Journal of Automated Reasoning (2020) 64:911–945
https://doi.org/10.1007/s10817-019-09539-7

Formal Reasoning Under Cached Address Translation

Hira Taqdees Syeda[1,2] · **Gerwin Klein**[1,2]

Received: 14 April 2019 / Accepted: 19 December 2019 / Published online: 18 January 2020

Abstract
Operating system (OS) kernels achieve isolation between user-level processes using hardware features such as multi-level page tables and translation lookaside buffers (TLBs). The TLB caches address translation, and therefore correctly controlling the TLB is a fundamental security property of OS kernels—yet all large-scale formal OS verification projects we are aware of leave the correct functionality of TLB as an assumption. In this paper, we present a verified sound abstraction of a detailed concrete model of the memory management unit (MMU) of the ARMv7-A architecture. This MMU abstraction revamps our previous address space specific MMU abstraction to include new software-visible TLB features such as caching of globally-mapped and partial translation entries in a two-stage TLB. We use this abstraction as the underlying model to develop a logic for reasoning about low-level programs in the presence of cached address translation. We extract invariants and necessary conditions for correct TLB operation that mirrors the informal reasoning of OS engineers. We systematically show how these invariants adapt to global and partial translation entries. We show that our program logic reduces to a standard logic for user-level reasoning, reduces to side-condition checks for kernel-level reasoning, and can handle typical OS kernel tasks such as context switching.

Keywords TLB · Cached address translation · Program verification · Isabelle/HOL · ARM

1 Introduction

We present a program logic in the interactive proof assistant Isabelle/HOL [23] for verifying programs in the presence of an ARM-style memory management unit (MMU). The MMU model is a sound abstraction of two-level page tables and a two-stage translation lookaside buffer (TLB) for caching complete and partial page table walks with address space identifiers (ASIDs) and global entries. While program logics for reasoning in the presence of address

✉ Hira Taqdees Syeda
syeda.hira.taqdees@gmail.com

Gerwin Klein
Gerwin.Klein@data61.csiro.au

1 Data61, CSIRO, Canberra, Australia

2 School of Computer Science and Engineering, UNSW, Sydney, Australia

translation exist [15], reasoning in the presence of *cached* address translation, i.e. a TLB, has so far remained an assumption in all large-scale operating system (OS) kernel verification projects.

Page table data structures encode a mapping from virtual to physical memory addresses. The OS kernel manages these, e.g. by adding, removing, or changing mappings, by keeping a page table structure per user process, and by maintaining invariants, such as never giving a user access to kernel-private data structures, ensuring that certain mappings are always present, or ensuring non-overlapping mappings between different page tables if so desired.

Since the TLB caches address translation, each of these operations may leave the TLB out of date w.r.t. the page table in memory, and the OS kernel must flush (or *invalidate*) the TLB before that lack of synchronisation can affect program execution. Since flushing the TLB is expensive, OS kernel designers work hard to delay and minimise flushes and to make them as specific as possible, using additional TLB features such as process-specific ASIDs to only invalidate specific sets of entries, or global entries for kernel mappings that are present in all processes. If this management is done correctly, the TLB has no effect other than speeding up execution. If it is done incorrectly, machine execution will diverge from the semantics usual program logics assume, e.g. wrong memory contents will be read/written, or unexpected memory access faults might occur.

Reasoning directly about programs under TLB-cached memory translation is hard, because the TLB introduces non-determinism even for otherwise deterministic programs, because global state changes even on memory reads, and because it introduces new failure modes that need to be avoided. We show in this paper how we can reduce this complexity using data refinement and arrive at a program logic that is well behaved for standard use cases such as user-level programs and OS code under fixed address translation, yet expressive enough to allow for the kind of optimisations OS developers need to achieve.

This paper builds on a re-design of our earlier work [25,26], to support global and partial translation entries. In the former [25], we developed an abstract machine model for memory operations in the presence of an ARMv7-A single-stage TLB, integrated this model with the extensive, well-validated Cambridge instruction set architecture model [8], and provided an abstraction roadmap for more recent implementations of the ARMv7-A architecture that feature a two-stage TLB. In later work [26] we extended the machine model with more MMU operations and presented a logic for reasoning about programs in the presence of cached address translation for the simpler MMU. The present paper combines both developments and shows the strength of the abstraction framework of [25] to scale up systematically from a single-stage TLB to a two-stage TLB to store complete and partial page table entries with ASIDs and global entries. More specifically, our contributions in the present paper are:

- an abstract MMU model with memory and TLB management operations in the presence of a two-stage TLB,
- improved fidelity of the machine model with respect to hardware; such as the formalisation of global TLB entries, and
- a program logic for such a two-stage TLB with ASIDs and global entries.

