Hardware Accelerated Cross-Architecture Full-System Virtualization

A

Hardware Accelerated Cross-Architecture Full-System Virtualization

Tom Spink, Harry Wagstaff & Björn Franke, University of Edinburgh, UK

Hardware virtualization solutions provide users with benefits ranging from application isolation through server consolidation to improved disaster recovery and faster server provisioning. While hardware assistance for virtualization is supported by all major processor architectures, including Intel, ARM, PowerPC & MIPS, these extensions are targeted at virtualization of the same architecture, e.g. an x86 guest on an x86 host system. Existing techniques for cross-architecture virtualization, e.g. an ARM guest on an x86 host, still incur a substantial overhead for CPU, memory and I/O virtualization due to the necessity for software emulation of these mismatched system components. In this article we present a new hardware accelerated hypervisor called CAPTIVE, employing a range of novel techniques, which exploit existing hardware virtualization extensions for improving the performance of full-system cross-platform virtualization. We illustrate how (1) guest MMU events and operations can be mapped onto host memory virtualization extensions, eliminating the need for costly software MMU emulation, (2) a block-based DBT engine inside the virtual machine can improve CPU virtualization performance, (3) memory mapped guest I/O can be efficiently translated to fast I/O specific calls to emulated devices, and (4) the cost for asynchronous guest interrupts can be reduced. For an ARM-based Linux guest system running on an x86 host with Intel VT support we demonstrate application performance levels, based on SPEC CPU2006 benchmarks, of up to $5.88\times$ over state-of-the-art QEMU and $2.5\times$ on average, achieving a guest dynamic instruction throughput of up to 1280 MIPS and 915.52 MIPS, on average.

CCS Concepts: •**Computing methodologies** → **Simulation tools;** •**Hardware** → **Simulation and emulation;**

Additional Key Words and Phrases: Virtualization

ACM Reference Format:
Tom Spink, Harry Wagstaff, and Björn Franke, 2016. Hardware Accelerated Cross-Architecture Full-System Virtualization *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Virtualization technology enables workload consolidation where multiple applications are deployed onto *Virtual Machines* (VMs), which then run on a single, more-powerful host machine. While virtualization may introduce some runtime overhead, processor manufacturers have developed support in hardware in the form of *instruction set extensions* (ISEs) to make their respective architectures efficiently and fully virtualizable, e.g. Intel VT or ARM Virtualization Extensions. They have also created additional mechanisms to allow I/O virtualization with less software overhead.

However, these virtualization extensions are geared towards *same*-architecture virtualization, where both the guest VM and the physical host machine share the same architecture. For *cross*-architecture virtualization, where guest VM and host architectures are different, translation between ISAs, emulation of the guest system's memory management unit (MMU), interrupt handling and I/O devices are typically implemented entirely in software, resulting in a substantial performance loss. For example, in full-system mode the gem5 architectural simulator [Binkert et al. 2011] takes about 30 minutes to boot into Linux on a current, mid-range host machine. While this performance level is sufficient for some computer architecture research, it is far too slow for any practical applications. QEMU [Bellard 2005], a popular cross-architecture full-system virtualizer using dynamic binary translation (DBT), is substantially faster, but

[0]New Paper, Not an Extension of a Conference Paper

still suffers an up to $20\times$ slow-down [Jones and Topham 2009] compared to native execution on the host.[1]

While same-architecture virtualization has become ubiquitous there are fewer, but nonetheless important applications for cross-architecture virtualization, e.g. Android software development using the QEMU-based Android Emulator shipped with the Android Studio [Gerber and Craig 2015], which provides a full-system ARM environment for software developers using an x86 host machine; building ARM Docker [Merkel 2014] containers on x86 machines; providing fast-forwarding in sampling based simulators [Sandberg et al. 2015]; or early-stage software development without a hardware target [Ceng et al. 2009]. All of these applications critically depend on fast cross-architecture virtualization due to unavailability or deliberate absence of a hardware platform supporting the chosen target ISA.

In this article, we develop a novel approach for speeding up *cross*-architecture virtualization, and implement these ideas in a new cross-architecture hypervisor called **CAPTIVE**. The *key idea* is to eliminate performance bottlenecks by exploiting the existing virtualization hardware extensions originally devised for *same*-architecture virtualization. We target four distinct *cross*-architecture virtualization challenges and make the following contributions:

(1) We show how virtual-to-physical address translation can be accelerated through the use of virtualization extensions, by mapping behavior of the guest MMU onto corresponding behavior of the host MMU – despite substantial differences between the two MMUs.
(2) We present a DBT system for the translation from the guest to host ISA, where a fast, block-based just-in-time (JIT) compiler that lives *inside* the native virtual machine compiles guest basic blocks to host native code.
(3) We develop an efficient mechanism to emulate the guest's memory mapped I/O devices, by exploiting the MMU to detect device accesses.
(4) Finally, we devise an interrupt handling scheme, which correctly honors the guest's instruction boundaries, even if one guest instruction is mapped onto several host instructions, thus implementing precise, yet efficient guest interrupts.

1.1. Motivating Example

It has been well established [Magnusson and Werner 1994; Chang et al. 2014; Wang et al. 2015; Hong et al. 2015] that emulation of a guest MMU is one of the most time-consuming parts of cross-architecture virtualization, therefore in this motivating example, we will focus on the memory address translation process required for virtualization. For this, consider the diagram in Figure 1, which shows the percentage of memory operations w.r.t. the total number of executed instructions in the SPEC CPU2006 integer benchmarks. About 50% of the instructions in these benchmarks perform memory accesses. This means if we are running these benchmarks in a virtualized ARM guest environment on an x86 host, on average, every other instruction demands an expensive virtual-to-physical memory translation using a virtualized ARM MMU, typically implemented in software. If we succeed in speeding up these address translations we will eliminate one of the most severe cross-architecture virtualization performance bottlenecks.

In a 32-bit ARMv7-A system with an ARMv7-A MMU, there are at most two levels of page tables representing a virtual memory area. To translate a virtual address into its corresponding physical address, the first-level page table (an L1 page table), pointed

[1]Native execution of a binary suitably compiled to the host ISA from the same sources, which have been used to build the binary for the guest's ISA.

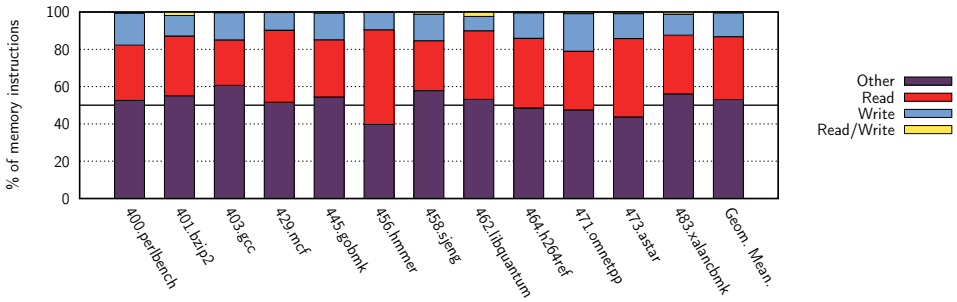


Fig. 1: Distribution of operations in the SPEC CPU2006 integer benchmarks. On average, around 50% of all instructions executed perform memory operations (either *read*, *write*, or *both*), which require expensive virtual-to-physical address translation using a software MMU.

to by the `TTBR` register, is indexed by bits 20–31 of the virtual address and the entry interrogated to determine if the mapping is to a *section* (a 1MB contiguous chunk of memory) or a small page (a 4kB contiguous chunk of memory). If the page table entry indicates a section, then the base address points to the physical base address of the memory. If the page table entry indicates a small page, then the base address points to the physical base address of a second-level page table (an L2 page table). The L2 page table is indexed by bits 12–19 of the virtual address, and the base address in the L2 page table entry points to the physical base address of the page corresponding to the mapping. A page table entry in the L1 (if pointing to a section) or L2 page table in addition to the base address pointer contains flags that indicate the access permissions of the page, e.g. whether or not the page is readable and writable, and if it is accessible whilst executing in the user privilege level.

In a 64-bit x86 system, there are four levels of page tables that represent a 48-bit virtual address space. Pointers are always 64-bit wide, but can only access 48-bits of virtual address space. Virtual addresses *must* be in *canonical form*, where bits 48–63 of the address are copies of bit 47. Any memory access to a non-canonical address will result in a general protection fault. Address translation operates in a similar fashion to ARM, with each level of page table being traversed to translate a 64-bit virtual address into a 64-bit physical address, subject to permissions which can be applied at any level of the page table–the higher level permissions taking precedence over the lower levels.

To avoid a costly page table walk for every memory access, both architectures employ a *translation lookaside buffer* (TLB), which caches the result of a previous hardware translation. If the page tables are modified, or the pointer to the top-level page table changed, the TLB *must* be flushed.

From this description it should be clear that the structure of the ARM and x86 MMUs are substantially different, yet fundamentally they both provide a mechanism for the translation of virtual addresses to physical addresses with access permission checking. In this article we propose to exploit this hardware address translation capability, and show how to map the behavior of an ARM MMU onto a *virtualized* Intel MMU. By intercepting ARM TLB invalidations and maintaining entries in the x86 page table that represent entries in the ARM page table, we can accelerate address translations. Instead of using a slow software implementation of the ARM MMU, guest address translations are redirected to the fast, host virtualized Intel MMU, which our system keeps consistent with the guest's ARMv7-A MMU. Using existing extensions originally devised for *same*-architecture virtualization we are able to speed up critical *cross*-architecture address translations over a purely software-based MMU implementation.

1.2. Overview

The remainder of this article is structured as follows. In Section 2 we provide background information on MMU virtualization in full-system simulators, Intel virtualization technology (Intel VT) and the kernel-based virtual machine (KVM) framework, on which our work relies. We then present our novel *cross*-architecture virtualization techniques in Section 3 and our results in Section 4. We discuss related work in Section 5, before we summarize and conclude in Section 6.

2. BACKGROUND

While this article is largely self-contained we assume a certain level of familiarity with the Intel VT extensions and also KVM. In this section we briefly introduce these technologies before describing our virtualization infrastructure in the next section.

