessed from other VMs via an IVC protocol using a shared (single-page) buffer for transferring the data, and a virtual interrupt to indicate to the driver that data is available. Completion is indicated by a virtual interrupt back to the client.

We world-switch timer state, which means that guests see virtual time rather than real time. This is appropriate in many circumstances even in embedded systems (a trivial example are the BogoMIPS calculation loops in Linux).

We provide a second-resolution real-time clock by wrapping the hardware RTC to produce ticks every second, and sending virtual interrupts to all VMs that have enabled the RTC device. VMs with real-time requirements would be allocated a timer as pass-through device, but we did not do this in our prototype.

5.5 Limitations

The main limitations of our prototype are the lack of VM priorities (VM scheduling is round-robin) and no support for multiple cores. These limitations will be removed in a forthcoming production version of the hypervisor.

Other limitations do not affect anticipated embedded-systems use cases (at least in the near future). One is the lack of support for dynamic creation and destruction of VMs; in most embedded use cases the number of VMs are fixed at least between firmware upgrades. Similarly, there is little benefit expected from deduplicating pages, as embedded use cases tend to be heterogenous, running different guest OSes in different VMs (eg. Linux and an RTOS) [Hei08].

Table 1: Estimated instruction latencies

Operation	Estimated cycles
L1 access	2-3
L2 access	10-15
Memory access	50-150
CP15 read	10-20
CP15 write	100
FPU access	10-20
Hyp entry	50

6. EVALUATION

6.1 Hypervisor size

Our prototype comprises 5,730 LOC. We estimate that VM priorities would add at most 500 LOC, while multicore support would add about 1,000 LOC. This is offset by removing about 1,600 LOC of unneeded functionality from the executive we used as the starting point of our implementation, so a fully-fledged hypervisor would still be less than 6 kLOC. This compares to 9 kLOC kernel plus 27 kLOC user-mode code in NOVA on x86 [SK10].

6.2 Performance

Given the lack of a hardware implementation of the architecture, or at least a cycle-accurate simulator, no real performance evaluation is possible. However, we can get very rough estimates by collecting traces and weighting instructions with their known or estimated latencies. We use the estimated latencies shown in Table 1, which are appropriate for an ARM A9 core and a typical memory system.

The most uncertain of these latencies is the cost of entering hyp mode, estimated at 50 cycles. This may look incredibly optimistic to those used to x86, where VM exits are more than an order of magnitude more expensive. Remember that ARM is a RISC architecture; kernel entry/exit costs of the order of ten cycles, compared to many hundreds on x86. Also, the ISA avoids inherently-expensive operations, such as automatic saving of guest state (see Section 3.3).

For a number of microbenchmarks we measured instruction counts and converted them into approximate cycle counts under the above assumptions. The result is shown in Table 2, where *instr.* refers to the instruction count extracted form the simulation and *cycles* is the resulting estimate of execution time in cycles. Note that even if our estimate for a hypervisor trap was off by a factor of four (which seems extreme, given our experience with the architecture), this would add 150 cycles to each of the entries in the table, and would not change the picture significantly.

The hypervisor entry/exit costs are the estimated mode-switch costs plus the software cost of saving/restoring enough state to execute C code in the hypervisor. *Hypervisor entry* is the entry cost for a hypercall, while *IRQ entry* is the cost of entering hyp mode via an interrupt vector. The comparable x86 figure (NOVA) is 4,000 cycles. *Page fault* is the cost of handling the case where the guest faults on a (guest-physical) page which has not yet been mapped by the hypervisor.

The *device emulation* figures refer to trapping and emulating access to a device register, and *acc* refers to the case where this is accelerated by hardware support for emulation. The results show that device emulation is expensive even with emulation support, and production systems will likely continue to use para-virtualization of device drivers.

World switch refers to switching VM context, using ARM's multi-register operations for efficiently saving and restoring state. Much of this state is kept in co-processor (MMU) or core-external (virtual interrupt controller, devices) registers which are more expensive to access than internal registers. *Lazy FPU switch* refers to the cost of switching the

Table 2: Microbenchmarks: Estimated overheads

Operation	Instr.	Est. Cycles
Hypervisor entry	89	450
IRQ entry	239	700
Hypervisor exit	31	200
Page fault	356	1500
Device emulation	249	1040
Device emulation (acc)	176	740
World switch	2824	7555
Lazy FPU switch	127	950

FPU lazily, i.e. when a guest application attempts to access it.

7. EXPERIENCE

We found that the ARM virtualization extensions significantly reduce complexity of a hypervisor (compared to para-virtualization) and are also likely to reduce virtualization overheads.

