Verification of TLB Virtualization Implemented in C*

Eyad Alkassar[1], Ernie Cohen[2], Mikhail Kovalev[1], and Wolfgang J. Paul[1]

[1] Saarland University, Saarbrücken, Germany
{eyad,kovalev,wjp}@wjpserver.cs.uni-saarland.de
[2] European Microsoft Innovation Center (EMIC), Aachen, Germany
ecohen@microsoft.com

Abstract. Efficient TLB virtualization is a core component of modern hypervisors. Verifying such code is challenging; the code races with TLB virtualization code in other processors, with other guest threads, and with the hardware TLBs, and implements an abstract TLB that races with other abstract TLBs and guest threads. We give a general methodology for verifying virtual device implementations, and demonstrate the verification of TLB virtualization code (using shadow page tables) in the concurrent C verifier VCC. To our knowledge, this is the first verification of any kind against a realistic model of a modern hardware MMU.

Keywords: Shadow Page Tables, TLB, Hardware/Software Interaction, Automatic Verification, Virtualization.

1 Introduction

A major challenge in the formal verification of low-level system software is dealing with devices. Because devices run in parallel with software, they typically necessitate concurrent program reasoning even for single-threaded software. Existing verifications of software that interacts with devices (e.g., [3]) rely on transforming the software and hardware into a single transition system. Similarly, verifying correct device virtualization requires reasoning about simulation, which is usually viewed as a relation between transition systems (rather than a property of a single program), and so likewise depends on treating the entire program as a big transition system, an approach sufficient for toy programs but hardly appetizing for industrial-scale software verification.

In this paper, we show how software that interacts with and simulates devices can be verified directly in VCC [4], a verifier for concurrent C. VCC provides several features that are central to our methodology:

– it supports verification of programs with fine-grained concurrency, which is generally needed for programs that deal with devices,

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft XT project under grant 01 IS 07 008.

R. Joshi, P. Müller, and A. Podelski (Eds.): VSTTE 2012, LNCS 7152, pp. 209–224, 2012.
© Springer-Verlag Berlin Heidelberg 2012

- it provides ghost state, which we use to represent the abstract state of a virtual device,
- it provides two-state object invariants (i.e., invariants that constrain not only the states of an object but its atomic state transitions also), which we use to express the desired behavior of the virtual device,
- it provides ghost code, which we use to witness the existence of simulation-preserving updates to the abstract state, allowing forward simulation to be expressed using ordinary program reasoning.

We apply our methodology to verify the (partial) correctness of C code (the hypervisor) that virtualizes the MMU of an x64-like processor[1]. The purpose of such virtualization is to provide, to several virtual machines (VMs), each running its own operating system, the illusion that every VM is running alone on a physical machine, even though different machines might try to configure their page tables to use the same physical addresses. To provide this illusion, the algorithm provides an additional level of address translation beyond what is provided by the hardware MMU. It does this by maintaining a separate set of *shadow page tables* (SPTs) for each VM, each SPT shadowing a page table of the guest (GPT). These SPTs are the tables actually used by the hardware for address translation, but are kept invisible to the VMs.

The guest TLB-controlling instructions, such as TLB invalidation (`INVLPG`) or modification of control registers (e.g., *CR3* register), are intercepted by hardware and virtualized by the hypervisor. When a memory access by the guest results in a page fault ($\#PF$) the hypervisor emulates the steps of the virtual MMU by walking the GPTs, setting access (A) and dirty (D) bits in the GPTs, and caching the translations in the SPTs. A correct SPT algorithm guarantees that the *virtual TLB* (VTLB), provided to the VM by the hardware TLB (HTLB) together with the intercept handlers, behaves accordingly to the hardware specification and provides appropriate translations to the guest running in the VM.

The verification target is interesting for several reasons. First, efficient MMU virtualization is perhaps the trickiest part of building correct hypervisors for modern processor architectures (particularly for machines without hardware SLATs). Second, precise reasoning about MMU behavior is central to the correctness of efficient memory management software; because flushing part or all of a TLB is expensive, memory managers often use clever tricks to avoid flushes whenever possible by allowing the TLBs to drift out of synchronization with the page tables (PTs), and using obscure properties of TLB behavior [5] (e.g., that the MMU will set an accessed bit in a nonterminal page table entry (PTE) atomically with its read of that entry, rather than in a separate memory transaction) to deduce that certain translations are not in the TLB. Third, in spite of the critical importance of MMU behavior, it has never been seriously treated in OS kernel verification. For example, the Verisoft project [3] used a synthetic hardware model without TLBs, while the L4.verified project [6] explicitly

[1] While the proof involves specification of many additional functions and data structures, the top-level result depends only on the definitions of the hardware behavior and the coupling relation between the virtual and concrete states (Section 4).

assumed that the TLBs were kept in sync with the page tables, essentially making the TLBs transparent to software. The similar approach was chosen in the Nova micro-hypervisor verification project [7], which used an abstract model of IA-32 hardware with MMU, but without the TLB.