2.1. Intel VT

Intel VT [Intel 2016] is a set of hardware extensions introduced by Intel to provide support for virtualization of an x86 processor. It provides new instructions for setting up and managing these virtual machines, transitioning between hypervisor and virtualized execution, and support for virtualizing the guest MMU with *extended page tables* (EPT). EPT provides an extra level of page tables that are walked by hardware when a virtual memory address that is not present in the TLB is encountered whilst running in the virtual machine. CAPTIVE utilizes Intel VT extensions by creating a virtual machine with the KVM infrastructure provided by the Linux kernel.

2.2. KVM

Hardware accelerated virtualization is well supported by multiple processor vendors, such as Intel with Intel VT and AMD with AMD-V. Whilst these extensions produce the same effect, i.e. they create an abstract computing platform on which to run unmodified operating systems, they are implemented and accessed completely differently. KVM [KVM 2016] is a *virtual machine monitor* implemented as part of the Linux kernel, which can utilize any supported hardware virtualization, on any platform. Its job is to abstract the details of the virtualization extensions, and to provide a generic interface for creating and managing a virtualization environment. KVM fully supports virtualization extensions such as Intel EPT or AMD RVI, which is used to accelerate a virtualized MMU. KVM itself is only an interface to hardware virtualization extensions, it will not work in the absence of these.

KVM was developed in tandem with QEMU, but does not depend on QEMU—it is an independent technology that is part of the standard Linux kernel.

In CAPTIVE, we utilize KVM to create a virtual machine backed by Intel VT or AMD-V in a platform-agnostic way. The use of KVM, and consequently the use of Intel VT is described in more detail in Section 3.1.

3. VIRTUALIZATION INFRASTRUCTURE

Our virtualization infrastructure consists of three main components:

(1) A **hypervisor** component, which runs on the host machine and uses KVM to control the host's hardware virtualization extensions.
(2) An **execution engine** component, which runs inside a normal virtual machine on the host.
(3) An **architectural implementation**, which defines the behavior of the architecture being virtualized.

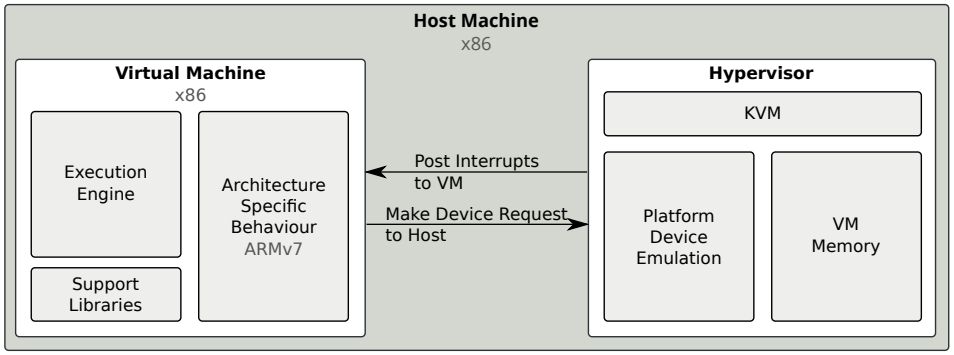


Fig. 2: A high-level overview of the virtualization infrastructure. The CAPTIVE hypervisor runs in the operating system (Linux) of the host machine, and creates a virtual machine that is the same architecture as the host (x86). Inside this virtual machine, the CAPTIVE execution engine virtualises a guest platform (ARMv7) by mapping the behaviour of the guest, to the behaviour of the host.

For the rest of this article, we will be assuming that we are virtualizing an **ARMv7-A guest** architecture (modelling a *RealView Platform Baseboard for Cortex-A8* [ARM 2011b]), on a standard **x86-64 host** machine with Intel VT virtualization extensions.

Due to the multi-layer nature of this system, it is important to define some terminology at this point to establish exactly what terms will describe what aspects of the system.

Definition 3.1 (*Host*). The **host** machine is the system on which we will be running the virtualized platform. This article will use an x86-64 machine as the host machine.

Definition 3.2 (*Guest*). The **guest** machine is the target platform that we wish to virtualize. This article will use an ARMv7-A platform as the guest machine.

Definition 3.3 (*Native Virtual Machine (VM)*). The hardware extensions provided by the **host** machine naturally provide a *same*-architecture virtual machine, e.g. using QEMU with KVM on x86 would result in an x86 virtual machine. In our case, the **Native Virtual Machine (VM)** refers to the virtual machine provided by these hardware extensions, to which we utilize in our infrastructure. Therefore, in this article, the **Native VM** is of the *same architecture* (x86-64) as the **host**.

Definition 3.4 (*Hypervisor*). The **hypervisor** is the software which runs on the **host** machine (within the host operating system), and is responsible for providing support to the **Native VM**, along with software implementations of guest platform devices (e.g. a timer device, or an interrupt controller).

Definition 3.5 (*Execution Engine*). The **execution engine** is a bare-metal application that runs inside the **Native VM** (without an operating system) and provides the CPU virtualization necessary to run the cross-ISA guest instructions. It maintains the state of the guest platform being emulated, and executes *platform specific* behavior, such as what happens when an MMU fault occurs or an external interrupt is signalled.

3.1. Overview

This section will give an overview of the operation of CAPTIVE. It will continue to use the example of an x86-64 *host* machine, and an ARMv7-A *guest* machine, implementing

the RealView Platform Baseboard Cortex-A8 [ARM 2011b]. This example leads to the following assumptions:

(1) We are virtualizing a 32-bit guest system on a 64-bit host system.
(2) We depend on KVM as a framework to provide access to Intel VT virtualization extensions.
(3) We utilize (but do not depend on) the *address-space identifier* (ASID) feature of the guest operating system to accelerate context switches.
(4) We exploit the fact that ARM page tables have similar access permissions to x86 page tables, enabling us to efficiently map MMU behavior.

Our system starts by instantiating a native virtual machine on the host, using the KVM framework. KVM is the interface to the Intel VT virtualization extensions, and starts by initializing the required structures that represent a virtual machine on an Intel processor (e.g. creating the VMCS structure, and issuing the `VMXON` instruction). Then, a single virtual x86-64 CPU is requested from KVM which will ultimately represent the virtual ARM processor.

This native VM is configured with physical memory that represents the physical memory provided by the guest platform—in this case, 2GB of physical memory is installed. An additional block of physical memory is also installed that contains the execution engine binary, and heap space for memory allocations and the translated code cache. The guest kernel (an ARMv7-A Linux kernel), is written into the guest physical memory at the correct location (as specified by the platform boot protocol). Finally, the virtual x86-64 CPU is configured to start up in 64-bit mode (by manipulating the virtual CPU state structure), and with the entry-point of the execution engine. The native VM is then started, and control is transferred to the execution engine running inside.

Once inside the native VM, we have full control of a bare-metal x86-64 machine, the execution engine is essentially an x86-64 kernel, with full privileged access to this virtual machine. We configure the virtual memory of the native VM in such a way as that the lower 4GB portion represents either a one-to-one mapping of guest physical memory (if the guest MMU is turned off) or the actual virtual memory mapping of the guest machine (detailed in Section 3.3). The execution engine itself resides in the high portion of virtual memory, and certain other virtual memory areas are mapped to the heap and stack.

When first started, the execution engine performs some platform initialization of the native VM, such as setting up the native (x86) page tables and the interrupt descriptor table (IDT), and eventually begins executing guest ARM code. We have already populated guest physical memory with the guest kernel we are about to execute, so execution begins from this entry-point, using JIT compilation of the guest ARM instructions to native x86-64 instructions (detailed in Section 3.2).

Any access by guest instructions to the memory of the guest machine is performed with a normal memory access, without having to translate/transform any virtual memory addresses – they are simply made to the address to which they would be made if running on a non-virtualized ARM system.

The guest platform we are virtualizing is a 32-bit platform and so any memory access can only be in the 0-4GB (2^{32}) range of lower virtual memory. Virtualization of a 64-bit platform is outside the scope of this article, but will require an extra layer of software indirection and is planned for future work. When an access to a particular address occurs for the first time, a page fault is generated and handled by installing a mapping of the corresponding virtual page to guest physical memory (subject to the operation of the ARM guest MMU).

External interrupts generated by devices (such as a timer device) are propagated as real interrupts into the guest system, which causes a flag to be set to indicate that

Listing 1: High-level instruction format description.

```
%% Define the MOVW instruction format: 4-bit condition, 4-bit operand, ...
ac_format <Type_MOVW> = "%cond:4 %op:4 %subop:4 %rn:4 %rd:4 %imm32:12";

%% Link movt, movw instructions to the MOVW instruction format
ac_instr<Type_MOVW> movt, movw;

%% Define the ARM instruction set architecture
ISA_CTOR(arm) {
  %% Instruction: movt, where (op = 0x3) and (subop = 0x4)
  movt.set_decoder(op = 0x3, subop = 0x4);
  %% Assembly mnemonic
  movt.set_asm("movt[%cond] %reg, #%imm", cond, rd, imm32);
  %% Instruction behavior is defined in the "movt_behavior" function
  movt.set_behavior(movt_behavior);

  %% Instruction: movw, where (op = 0x3) and (subop = 0x40)
  movw.set_decoder(op = 0x3, subop = 0x0);
  %% Assembly mnemonic
  movw.set_asm("movw[%cond] %reg, #%imm", cond, rd, imm32);
  %% Instruction behavior is defined in the "movw_behavior" function
  movw.set_behavior(movw_behavior);
}
```

Listing 2: Semantic description of the behavior of the `movt` and `movw` instructions.

```
%% Instruction behavior for "movt"
execute(movt_behavior)
{
  uint32 orig = read_register_bank(RB, inst.rd) & 0xffff;
  uint32 rn = inst.imm32 | (inst.rn << 12);

  uint32 result = orig | (rn << 16);
  write_register_bank(RB, inst.rd, result);
}

%% Instruction behavior for "movw"
execute(movw_behavior)
{
  uint32 result = inst.imm32 | (inst.rn << 12);
  write_register_bank(RB, inst.rd, result);
}
```

Fig. 3: Instruction formats and semantics are specified using an ArchC-like high-level architecture description language. Example showing the specification of formats (in Listing 1) and semantics (in Listing 2) of the `movt` and `movw` instructions, respectively.

native code should stop executing at the next *safe point*. At a minimum, a *safe point* is an instruction boundary, but we insert safe points at guest basic block boundaries to improve performance.

