Verifying Shadow Page Table Algorithms

Eyad Alkassar*, Ernie Cohen†, Mark Hillebrand†, Mikhail Kovalev*, and Wolfgang J. Paul*
*Saarland University, Saarbrücken, Germany
{eyad,kovalev,wjp}@wjpserver.cs.uni-saarland.de
†European Microsoft Innovation Center (EMIC GmbH), Aachen, Germany
{ecohen,mahilleb}@microsoft.com

***Abstract*—Efficient virtualization of translation lookaside buffers (TLBs), a core component of modern hypervisors, is complicated by the concurrent, speculative walking of page tables in hardware. We give a formal model of an x64-like TLB, criteria for its correct virtualization, and outline the verification of a virtualization algorithm using shadow page tables. The verification is being carried out in VCC, a verifier for concurrent C code.**

I. Introduction

Virtual addressing is the most common way for a *host* program (typically an OS or hypervisor), to virtualize the memory of a *guest* program. In a typical implementation, the translation from virtual addresses (VAs) to physical addresses (PAs) is controlled by page tables (PTs) in memory; the hardware concurrently walks these page tables, setting accessed (A) and dirty (D) bits in the page table entries (PTEs) as appropriate, and caching the translations in a translation lookaside buffer (TLB). When the guest addresses memory, the processor uses the TLB to translate virtual to physical addresses. If a suitable translation is not available, a hardware page fault (#PF) throws control to the host, giving it an opportunity to intercede. The processor automatically flushes the TLB in certain circumstances (e.g., on an address space switch), but it is generally up to the host to manage the coherency of the TLB.

Virtual addressing does not in itself provide correct virtualization for guests that edit their own page tables, such as operating systems. A standard solution[1] to this problem is to control guest address translation using a separate set of *shadow page tables* (SPTs), invisible to the guest, each of which "shadows" one of the guest page tables (GPTs). When a guest memory access results in a #PF, the host #PF handler walks the GPTs (simulating the TLB hardware), setting A and D bits in the GPT entries, and caching the translation (perhaps with an additional level of translation) in the SPTs. This allows the hardware to subsequently walk the SPTs to cache the SPT translations into the hardware TLB (HTLB). Thus, the SPTs, the #PF handler, and the HTLB act in concert to provide a virtual TLB (VTLB) to the guest.

Because walking the page tables slows down program execution, high-performance memory managers running in a guest are often very aggressive in their use of the TLB, flushing translations only when absolutely necessary, allowing the TLB to cache stale translations to some extent, with its correctness depending on fine details of the TLB semantics (e.g., exactly when A bits are set). High performance hypervisors are equally aggressive in flushing translations from the SPTs and HTLB only when necessary; for example, the SPT algorithm in the Hyper-V$^{\text{TM}}$ [3] hypervisor shares SPTs between different processors and address spaces, and selectively write-protects GPTs from guest edits to keep them in sync with their SPTs (so that they don't have to be flushed on a guest address-space switch) [4]. This combination makes SPT algorithms difficult to test (particularly since an error is likely to manifest in a guest failure long after the SPT entry leading to it has been flushed); for example, one bug in the aforementioned SPT algorithm required seven thread switches to manifest. This makes SPT algorithms an ideal target for formal verification.

We describe the verification of a simple shadow page table algorithm. We formulate the main invariants and present a verification pattern in VCC, an automatic verification environment for concurrent C code (available at http://vcc.codeplex.com/).

II. TLB Virtualization Problem

A. Hardware Model

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 word (64 bits) addressable memory, 45-bit long VAs, and PAs 49 bits long. We call the top-most 36 bits (for the VAs) or 40 bits (for the PAs) the page frame number (PFN). We decompose a virtual address $a \in \mathbb{B}_{45}$ into page table indices $a.px[i]$ for $i \in [1:4]$ of 9 bits each and a 9-bit physical page displacement $a.px[0]$.

An x64 multi-core/multi-processor machine is modeled with the record $h :: x64conf$, where $h.p[i]$ denotes the hardware configuration of the processor i, and $h.mm :: \mathbb{B}_{49} \mapsto \mathbb{B}_{64}$ represents the shared memory of the system. A processor configuration consists of a register $CR3$ giving the address of the root PT, a (processor local) TLB tlb, and an uninterpreted variable $state$ encapsulating the rest of the processor state. A PT consists of 512 PTEs, each being a struct with five fields: the SPT page frame number pfn at which the entry is pointing, accessed and dirty bits a and d, a present bit p, and the set of access rights r (e.g., writing access $r[rw]$). We define the less-or-equal operator on set of rights as $r1 \leq r2 = \forall j. r1[j] \leq r2[j]$.