It is worth noting that both formalisation and abstraction of global entries in the refinement chain are non-trivial additions to the previous ASIDs-only MMU models [25,26]. Global entries fundamentally compromise the intrinsic mutual exclusion of ASIDs-only TLB entries, since they provide translation for *all* ASIDs. In this paper, we show how to cater for this compromise, and still reach a sound yet easy to reason about abstract MMU model. One of the highlights of this MMU model is that it automatically reduces to the ASIDs-only MMU model if an implementation does not make use of global translation entries.

Apart from the logic itself and its soundness, we demonstrate in a small case study that the logic can be used to reason effectively and efficiently about kernel code, that it reduces to simple side-condition checks on kernel code that does not modify page tables, and that the logic reduces to standard Hoare logic for user-level code. The logic is generic and can easily be adapted to, for instance, the shallow embedding specifications of the verified seL4 microkernel [12] use, or the more deeply embedded C semantics of the same project. It should also transfer readily to other settings such as the lower levels of CertiKOS [10] in Coq.

This paper is organised as: after presenting related work in Sect. 2, Sect. 3 introduces the Isabelle/HOL notation we use in this paper. Sect. 4 describes our formal MMU model, the TLB, and its maintenance operations for the ARMv7-A architecture, as well as the integration with the Cambridge ARM model. Sect. 5 presents the series of step-wise data refinements that each simplify reasoning. Sect. 6 presents the syntax and semantics of a small example language, as well as the program logic, Sect. 7 shows the main reduction theorems that simplify reasoning, and Sect. 8 concludes with examples, including context switching. The corresponding Isabelle/HOL theories are available online [27].

2 Related Work

The TLB has the nice property that it has no effect on the execution of a program apart from making it faster, *if* it is used correctly. For this reason, all large-scale formal OS kernel verifications so far have left correct TLB management as an assumption. This includes the OS kernel verification work in seL4 [11,12] and CertiKOS [10], which both do reason about page table structures, but omit the TLB. Similarly, Daum et al. [6] reason about user-level programs on top of seL4, including page tables, but not about the TLB. The certified abstraction layers over deep specifications in the CertiKOS project [9] are similar in spirit with our refinement chain for TLB abstraction, and the deep specification framework as such should apply to the abstractions presented here. The general idea of refinement stacks in operating system verification goes back to at least PSOS [7,22].

Kolanski et al. [13–15] develop an extension of separation logic to formally reason about page tables, virtual memory access, and shared memory in Isabelle/HOL. However, the memory model does not include the TLB and does not address TLB caching, consistency and invalidation, which we add here.

Alkassar et al. [2] verify x64-like TLB virtualisation; they model the TLB as a set of complete and partial page table walks and provide semantics for address translation and TLB-invalidation operations. In the continuation of this work, Kovalev [16] states a reduction theorem for page table walks in ASID 0 for a specific hypervisor setup. However, while other parts of this development are mechanised, this reduction theorem is not. As we will see in Sect. 6, the restriction to one ASID makes the model too conservative for more general OS code. These efforts focus on verifying TLB virtualization, and as such they provide a TLB abstraction function on the host configuration to formulate the virtualised TLB for every guest. We also model the TLB as a set of page table walks, but our aim is a generic reasoning framework for verifying programs (both kernel-level and user-level) in the presence of a TLB.

Barthe et al. [5] formalise a virtualisation model for the ARM architecture featuring the TLB and cache, and provide abstract reasoning about cache-based side-channels. They model the TLB as a partial map from virtual addresses to machine addresses, and provide axiomatic semantics of hypervisor actions. They reason about cache-based attacks and countermeasures, prove flush-enforced isolation between guest operating systems upon context switch, and reason about the transparency provided by the virtualization model to the guest operating system. In contrast, the work presented here provides a program logic and a proof that the abstraction of the TLB is sound with respect to a detailed concrete base-level model of the hardware. Our model shows that a pure map from virtual to machine addresses as the TLB model is too simple for the full behaviour of real operating systems.