3.2. CPU Virtualization

Same-architecture virtualization is easily supported by modern processors that include hardware support for virtualization. Technologies such as Intel VT and AMD-V allow guest code to run directly on the host processor, without modification or instrumentation for maximum performance. Certain privileged operations (such as changes to control registers, and TLB invalidations) are trapped by the host CPU and handled by a hypervisor (for example KVM).

This virtualization is trivial in the same-architecture case, because both the guest and the host have the same *instruction set architecture* (ISA) and are therefore binary compatible. However, this presents a problem for cross-architecture CPU virtualization, as the ISAs are different, and completely incompatible.

In a traditional full-system simulator, techniques such as *interpretation* or *dynamic binary translation* (DBT) are used to virtualize the guest ISA on the host ISA, the former being straightforward to implement, and the latter being recognized as one of the fastest ways [Ung and Cifuentes 2000; Ebcioglu et al. 2001; Bellard 2005] to emulate guest instructions on a host machine. Emulation of guest instructions is a necessity for cross-architecture virtualization, and techniques for doing so have been well studied and presented in DBT improvement articles such as [Kumar et al. 2005; Guha et al. 2010].

Our approach to instruction emulation is based on a basic block *just-in-time* (JIT) compiler engine, which takes guest basic blocks discovered at runtime, and compiles them into corresponding host basic blocks. Block compilation is synchronous to the execution of the guest system, and occurs on-demand when a translation for a particular guest basic block is not available. Generated host code is stored in a code cache, for later use. This is similar to the approach taken by QEMU, except for two important differences: (1) we generate code independent of the virtual address of the guest basic block, and (2) the JIT compiler in CAPTIVE lives *inside* the native virtual machine as part of the execution engine.

QEMU has implemented an advanced caching strategy that initially uses a fast first-level cache indexed by virtual address to look up the code associated with a guest basic block, which is invalidated when page mappings change. As basic-blocks are translated for specific *virtual* addresses, the virtual PC may be constant-folded into the translations. However, the translation cannot be re-used if the same physical address is mapped to different virtual addresses. To handle this situation, when a miss occurs in the first-level cache, a second-level cache that is indexed by virtual PC, physical PC and memory access flags is consulted. If this cache misses, then the guest basic block is translated. This results in a guest basic block being translated for each distinct virtual address mapping. In contrast, CAPTIVE always indexes the code cache by physical PC, and translates code in a way that is independent of its virtual address, meaning we can re-use the same translation for multiple virtual addresses.

The JIT compiler is generated from a high-level instruction description, where instruction formats are defined in an ArchC-like *domain specific language* (DSL) [Azevedo et al. 2005], and the semantic behaviors are specified in a C-like DSL. In an offline pass, this architecture definition is parsed, and a high-speed instruction decoder and instruction IR generator are produced, which are then used by the execution engine to compile guest code.

Listing 1 shows an extract of an instruction-format description from our ARMv7-A model, which is used to generate an instruction decoder. Line 2 contains a bit-level representation of the instruction format, and line 5 declares two instructions (`movt` and `movw`) that conform to this pattern. Lines 10 and 17 further specialize the pattern, specific to the two instructions, by placing constraints on the values of the fields defined in the instruction format. Lines 12 and 19 assist debugging by producing a disassembly format for the instructions, in a `printf`-style declaration. Finally, lines 14 and 21 attach semantic behaviors to the instructions. Listing 2 shows the C-like DSL that describes the behaviors of the corresponding instructions. These descriptions are fed into an offline generator, which produces a fast instruction decoder, and a low-level intermediate representation (IR) emitter. The instruction decoder is used by the execution engine to decode *guest* basic blocks. The IR emitter is called for each decoded instruction, which produces a sequence of low-level IR instructions.

Listing 3: ARM guest basic block

```
ldr   r3, [r4]        // Load value from memory
cmp   r3, #0x1000000 // Compare value to constant
movcs r0, #1          // Set r0 to 1, if carry flag
bcs   5412dc          // Branch if carry flag
```

Listing 4: Execution Engine IR

```
    // Load r4
b0: ldreg       i4 $0x10, i4 v0
    // Read memory
    ldmem       i4 v0, i4 v0
    // Store value in r3
    streg       i4 v0, i4 $0xc
    // Subtract, and update guest
        flags
    sbc flags   i4 $0x1000000,
                i4 v0,
                i1 $0x1
    inc-pc      i4 $0x8
    // Read carry flag, and branch
    ldreg       i4 $0x140, i1 v0
    branch      i1 v0, b1, b2

    // Set r0 to 1
b1: streg       i4 $0x1, i4 $0x0
    inc-pc      i4 $0x4
    // Calculate branch target
    ldpc        i4 v0
    sub         i4 $0x1d8, i4 v0
    and         i4 $0xfffffffe,
                i4 v0
    streg       i4 v0, i4 $0x3c
    jmp         b3
    // Increment PC
b2: inc-pc      i4 $0x8
b3: ret
```

Listing 5: x86 host basic blocks

```
    mov   0x10(%rdi),%eax
    mov   (%rax),%eax     // Load value from memory
    mov   %eax,0xc(%rdi)
    sub   $0x1000000,%eax // Compare to constant
    setae 0x140(%rdi)
    seto  0x143(%rdi)
    sete  0x141(%rdi)
    sets  0x142(%rdi)
    lea   0x8(%r15d), %r15d // Increment PC
    cmpb  $0, 0x140(%rdi)
    jnz   1f                // Skip if not carry
    movl  $0x1,(%rdi)       // Set r0 to 1
    lea   0x4(%r15d), %r15d // Increment PC
    lea   -0x1d8(%r15d),%eax
    and   $0xfffffffe,%eax
    mov   %eax,%r15d  // Update PC to branch target
    jmp   2f
1: lea   0x8(%r15d), %r15d // Increment PC
2: // Epilogue
    //
```

Fig. 4: Example inputs and outputs during the JIT compilation phase of CPU virtualization. ARM guest code is initially translated to an internal representation for optimization, before x86 host code is generated and emitted.

3.2.1. Native Code Production. Our execution engine compiles *guest* basic blocks at a time, but will extend to a trace-based approach if the branch targets are static and land on the same *guest* memory page. Guest basic blocks are terminated at page boundaries for memory protection purposes. Normal control flow out of a block is optimized utilising techniques from [Spink et al. 2014], which includes directly chaining to other basic blocks that are part of the same memory page to avoid costly returns to the main execution loop. If a translation does not exist, or the destination does not live on the same page, control is returned to the main execution loop, which will then handle the situation accordingly. We dedicate an x86 host register (`%r15d`) to tracking the guest PC, which significantly improves performance by avoiding an increment to a memory location (i.e. the emulated guest register file) on each instruction.

As each instruction is being translated to a sequence of IR instructions, we employ the partial evaluation technique described in [Wagstaff et al. 2013] to constant-fold values known at compilation time into the IR. This technique also allows us to evaluate control flow within an instruction that can be resolved at compilation time too, i.e. control flow that depends on values which are constant.

After producing IR that represents the basic block being translated, we then perform some basic optimization passes (such as liveness analysis, control flow optimization and dead code elimination) before performing a linear-scan register allocation pass. After allocating registers, we perform some final optimizations (such as register value re-use and dead code elimination) and then forward the IR to the instruction lowering pass which translates IR instructions into corresponding host instructions.

Although there are existing JIT compilers, we choose to implement our own JIT compiler for three main reasons:

(1) It is not feasible to use compilers such as GCC and LLVM with our system, because they have dependencies on external libraries, such as `libc` for example. The JIT compilation is performed *inside* the native VM, which is not running an operating system and does not have a C library.
(2) Compilers such as GCC and LLVM are too slow to be used as JIT compilers [Brandner et al. 2009], as they would introduce an unacceptable amount of compilation latency. Our JIT is specialized for fast dynamic binary translation, much like TCG in QEMU.
(3) Existing JIT compilers, e.g. TCG in QEMU, are not well suited for automatic retargeting from a high-level architecture description and would require additional modifications to support bare-metal code generation.

Listing 3 shows an example ARM guest basic block that is encountered during the Linux kernel boot process. The IR emitter iterates over this block and after an optimization pass produces the IR shown in Listing 4. Finally, a quick template-based lowering pass produces the native x86 machine code shown in Listing 5.

It can be seen here that multiple *host* basic blocks are produced from a single *guest* basic block. In this example, this occurs because of a predicated ARM instruction (`movcs`) that may or may not be executed, depending on the current state of the flags. Since we do not class predicated instructions as basic block terminators, additional control flow is required to account for this.

3.2.2. Exploitation of Hardware Features. Given that we are operating in a bare-metal environment, we have full control of an x86 machine and so exploit our ability to use privileged instructions and access normally privileged features to generate efficient native code. Furthermore, we exploit our ability to switch the virtualized CPU into and out of privileged mode (ring 0 in x86 terminology) so that we can execute ARM guest code that usually runs in ARM user mode in x86 user mode, and ARM guest code that usually runs in ARM system mode in x86 privileged mode. This enables us to utilise the user/kernel memory protection available in x86 page tables, by mapping it to the corresponding ARM page table permissions. This is something that a user-space system like QEMU cannot do, as it is constrained by the host operating system.

We implement fast mode switching in x86 by using the `syscall` and `sysret` instructions, which provide efficient means of transitioning between user and privileged mode. We make use of the general-purpose segment register `FS` to point to a per-CPU data structure, which contains the state of the emulated ARM CPU, and also the `GS` segment register for efficient user-mode emulated memory accesses as described in Section 3.3.3.

We utilise the x86 *call gate* mechanism for invoking functions from user mode that require kernel mode permissions. This is an alternative to the slower software-interrupt based mechanism (i.e. using the `int` instruction).