The TLB state is modeled as a set of page table *walks*, each of which summarizes a partial or complete traversal of

[1]Recent Intel and AMD processors provide a hardware alternative, in the form of an extra address translation layer not visible to the guest OS [1], [2].

TABLE I: Semantics of the Abstract TLB

Transition name	**Guard on the configuration h**	**Resulting configuration h'**
$createw_{tlb}(i, va, r)$		$h/\{.p[i].tlb = h.p[i].tlb \cup \{winit(h.p[i].CR3, va, r)\}$
$deletew_{tlb}(i, w)$	$w \in h.p[i].tlb$	$h/\{.p[i].tlb = h.p[i].tlb \setminus \{w\}\}$
$extendw_{tlb}(i, w)$	$w \in h.p[i].tlb \wedge pte(h, w).a \wedge pte(h, w).p$ $\wedge (pte(h, w).d \vee w.l > 1 \vee \neg w.r[rw]) \wedge w.l > 0$	$h/\{.p[i].tlb = h.p[i].tlb \cup \{wext(h, w)\}\}$
$setaccess_{tlb}(i, w)$	$w \in h.p[i].tlb \wedge w.l > 0 \wedge pte(h, w).p$	$h/\{.mm[w.pfn][w.va.px[w.l]].a = 1\}$
$setdirty_{tlb}(i, w)$	$w \in h.p[i].tlb \wedge w.l = 1 \wedge \neg fault(h, w)$ $\wedge w.r[rw] \wedge pte(h, w).r[rw] \wedge pte(h, w).a$	$h/\{.mm[w.pfn][w.va.px[w.l]].d = 1\}$
$mov2cr3_{cpu}(i, pto)$	$h.p[i].tlb = \emptyset \wedge instr(h.p[i]) = mov2cr3$	$h/\{.p[i].state = step(h, i), .p[i].CR3 = pto\}$
$invlpg_{cpu}(i, va)$	$h.p[i].tlb \cap \{w \mid w.va = va \vee w.l > 0\} = \emptyset$ $\wedge instr(h.p[i].state) = invlpg$	$h/\{.p[i].state = step(h, i, va)\}$
$transl\_ok_{cpu}(i, va, r, pa)$	$w \in h.p[i].tlb \wedge w.va = va \wedge w.l = 0$ $\wedge r \leq w.r \wedge instr(h.p[i]) = mem\_instr$	$h/\{.p[i].state = step(h, i, pa)\} \wedge pa = w.pfn \circ va.px[0]$
$transl\_pf_{cpu}(i, va, r, f)$	$w \in h.p[i].tlb \wedge w.va = va \wedge fault(h, w)$ $\wedge r \leq w.r \wedge instr(h.p[i]) = mem\_instr$	$h/\{.p[i].state = step(h, i, f)\} \wedge f$

the page tables for a given VA. Each walk is given by a virtual address va, a level l giving the number of page table levels remaining to be walked,[2] the page frame number pfn of the next page table to be used for translation, 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.

The function $winit(pfn, va, r)$ returns a walk with level 4 and the other components initialized according to the given parameters. The extension $wext(h, w)$ of a walk w in the hardware configuration h is defined as follows ($s/\{.c = v\}$ denotes update of field c of struct s to value v):

$$
\begin{aligned}
&pte(h, w) = h.mm[w.pfn \circ w.va.px[w.l]] \\
&fault(h, w) = \neg pte(h, w).p \vee \ \neg(w.r \leq pte(h, w).r) \\
&wext(h, w) = \\
&\qquad w/\{ \ .pfn \quad = pte(h, w).pfn, \\
&\qquad\qquad .l \qquad = w.l - 1, \\
&\qquad\qquad .r \qquad = \lambda i. \ w.r[i] \wedge pte(h.w).r[i] \ \}
\end{aligned}
$$

A complete walk can be used to address memory iff the walk's VA matches the requested VA and the walk provides rights (write, execute, etc.) at least equal to those requested. A #PF can be generated only from a partial walk leading to a PTE that is non-present or provides insufficient rights.

Table I gives the behavior of our TLB model, expressed as a transition relation on the hardware configuration h. For some operations we introduce additional parameters, such that virtual address va, physical address pa, and #PF flag f in case of a CPU address translation. While the first five actions (indexed with tlb) model autonomous behavior of the TLB, the last four actions (indexed with cpu) abstractly model the CPU behavior, using the uninterpreted function $step()$ to update the CPU state. Note also that operations such as `INVLPG` that flush the TLB are modeled instead as blocking when the TLB contains offending entries; these models are equivalent because the TLB is allowed to delete walks at any time.