In the rest of the paper we present an overview of the techniques we used for modelling the concurrent hardware and for verifying the code in VCC, formulate the VTLB correctness, and provide the annotated portions of the code crucial for our verification methodology.

2 Verification Methodology

2.1 VCC Background

VCC is a (hopefully) sound, deductive verifier for concurrent C code. As in many other modern, function-modular software verifiers, each function is specified using preconditions, postconditions, and framing information (writes clauses); when verifying the body of a function, the specifications of functions are used to provide the meanings of function calls. VCC allows *ghost data* of various kinds (locals, fields of data structures, and function parameters); ghost data is typically used to represent the abstract value of a data structure. Ghost data can use mathematical types not provided by C (integers and maps of arbitrary size).

Where VCC most differs from other deductive verifiers is in its treatment of data structures and concurrency. In each state, each object (i.e., each instance of a C structured type) is classified as *open* or *closed*, and has a unique *owner*. Only closed objects and threads can own other objects, and only threads can own open objects. A thread can sequentially access only data in an open object that the thread owns[2]; this guarantees that sequential operations in different threads cannot interfere.

The *volatile* keyword has a special meaning in VCC, identifying the shared data which may be changed in an atomic step without opening an object. VCC allows each object type to be annotated with 2-state invariants specifying how the system state may change (in a single atomic step). The invariant of an object is only required to hold in transitions that begin or end with the object closed. (Each type implicitly includes invariants stating that non-volatile fields of the object do not change.) Unlike other verifiers, VCC object invariants can mention arbitrary parts of the state, necessitating the following check: a transition is said to be *legal* if it satisfies the invariants of all objects that it modifies; an object invariant is said to be *admissible* iff it is guaranteed to be preserved by all legal transitions that do not update the object. VCC requires all object invariants to be admissible; this check is made on the basis of type definitions (without looking at code).

A typical idiom is for an invariant of an observer object to assert a property of a subject object, this invariant holding because it was true at the time the

[2] It can also sequentially read data that it knows (by virtue of object invariants) not to be changing.

observer was formed and it is maintained by the subject. Because the subject cannot be relied on to maintain any invariant when it is open, the observer invariant must also guarantee that the subject stays closed. For this invariant to be admissible, the subject must include an invariant that once closed, it remains closed until there are no closed observers observing it. This idiom is very common and useful (particularly for ghost objects), so VCC provides syntactic support for it: subjects supporting such observers are said to be *claimable*, and the supported observers are called *claims*. Each claimable object maintains a count of the number of outstanding claims on it; the object cannot be opened when the claim count is nonzero. When forming a claim that claims a set of subjects and a particular property of the state, VCC does the corresponding admissibility check "inline" to avoid having to introduce a separate object type.

One way to make an observer invariant admissible is for the subject invariant to explicitly require a check of the observer invariant whenever the subject changes state in a way that might violate this invariant. We describe this invariant by saying that the observer *approves* such state changes. Approval provides a general technique for semantic subclassing of concurrency in VCC. A type with an invariant that does not use approval corresponds to a "closed" class (w.r.t concurrency), in the sense that clients cannot strengthen the invariant of an object of the type. A type in which certain updates are approved by a client allow the client to effectively strengthen the invariant of the type as needed (to the extent allowed by his approval). This is the norm when implementing concurrent abstract data types. Approval of a volatile field of an object by a thread that owns the object has the effect of making the field sequential from the standpoint of the owning thread (except that it must still update the field with atomic actions).

2.2 Modelling Hardware

When verifying a multi-threaded program that operates on shared data, the normal procedure is to choose invariants on the shared data that are suitably strong to verify the particular program. To specify a multi-threaded system, such as a multiprocessor, we do not know what particular program will be run, and so have to choose a "generic" invariant for the shared data i.e., the strongest invariant maintained by the system. To construct this invariant, we introduce a ghost variable indicating which component of the system is executing at each step, and require that changes of the shared state (in the current verification, the shared memory) are approved by the currently executing component.

There are two natural ways to model hardware devices in VCC. One is to model a device with a program that exhibits all possible behaviors of the device. This is convenient for verifying a program designed to run in parallel with the device; the verification of the device code guarantees that the invariants required by the program are not so strong as to be potentially broken by actions of the hardware.

The other way to model a device is by modelling the device as an object, where the 2-state invariant of the object gives the allowed transitions of the

hardware. This model is convenient for showing correct simulation of the device: we model the virtual device state with ghost data, and show that each update to this data (along with updates to concrete data, like shared memory) satisfies the 2-state invariant of the hardware. (In addition, we normally maintain a 1-state coupling invariant between the concrete state and the virtual state, which has to be maintained by updates to either concrete or virtual state.) In principle, an object model could be used in place of the hardware thread, but there are technical difficulties in doing this that go beyond the scope of this paper, so the current verification uses both models.