3.2.3. Code Cache. In order to improve execution performance, translated guest basic blocks are kept in a code cache, indexed by physical address. The benefit of using physical addressing is that if the guest page tables are invalidated, we do not need to invalidate any compiled code. In fact, the only time we have to invalidate code is in the presence of self-modifying code, or more generally when a page that has previously been executed is written to. We detect this occurrence by marking (physical) pages that have been executed with a flag, and protecting those pages from being written to. When a memory fault occurs because of a write, and the page has been flagged, all cached code corresponding to that page is invalidated. If the memory fault was to an address

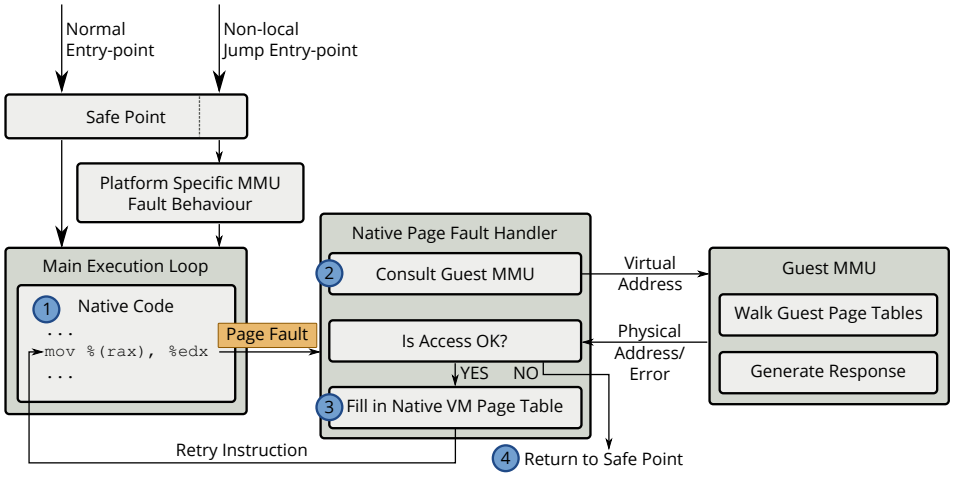


Fig. 5: Operation when virtualizing memory accesses. A memory access (1) is performed as a single native instruction, which when accessing a virtual address for the first time will cause a page fault in the Native VM. The native page fault handler will (2) consult the guest MMU implementation, to determine if the mapping is valid, then either (3) fill in a native page table entry, or (4) perform a non-local jump from the page fault handler back to a safe point to invoke platform specific memory fault handling.

on the page that was currently executing, we return to the main execution loop via a non-local jump, since we cannot return to cached code that represents instructions that were potentially modified.

3.3. MMU Virtualization

One of the most important parts of full-system virtualization is the faithful emulation of the *memory management unit* (MMU), which if implemented incorrectly will lead to an unusable system, and if implemented poorly can lead to severe performance penalties. Hardware extensions for same-architecture virtualization provide accelerated means of virtualizing the MMU of a guest machine on the host, but a problem arises when virtualizing a guest with a different architecture. As described in the motivating example (Section 1.1), the MMUs between two different architectures behave quite differently, and traditional full-system cross-architecture virtualization uses a (correct, but slow) software MMU implementation to emulate this subsystem. Thus, much work has been done [Wang et al. 2015; Chang et al. 2014; Hong et al. 2015] in the area of software MMUs to reduce the translation penalty and hence increase overall throughput of the virtualization system.

Fundamentally, the function of the MMU is to translate a *virtual address* to a *physical address*, applying any permissions that may be defined for that access. Usually, this mapping is represented with *page tables*, with various levels of indirection to suit the granularity of the mapping. Full-system virtualization requires that every instruction that accesses virtual memory is subject to the behavior of the MMU. For the same-architecture case, memory instructions are mapped one-to-one, and the hardware extensions take care of performing the virtual-to-physical translation and permissions checking, but for cross-architecture virtualization, each memory access must be emulated in such a way as to perform the translation and permissions checking subject to the behavior of the guest platform.

Software approaches when faced with a memory access (in the base case), will traverse the guest page table to resolve the physical address, and check that the access satisfies the permissions imposed by the translation. These accesses will be subsequently sped up by introducing a software cache, much like a software *translation lookaside buffer* (TLB), so that future memory accesses do not incur a penalty of a costly page table walk. When the guest page tables change, the software TLB will be flushed, and the process will start again.

Since the native VM is a bare-metal environment, we can take full control of the native MMU, and use it to reflect the mappings of the guest, allowing us to use unmodified guest virtual addresses, and to emulate the access with a single native instruction.

Definition 3.6 (*Native MMU*). The **native MMU** is the MMU that is part of the **Native VM**. In this example, the MMU is an x86-64 MMU, which has a 4-level hierarchy.

Definition 3.7 (*Native Page Table Entry*). A **native page table entry** is an entry in the page table of the **native MMU**.

Definition 3.8 (*Guest MMU*). The **guest MMU** is a (software) implementation of the MMU that is part of the guest machine. In this example it is an implementation of an ARMv7-A MMU. It is implemented as a service that takes a virtual address, and returns either success (along with the corresponding physical address and a bitmask of allowed permissions), or failure (along with the type of failure).

Our approach to cross-architecture virtualization of the MMU is to present the lower 4GB (i.e. virtual addresses `0x0` to `0xffffffff` in the native VM's 48-bit address space) of virtual memory to the execution engine, as the 4GB (2^{32}) of virtual memory required for our 32-bit guest machine (see Figure 7). This area is now an exact 1:1 mapping of guest virtual addresses to native VM virtual addresses. Figure 5 shows how the various components work together. When we virtualize a memory access from the guest (whether a load, store or fetch), we perform that access on the *unmodified* memory address directly, which will of course (for the 32-bit system we are virtualizing) lie in the lower 4GB region. The first time a memory address is accessed, it will cause a page fault inside the native VM, and at this point we consult the software implementation of the guest's MMU. The response is either the corresponding *guest* physical address, or a fault condition. If the access is to be allowed, we then populate the x86 page table of the native VM with an entry that maps the associated virtual page to the corresponding physical page of the guest, and return to executing code. Further accesses to this page will now go via the native VM's page tables, and hardware TLB.

To improve performance, when the guest MMU is asked for the translation, it also returns the allowed permissions associated with that mapping (such as read/write and user/privileged), so that the native VM's page tables can be pre-populated with this information. This means that a read to a page that is also permitted to be written to will only fault once—the first time it is accessed.

On a 64-bit x86 machine, there are four levels of page tables, which we will refer to as **L4** thru **L1**. The first entry in the (top-level) **L4** page table represents the lower 0–512GB of virtual memory, and we reserve this region to contain the entire 4GB virtual address space of the guest, starting at virtual address `0x0`. This means we can apply permission flags to the first entry in this table, to control the entirety of the 4GB virtual address space. If the guest alters the content of their page tables, just as on actual hardware they are required to issue a TLB flush instruction, which we intercept and use as a signal invalidate the native VM's page tables. As the native VM has a four-level page table, we deny access to the entire lower 4GB area of virtual memory, by clearing the *page present* flag in the first entry of the **L4** page table, then perform a native TLB flush. This makes invalidations very quick to perform. The next time

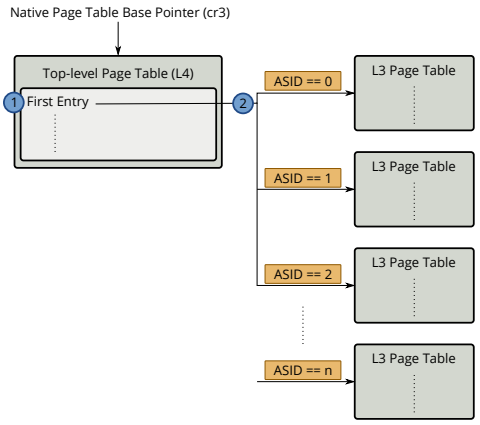


Fig. 6: The top-level (**L4**) page table remains static, and the pointer to the **L3** page tables (1) are tracked with the ASID (2).

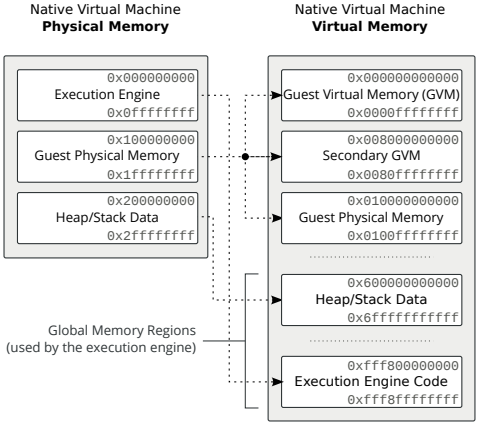


Fig. 7: Native VM *Physical* and *Virtual* memory organization. We reserve the bottom 512GB to contain the entire 4GB virtual address space.

a memory access happens, a page fault will occur, and the page tables will be rebuilt. This invalidation technique also applies when the guest changes the value of their own page table base pointer, which involves an implicit TLB flush.

An important corner-case to consider is the behavior of the system when an unaligned memory access spans a page boundary, e.g. a 32-bit memory access to the last byte of a page. This situation is handled automatically by CAPTIVE, as the page faulting behavior between the guest and host systems is identical.

3.3.1. Address-space Identifier. Usually, changing the page table base pointer naturally causes a TLB invalidation, as the previous mappings are no longer valid. However, since the page table base pointer is changed on every context switch, this can lead to a severe performance penalty, especially in our virtualization environment when the native page tables need to be rebuilt each time. An approach to reduce this penalty is described by [Wang et al. 2015] as "Private SPT", which utilizes the ARM address space identifier (ASID) register to quickly switch between pre-populated mappings.

We take inspiration from this approach, and use the ASID register to point to multiple **L3** mappings, as shown in Figure 6. The top-level native page table (the **L4** page table) remains static, but when the current ASID is changed by the guest, we replace the base pointer to the **L3** page table (in the first slot of the **L4** page table), and invalidate the native TLB. As previously described, the first entry in the **L4** page table is used solely for the purpose of managing guest virtual memory, so even though it represents an address space >4GB, it simplifies both our fast invalidation technique, and changing the corresponding page tables that represent the guest 4GB address space.

If this is the first time the ASID has been seen, the normal page-fault lazy resolution process will occur as described previously, but if the ASID has already been encountered, the page tables already contain mappings ready to be used (unless they were explicitly invalidated), without incurring any page faults.