Nemati et al. [21] verify the isolation properties of a hypervisor that uses direct paging on the ARMv7 architecture to virtualize the CPU memory subsystem. They develop a formal CPU model in the HOL4 theorem prover [24] by extending the formal ARM ISA model [8] with an MMU. The MMU model provides address translation through two-level page tables, and the CPU state includes system MMU registers. Nemati et al. [20] extend this work further to design, implement and verify an MMU virtualization platform for the ARMv7-A architecture, with Linux as an untrusted guest. The focus of their work is to virtualize the ARMv7 memory subsystem, but they do not model the TLB for reasoning about programs in general.

Lutsyk [18] provides a paper-and-pencil correctness proof for a pipelined multi-core implementation of the MIPS-86 ISA, extended with hardware virtualisation. This work also models the TLB as a set of walks, but is interested in wether the hardware circuits correctly implement the TLB semantics, whereas we are concerned with reasoning about programs on top.

Achermann et al. [1] are also more concerned with hardware correctness than program correctness. They present a methodology for formalising the physical address space of Systems-on-Chip (SoCs), and demonstrate it by modeling the MIPS R4600 TLB. They develop a refinement stack for reasoning about the physical address interconnects of the TLB, and conclude “the impossibility of correctly configuring a MIPS R4600 TLB as specified in its documentation” [1].

3 Notation

This section introduces Isabelle/HOL syntax used in this paper, where different from standard mathematical notation.

Isabelle denotes the space of total functions by ⇒, and type variables are written `'a`, `'b`, etc. The notation `t::τ` means that HOL term `t` has HOL type `τ`. The `option` type

datatype `'a option = None | Some 'a`

adjoins a new element `None` to a type `'a`. We use `'a option` to model partial functions, writing `⌊a⌋` instead of `"Some a"` and `"'a ⇀ 'b"` instead of `'a ⇒ 'b option`. The `Some` constructor has an underspecified inverse called `the`, satisfying the equation `the ⌊x⌋ = x`, and the function `these` converts an `'a option set` to an `'a set` with `these S = {y | ∃x∈S. x = ⌊y⌋}`. The set of values returned by a total function `f` is denoted by `range f`, i.e. `range f = {y | ∃x. f x = y}`, whereas the range of a partial function is defined as `ran f = {y | ∃x. f x = ⌊y⌋}`.

Function update is written `f(x := y)` where `f::'a ⇒ 'b`, `x::'a` and `y::'b`, and `f(x ↦ y)` stands for `"f (x:= Some y)"`.

Isabelle’s type system does not include dependent types, but can encode numerals and machine words of fixed length. The type `'n word` represents a word with `n` bits; concrete types include e.g. `32 word` and `64 word`.

The Cambridge ARM formalisation [8] models the CPU state as a record type `state`. For every record field, there is a *selector* function of the same name. For example, if `s` has type `state` then `MEM s` denotes the value of the `MEM` field of `s`, and `s(|MEM := id|)` will update `MEM` of `s` to be the identity function `id`.

The ARM formalisation uses the state monad to model state transformers. The state monad encodes computations with side effects in a purely functional model. For result type `'a` and state type `'s`, the associated monad type, abbreviated `('s, 'a) state_monad`, is `'s ⇒ 'a × 's`. That is, a function from current state to next state together with a computation result. A pure state transformer is typically denoted by the one-valued return type `unit`, that is, `'s ⇒ unit × 's`. The two monad constructors `return` and `bind` are defined as follows:

```
return a ≡ λs. (a, s)
bind f g ≡ λs. let (v, s') = f s in g v s'
```

The constructor `return` simply injects the value `a` into the monad type, passing the state unchanged, while `bind` sequentially composes a computation `f`, and a computation `g` (a function from the return type of `f`). We use `do` syntax for longer computations:

```
bind f g ≡ do { x ← f; g x }
```

For fetching and updating a particular parameter from the state, the Cambridge ARM model uses the functions `read_state` and `update_state`:

```
read_state f ≡ λs. (f s, s)        update_state f ≡ λs. ((), f s)
```

We abbreviate multiple `read_state` calls into tuple notation, written as `read`:

```
(a,b) ← read (f,g)
```

and write `update_state` as `update`.

4 A Formal MMU Model for the ARMv7-A Architecture

In this section we present a formal MMU model including address translation, memory operations, and MMU maintenance operations for the ARMv7-A architecture with a two-stage TLB with ASIDs and global entries. The ARM architecture provides multiple address translation modes that differ in the number of levels and number of bits being translated. Without loss of generality for the treatment of the two-stage TLB we focus on one of these modes here—the others are analogous. This mode provides four sizes of pages (small, large, section, and super section; cf. [3, Chapter B3]) and a two-level page table structure. Depending on the virtual address being translated, the first-level of a page table structure provides either the base address of a section, a super section, or the pointer to a second-level page table containing the base addresses for small and large pages. The location of the root of a page table structure in the main memory is determined by a hardware register, the translation table root register TTBR0.