Compared to x86 there is some added complexity, for example saving and restoring VM state in software, but this is minor (about 200 LOC). The advantage of the ARM model is that the hypervisor can optimise the handling of VM state. For example, if an interrupt is received which is destined for the presently running VM, no state needs to be switched beyond what is needed to run the hypervisor, while x86 would save and restore redundant state in this case. The downside is that a full world switch seems to be more expensive than on x86. However, this cost could be reduced if there was a single (pipelined) instruction for saving the complete MMU (CP15) context, a saving which could be of the order of 1000+ cycles.

IRQ handling is also simple and fast on ARM, as the hypervisor can inject interrupts "blindly", while on x86 the hypervisor needs to check masks and guest priorities, and modifications of masks by the guest trap into the hypervisor. One drawback of the ARM model is that interrupt handling is configured all-or-nothing. Unless all IRQs can be handled by the guest (which seems an unusual situation), the hypervisor must be invoked on each IRQ and forward it to the respective guest. Per-IRQ configuration would be a significant improvement (shaving an estimated 700 cycles off IRQ processing), even though it would slightly increase the (anyway high) cost of a world switch (for saving/restoring the IRQ bitmap).

Instruction-emulation support is one of the strengths of the ARM model, which dramatically reduces hypervisor complexity and significantly improves performance. The exception are write-back instructions, for which no support is available. Given that the instruction has already been loaded into the core, having the hardware save it into a dedicated hypervisor register could reduce emulation cost by an estimated 250 cycles. However, this may not add much to overall performance, as emulating write-back instructions seems a rare event.

In fact, given the thorough virtualization of the processor hardware, including the MMU, instruction emulation is mostly needed for virtualising devices. However, pure virtualization of device-register accesses is likely very expensive even where the architecture fully supports emulation. We therefore expect that drivers for most shared devices will continue to be para-virtualized, making the lack of emulation support for write-back even less critical.

8. CONCLUSIONS AND FUTURE WORK

In the course of this project we encountered numerous simulator bugs which were new to ARM (but confirmed as bugs). This is strong indication that no-one had exercised the simulator as we have, and most likely means that we were the first to produce a fully-functional hypervisor for hardware-supported virtualization on ARM able to run unmodified Linux guests.

We found that while the ARM extensions superficially look very similar to those of the x86 architecture, the RISC nature of ARM allows for a much simpler implementation. Especially the support for instruction emulation benefits from the RISC design, and allows all required emulations to be done with very little code. While our prototype still lacks a few features, most importantly VM priorities and support for multicores, it is clear that a fully-functional ARM hypervisor can be implemented in around 6,000 LOC, vastly smaller than on x86. Work on turning the prototype into a commercial product is under way.

Acknowledgements

We would like to thank the OK Labs engineering team for their support, especially Carl van Schaik for sharing his thorough knowledge of the ARM architecture. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

9. REFERENCES

[AA06] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *12th ASPLOS*, San Jose, California, USA, Oct 2006.

[ARM10] ARM Architecture Group. *Virtualization Extensions Architecture Specification*, 2010. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b_virtualization_extns/index.html.

[ARM11] CoreLink system controllers for AMBA. http://www.arm.com/products/system-ip/controllers/index.php, 2011.

[Bha09] N. Bhatia. Performance evaluation of Intel EPT hardware assist. Technical report, VMWare, 2009.

[FO06] J. Fisher-Ogden. Hardware support for efficient virtualization. Technical report, University of California, San Diego, 2006.

[Gree10] INTEGRITY Secure Virtualization. http://www.ghs.com/products/rtos/integrity_virtualization.html, May 2010.

[Hei08] G. Heiser. The role of virtualization in embedded systems. In *1st WS Isolation & Integration Emb. Syst.*, pages 11–16, Glasgow, UK, Apr 2008. ACM SIGOPS.

[Hei09] G. Heiser. Hypervisors for consumer electronics. In *6th IEEE Consumer Comm. & Networking Conf.*, pages 1–5, Las Vegas, NV, USA, Jan 2009.

[HL10] G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *1st APSys*, pages 19–24, New Delhi, India, Aug 2010.

[HSH$^+$08] J.-Y. Hwang, S.-b. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer Comm. & Networking Conf.*, pages 257–261, Las Vegas, NV, USA, Jan 2008.

[Kro09] K. L. Kroeker. The evolution of virtualization. *CACM*, 52(3):18–20, 2009.

[NSL$^+$06] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology J.*, 10(3), Aug 2006.

[OKL11] Open Kernel Labs Website. http://www.ok-labs.com/about/about-ok-labs, Jan 2011.

[RedB10] Red Bend Software website. http://www.redbend.com/, Dec 2010.

[SK10] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *5th EuroSys Conf.*, Paris, France, Apr 2010.

[VMwa10] VMware mobile virtualization platform website. http://www.vmware.com/products/mobile/, Dec 2010.