We trap the special invalidation instructions issued by the guest to invalidate TLB entries by ASID, and use these signals to invalidate the page tables that we have associated with that particular ASID.

This optimization only holds for guest platforms that have the concept of an ASID, and guest kernels that actually use it (a limitation also encountered by [Wang et al.

Listing 5: ARM input assembly

```
; Read memory at address PC + 92 + 8
; (0x100a0) into r0
ldr  r0, [pc, #92]
```

Listing 6: CAPTIVE output assembly.

```
; Read memory from PC + offset + 8
mov    0x64(%r15d), %eax
mov    %eax, (%rdi)      ; Store into r0
lea    0x4(%r15d), %r15d ; Increment PC
```

Listing 7: QEMU output assembly.

```
mov    $0x100a0,%ebp   ; Prepare memory address
mov    %rbp,%rdi
mov    %ebp,%esi
; Calculate cache entry address
shr    $0x5,%rdi
and    $0xfffffc03,%esi
and    $0x1fe0,%edi
lea    0x2c18(%r14,%rdi,1),%rdi
cmp    (%rdi),%esi     ; Check cache tag
; Restore destination address
mov    %ebp,%esi
jne    0x7f4d682a718f  ; Cache-miss?
add    0x10(%rdi),%rsi
mov    (%rsi),%ebp     ; Read memory
mov    %ebp,(%r14)     ; Store into r0
```

Fig. 8: An example of a PC-relative load instruction being translated by CAPTIVE and QEMU. CAPTIVE tracks the (virtual) `PC` in `%r15d`, and emits three instructions for this memory access whereas QEMU emits 13 instructions that involve interrogating its address cache.

2015]). However, it is possible to extend this approach to track the guest platform's page table base pointer, and maintain a set of mappings for "seen" page table bases.

3.3.2. Native VM Memory Layout. As we are operating inside a native virtual machine, we have full control over the virtual machine's virtual memory and so we exploit this opportunity for manipulating the virtual page mapping arbitrarily. We establish page mappings for the execution engine and heap/stack data areas, and mark these entries as *global*, so that they are not flushed from the TLB when the TLB is flushed. We provide a one-to-one mapping of *guest* physical memory, in the virtual memory space so that we can quickly access data by guest physical address. This is useful for the emulated MMU, as it uses physical address pointers to traverse the guest page tables.

3.3.3. Secondary Guest Virtual Memory. The secondary guest virtual memory mapping is part of an optimization for handling ARM `ldrt` and `strt` instructions, which perform memory accesses subject to user-mode memory permission checking, whilst executing in kernel mode. These instructions are notoriously difficult to optimize [Ding et al. 2012], as they invoke behavior that must be specially handled. As they are defined, there is no direct mapping of this behavior from an ARM system to an x86-64 system, however to maintain performance we employ a second region of guest virtual memory to optimize these accesses specially.

Since it is known at JIT compilation time that a particular memory access has these special semantics, we can emit an optimized `mov` instruction, that offsets the calculated memory address against a base pointer held in the x86 `GS` register. This base pointer points to the base of the second virtual memory region, and so all memory accesses are made into this second region. Then, when a page fault occurs we apply the appropriate semantics when faulting the page in. Whilst this may sound like a guest architecture-specific optimization, it is implemented independent of the target architecture, and so may be used (or not) by any platform that requires it.

3.3.4. Comparison to QEMU. QEMU uses software-based MMU virtualization, and Listing 5 shows an example ARM instruction that accesses memory, from a PC-relative offset. This instruction loads a value from memory, residing at the address $PC + 92 + 8$. Listing 7 shows the QEMU generated native code for this single instruction, which

involves accessing a software cache, with a branch to a handler if a cache miss occurs. Our output code (shown in Listing 6) consists of performing the memory access directly on memory itself, using the unmodified value from the guest instruction and access permissions.

The other slight difference is the optimization performed for a PC-relative lookup. In QEMU's case, it can constant-fold the address of the memory access (`0x100a0`) into the generated assembly because it generates basic blocks for virtual pages. However, we generate basic blocks for physical pages, which may be accessed by any virtual address, and hence must read the `PC` register each time we wish to use it. As we map the guest `PC` to a host register, this improves code quality and adds virtually no performance penalty. This improvement in code quality is not because of an improvement in the quality of the JIT itself, but rather we have the ability to make memory accesses in this fashion.

3.4. Device Virtualization

In order to faithfully emulate a guest platform, we must also emulate the devices present on that platform, e.g. timer devices, interrupt controllers, I/O devices, etc. In order to do this, the hypervisor contains software emulations for the various devices that make up the platform. On a *real* guest platform, these devices are accessed by the guest through the memory subsystem; they are mapped into the physical memory space (and then mapped by the guest operating system into the virtual address space) and device registers are written to and read from with normal memory accesses. This approach to device communication increases flexibility (e.g. device accesses are subject to MMU translations and permissions checks), and reduces complexity for operating systems, but adds a layer of complexity to virtualization frameworks wishing to emulate devices in a particular platform, as they must detect these accesses to device memory, and handle them accordingly.

As our device implementation lives in the hypervisor (i.e. outside of the native VM), memory accesses by the guest must be trapped back to the hypervisor, so that they can be forwarded to the particular device being accessed. The most straightforward way to accomplish this with our infrastructure is to use the *memory-mapped I/O* (MMIO) feature of KVM to intercept memory accesses to regions of guest physical memory that correspond to devices, and handle them accordingly. This approach works well, but suffers from a severe performance penalty, as every access to a device must perform a costly **VM exit**, then the native guest instruction must be emulated by the hypervisor to fill in the data that was read, or to extract the data that is to be written.

Device accesses in a full-system occur quite frequently. For example, a Linux system configured with a 100Hz timer will be interrupted 100 times a second, and each interrupt requires the guest to interrogate the interrupt controller device to ascertain the cause of the interrupt, then the timer device to read timing related data, then write to the devices to acknowledge and complete the interrupt.

Another approach is to make a hypercall using *port-based I/O* (PIO) instructions, which have slightly faster VM exit sequences, but this suffers from a fundamental problem: detecting a device access. As mentioned previously, a device access to a memory-mapped device is indistinguishable from a normal memory access at the instruction-level—it is performed with a normal memory access instruction (e.g. `ldr` in ARM). Therefore, we need a way to detect access to device memory, and trap to the host using a faster *hypercall* mechanism. Since we are in control of the native VM MMU, and we know the locations of devices in physical memory (this is part of the platform configuration), we can simply mark any device page as inaccessible, so that every memory access traps in the native VM, rather than in the hypervisor.

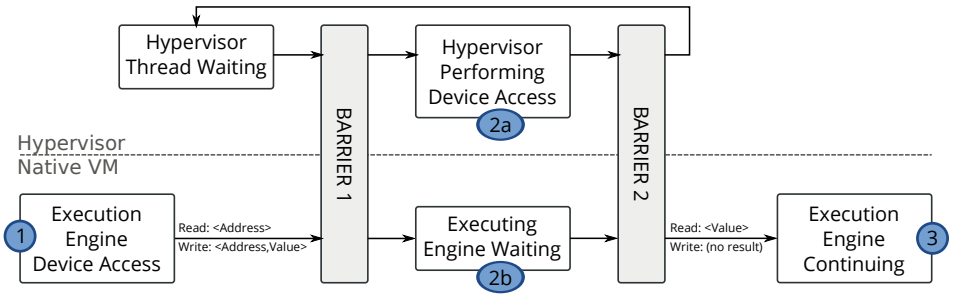


Fig. 9: An illustration of the fast device access operation, using synchronization barriers. When a device access is made (1), barrier 1 is entered by the guest (at which the host is already waiting) and the host performs the access on the emulated device (2a). Meanwhile, the guest waits for the host to complete the operation (2b). Then, when the access is complete, barrier 2 is entered by the host and execution by the guest continues (3).

Now that we are receiving a page fault in the native VM (which is faster than trapping to the hypervisor), there are two approaches to take:

(1) Translate the device access into a (slightly) faster PIO access, which still results in a VM exit, or
(2) use a message-passing implementation to communicate with the hypervisor, avoiding a VM exit.

We wish to avoid VM exits at all costs, as they introduce a significant amount of overhead [Ott 2009]. A VM exit with Intel VT and KVM requires storing the entire state of the virtual machine, and performing a context switch back to user-space code. Returning to the VM (a VM entry) involves restoring this saved state.

For this reason, we implement (2) and once the native VM receives a page fault to a device memory page, we communicate with a hypervisor thread using a synchronization barrier system. This avoids a costly VM exit, as the virtualized CPU is simply spinning on a barrier, waiting for a response from the hypervisor. This sequence is shown in Figure 9. When a device access is to be made Fig. 9 (1), a data structure is prepared by the execution engine inside the native VM, and a synchronization barrier is entered. A hypervisor thread (which is already waiting on this barrier) resumes execution and deals with the device access request Fig. 9 (2a). Meanwhile, the guest waits on a second barrier Fig. 9 (2b) whilst the hypervisor is servicing the request, and when the request is complete, the hypervisor writes the result back into the data structure, and enters the barrier. This causes the execution engine to resume execution Fig. 9 (3), extracting the necessary data from the request structure. The guest cannot proceed until the hypervisor has signalled that the data has been processed by the emulated device, and this is the reason for the second barrier.