3 Specification

The type of n-bit strings $\{0,1\}^n$ is denoted by $\mathbb{B}_n$. We interpret a string $a \in \mathbb{B}_{64}$ either as a 64-bit string, a natural number, or a PTE. We consider a quad-word (64 bits) addressable memory, 45-bit long (quad-word) virtual addresses (VAs), and (quad-word) physical addresses (PAs) 49 bits long (which correspond to architecturally defined byte addresses of 48 bits for virtual and 52 bits for physical addresses). We call the top-most 36 bits (for the VAs) or 40 bits (for the PAs) the page frame number (PFN). We use the operator $=_l :: \mathbb{B}_{36} \mapsto \mathbb{B}_{36} \mapsto \mathbb{B}$ to compare the bits $[35 : 9 \cdot l]$ of two virtual PFNs.

3.1 Host Hardware Model

We model an x64 multi-core/multi-processor machine in the AMD64 style [1] with the record $h :: \mathit{x64conf}$. Shared memory of the system is denoted by $h.mm :: \mathbb{B}_{49} \mapsto \mathbb{B}_{64}$. The hardware configuration of a processor $h.p[i]$ consists of a register *CR3* giving the address of the root PT, a (processor local) TLB *tlb* tagged with *address space identifiers* (ASIDs), a register *asid* providing the tag of the current address space running on the processor, and an uninterpreted variable *state* encapsulating the rest of the processor state. In order to implement and specify the TLB lazy flushing mechanism (which exploits TLB tags), we introduce the function $\mathit{asid}_{gen}(p)$, returning the *generation* of the tags which are still valid in the TLB.

A single PT occupies one memory page (4KB long) and consists of 512 PTEs, each being a 64-bit long union containing the page frame number *pfn* at which the entry is pointing (40-bits long), accessed (A) and dirty (D) bits a and d, a present bit p, and the bits denoting access rights. To simplify reasoning, we introduce the set of access rights r where e.g., $\mathit{pte}.r[rw]$ denotes the write access bit $\mathit{pte}.rw$, with operations of rights comparison and rights restriction (i.e., addition):

$$r_1 \leq r_2 \stackrel{\text{def}}{=} \forall j : r_1[j] \leq r_2[j] \qquad r_1 + r_2 \stackrel{\text{def}}{=} \lambda i : r_1[i] \wedge r_2[i] \ .$$

We model the TLB state as a set of page table *walks*, each of which summarizes a partial or complete traversal of the page tables for a given VA. Each walk is given by a virtual PFN *vpfn*, a level l giving the number of page table levels

remaining to be walked[3], the page frame number *pfn* of the next page table to be used for translation, the tag *asid* of the address space where the walk was performed and a set r of access rights giving all rights not denied by the walk gathered thus far. A walk is *complete* if its level is 0, and *partial* otherwise.

For the transition system of the host TLB we use the non-deterministic TLB model from [2] extended with ASIDs. We distinguish between autonomous MMU steps, s.t. walk creation, extension, deletion and setting of A and D bits and the TLB-dependent abstracted processor steps, s.t. address invalidation (`INVLPG` and `INVLPGA`), writing of *CR3* register and TLB flush. We use the predicate $valid\_step(h, h')$ to denote that the hardware has performed a valid step of the transition system from configuration h to h'.

We assume that the hypervisor is running untranslated and abstract the TLB to have no walks in the zero ASID (in AMD64 the hypervisor code is always executed with ASID equals zero).

3.2 Virtual Hardware Model

A hypervisor provides to each VM (and to the guest code executed in the VM) the illusion of running on its own private hardware. In the paper we provide this illusion for a single VM (this can be easily generalized to multiple VMs mapped to disjoint host memory portions).

The specification model is an abstract (virtual) machine $g :: x64conf$, which resembles a slightly restricted hardware with the virtual memory $g.mm$ and virtual processors (VPs) $g.p[i]$, $i \in [0 \dots N_p]$. The hardware virtualization features, such as tagged TLB and virtualization instructions, are not available for the VM.

3.3 Memory Virtualization

The memory of the VM is mapped to some region of the memory of the host machine by means of the injective function $gpa2hpa :: \mathbb{B}_{40} \mapsto \mathbb{B}_{40}$, translating guest physical PFN into the host physical PFN. Translations of guest virtual to guest physical addresses are defined by the GPTs, which are located in the memory of the VM and can be modified by the guest without notifying the hypervisor. When the VM is running runs, the hardware TLB does not have access to the GPTs, but rather operates with SPTs allocated and maintained by the hypervisor.

SPT j of VP i is identified by a pair (i, j) and is obtained from the hypervisor memory by the function[4] $spt(i, j)$. The function $i2a(i, j)$ returns the host PFN of the SPT (i, j). We organize the SPTs for each VP as a tree of SPTs, and assign

[3] To simplify the presentation, we do not consider large pages and legacy addressing modes here, so each complete walk goes through exactly four page tables. Also, we do not consider global page translations.