As opposed to simpler single-stage implementations of ARMv7-A, more recent implementations of ARMv7-A, such as Cortex-A15, include a two-stage TLB that caches both, full address translations, and partial walks, cf. [4, Chapter 5]. The first stage caches entries that provide end-to-end address translations, i.e. results of complete page table lookups. In the formalisation we call this stage simply the TLB. The second stage, called the page directory cache (PDC), caches the results of partial page table lookups—up to the first level of the page table only. For a two-level page table this means that the PDC stores translation entries for sections and super sections, and pointers to the second-level page table containing small and large page translation entries.

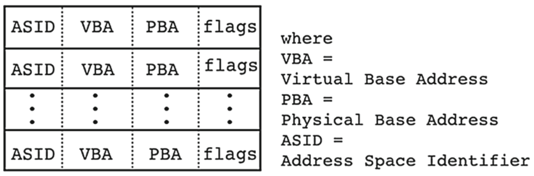

Fig. 1 An Abstraction of TLB Entries

For both stages, the architecture associates a process-specific tag called address space identifier (ASID) with TLB entries to enable translation caching for different processes. It also supports global page table entries, which match all ASIDs.

The ARM architecture manual [3] describes the two-stage TLB as a black box, i.e. by its external interface only. It does not specify the replacement strategy or its exact internal state. We use the same approach and base our abstraction for the TLB and the PDC directly on the architecture manual: we specify their operational model and lookup operation, but leave much of the implementation open. For page table operations we reuse Kolanski's existing ARM page table model [15], and integrate it with the two-stage TLB formalisation that we build up in this section to form the MMU model. Kolanski's model differentiates between virtual and physical address by type, and we continue in that tradition. He defines addresses `addr_t` as:

```
datatype ('a, 'p) addr_t = Addr 'a
```

where `'a` is the address size (e.g. `32` word) and `'p` is a tag which can be `physical` or `virtual`. For modeling addresses, we specialise `addr_t` as:

```
vaddr = "(32 word,virtual) addr_t"    paddr = "(32 word,physical) addr_t"
```

We use `addr_val (Addr a) = a` to extract the address.

We formalise the TLB and the PDC as a set of TLB and PDC entries respectively. A TLB entry consists of an ASID or global tag, a virtual base address, a physical base address, and flags for access control and other page attributes. Figure 1 gives a visual representation. Corresponding to the four page sizes of the architecture, there are four different sizes of TLB entries. In this paper, we formalise page table entries for small pages and sections only, as the remaining are analogous. This means, for this setting we have two types of TLB entries, one with 20-bit base addresses for small pages and one with 12-bits for sections. Formally:

```
  type_synonym tlb = tlb_entry set
datatype tlb_entry  =  EntrySmall "(asid option)" "(20 word)" "(20 word)" flags
                    |  EntrySection "(asid option)" "(12 word)" "(12 word)" flags
```

A global TLB entry has ASID field `None`, while `Some` ASID represents a process-specific translation entry. The record type `flags` represents the permissions of TLB entries, which include the non-global `nG` bit from the respective page table entry, as well access control such as read/write/execute rights. Similar to the TLB entry, a PDC entry specifies either a 32-bit base addresses for sections or 32-bit pointers to the second-level page tables. Formally:

```
type_synonym pdc = pdc_entry set
datatype pdc_entry  =  PDE_Table "(asid)" "(12 word)" "(32 word)"
                    |  PDE_Section "(asid option)" "(12 word)" "(32 word)" flags
```

We always associate an ASID with a `PDE_Table` entry since the corresponding page directory entries provide no means for encoding global page table pointers. The constructor `PDE_Section` can have either an ASID or a global tag depending on the `nG` bit of its `flags`.

With the TLB and the PDC state formalised, we now describe their lookup. For any given 32-bit virtual address, a lookup finds the corresponding TLB or PDC entry. A lookup can have three kinds of results:

```
datatype 'e lookup_type = Miss | Incon | Hit 'e
```

These results are: either there is no corresponding entry (`Miss`), or there is more than one matching entry, representing an inconsistency (`Incon`), or there is exactly one correct result (`Hit`). The `lookup_type` is polymorphic w.r.t. the entry type, so the function can be used for both TLB and PDC.