3.4.1. Device Implementations. Unlike traditional same-architecture virtualization, where the possibility exists to *para-virtualize* hardware that exists on the host for use by the guest, or simply pass-through real hardware devices (e.g. using Intel VT-d) this same kind of mapping does not exist for cross-architecture virtualization as it is unlikely that there are any 1-to-1 compatible devices available on the host system. Therefore, all guest platform devices are implemented in software, which faithfully emulate the behavior of the device they represent. An example of a device we implement in software is the ARM PrimeCell SP804, which is a two-channel timer device. This

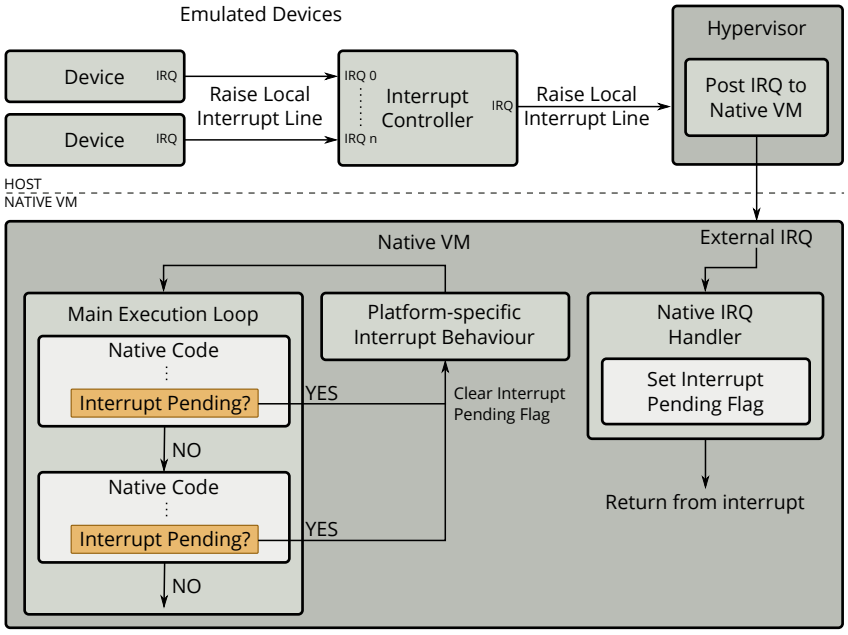


Fig. 10: An illustration of the injection of an IRQ into the native virtual machine, to indicate that an emulated IRQ line has gone high.

device is configured and interrogated by the guest through registers that are memory mapped. It is also capable of raising interrupts when a timeout occurs, depending on the mode of operation of the timer.

In the future, we hope to map similar devices (e.g. timer devices, etc) to existing hardware devices. Even though their interfaces may be incompatible, it may be possible to configure the behavior of the devices in similar ways and avoid having to use full software implementations of the device.

3.4.2. Device Interrupts. Platform devices may raise interrupts to indicate that an event has occurred, such as a timer has timed-out, or data is ready to be read. On a physical platform, an interrupt controller would aggregate the individual interrupts from each device, and trigger a physical interrupt line on the CPU, to indicate that an interrupt has been raised. The CPU would enter its external interrupt handling routine, and interrogate the interrupt controller to work out which device(s) raised the interrupt. The RealView Platform Baseboard Cortex-A8 [ARM 2011b] has such a setup with an ARM generic interrupt controller (GIC), that receives interrupts from devices and posts these to the CPU. We implement the GIC in software, but post real IRQs to the guest system, when the interrupt controller triggers a physical interrupt line on the CPU.

3.5. IRQ Virtualization

As described in the previous section, emulated devices may issue interrupts to the guest system by means of an interrupt controller. For the platform we are virtualizing, the interrupt controller is an ARM *generic interrupt controller* (GIC), which aggregates interrupts from other platform devices, and presents them to the CPU.

Fundamentally, the CPU has a single physical interrupt line that is raised when an interrupt is pending, and lowered when the interrupt is acknowledged. This interrupt line is toggled by our emulated GIC, and is visible to the virtualized CPU. On the rising

System	Dell™ PowerEdge™ R610		
Architecture	x86-64	*Model*	Intel™ Xeon™ E5-1620
Cores / Threads	4/8	*Frequency*	3.50 GHz
L1-Cache	1× 4× 32kB (*I$ & D$*)	*L2-Cache*	1× 4× 256kB
L3-Cache	1× 10 MB	*Memory*	16 GB

Table I: Host Machine Description

edge of the interrupt line, we inject a native IRQ into the guest machine, to inform it that the line has been raised. These interrupts of course happen asynchronously, e.g. a timer device will run as a separate thread on the host machine, and when its timeout occurs, it will trigger its own interrupt line, propagating through the interrupt controller and into the guest. The ideal situation would be to immediately invoke the platform-specific interrupt handling code, on the rising edge of the interrupt line, but this is not feasible for two reasons:

(1) The guest may not be running in native code (it may be handling a page fault) and,
(2) single guest instructions are compiled to multiple host instructions, which means the interrupt may happen part-way through the emulation of a guest instruction.

This is unacceptable, as guest instructions are not necessarily re-entrant and may have partially changed the state of the guest system mid-way through. Guest instructions need to appear to be atomic, and so they must have completed before we can divert to the interrupt handling behavior.

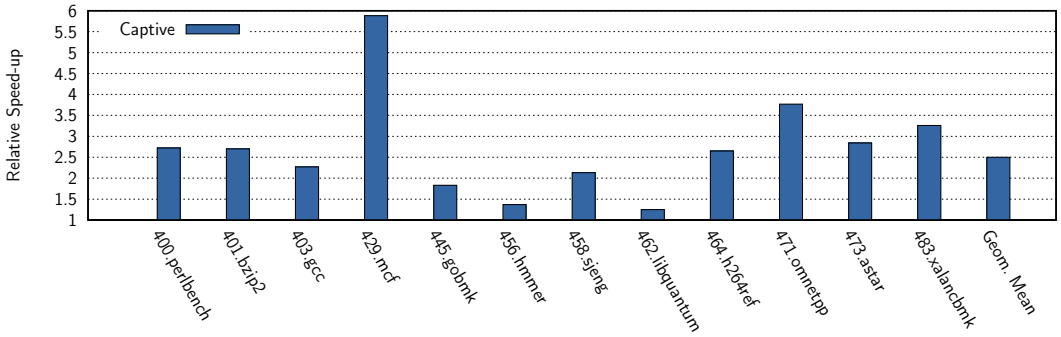
We solve this by setting an interrupt pending flag when in the native IRQ handler, to indicate that the emulated interrupt line has gone high. This flag is checked by native code at the end of a guest basic block, before it chains to the next. If the flag is set, it is cleared and we leave native code to perform the guest platform behavior associated with servicing an interrupt. This defers asynchronous interrupt checking to basic block boundaries, which significantly improves performance over checking on instruction boundaries.

4. EXPERIMENTAL EVALUATION

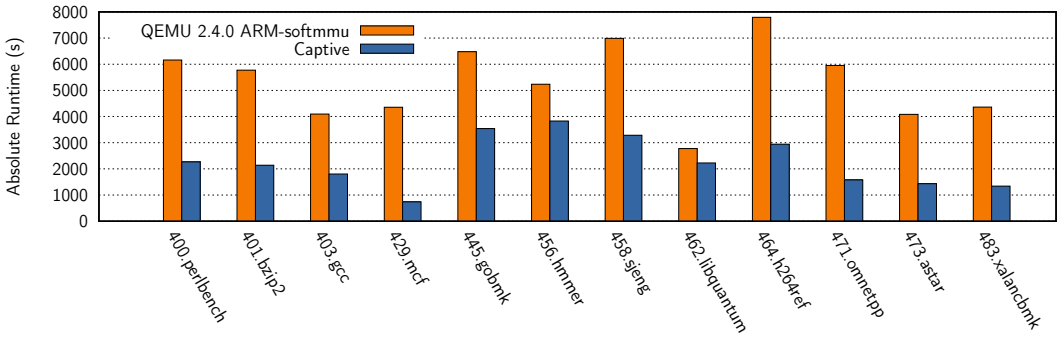
In this section, we evaluate the performance of our system using industry standard benchmarks and compare CAPTIVE to the state-of-the-art cross-architecture virtualizer QEMU. We use the SPEC CPU2006 integer benchmark suite, as it is widely considered to be representative of real-world workloads. For our key results, we are using the *reference* input set, which requires a minor modification to the guest platform to increase the available guest physical memory for running the benchmarks. The amount of physical memory presented to the guest system is independent of the amount of physical memory available on the host system, as it is defined by the platform being emulated. We implement a RealView Platform Baseboard Cortex-A8 [ARM 2011b] platform, which specifies only 512MB of physical memory [ARM 2011a], but this is insufficient for running the reference input set of the benchmark suite. To overcome this limitation, we artifically increase the amount of physical memory in the guest platform to 2GB in both CAPTIVE and QEMU, enabling the benchmark suite to run.

4.1. Experimental Setup

The platform that we are virtualizing is a RealView Platform Baseboard for Cortex-A8, which is fully supported by QEMU. We are running a vanilla ARM Linux 4.3.0 kernel, with the default configuration for the platform, except for the addition of a VirtIO block device to provide storage to the guest and an increase in physical memory as



(a) Relative speed-up of the SPEC benchmark suite using the reference input set, in CAPTIVE over QEMU–higher is better.



(b) Absolute runtime of the SPEC benchmark suite using the reference input set in seconds–lower is better.

Fig. 11: Key Results: (a) shows relative speed-up, and (b) shows absolute run time. On average, CAPTIVE is $2.5\times$ faster than QEMU.

described previously. We are using an Arch Linux ARM user-space. The host machine is described in Table I.

4.2. Key Results

Our *key results* compare the performance of our system to QEMU version 2.4.0. Figure 11a shows the relative speed-up of CAPTIVE, compared to QEMU. In all cases we outperform QEMU, and on average by a factor of $2.5\times$. Figure 11b shows the absolute runtime of each benchmark in seconds.

Of interest is 429.mcf, which gains a performance improvement of $5.88\times$. This is due in part to the benchmark responding well to our optimizing DBT system, which produces highly optimal runtime code based on the dynamic behaviour of the benchmark, versus the static optimization that is performed at compile time.

Only two out of twelve benchmarks show speed-ups less than $1.5\times$, yet still outperform the baseline QEMU. Given the acceptance of SPEC as a realistic workload, there are multiple characteristics that can affect simulation performance, and it is clear that the range of benchmarks exercise the simulation system in numerous ways, making it difficult to pin-point any particular feature that causes fluctuations in performance.