[4] We assume configuration C to be an implicit parameter in all functions dependent on the memory state.

to each SPT a level ranging from 4 (top-level) to 1 (terminal). The entries of non-terminal SPTs point to other SPTs, while the entries of terminal SPTs point to memory of the VM (under the $gpa2hpa$ map). The predicate $walks\_to(i, j, px, j')$ denotes that SPT (i, j) points to SPT (i, j'):

$$walks\_to(i, j, px, j') \stackrel{\text{def}}{=} (spt(i, j)[px].pfn = i2a(i, j')) \ .$$

The index of the top most SPT of VP i is denoted by $iwo(i)$. The predicate $used(i, j)$ checks whether SPT (i, j) is in use by its VP or is free otherwise, the function $l(i, j)$ returns the level of the SPT (i, j) if it is currently used by the VP. The function $vpfn(i, j)$ returns the prefix of the SPT (VA range for the addresses of the walks that might go through to this SPT) and the function $r(i, j)$ provides the accumulated rights from the top-level SPT to the SPT (i, j). The ASID of the VP i is obtained by the function $asid(i)$ and the ASID generation by the function $asid_{gen}(i)$. The predicate $re(i, j)$ checks whether the SPT is *reachable* from the root $iwo(i)$.

Guest instructions and exceptions that operate on the TLBs are intercepted by the hypervisor so that they can be virtualized in the SPTs.

3.4 Correctness of TLB Virtualization

The abstract VM g is implemented on a host machine h running the hypervisor code. The VM abstraction and the implementation are linked by a coupling invariant. Correctness of the hypervisor is established by proving the simulation between the abstract VM and the VM implementation running atop of the hypervisor. More detailed, we have to show that (i) the coupling invariant is maintained and (ii) the host transitions can be abstracted to valid VM steps (i.e. respecting the hardware transition relation). These properties are encoded by the following invariant:

Invariant 1. *Let h and h' be states of the host hardware machine and the coupling invariant holds between h and g. Then it follows*

$$coupling(h, g) \wedge valid\_step(h, h') \implies \exists g' : valid\_step(g, g') \wedge coupling(h', g') \ .$$

For correct TLB virtualization, we have to consider (i) those parts of the coupling invariant related to the TLB and registers used for address translation and (ii) MMU-related steps of the host.

A VTLB, being part of the virtual hardware g, has to correctly simulate every address translation performed by the HTLB (as well as flushes and autonomous TLB steps). Intuitively, this means that when the guest code is performing a memory access to a guest physical address a (i.e., the VTLB of the VM returns address a for this memory operation), the HTLB should return the translated address $gpa2hpa(a)$. To make this possible, we have to couple every complete walk in the HTLB with the respective ones in the VTLB. We do this by linking the VTLB to the two components of the implementation: the HTLB and the SPTs.

Table 1. Main invariants of the SPT algorithm

Invariant name	Invariant property
htlb_walks	$w \in h.p[i].tlb \wedge valid(h,i,w.asid) \implies w \in W[i]$
vtlb_walks	$w \in W[i] \wedge w.l = 0 \wedge w.asid = asid(j) \implies hw2gw(w) \in g.p[j].tlb$
running_asid	$h.p[i].asid \neq 0 \implies valid(h,i,h.p[i].asid)$
distinct_asids	$i \neq j \wedge vp2hp(i) = vp2hp(j) \wedge asid(i) = asid(j)$ $\implies asid_{gen}(i) \neq asid_{gen}(j)$
partial_walks	$w \in W[i] \wedge w.l \neq 0 \wedge asid(j) = w.asid \implies w \in rwalks(j)$ $\wedge w \in rwalks(j) \implies w \in W[vp2hp(j)]$
reachability	$re(i, iwo(i)) \wedge (re(i,j) \wedge walks\_to(i,j,px,j') \implies re(i,j'))$
complete_walks	$w \in cwalks(j) \implies w \in W[vp2hp(j)]$
coupling_gwo	$g.p[i].CR3 = gwo(i)$

Formally, we want the coupling invariant to establish the following property over the HTLB:

$$w \in h.p[i].tlb \wedge w.asid = h.p[i].asid \wedge w.l = 0 \implies hw2gw(w) \in g.p[j].tlb \ , \tag{1}$$

where j is the ID of the currently running VP and the function $hw2gw(w)$ transforms a host walk to the respective walk of the VTLB by applying the inverse of the function $gpa2hpa$ to $w.pfn$.

However, to make (1) inductive we have to argue about HTLB walks not only in the currently running ASID, but in all ASIDs which could possibly be scheduled to run without a preceding TLB flush. If an ASID a could be scheduled to run on a host processor (HP) i without a flush, we call it *valid* and we define the set of valid ASIDs in the following way

$$valid(h,i,a) \stackrel{\text{def}}{=} \exists j : vp2hp(j) = i \wedge a = asid(j) \wedge asid_{gen}(h.p[i]) = asid_{gen}(j) \ ,$$

where the function $vp2hp(i)$ identifies the HP on which VP i is scheduled to run.

For a better partitioning of the invariants in data structures (Sec. 4), we introduce the superset $W[i]$, holding all walks possibly residing in the HTLB of HP i. The desired property (1) is now obtained by the invariants *htlb_walks*, *vtlb_walks*, and *running_asid* (Tab. 1) with the help of *distinct_asids*, which ensures the uniqueness of a VP with a given valid ASID.

