Modularizing CPU Semantics for Virtualization

Paolo Giarrusso
BedRock Systems, Inc
paolo@bedrocksystems.com

Abhishek Anand
BedRock Systems, Inc
abhishek@bedrocksystems.com

Gregory Malecha
BedRock Systems, Inc
gregory@bedrocksystems.com

František Farka
BedRock Systems, Inc
frantisek@bedrocksystems.com

Hoang-Hai Dang
BedRock Systems, Inc
hai@bedrocksystems.com

Abstract

Hardware-assisted virtualization simplifies *implementing* virtual machine monitors (VMM), but verifying a VMM using hardware virtualization requires a precise model of the virtualization. A natural specification for a VMM is that it refines the bare-metal hardware *specification*; however, a simple backwards simulation on the processor semantics would require case distinction on all instructions, showing that they refine the non-virtualized counterparts. This case analysis is prohibitively expensive for modern architectures. Further, correctness of hardware virtualization is mainly the responsibility of the ISA architects, not the VMM implementer. In this paper, we propose a modularization of hardware semantics that succinctly captures the difference between the "bare-metal" behavior and hardware-virtualized behavior of a processor. We illustrate how this helped us modularize the verification of our VMM.

1 A Family of Processor Models

To facilitate reasoning about how virtualization state affects the behavior of the processor, we want a processor model that allows us to model virtualization state precisely while abstracting the semantics that are not virtualization-relevant, e.g. arithmetic operations. Decomposing the processor state into virtualization-relevant state and non-virtualization relevant state allows us to precisely model the former while abstracting the later. Following the pattern of SAIL [1], we postulate a description of each instruction's semantics in a core language and describe the semantics of the processor as executing a tree of instructions (ToI) [2], which allows processors to interleave and speculate the execution of multiple instructions. In SAIL, the effects in this core language are memory operations; in our virtualization model, we add effects for accessing virtualization-relevant state. These virtualization effects are interpreted by event handlers that are still internal to the processor as shown in Figure 1. Virtualization events from the RTL core are either handled using internal state of the handler producing τ steps for the full processor model, or are translated to external events such as

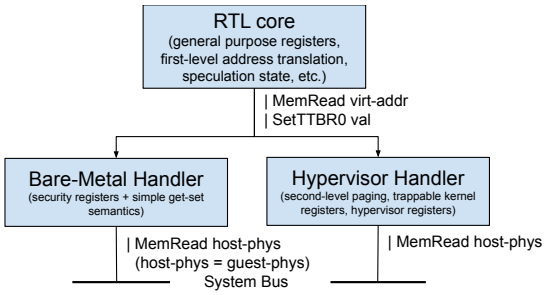

Figure 1. Decomposition of the processor model to isolate virtualization-relevant state.

memory accesses using the virtualization state, e.g. 2nd-stage page tables (see section 1.2).

1.1 Example: Control Registers

To better understand the decomposition, consider the virtualization flag on ARM that enables trapping on changes to the page table base register (TTBR0). To ensure that the virtualization handler gets access to all changes to this register, we remove TTBR0 and the flag from the RTL core state and only provide access to them via get and set events. The bare-metal handler (with no virtualization extensions) simply handles these events by manipulating its own value of TTBR0. The virtualized handler would instead consult the flag and inform the RTL core that it mustn't complete the write, but rather trap. If the write was executed out of order, the RTL core will defer the trap appropriately to emulate in-order execution.

This decomposition eases reasoning precisely about the behavior of the flag, while abstracting the details of which instructions actually produce the relevant events. To prove virtualization of TTBR0 correct, we simply prove that "TTBR0 emulation works": our VMM's software handler for trapped accesses to TTBR0 refines untrapped accesses to TTBR0 in the non-virtualized semantics.

1.2 Example: Memory

Memory accesses in virtualized CPUs add extra steps compared to unvirtualized CPUs, which we encapsulate in effect handlers. Virtualized CPUs have two stages of address translation. The first stage translates guest virtual addresses to

PRiSC'24, ,

guest physical addresses (GPAs) or invokes the guest page-fault handler. This is guest behavior and is encapsulated entirely in the RTL core. The second stage is exclusive to virtualized CPUs and translates GPAs to host physical addresses (HPAs), or faults to the hypervisor. This translation is implemented by the handler for virtualized systems. For unvirtualized systems, the translation is the identity.