We say an entry *matches* a pair of ASID `a` and virtual address `va` when the top bits of `va` equal the virtual base address of the entry and it is either a global entry or has the same ASID `a`. Let `range_of e` be the set of addresses matched by entry `e`, and `asid_of e` the ASID of `e`, then we can define the `lookup` operation as follows.

```
entry_set :: 'e set ⇒ asid ⇒ vaddr ⇒ 'e set
 entry_set tp a va ≡
 {e ∈ tp | va ∈ range_of e ∧ (asid_of e = None ∨ asid_of e = ⌊a⌋)}

lookup :: "'e set ⇒ asid ⇒ vaddr ⇒ 'e lookup_type"
 lookup tp a va ≡
 let S = entry_set tp a va
 in if S = ∅ then Miss else if |S| = 1 then Hit (the_elem S) else Incon
```

where `the_elem {x} = x`. The parameter `tp` is either a TLB entry set or a PDC entry set, and the functions `range_of` and `asid_of` are instantiated for both.

A given TLB or a PDC can be split into its global and non-global entries as:

```
glb_entries tp = {e ∈ tp | asid_of e = None}
nglb_entries tp = {e ∈ tp | ∃a. asid_of e = ⌊a⌋}
```

This covers the base model of the two-stage TLB itself. We now present a formal MMU model based on the ARM architecture manual [3] integrated with the instruction set architecture (ISA) semantics by Fox and Myreen [8]. This ISA model is very detailed and extensively validated, but it assumes a flat, total function `MEM :: 32 word ⇒ 8 word` without address translation as its model for memory. We keep `MEM` as the basic model for physical memory, but generalise it to the partial function `MEM :: "paddr ⇀ 8 word"` to express that it works on physical addresses and that not all physical address might be backed by memory in the machine. If a computation accesses non-existing memory, an exception will be raised. We then change all read and write instructions that access main memory to not go to physical memory directly, but to go through the two-stage TLB and address translation first. The existing ISA model conveniently provides a narrow interface to memory with the functions `mem_write` and `mem_read` that all other memory accesses go through, so we concentrate our work there.

Since our plan for Sect. 5 is to provide a series of MMU models that differ in the TLB abstraction, making them simpler and easier to reason about as we progress, we design the interface between the rest of the ARM model and the MMU as a type class `mmu` in Isabelle that we can instantiate. Separate instances will give us separate models between which we then can prove refinement theorems. To get there, we first need to model the rest of the MMU. Figure 2 gives an overview. To formalise this picture, we extend the original `state` record of the ISA model with two additional hardware registers: the page table root register `TTBR0`, and the current ASID register `ASID`. We then use Isabelle's extensible records [19] to extend `state` with the pair `tlb × pdc` which will contain the two-stage TLB hardware state we modeled earlier in this section.

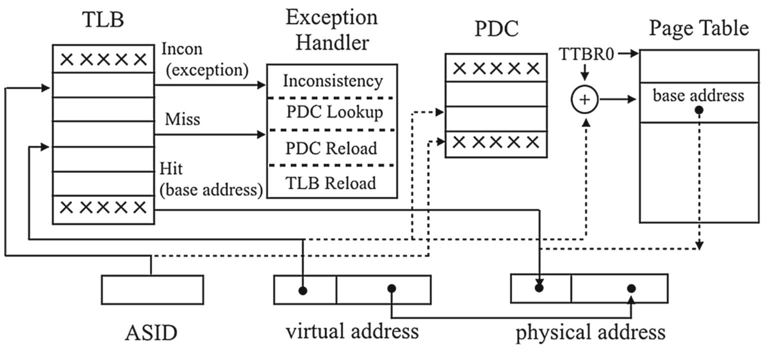

Fig. 2 ARMv7-style Memory Management Unit

The main interface for the rest of the ARM model to the MMU is wrapped up in the type class `mmu`:

```
class mmu =
  mmu_translate :: vaddr ⇒ 'a state_scheme ⇒ paddr × 'a state_scheme
  mmu_read :: "vaddr × nat ⇒ 'a state_scheme ⇒ bl × 'a state_scheme"
  mmu_write :: "bl × vaddr × nat ⇒ 'a state_scheme ⇒ unit × 'a state_scheme"
  update_TTBR0 :: paddr ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
  update_ASID :: "asid ⇒ 'a state_scheme ⇒  unit × 'a state_scheme"
  flush :: flush_type ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
```

where `'a state_scheme` are the potential extensions of the existing record type `state`. The interface for the values being read and written in the ARM model is via `bl = bool list` instead of machine words directly, which we keep here. The `nat` parameter indicates how many bytes to read/write, e.g. one byte, a word, a double word, etc. The OS kernel updates the page table root and ASID registers during context switches which in turn requires TLB maintenance; therefore we introduce `update_TTBR0` and `update_ASID` instructions. The `flush` instruction, as the name describes, invalidates the two-stage TLB entries for maintaining coherency.