To maintain *htlb_walks* when the HTLB is extending a walk, we have to define the content of $W[i]$ and argue about all SPTs, which could be walked by the HTLB in a given configuration.

Our algorithm ensures that the HTLB only accesses *reachable* SPTs i.e., those linked in the SPT tree. The set of all partial walks of VP i sitting on reachable SPTs is defined as

$$\begin{aligned} rwalks(i) \stackrel{\text{def}}{=} \{w \mid & re(i,j) \wedge w.r \leq r(i,j) \wedge w.pfn = i2a(i,j) \wedge w.l = l(i,j) \\ & \wedge w.vpfn =_{w.l} vpfn(i,j) \wedge w.asid = asid(i)\} \ . \end{aligned}$$

Invariant *partial_walks* (Tab. 1) relates partial walks from $W[i]$ with the walks over the reachable SPTs. Invariant *reachability* helps to maintain *partial_walks* when the HTLB extends a walk (going from one reachable SPT to another).

A straightforward way to identify the complete walks in $W[i]$ is to argue about all terminal shadow PTEs (SPTEs) that could have possibly been walked by the HTLB since the last flush [2]. The task however is cumbersome: a single SPT could be reused for shadowing different GPTs without a complete flush of the HTLB. In this case the HTLB could have walked some SPTE twice - before and after it was reused for a new shadowing. In our approach we only keep track of the terminal SPTEs belonging to reachable SPTs, which is enough to justify the new walks added to the HTLB w.r.t the VTLB. Additionally, we make sure that the VTLB (and the set $W[i]$) drops only the walks which are no longer present in the HTLB.

For a walk through a (terminal) SPT (i, j) let $spte = spt(i, j)[w.vpfn[8:0]]$. Then the set of complete reachable walks of VP i is defined as

$$cwalks(i) \stackrel{\text{def}}{=} \{w \mid re(i,j) \wedge l(i,j) = 1 \wedge w.r \leq r(i,j) + spte.r \wedge w.l = 0 \wedge spte.p \\ \wedge\, w.vpfn =_1 vpfn(i,j) \wedge w.asid = asid(i) \wedge w.pfn = spte.pfn\} \ .$$

Invariant *complete_walks* (Tab. 1) relates the complete walks over the reachable SPTs with the complete walks in $W[i]$.

Finally, invariant *coupling_gwo* couples the *CR3* register of the VM with the guest walk origin, which is necessary for creating a new walk in the VTLB.

4 Implementation and Verification in VCC

We use ghost data to maintain both the state of the virtual hardware and the state of the host hardware other than the memory (Fig. 1). The hardware transition relation is formulated as a 2-state invariant of the hardware data structure. We use the same data types for modeling the host hardware and for the specification of the abstract VM.

The autonomous part of the host hardware state (e.g., HTLB) is modelled with volatile data and is allowed to change non-deterministically. We locate the host hardware state in the ghost memory, but we do allow limited information flow between some of its fields (e.g., registers and TLB) and the concrete program[5]. We do not restrict the memory updates of the host hardware in the transition relation, since that would require approval of the whole VCC memory by the hardware data structure, making memory changes in the code through regular variable assignments impossible. Instead, we allow VCC software invariants to specify memory transitions on C level. As a result, the autonomous hardware is verified as a C thread with the same annotations on type definitions as the main program.

[5] This is done for lack of a dedicated hybrid type capturing implementation state other then the main memory.

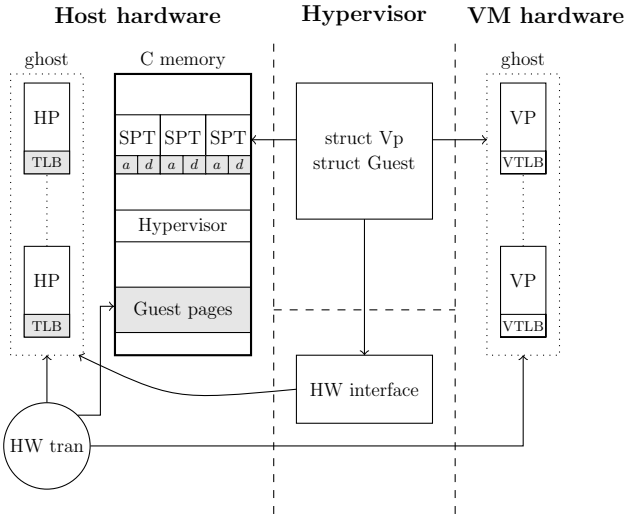

Fig. 1. Approval scenario for the SPT algorithm

The state of the virtual hardware (excluding the memory) is also located in the ghost memory. In contrast to the host hardware, we do specify memory framing for the hardware transitions of the VM. The memory of the VM is abstracted from the portions of the C memory allocated to the machine w.r.t the function *gpa*2*hpa*. To ensure that every update of the virtual memory is justified by the transition relation of the VM, we model the memory of the VM as volatile data approved by the virtual hardware.