To prove virtualization of memory correct, we must simply prove that "memory emulation works": accessing virtualized memory refines accessing non-virtualized memory. If the guest is reading GPA 0x4000, the VMM must ensure the second-stage translation maps that address to a host address with the correct value, *or* the VMM must trap the access and emulate it, for example by probing a virtual device.

Justifying this split is more subtle: in hardware, page table translations for both stages are carried out by a single memory management units (MMUs) and results are cached by translation lookaside buffers (TLBs). Yet, we abstract from the hardware and split TLB and MMU between the RTL core and event handler: the RTL core handles first-stage translation and encapsulates associated TLB state, while the event handler implements second-stage translation (including associated TLB state).

2 Proof Sketch of our VMM

The top-level specification of VMM correctness is a refinement stating that the VMM running a guest binary refines the behavior of a bare-metal computer without virtualization extensions running the same binary. The full simulation relies on decomposing the full computer LTS, which is comprised of memory, devices and processors. Here we focus on the simulation for a single CPU leveraging the model above.

Figure 2 illustrates the execution of the CPU and how it preserves the simulation relation. Each rectangle denotes a state of the core RTL semantics of a CPU core and excludes the state of the handlers. The bottom row corresponds to the implementation which includes the VMM software and the host with virtualization extensions. The top row corresponds to the spec: the idealized bare metal CPU. Modern CPU core threads can execute steps of multiple instructions in parallel. Each circle in a rectangle represents the state of an execution of an instruction. The arrows indicate program order. In general, this can look like a tree of instructions because at branch instructions, modern CPUs can speculatively execute parts of multiple branches. Each "s" in the circle denotes an RTL micro-step that remains to be executed. Each red circle denotes an instruction whose execution has run into a fault, e.g. because it tried to change the TTBR0 register.

The main advantage of our modularization is that simulation relation between the RTL core parts are simple: it says that the core RTL states as shown in the diagram are almost identical, except in case of red instructions, where

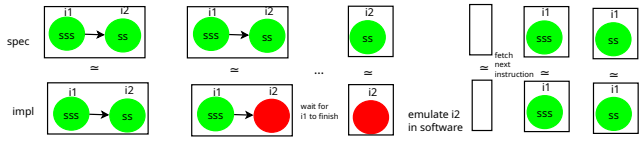

Figure 2. The simulation relates trees of instructions (boxes). Instructions (circles) match exactly except when the instruction is faulting (red). Virtualization state is not shown.

the corresponding instruction in the bare metal is stuck before the faulting step. In such cases, first, the virtualization hardware waits for previous instructions to fully finish, then saves the register state and traps to the VMM, providing it the register state at the end of the last-finished instruction (i1 in this case). ARM has some "decode-assist" registers, which provides VMM implementations with an encoding of the instruction i2 and often even information gathered while executing i2 until the fault happened. For example, if a load instruction faulted because the second-stage page table had missing read permission, decode-assist registers may include the faulting address. The VMM uses decode assists and the full register state to emulate the instruction i2 in software to match what would have happened in the bare-metal. After the emulation, the VMM "returns" to hardware virtualization with an empty pipeline, which matches the bare-metal.

The actual proof has more complexities. For example, sometimes, the information in decode-assist registers is insufficient to precisely determine the faulting instruction. In these cases, the VMM fetches the program counter from the register state, fetches the instruction, and emulates it in full. Because other guest cores could be running concurrently, the instruction at the address in the instruction pointer register may have changed since the trap happened, so we cannot assume that the emulation will align with the steps that the specification has already taken. To handle this situation, we assume that a bare-metal CPU can spontaneously delete any leaf instruction node in the tree of instructions, as long as the instruction has not committed. Note that instruction executions that have faulted cannot have committed externally visible effects.

References

[1] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA)*. 608–621. https://doi.org/10.1145/2837614.2837615

[2] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (dec 2017), 29 pages. https://doi.org/10.1145/3158107