4.3. Comparison to Existing Techniques

One of the most recent efforts to improve memory address translation performance in full-system simulators is in [Wang et al. 2015] (herein referred to as HSPT), which

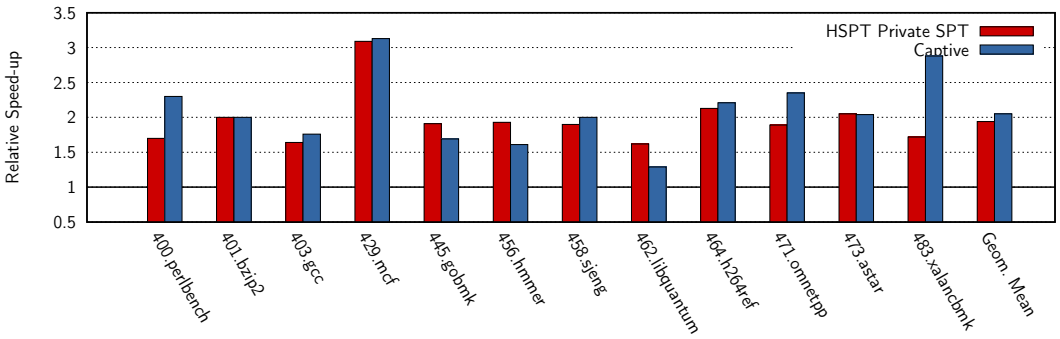


Fig. 12: Relative performance improvement of SPEC benchmarks by HSPT and CAPTIVE over the Android Emulator baseline–higher is better.

describes a practical implementation of an *embedded shadow page table*, using Linux system calls (specifically `mmap`) to create an efficient *guest*-virtual to *host*-physical mapping similar to our own approach. In order to compare CAPTIVE to the HSPT implementation, we have extracted the published results from [Wang et al. 2015] and implemented the same experimental setup, by comparing the performance of CAPTIVE to the same version and configuration of the Android Emulator as used by HSPT. This enables us to make a relative performance comparison against the same baseline, even in the presence of different host machines.

Figure 12 shows that HSPT have achieved an average improvement of $1.94\times$ (geometric mean) over the Android Emulator, using the *Private SPT* technique, whereas on average, CAPTIVE achieves a performance improvement of $2.05\times$ (geometric mean).

In the majority of cases, we equal or surpass the speed-up presented by HSPT, in particular 483.xalancbmk in CAPTIVE shows a much greater speed-up of $2.88\times$, compared to $1.72\times$ in HSPT. This is due in part to the I/O nature of this particular benchmark, and our optimized I/O and IRQ handling techniques give us a clear advantage here.

4.4. I/O Performance

In this section, we evaluate the performance of our I/O virtualization, using the standard Linux I/O performance measuring tool `hdparm`. We measure the I/O performance on a variety of virtualization configurations, including taking a measurement of the host system. We also introduce Oracle VirtualBox as another virtualization platform that uses Intel VT extensions, and as such only supports same-architecture virtualization. For measurement of same-architecture virtualization I/O performance, VirtualBox and QEMU are given an x86 Linux distribution containing the `hdparm` tool. For cross-architecture virtualization, QEMU and CAPTIVE are provided with a file-system that exists as a normal file on the host machine's file-system. For QEMU/ARM and CAPTIVE/ARM, the platform device used to communicate this data back and forth is a VirtIO block device, which is fully supported by both hypervisors. VirtIO is a virtualization technology that enables efficient paravirtualization of various platform devices, such as network and disk devices. Our emulated disk is based on a VirtIO block device, and is the only paravirtualized device in the platform.

Table II shows the absolute I/O throughput of the virtualization configurations, along with throughput on the native host platform, using two distinct metrics: *cached* and *buffered*.

Cached reads are subject to the Linux kernel page cache, and as such represent the performance at which disk data can be accessed from the page cache in the guest sys-

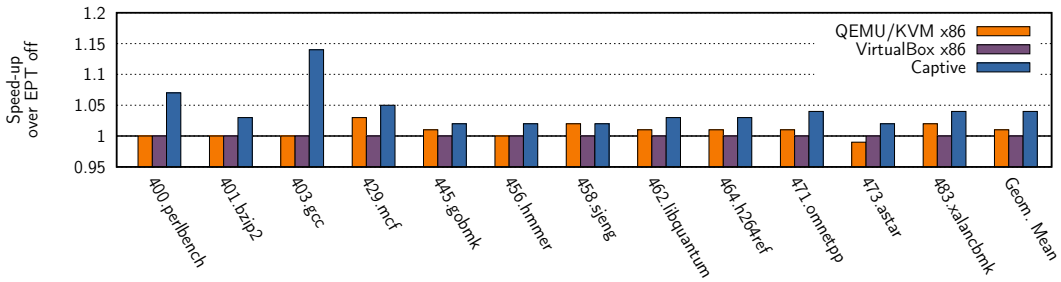


Fig. 13: Relative performance improvement gained by turning on Intel's *extended page tables* (EPT) for the SPEC CPU2006 integer benchmark suite–higher is better. We show that the use of EPT has virtually no effect on the virtualization performance for the SPEC CPU2006 benchmark suite.

tem. VirtualBox and QEMU/KVM make these accesses at virtually the same rate as the host platform, since there is no virtualization overhead for memory accesses. For QEMU/DBT, the accesses to this cache are subject to the software MMU implementation, and therefore incur a significant access penalty. For QEMU/DBT, in the x86-on-x86 case this causes a slow-down of $9.79\times$, and a slow-down of $78.87\times$ for the ARM-on-x86 case. In CAPTIVE, the slow-down is only $7.3\times$, improving over QEMU by $10.8\times$.

Buffered reads indicate the rate at which data can be accessed directly from disk—bypassing the page cache. For these experiments, *host* caching was disabled in each hypervisor, causing all accesses to the virtual disk device to go directly to the host file-system, and then onto the underlying storage medium. All hypervisors suffer a slow-down over native for this case, as there will be overhead in accessing the virtual disk on the host file-system, but the slow-down over native for CAPTIVE is only $1.11\times$, compared to QEMU/ARM being $1.64\times$. Virtualization of the x86 guest machines on VirtualBox, QEMU/KVM and QEMU/DBT all have even worse slow-downs, but this may be due to the implementation of the virtual disk device, which for these three hypervisors is an emulated IDE disk drive, as opposed to the para-virtualized VirtIO device used in QEMU/ARM and CAPTIVE.

4.5. Additional Hardware Support for MMU Virtualization

The latest version of Intel VT includes hardware support for accelerating virtualized guest page tables, which is branded as *extended page tables* (EPT). KVM can make full use of this technology, and this section evaluates the performance improvement of EPT over non-EPT backed virtualization. We use QEMU/KVM and Oracle VirtualBox (which fully supports EPT) to measure the impact of EPT on same-architecture virtualization. We have run six experiments to produce this data, three with EPT disabled

Hypervisor	Execution	Arch.	Cached	Buffered
None	Native	x86	12384.21 MB/s	173.52 MB/s
VirtualBox	Intel-VT	x86	11941.43 MB/s	91.64 MB/s
QEMU	KVM	x86	11881.06 MB/s	102.72 MB/s
QEMU	DBT	x86	1265.03 MB/s	79.80 MB/s
QEMU	DBT	ARM	157.02 MB/s	105.77 MB/s
CAPTIVE	KVM/DBT	ARM	1695.29 MB/s	155.72 MB/s

Table II: Absolute I/O throughput for various configurations of execution environments. Cached reads are subject to the Linux kernel's page cache, and buffered reads go directly to the real or emulated disk device.

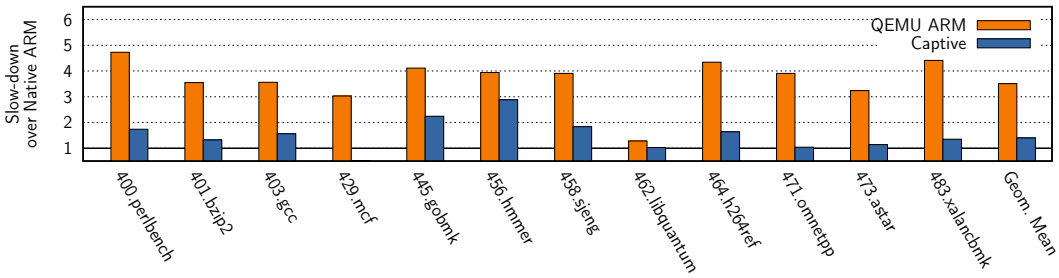


Fig. 14: Relative slow-down of QEMU and CAPTIVE over native execution on a physical ARM platform (ODROID-XU using Samsung Exynos5422 Cortex-A15 2.0Ghz quad-core and Cortex-A7 quad-core CPUs)–lower is better. On average, CAPTIVE is $1.4\times$ slower than the hardware platform, compared to a $3.51\times$ slow-down for QEMU.

in the respective hypervisor, and three with EPT enabled. We then present the relative speed-up of each hypervisor with EPT enabled, over EPT disabled. For QEMU/KVM and Oracle VirtualBox the experiments were naturally made on a virtualized x86-64 system, with x86-64 versions of the SPEC benchmark suite. For CAPTIVE, we have used the same setup as described in Section 4.1, with EPT enabled and disabled.

The data shows that in our experiments, EPT does not make any significant improvement on the workloads we have tested. This is contrary to some published experiments, e.g. VMware have conducted a performance evaluation of EPT in [VMware 2009], which shows that EPT can improve performance of MMU-intensive benchmarks by 48%, and MMU-microbenchmarks by up to 600%. However, our measurement of the impact of EPT on the SPEC CPU2006 benchmarks shows that the performance increase to be negligible, which is also the conclusion drawn by Buell et al. [2013] and Merrifield and Taheri [2016]. Figure 13 shows the relative performance improvement of the SPEC benchmark suite, running on both a virtualized x86 system (using QEMU/KVM and Oracle VirtualBox) and on a virtualized ARMv7-A system (using our virtualization hypervisor). On average, there is virtually no improvement for QEMU and VirtualBox, and only 3% for CAPTIVE.

4.6. Slow-down over Native Execution on High-End Hardware

We have evaluated the performance of CAPTIVE, compared to execution of the benchmarks on an ARM hardware platform. We have collected run times on an ODROID-XU, and Figure 14 shows the relative slow-down of both CAPTIVE and QEMU. On average, CAPTIVE is $1.4\times$ slower than native execution of SPEC on an ARM platform, whereas QEMU is $3.51\times$ slower. Again, of interest is the 429.mcf benchmark that actually shows a speed-up over native. This is again partly due to the JIT compiler discovering optimizations that can be made dynamically, but also due to larger CPU cache (e.g. L1) sizes on the host platform.