The updates of the virtual hardware, simulating the steps of the VM, are performed by the ghost code in atomic statements, guaranteeing that the transition relation and coupling invariant are maintained by every update. When the step of the virtual hardware involves accessing the implementation memory (e.g., fetching of a GPTE by the $\#PF$ handler), the update to the virtual configuration is done in the same atomic block as the memory access. This allows to simulate a step of the VM on the virtual memory abstracted from the C implementation memory.

The correctness (coupling) invariants from Tab. 1 are specified as 1-state invariants over the data structures of the hypervisor and over the simulated virtual hardware. More precisely, the invariants specific to a single virtual processor are included in the invariant of the implementation data structure of type `Vp` (Fig. 1) and the invariants establishing properties over the VPs altogether (s.t. the invariant *distinct_asids*) are specified in the data structure of type `Guest`. With each VP we associate a set of ghost fields used for maintaining correctness of the SPT algorithm (e.g., maps of allocated and reachable SPTs for this VP).

The properties of the overall system which have to be maintained by software and hardware steps are specified in the so called *hardware interface*. For instance, it specifies for each HP a map $W[i]$ (Sec. 3.4), which contains all walks possibly

residing in the HTLB of that processor, and states the invariant *htlb_walks* (Tab. 1). The hardware interface is purely ghost, since it is only used for specification rather than to implement concrete data structures or hardware components. To check that the invariants of the hardware interface are maintained by all possible hardware transitions, we have to explicitly invoke each of them in the MMU thread.

4.1 Specification

The processor state[6] is modeled using the struct type `Processor`.

```
spec(typedef struct _Processor {
    Procx i; // Processor id
    bool v; // flag for virtual
    volatile Asid asid; // processor ASID
    volatile Tlb tlb; // TLB (a map of walks)
    Hardware *h; // pointer to hardware container
    Hwinterface *hwi; // pointer to HW interface
    inv(approves(h, tlb, asid)) // approval by hardware
    inv(!v ==> approves(hwi, tlb, asid)) // approval by HWI
    inv(approves(owner(this), asid)) // thread approval
    inv(v ==> approves(owner(this), tlb)) //thread approval
} Processor;)
```

Fields of the processor which may change only by instruction execution (as e.g. registers) are approved by the running thread. (We mark these sequential fields volatile to allow them being controlled by the 2-state transition invariant of the hardware.) The flag v is used to distinguish between the host and the virtual hardware. For the virtual hardware the TLB of the processor is also approved by the running thread (the steps of the VTLB are always explicitly performed by intercept handlers). For the host hardware, the TLB and the current ASID register are approved by the hardware interface, where we state software dependent properties on these fields (see Fig. 1 for dependencies between data structures used for hardware modeling and implementation of the algorithm).

The data structure `Hardware` encapsulates all processors and defines via 2-state invariants all valid hardware transitions. To ensure that the processor respects this transition relation, it has to approve all processor fields.

```
spec(typedef struct _Hardware {
    Processor *p; // array of processors
    bool v; // flag for virtual
    claim_t cm[Ppfn], cp[Procx]; //claims on processors and memory
    Ppfn gpa2hpa[Ppfn]; // memory translation (for VM)
    volatile Procx i; // index of acting processor
    volatile Action act; // type of action
    volatile Walk w; // TLB walk for the action
    inv(forall(Procx i; claims_obj(cp[i], &p[i]))) // Claims on processors
    // Claims on memory (for VM)
    inv(forall(Ppfn a; gpa2hpa[a] ==> claims_obj(cm[a], page(gpa2hpa, a))))
    inv(p_unch(p) && (!v || m_unch(abs_m(gpa2hpa))) ||
      act == TLB_SET_AD && tlb_setad(p, i, w, old(read_pte(w,gpa2hpa,v)))
        && (!v || m_upd(abs_m(gpa2hpa), w)) ||
      act == CORE_INVLPGA && core_invlpga(p, i, v)
        && (!v || m_unch(abs_m(gpa2hpa))) || ...) // Transition relation
} Hardware;)
```

[6] We expose only the most crucial parts of the data structures necessary to understand our methodology, omitting e.g., valid ASIDs and *CR3* registers here.

```
typedef struct _Spt {
    volatile Pte e[512];
    spec(volatile Pte ge[uint];)
    inv(approves(owner(this), ge))
    inv(sptes_eq_except_a_and_d(e, ge))
} Spt;
```

```
typedef struct vcc(claimable) _Gpt {
    volatile Pte e[512];
    spec(_Hardware *h;)
    inv(approves(h, e))
} Gpt;
```

Listing 1. SPTs and GPTs

The current hardware transition is specified by variables i, act, and w, where i identifies the acting processor, act the action type and w the walk targeted by the action in case of a TLB transition. When we prove simulation for the VMs, we use these variables to choose a certain step we want to simulate. In case of the host hardware these variables allow us to explicitly go over all possible TLB steps in the MMU thread, showing that none of them violate the VCC software invariants.