[2]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 tagged TLBs or global page translations.

B. Correctness Criteria

A hypervisor provides to each guest the illusion of running on its own private memory, processors and TLBs. In the following we provide this illusion for a single guest,[3] modeled as a virtual machine $g :: x64conf$.This guest g is implemented on a single host machine running the hypervisor code, linked to this implementation by a coupling invariant. Hypervisor correctness is established by proving that execution of the host machine preserves this invariant, and that g behaves accordingly—in particular, that (i) the VTLBs $g.p[j].tlb$ of the guests satisfy the transition relation of Table I, and (ii) that any virtual memory access of this virtual processor is justified by a complete walk in its VTLB.

Given the transitive closure $\rightarrow^*_{tlb}$ of permissible TLB steps as defined in Table I, we can formulate the first property in form of forward simulation:

Invariant 1: Let h and h' be pre and post states of a host step, and g and g' be the abstracted guest machine states respectively. Then the changes to the TLB of any virtual processor (VP) j form a valid TLB transition:

$$g.p[j].tlb \rightarrow^*_{tlb} g'.p[j].tlb \qquad (1)$$

To formulate the second invariant we need to introduce parts of the coupling invariant. The function $vp2hp(j)$ defines for a VP j on which host processors it is currently scheduled to run. The memory of the guest is mapped to some region of the shared memory of the hardware machine. Then, the memory mapping is defined by the injective function $gpa2hpa :: \mathbb{B}_{40} \mapsto \mathbb{B}_{40}$, which maps a guest physical PFN into the host physical PFN.

The second invariant establishes a relation between the VTLB and the implementation model. The walks contained in the HTLB should be present in the VTLB with respect to the guest memory projection. A function $hw2gw(w)$ translates a complete host walk w into a respective guest walk applying the $gpa2hpa^{-1}$ mapping to the field $w.pfn$ and leaving the other fields of the walk w unchanged.

[3]This can be easily generalized to multiple guests mapped to disjoint host memory portions.

The VTLB is given by the result of an abstraction function on the host configuration. In this case it will usually contain more complete walks than the HTLB. Nevertheless, every complete walk present in the HTLB should correspond to a complete walk in the VTLB.

Invariant 2: Let a complete walk w be present in the HTLB. Then a VTLB contains the walk $hw2gw(w)$.

$$w \in h.p[i].tlb \wedge w.l = 0 \implies \qquad (2)$$
$$\exists j.\ vp2hp(j) = i \wedge hw2gw(w) \in g.p[j].tlb$$

In order to infer the second invariant after a TLB step we also need an invariant dealing with partial walks in the HTLB. The statement of this invariant depends on the definition of the VTLB. Note, that the VTLB is not obliged to store partial walks, because their presence in the TLB is not mandatory.

III. SPT Algorithm

In this section we describe a basic implementation of a shadow page table algorithm, define the abstracted VTLBs, and state the required invariants to prove the correctness criteria from the previous section.

A. Overview on the Implementation

We assume that the $gpa2hpa$ map is static, and that appropriate data structure and functions are given to store and query it. The SPTs are located in an array $\texttt{SPT}[0 : (n-1)]$. We define the functions $i2a :: \mathbb{N} \mapsto \mathbb{B}_{40}$ to return the host PFN of the SPT stored in each array element, and the function $a2i :: \mathbb{B}_{40} \mapsto \mathbb{N}$ as the inverse function on these PFNs. Every SPT may either be free or in use by a single VP j (more precisely the HTLB of the host processor it runs on). For each VP j we let $gwo(j)$ denote the current value of its CR3 register, and $hwo(j)$ denote the CR3 used on the host when the guest is actually running. The latter CR3 designates the top-level SPT used for VP j in the SPT array. We organize the SPTs for each VP as a tree of SPTs, and assign to each SPT a level ranging from 4 (top-level) to 1 (terminal) and a VA range for the addresses of the walks that might go through to this SPT (*prefix* of the SPT). The entries of non-terminal SPTs point to other SPTs, while the entries of terminal SPTs point to memory of the guest (under the $gpa2hpa$ map). The predicate $walks\_to(i, px, j)$ denotes that SPT with index i points to SPT j.

$$walks\_to(i, px, j) = (\texttt{SPT}[i][px].pfn = i2a(j))$$

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

Every SPT has an additional Page Table Info (PTI) data structure associated with it, which keeps auxiliary information about SPTs. The fields of $\texttt{PTI}[i]$ include $gpfn$ (the guest physical PFN of the GPT) and l (the level of the SPT).