5. RELATED WORK

Instruction set simulation is an active field of research and a large number of techniques for the efficient implementation of either user mode or full system simulators have been published, e.g. [Böhm et al. 2011; Böhm et al. 2010; Witchel and Rosenblum 1996; Binkert et al. 2011; Patel et al. 2011; Sandberg et al. 2015; Yourst 2007; Bellard 2005; Ding et al. 2011; Magnusson et al. 2002; Qin and Malik 2003; AMD Developer Central 2010]. In Table III we provide an overview of well-known simulators, their capabilities and implementation techniques.

Simulator	Engine	Full-System	Multi-Core	Detailed	Hardware Accelerated	Target ISA
ArcSim	Parallel DBT	Yes	Yes	Config.	No	User Retargetable
Embra	DBT	Yes	Yes	Cache	No	MIPS R3000/R4000
gem5	Discr. Event	Yes	Yes	Yes	No	User Retargetable
MARSS	DBT	Yes	Yes	Yes	No	Intel x86
OVPSim	DBT	Yes	Yes	No	No	Multiple available
pFSA	Direct Exec.	Yes	No	Sampling	For same ISA	Intel x86
PTLsim	Virtualization	Yes	No	Yes	No	Intel x86-64
QEMU	DBT	Yes	Yes	No	No	Multiple available
QEMU	KVM	Yes	Yes	No	For same ISA	Multiple available
PQEMU	DBT	Yes	Yes	No	No	ARM11MPCore
Simics	Interpreter	Yes	Yes	Approx.	No	Multiple available
Simit-ARM	DBT	Yes	No	No	No	ARM v5
SimNow	DBT	Yes	Yes	(COTSon)	No	Intel x86, AMD64
CAPTIVE	DBT	Yes	Yes*	Cache*	**Yes**	User Retargetable

(*) Multi-core virtualization and optional cache modelling are beyond the scope of this article.

Table III: Comparison of simulators: techniques and capabilities.

Early work in the context of Simics introduced a software caching mechanism, which improved the performance of interpreted memory operations by reducing the number of calls to complex memory simulation code [Magnusson and Werner 1994]. More recently, Chang et al. [2014], Wang et al. [2015] Hong et al. [2015] have presented novel schemes for speeding up address translation in full-system simulators. These two schemes are the closest matches to our work documented in the literature. In [Chang et al. 2014] a shadow page table – called embedded shadow page table (ESPT) – is embedded into the address space of a cross-ISA dynamic binary translation (DBT) system. Similar to CAPTIVE, ESPT uses the hardware memory management unit in the CPU to translate memory addresses, instead of software translation. However, the original ESPT approach has a few drawbacks. For example, its implementation relies on a loadable kernel module (LKM) to manage the shadow page table. Using LKMs is less desirable for system virtual machines due to portability, security and maintainability concerns. Hence, a different implementation – called HSPT – adopts a shared memory mapping scheme to maintain the shadow page table using only `mmap` system calls [Wang et al. 2015]. In section 4.3 we show a side-by-side performance comparison between this improved HSPT scheme and the approach taken by CAPTIVE. Dynamic resizing of a software TLB is proposed in [Hong et al. 2015]. Using per-page-table utilization information the size of the software TLB is adjusted for each process separately.

6. SUMMARY, CONCLUSION AND FUTURE WORK

We have introduced new techniques for cross-architecture virtualization, using hardware accelerated processor extensions and implemented these ideas in a hypervisor called CAPTIVE. The key contribution is the mapping of guest system MMU behavior to host system MMU behavior, and we improve over the state-of-the-art simulator QEMU on average $2.5\times$. We show that CAPTIVE is better than existing techniques to improve MMU virtualization. There are three major routes that we wish to take to extend our work:

(1) We wish to extend the capability of our system to 64-bit guests, and find efficient ways of exploiting our existing infrastructure to handle the larger address space.
(2) We wish to implement a multicore version of the execution engine, to support platforms with multiple (possibly heterogeneous) processor cores.
(3) We wish to explore the possibility of mapping guest platform devices to real host devices, e.g. timer devices, eliminating hypervisor emulation overhead.

REFERENCES

AMD Developer Central. 2010. AMD SimNow simulator. http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/. (2010).

ARM. 2011a. About the PB-A8. (2011). http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/BABCHBFC.html"#CHDFGCFB Retrieved 02-June-2016.

ARM. 2011b. RealView Platform Baseboard for Cortex-A8 User Guide. (2011). http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0417d/index.html Retrieved 02-June-2016.

Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. 2005. The ArchC Architecture Description Language and Tools. *Int. J. Parallel Program.* 33, 5 (Oct. 2005), 453–484. DOI: http://dx.doi.org/10.1007/s10766-005-7301-0

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: http://dx.doi.org/10.1145/2024716.2024718

Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. 2011. Generalized Just-in-time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 74–85. DOI: http://dx.doi.org/10.1145/1993498.1993508

Igor Böhm, Björn Franke, and Nigel P. Topham. 2010. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (ICSAMOS 2010), Samos, Greece, July 19-22, 2010*, Fadi J. Kurdahi and Jarmo Takala (Eds.). IEEE, 1–10. DOI: http://dx.doi.org/10.1109/ICSAMOS.2010.5642102

Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. 2009. Fast and accurate simulation using the LLVM compiler framework. In *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*.

Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and H. Reza Taheri. 2013. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. VMware technical journal. (2013). https://labs.vmware.com/vmtj/methodology-for-performance-analysis-of-vmware-vsphere-under-tier-1-applications

Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. 2009. A High-level Virtual Platform for Early MPSoC Software Development. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '09)*. ACM, New York, NY, USA, 11–20. DOI: http://dx.doi.org/10.1145/1629435.1629438

Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Efficient Memory Virtualization for Cross-ISA System Mode Emulation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*. ACM, New York, NY, USA, 117–128. DOI: http://dx.doi.org/10.1145/2576195.2576201

J. H. Ding, P. C. Chang, W. C. Hsu, and Y. C. Chung. 2011. PQEMU: A Parallel System Emulator Based on QEMU. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*. 276–283. DOI: http://dx.doi.org/10.1109/ICPADS.2011.102

Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. 2012. ARMvisor: System virtualization for ARM. In *Ottawa Linux Symposium*.

K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. 2001. Dynamic binary translation and optimization. *IEEE Trans. Comput.* 50, 6 (Jun 2001), 529–548. DOI: http://dx.doi.org/10.1109/12.931892

Adam Gerber and Clifton Craig. 2015. *Learn Android Studio: Build Android Apps Quickly and Effectively* (1st ed.). Apress, Berkely, CA, USA.

Apala Guha, Kim hazelwood, and Mary Lou Soffa. 2010. DBT Path Selection for Holistic Memory Efficiency and Performance. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '10)*. ACM, New York, NY, USA, 145–156. DOI: http://dx.doi.org/10.1145/1735997.1736018

Ding-Yong Hong, Chun-Chen Hsu, Cheng-Yi Chou, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. 2015. Optimizing Control Transfer and Memory Virtualization in Full System Emulators. *ACM Trans. Archit. Code Optim.* 12, 4, Article 47 (Dec. 2015), 24 pages. DOI: http://dx.doi.org/10.1145/2837027

Intel. 2016. Intel Virtualization Technology (Intel VT). (2016). http://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html Retrieved 26-April-2016.

Daniel Jones and Nigel Topham. 2009. High Speed CPU Simulation Using LTU Dynamic Binary Translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '09)*. Springer-Verlag, Berlin, Heidelberg, 50–64. DOI: http://dx.doi.org/10.1007/978-3-540-92990-1_6

Naveen Kumar, Bruce R. Childers, Daniel Williams, Jack W. Davidson, and Mary Lou Soffa. 2005. Compile-time Planning for Overhead Reduction in Software Dynamic Translators. *Int. J. Parallel Program.* 33, 2 (June 2005), 103–114. DOI: http://dx.doi.org/10.1007/s10766-005-3573-7

KVM. 2016. KVM. (2016). http://www.linux-kvm.org/page/Main_Page Retrieved 26-April-2016.

Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: a Full System Simulation Platform. 35, 2 (Feb. 2002), 50–58. http://dlib.computer.org/co/books/co2002/pdf/r2050.pdf; http://www.computer.org/computer/co2002/r2050abs.htm

Peter S. Magnusson and Bengt Werner. 1994. *Some Efficient Techniques for Simulating Memory*. Technical Report R94. Swedish Institute of Computer Science technical report.

Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

Timothy Merrifield and H. Reza Taheri. 2016. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*. ACM, New York, NY, USA, 25–35. DOI: http://dx.doi.org/10.1145/2892242.2892258

David Ott. 2009. Virtualization and Performance: Understanding VM Exits. (2009). https://software.intel.com/en-us/blogs/2009/06/25/virtualization-and-performance-understanding-vm-exits Retrieved 07-June-2016.

A. Patel, F. Afram, S. Chen, and K. Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*. 1050–1055.

Wei Qin and S. Malik. 2003. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Design, Automation and Test in Europe Conference and Exhibition*. 556–561. DOI: http://dx.doi.org/10.1109/DATE.2003.1253667

A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. 2015. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*. 183–192. DOI: http://dx.doi.org/10.1109/IISWC.2015.29

Tom Spink, Harry Wagstaff, Björn Franke, and Nigel Topham. 2014. Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. ACM, 3–12.

David Ung and Cristina Cifuentes. 2000. Machine-adaptable Dynamic Binary Translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. ACM, New York, NY, USA, 41–51. DOI: http://dx.doi.org/10.1145/351397.351414

VMware. 2009. *Performance Evaluation of Intel EPT Hardware Assist*. Technical Report. VMware. https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. 2013. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *Proceedings of the Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 21, 6 pages. DOI: http://dx.doi.org/10.1145/2463209.2488760

Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. 2015. HSPT: Practical Implementation and Efficient Management of Embedded Shadow Page Tables for Cross-ISA System Virtual Machines. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 53–64.

Emmett Witchel and Mendel Rosenblum. 1996. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*. ACM, New York, NY, USA, 68–79. DOI: http://dx.doi.org/10.1145/233013.233025

M. T. Yourst. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*. 23–34. DOI: http://dx.doi.org/10.1109/ISPASS.2007.363733