In the hardware transition system we make a distinction between the steps of the host hardware and the steps of the virtual hardware, having memory framing only if the v bit is set. For the virtual hardware we also require that it claims all memory pages allocated the the VM. We obtain those pages by the map $gpa2hpa$ translating guest physical addresses to host physical addresses (constructed during VM initialization and maintained during memory allocation). An arbitrary memory page of the VM is modeled as a GPT (Listing 1) consisting of guest PTEs (GPTEs) which are approved by `Hardware`.

4.2 Implementation

A SPT (Listing 1) contains an array of SPTEs. Since the A/D bits of a SPTE may be accessed concurrently by hardware, we have to mark the complete entry as volatile. All other bits may only be accessed by the software currently running on the processor. Since thread approval can not be stated bit-wise we have to introduce an approved ghost copy of each SPTE (identical to the original one except for A/D bits).

The SPT algorithm itself consists of a number of intercept handlers. The most crucial ones are considered below.

#PF Intercept. When a #PF is intercepted, the hypervisor walks the GPTs to obtain a set of GPTEs used for the translation of the faulty VA. Every access to a GPT is performed inside an atomic block. During the walk we simulate the respective VTLB steps (initializing a walk, setting A/D bits, extending a walk), which makes the GPT walker the core part of the #PF handler. If the walk extension is unsuccessful (rights violation or present bit not set), we simulate the #PF -signalling step of the VP, inject #PF to the VM and return. After successful walking of GPTs the handler walks the SPTs and finds the first SPTE which is not in-sync with the associated GPTE. The subtree pointed by this SPTE is freed and new subtree (being in-sync with the fetched GPTEs) is allocated

and attached to the SPTE. The set of walks $W[i]$ is updated to hold the newly attached walks and to drop the detached ones. Non-dirty terminal SPTEs are marked write protected to propagate a D bit to the VM when it is being set by the HTLB. In case of detaching a subtree we perform a hardware `INVLPGA` to ensure that the HTLB is not sitting on the freed SPTs.

Flush Intercept. When the TLB flush is intercepted, the handler frees all the SPTs of the VP, allocates a fresh top-level SPT (which has all its entries set to non-present), assigns an unused ASID for the VP and simulates the VTLB flush step. If all the ASIDs are already in use, the handler flushes the HTLB and increments the ASID generation of the HP. (Currently we explicitly assume that the ASID generation doesn't overflow.) At this point, all ASIDs which were previously assigned to VPs running on this HP become invalid. The handler then gives the first ASID to the intercepted VP and makes it valid by setting the ASID generation of the VP to the current one of the HP. Set $W[i]$ in the hardware interface is updated to hold only walks sitting on the fresh top-level SPT.

Every time when some VP is scheduled to run we check whether the ASID generation of the VP is equal to the ASID generation of the HP. If this is not the case (i.e., some other VP has increased the ASID generation of the HP), we allocate an unused ASID for the VP and proceed in the same way as in the case of a flush intercept.

`INVLPG` **Intercept.** In case of the `INVLPG` intercept the handler walks down the SPTs for the invalidated address and marks a terminal SPTE non-present. Then it performs a hardware `INVLPGA` on the faulty VA in the ASID of the intercepted VP. The complete walk through the modified terminal SPTE is removed from the set W and the `INVLPG` step of the virtual VP is simulated.

4.3 Verification

We consider verification examples of the code of the #PF handler simulating the step of the VTLB and, of a single HTLB transition performed in the MMU thread.

VTLB Steps Simulation. The VTLB operates on the shared (volatile) memory of the VM and races with other VTLBs and with the running VPs. Hence, the memory of the VM may change arbitrarily in between atomic accesses to it. To simulate a VTLB step corresponding to the operation on the memory of the VM, we have to perform the simulation in the same atomic block where the handler reads/writes GPTs. We also need to have access to the state of the virtual processor and to the transition relation of the virtual hardware. The VP and everything in its ownership domain is thread local, while an instance of `Guest` is shared between all the VPs and is claimed to be closed by a claim *gc* (i.e., we can not update sequential data of the guest, but can assert its invariant).

As an example of a VLTB step we consider the setting of A/D bits for a top-level walk (performed in a GPT walker before we fetch a top-level GPTE for walk extension):

```
pfn = gpa2hpa(vp->gwo, guest);
if (pfn == 0) return 0; // non-allocated guest address
gpt = (Gpt *)(pfn << 12);
px = compute_idx(vpfn, 4);
while (!cmp_result)
  writes(vp)
  inv(thread_local(vp) && claims(gc, guest) && ...)
{
    atomic(gpt){old_pte = gpt->e[px];} //fetching GPTE
    unwrap(vp); // opening thread-local object
    atomic(...){ // setting A and D bits
        if (old_pte.p) { // modifying and writing GPTE
            cmp_result = (old_pte == (rw && old_pte.rw)
                ? asm_cmpxchg(&gpt->e[px], old_pte, SET_AD(old_pte))
                : asm_cmpxchg(&gpt->e[px], old_pte, SET_A(old_pte)));
            spec(if (cmp_result) { // fixing step parameters
                guest->g.i = vp->i;
                guest->g.act = TLB_SET_AD;
                guest->g.w = top_level_walk(vp->gwo, vpfn);
             })
        } else // don't do update if the entry is not present
            cmp_result = 1;
    }
    wrap(vp); // closing thread-local object
}
```

Since the x64 architecture does not provide an instruction performing an atomic read-modify-write operation we use a loop in which we fetch an entry, modify it, and then write it back if the entry has not been changed in between. Writing to a GPTE is done by an interlocked compare-exchange operation. To specify the behaviour of compare-exchange we define a C function `asm_cmpxchg` reflecting the effect of the interlocked operation on the C memory. If the compare-exchange is successful, we simulate the setting of A/D bits by the VTLB.