The algorithm maintains the SPTs by handling the following intercepts:

1) Flushing/Switching of CR3: Flushes of the guest (e.g., by executing `mov2cr3`) are intercepted by the hypervisor. The intercept is handled by freeing all the VP's SPTs, allocating a fresh top-level SPT (which has all its entries set to non-present), and executing an HTLB flush.

2) #PF intercept: When a host #PF is intercepted, the hypervisor walks the GPTs to determine the reason for the fault. If a GPTE is reached that has insufficient rights for the page-faulting operation or has the present bit not set, a page fault is injected into the guest (and the hypervisor returns). Simultaneously with walking the GPTs, the hypervisor also walks the associated SPTs down from the top-level SPT. If a non-present (non-terminal) SPTE is encountered during the walk, we update the SPTE to point to a newly allocated, zero-filled SPT; the new SPT shadows the GPT referenced by the corresponding GPTE. For a present SPTE we check whether rights and PFN still correspond to the GPTE. If not, the old SPT subtree is detached (and a hardware `INVLPG` executed on the faulty VA) before allocating, initializing, and pointing to a new SPT as before. A GPTE's A bit is set when the #PF handler walks it; A bits in SPTEs are always set. Note that all not dirty terminal SPT entries are kept write protected to propagate a D bit to the guest before it is set by the HTLB.

3) `INVLPG` *intercept:* The implementation walks down the SPTs for the `INVLPG` address and, when reaching a terminal SPTE, marks it non-present. Then it performs a hardware `INVLPG` on the faulty VA.

B. VTLB Abstraction

To define the virtual TLB abstraction and verify the invariants we introduce *ghost fields* in the PTI structure, which are used only for verification but are ignored by the implementation. The field $vpid$ stores the index of the VP using the associated SPT, the field $vpfn$ stores the SPT's prefix, the field r stores the accumulated rights from the top-level SPT to the given SPT, the reachability bit re distinguishes whether the HTLB can walk the SPT, and for terminal SPTs, the array $gp[0 : 511]$ in the PTI stores the *ghost present* bits of the terminal PTEs denoting whether the complete walks through the PTEs might be present in the HTLB.

Next, we define the coupling invariant. Every guest component is abstracted from the hardware machine h. General purpose registers of the VP j are either loaded into the hardware registers of the processor $vp2hp(j)$ or are stored in some implementation data structure.

We define the VTLB as the set of walks corresponding to the complete walks that might be cached by HTLB. Formally, we use the ghost fields of the PTI data structure to construct the set $walks(i)$, containing walks *sitting* on the SPT i.

$$walks(i) = \{w \mid w.r \leq \texttt{PTI}[i].r \wedge w.pfn = i2a(i)$$
$$\wedge\, w.l = \texttt{PTI}[i].l \wedge \forall j \in [\texttt{PTI}[i].l + 1 : 4].\ w.va.px[j]$$
$$= \texttt{PTI}[i].vpfn[9 \cdot j - 1 : 9 \cdot (j-1)]\}$$

We define the set of indices of the terminal SPTs belonging to the VP j by

$$tSPT(j) = \{i \mid \texttt{PTI}[i].l = 1 \wedge \texttt{PTI}[i].vpid = j\}.$$

The set of complete walks of the VP j is defined as follows:

$$cwalks(j) = \{wext(h, w) \mid w \in walks(i) \wedge i \in tSPT(j)$$
$$\wedge \texttt{PTI}[i].gp[w.va.px[0]] \wedge w.r \leq \texttt{SPT}[i][w.va.px[0]].r\}$$

The VTLB is defined as a translation of complete shadow walks into the respective walks over the GPTs:

$$g.p[j].tlb = \{hw2gw(w) \mid w \in cwalks(j)\}$$

Note, that the VTLB definition does not use the implementation SPTE present bit, because some complete walks through present SPTEs may be flushed out of the HTLB by INVLPG.

C. Invariants

In this section we specify implementation dependent invariants used to prove Invariants 1 and 2.

We introduce the notion of *reachable* SPTs, to mark those SPTs which may have been walked by the HTLB since the last flush. Thus we store where partial HTLB walks may reside.