The invariants of the virtual hardware are checked automatically at the end of the atomic block, ensuring that a selected TLB step is performed accordingly to the transition relation. Since we operate only with one VP, VCC doesn't need to check the invariants of other VPs. The invariants of the hardware interface also are untouched here, because the set of the reachable walks remains unchanged.

MMU Thread. For soundness of the approach we have to emulate the behaviour of a hardware MMU in a C thread. There are two reasons why the two-state invariant of `Hardware` describing the MMU behaviour alone is not enough. First, the only place where VCC checks that the invariant of the concurrent object being modified holds is the atomic block where the writing to the object is done. Moreover, to check this VCC first has to ensure that the address being written belongs to a typed object. In the MMU thread we guarantee that all MMU writes are done to the SPTs of a running VP and these writes do not violate the invariants of SPTs.

The second reason why we need a software MMU thread is the presence of the observer: the invariants of the hardware interface should not restrict the hardware transition system in any sense. Since the transition invariant contains

a disjunction of steps, we have to ensure that the invariant of the hardware interface holds for every step from the disjunction. Note, that the invariant of the hardware interface has to be checked not only for MMU steps, but for other hardware steps as well. This check is done in the intercept handlers every time we execute a certain TLB-dependent processor step (e.g., `INVLPGA`).

The MMU thread consists of a number of atomic actions each performing a single MMU step. As an example, we again consider the setting of A/D bits by the TLB of the hardware processor *hp*.

```
atomic(...) {
    spt = (Spt *)(w.pfn << 12);
    px = compute_idx(w.vpfn, w.l);
    assume(hp->tlb[w] && w.l != 0 && w.asid == hp->asid
        && hp->asid > 0 && spt->e[px].p); // assuming guard
    vp = guest->hp2vp[hp->i][hp->asid]; // get the running VP
    assert(inv(vp)); // asserting invariant of running VP
    begin_update(); // start of update in the block
    spt->e[px] = (w.l == 1 && w.r[rw] && spt->e[px].rw)
        ? SET_AD(spt->e[px])
        : SET_A(spt->e[px]); // performing a write
    guest->h.i = hp->i; // fixing step parameters
    guest->h.act = TLB_SET_AD;
    guest->h.w = w;
}
```

With the help of the invariant of the running VP (obtained from the current ASID of the HP), VCC derives that the memory write is done to the SPT owned by that VP and the system invariants are maintained. Note, that the only invariants which might get broken by the HTLB step are the invariant of the hardware interface (if the HTLB adds a walk which is not present in $W[i]$) and the invariant of the SPT itself (if the HTLB modifies other bits than A/D).

5 Conclusion and Future Work

We have demonstrated the verification of a concurrent program dealing with devices using an automatic C code verifier. We have given a general methodology for verification of virtual device implementations, specified TLB virtualization with SPTs and formally verified a SPT algorithm.

The implementation of the SPT algorithm contains ca. 700 lines of C code (including initialization of data structures) and ca. 4K lines of the annotations which include function contracts, loop invariants, data invariants, ghost code, and (proof) assertions. Roughly a third of annotations comprise function and block contracts and another third is ghost code for maintaining ghost fields, showing simulation, and running MMU thread (which is purely ghost). The overall proof time is ca. 18 hours on one core of 2GHz Intel Core 2 Duo machine. The estimated person effort is 1.5 person-years, including VCC learning period.

There are two possible directions of future work. The first one is to integrate the proof and the specification to a prototypical hypervisor being developed and verified at the Saarland University. In particular, this requires adapting the proof to be done on top of the kernel layer of the hypervisor, rather than on top of the

real hardware (the support for this scenario is already included in our models). The second direction is to verify a more sophisticated version of the algorithm, which uses write-protection of GPTs and sharing of SPTs.

References

1. Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.14 edn. (September 2007)
2. Alkassar, E., Cohen, E., Hillebrand, M., Kovalev, M., Paul, W.: Verifying shadow page table algorithms. In: Formal Methods in Computer Aided Design (FMCAD) 2010, pp. 267–270. IEEE, Lugano (2010)
3. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 71–85. Springer, Heidelberg (2010)
4. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
5. Intel Corporation: TLBs, Paging-Structure Caches, and Their Invalidation (April 2007)
6. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 207–220. ACM, New York (2009)
7. Tews, H., Weber, T., Völp, M., Poll, E., van Eekelen, M., van Rossum, P.: Nova micro–hypervisor verification. Tech. Rep. ICIS–R08012. Radboud University Nijmegen (May 2008)