We maintain reachability using the ghost flag $\texttt{PTI}[i].re$ and the following invariants.

$$\forall j.\ \texttt{PTI}[a2i(gwo(j))].re \tag{3}$$

$$\texttt{PTI}[i].re \wedge walks\_to(i, px, i') \implies \texttt{PTI}[i'].re \tag{4}$$

$$w \in h.p[k].tlb \wedge w.l > 0 \tag{5}$$

$$\implies w \in walks(a2i(w.pfn)) \wedge \texttt{PTI}[a2i(w.pfn)].re$$

$$\texttt{PTI}[i].re \wedge \texttt{SPT}[i][px].p \implies \texttt{PTI}[i].gp[px] \tag{6}$$

Invariant 3 states that the top level SPT is always reachable. Invariant 4 states that if a SPT is reachable then its descendants in the SPT tree are also reachable. Invariant 5 states about the partial walks in the HTLB, namely that all partial walks in the HTLB are sitting on the reachable SPTs. Invariant 6 states a connection between the reachable bit of the terminal SPT and the ghost present bits of the terminal SPTEs. It states that if a present bit in the SPTE of a reachable terminal SPT is set, then the ghost present bit for this entry is also set. Note, that for maintaining the invariants, whenever we detach a shadow subtree we have to perform a hardware INVLPG and reset the reachability bits for the SPTs in the subtree.

IV. Verification

To verify the C implementation of the algorithm we use VCC, a deductive verifier for concurrent C code.VCC extends C with ghost data (possibly of non-C types, such as mathematical integers and maps), ghost functions, function contracts, and (2-state) data invariants that constrain how fields of a "valid" object are allowed to change in a legal system step. The use of 2-state invariants both allows us to model abstract automata (like TLBs) and to prove forward simulations internally (via code annotation), rather than at the meta-level.

In VCC, the TLB state and its transition relation can be specified as a ghost type and a ghost predicate. These are used at two places: (i) we represent the HTLB as a C struct type with a field of this type storing its contents and the TLB transition relation used as a 2-state invariant on this field. (ii) we use the TLB transition relation on the abstracted VTLB state as a 2-state invariant of the SPT structures, obliging VCC to show that SPT updates satisfy TLB semantics.

Next, we (very briefly) sketch the correctness arguments for INVLPG handling and HTLB steps.

1) INVLPG *intercept:* The INVLPG intercept handler with annotations is shown below:

```
Walk ws[4]; Pte pte;
ws[4] = initwalk(gwo[j], va, r); y = 4;
while (ws[y].l && !ws[y].f) {
  atomic (SPT) { /* atomic read */
    pte = SPT[a2i(ws[y].pfn)][px(va,y)]; }
  ws[y−1] = wext(pte, ws[y]);
  y = y−1; }
if (l == 0)
  atomic (SPT) { /* atomic write */
    SPT[a2i(ws[1].pfn)][px(va,0)].p = 0; }
asm_invlpga(va);
spec( /* ghost code */
  for(k = 0; k < n; k++) {
    if (PTI[k].l == 1 && PTI[k].vpfn == pfn(va) && PTI[k].vpid == j)
      PTI[k].gp[px(va,0)] = 0; } )
```

The ghost construct ***atomic***(o) checks that the subsequent block of statements is being executed atomically by the implementation, with no updates other than on volatile fields of the designated object o. The ***spec***(...) wraps regular ghost code—in our case the code that resets the ghost present bits for the invalidated VA by iterating over all terminal SPTs. Since the VTLB abstraction depends on both ghost present bits and terminal SPT entries, the above ghost and implementation code implicitly alters the VTLB content.

The intercept handler emulates the following guest steps preserving Invariant 1: (i) for each $w \in g.p[j].tlb$ such that $w.va = va$ do $deletew_{tlb}(j, w)$, and (ii) $invlpg_{cpu}(j, va)$.

The other invariants hold due to the fact that after the hardware INVLPG is performed every walk belonging to HTLB is a walk that was sitting there before the intercept happened.

2) Hardware TLB steps: The HTLB can add walks, extend walks, remove walks, set A and D bits. The VTLB in this case remains unchanged. Invariant 5 follows from Invariants 3 and 4. Invariant 2 follows from Invariants 5 and 6, and the VTLB definition, specifically from the fact that VTLB contains all complete walks over the SPTs accessible by the VP.

The verification of the complete SPT algorithm in VCC is an ongoing effort.

Acknowledgment

This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft XT project under grant 01 IS 07 008. Work of the third author done while at DFKI GmbH, Saarbrücken, Germany.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3B*, Intel Corporation, June 2009.

[2] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 3rd ed., Advanced Micro Devices, Sep. 2007.

[3] Microsoft Corp., "Windows Server 2008 R2 – virtualization with Hyper-V," http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx, 2008.

[4] J. T.-J. Sheu, M. D. Hendel, L. Wang, E. S. Cohen, R. A. Vega, and S. A. Nanavati, "Reduction of operational costs of virtual TLBs," U.S. Patent 20 080 134 174, Jun. 5, 2008.