We now explain the instantiation of each of the parameters of type class `mmu` for our MMU model, these functions also constitute the base model of our refinement chain in Sect. 5. We begin by presenting the interface between TLB and page table present in main memory.

4.1 Page Table Walk

For loading the two-stage TLB from the two-level page table, we classify page table walks into:

```
datatype pt_walk_typ   =   Fault
                       |   Partial_Walk "pdc_entry"
                       |   Full_Walk "tlb_entry" "pdc_entry"
```

For a virtual address `va`, a page table walk resulting in `Fault` represents an unmapped virtual address, `Partial_Walk` with a PDC entry means that either the virtual address `va` belongs to a mapped section, or its translation can be further achieved using a page table present at a specific location in the memory. A `Full_Walk` with a TLB and a PDC entry represents that the virtual address `va` is fully mapped: it either belongs to a section, or to a mapped small page of the memory. We then define page table-directory walk `ptd_walk` as:

```
ptd_walk :: asid ⇒ heap ⇒ paddr ⇒ vaddr ⇒ pt_walk_typ
ptd_walk a m rt va ≡
case pdc_walk a m rt va of
    None ⇒ Fault
  | ⌊pde⌋ ⇒ case pdc_to_tlb pde m va of None ⇒ Partial_Walk pde
                                         | ⌊entry⌋ ⇒ Full_Walk entry pde
```

where heap = "paddr ⇀ byte". For a given virtual address va, the function pdc_walk walks the first-level page table present at the root rt in memory mem under the ASID a, and returns the first-level page table entry encoded as a pdc_entry. For an invalid pdc_walk we return Fault, while for a valid pdc_walk we complete the remaining page table walk using the function pdc_to_tlb and return either Partial_Walk or Full_Walk. The ASID of the resultant entries is determined by the access permissions of the respective page table entries to decide whether it is global or not, and the current ASID when it is not.

Note that a Partial_Walk with a PDE_Table entry does not mean that a Full_Walk for the virtual addresses va has to exist. When we are interested in full walks only, we use the function pt_walk that returns a tlb_entry option, with None for partial walks and faults.

In summary, we have three variants of page table walks, namely pdc_walk for partial walks only, pt_walk for full walks only, and ptd_walk for the combination.

4.2 Memory Operations

We now explain memory operations including address translation, memory read and write for our base MMU model. Address translation for memory operations is defined as:

```
mmu_translate va = do {
  update (λs. s⦇TLBs := TLBs s - tlb_evict s⦈);
  (m, rt, a, tlb, pdc) ← read (MEM, TTBR0, ASID, TLBs);
  case lookup tlb a va of
  Miss ⇒
    case lookup pdc a va of
    Miss ⇒
      case ptd_walk a m rt va of Fault ⇒ raise PAGE_FAULT
      | Partial_Walk pde ⇒ do {
          update (λs. s⦇TLBs := TLBs s ∪ (∅, {pde})⦈);
          raise PAGE_FAULT }
      | Full_Walk entry pde ⇒ do {
          update (λs. s⦇TLBs := TLBs s ∪ ({entry}, {pde})⦈);
          return (va_to_pa va entry) }
    | Incon ⇒ raise IMPLEMENTATION_DEFINED
    | Hit pde ⇒
        let entry = pdc_to_tlb pde m va
        in if fault entry then raise PAGE_FAULT
           else do {
               update (λs. s⦇TLBs := TLBs s ∪ ({the entry}, ∅)⦈);
               return (va_to_pa va (the entry)) }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

The function mmu_translate first evicts an underspecified set of entries from the TLB and the PDC. This models the fact that the architecture does not define the replacement strategy and the programmer must assume that any entry could be evicted at any time.[1] Since the rest of the Cambridge ARM model is deterministic, we use an oracle function `tlb_evict` here instead of true non-determinism.

The next step in `mmu_translate` after reading out the hardware state is to do a full TLB lookup for the virtual address `va` to be translated under the current ASID. If the result of that lookup is `Incon`, the machine raises an unrecoverable exception and halts, expressing the fact that in normal operation, this state should never be encountered. If the result is `Hit entry`, we translate this TLB entry to the corresponding physical address `pa` using the function `va_to_pa` and return that address. A full formalisation would at this point additionally check flags and access rights and generate the appropriate exception information where needed.

If the result of the TLB lookup is `Miss`, we perform a PDC lookup, a potential TLB and PDC reload and the address translation. If the result of the PDC lookup is `Incon` the machine raises an unrecoverable exception and halts. If the PDC lookup results in `Hit pde`, we complete the translation for address `va` from the page directory entry `pde` and store the result in the TLB. If the result of the PDC lookup is `Miss`, we perform a full page table walk in memory using the function `ptd_walk`, and potentially reload both PDC and TLB. If the result of the `ptd_walk` is a `Fault`, we raise this fault, which will cause the machine to jump to the appropriate exception handler. If we get `Partial_Walk` we reload the PDC and again raise a page fault. If the result of the page table walk is `Full_Walk` with TLB and PDC entries, we simply add them to the TLB and PDC respectively, and execute address translation as in the `Hit` case.

Reusing the original functions `mem_write` and `mem_read` from the ARM model for physical memory, the definition of memory operations are straightforward:

```
mmu_write (val, va, sz) =
do { pa ← mmu_translate va; when_no_exc mem_write (val, pa, sz) }

when_no_exc f = do {
  exception ← read exception;
  if exception = NoException then f else return ()
}

mmu_read (va, sz) = do { pa ← mmu_translate va; mem_read (pa, sz) }
```

Both `mmu_write` and `mmu_read` first perform address translation, and then their original purpose, but using translated addresses instead. In case of an exception in `mmu_translate`, the write function does nothing to give the translation exception precedence, while the pure read function can continue, because it does not change the state.

4.3 MMU Operations

We now explain the MMU operations for updating the page table root and ASID register, as well as the TLB flush operations.

In this base model the instructions `update_TTBR0` and `update_ASID` merely do what their names describe. They do not evict anything from the TLB:

```
update_TTBR0 r = update (λs. s⦇TTBR0 := r⦈)
update_ASID a = update (λs. s⦇ASID := a⦈)
```

We formalise the flush operations provided by the ARM manual as:

[1] ARM also provides locked down entries that will not be evicted automatically. These could be modelled easily here by excluding them from the eviction set.

```
datatype flush_type  =  FlushTLB          |  FlushVR "(vaddr set)"
                     |  FlushASID asid    |  FlushAVR asid "(vaddr set)"
```

The flush instructions operate on both TLB and PDC. The instantiation for this base model is:

```
flush f ≡
case f of FlushTLB ⇒ update (λs. s(|TLBs := (∅, ∅)|))
| FlushVR vset ⇒ update (λs. s(|TLBs := flush_vset (TLBs s) vset|))
| FlushASID a ⇒ update (λs. s(|TLBs := flush_asid (TLBs s) a|))
| FlushAVR a vset ⇒ update (λs. s(|TLBs := flush_asid_vset (TLBs s) a vset|))
```

FlushTLB simply returns an empty TLB and PDC, whereas FlushVR takes a pair tlb × pdc and flushes the entries matching the given set of virtual addresses:

```
flush_vset (tlb, pdc) vset =
(tlb - (⋃x∈vset {e ∈ tlb | x ∈ range_of e}),
 pdc - (⋃x∈vset {e ∈ pdc | x ∈ range_of e}))
```

FlushASID flushes all entries under the given ASID:

```
flush_asid (tlb, pdc) a =
(tlb - {e ∈ tlb | asid_of e = ⌊a⌋}, pdc - {e ∈ pdc | asid_of e = ⌊a⌋})
```

And FlushAVR flushes the entries for the given set of virtual addresses under the given ASID:

```
flush_asid_vset (tlb, pdc) a vset =
(tlb - (⋃x∈vset {e ∈ tlb | x ∈ range_of e ∧ asid_of e = ⌊a⌋}),
 pdc - (⋃x∈vset {e ∈ pdc | x ∈ range_of e ∧ asid_of e = ⌊a⌋}))
```

By redirecting all other memory-related functions in the ARM model to go through the interface of the type class mmu, we arrive at a full operational model that supports address translation and MMU operations. The purpose of this paper is not to provide a fully detailed formalisation of existing hardware, but to present the main ideas on how to simplify reasoning in the presence of a two-stage TLB of the ARMv7-A architecture. Despite this focus, we have validated the model using the following methods:

– Close manual review of the formal specification against the behaviour described in the ARM reference manual [3].
– Specification review against expectations of experience kernel engineers.
– Formal symbolic testing through executing a number of memory instruction cycles in the theorem prover. By symbolic execution in the simplifier, we manually checked consistency against expected behaviour.

A full formalisation would need a more extensive test suite in the spirit of Fox and Myreen [8], and at least making the formal symbolic testing mentioned above more systematic and complete. Validating a fully detailed TLB model has additional subtleties compared to validating the behaviour of an ISA specification, because the precise internal TLB state of the hardware is neither specified in the architecture, nor easily inspectable in hardware. If one were to validate directly against hardware, it would therefore be necessary to use indirect observations. These could include cases where page faults are expected/not expected due to TLB state, for instance pages being accessible even though the current state of the page table disagrees. After developing a set of such test cases, hardware and specification could be run against each other, comparing outcome.

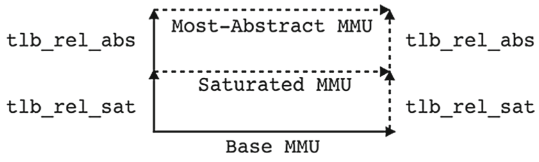
Fig. 3 Refinement Stack of MMU Models

In summary, we have so far extended the Cambridge ARM model by: a change of memory model to admit the notion of unmapped memory, the introduction of an MMU including two-stage TLB and page table lookup mechanisms, the extension with maintenance operations, and an adjustment of the subsequent memory operations to include the address translation layer.

5 MMU Abstraction

The MMU model of Sect. 4 gives us the ground truth of how hardware operates, and thereby the foundation for a logic for programs under the two-stage TLB, but the model is hard to reason about directly. From Sect. 4, we see that a two-stage TLB introduces:

1. non-determinism through unspecified entry replacement strategy,
2. potential state change caused by any mapped memory access, including reads,
3. potential (internally) inconsistent TLB and PDC state from multiple conflicting entries, and
4. potential (external) inconsistency between page table, TLB, and PDC.

The latter two are states the program must avoid. The first two introduce unnecessary complexity: a program that is otherwise deterministic should not require reasoning about nondeterminism, and a correctly operated TLB framework should not complicate reasoning about memory reads nor memory writes that are unrelated to page tables.

In this section, we show how we can construct a model that avoids the additional complexity and produces sufficient conditions for safe execution. In particular, we build a series of formal abstractions of the concrete MMU model of Sect. 4 that are increasingly easier to reason about, but preserve functionality and the optimisation opportunities OS developers must be able to exploit. We verify these step-wise abstractions by refinement theorems.

We observe as the first step in our abstraction chain that a TLB with fewer entries is always more consistent, and in this sense safer, than one with more entries. Formally, lookup results naturally form an order with `Miss` being the bottom element, and `Incon` the top:

$$\texttt{l} \le \texttt{l}' \equiv \texttt{l} = \texttt{Miss} \lor \texttt{l}' = \texttt{l} \lor \texttt{l}' = \texttt{Incon}$$

We can then prove monotonicity.

Lemma 1 $\texttt{tp} \subseteq \texttt{tp}' \Longrightarrow \texttt{lookup tp a va} \le \texttt{lookup tp}' \texttt{ a va}$

Proof By case distinction and unfolding the definitions. □

We use this observation in the abstraction chain by making the abstraction less safe, i.e. more inconsistent, with the standard refinement idea that if we manage to prove safe behaviour of the abstraction, we will also have proved safe behaviour of all possible actual executions.

Our refinement stack consists of two steps, shown in Fig. 3. In the first refinement, we remove eviction and at the same time saturate the PDC and the TLB with the mapped state of page table. This eliminates nondeterminism, state change for the memory reads and the

PDC lookup from the concrete MMU model. Next, we abstract the TLB and the PDC to an extent that no actual cache lookup is required: two-stage TLB inconsistencies are tracked using a record of virtual addresses, while memory and MMU operations are performed using the page tables present in main memory. For each level in the refinement stack we prove that its abstraction preserves a refinement relation and is sound with respect to its immediate concrete MMU model. We then join refinement levels in order to show the soundness of the most abstract model w.r.t. base model of Sect. 4.

The main burden on the proof engineer that we cannot hope to eliminate completely in general will be to show that the TLB and the PDC are currently in a consistent state for the address to be accessed. We formalise consistency of TLB and PDC for a virtual address as:

```
consistent m rt tlb pdc a va ≡
consistent_lookup tlb (pt_walk a m rt va) a va ∧
consistent_lookup pdc (pdc_walk a m rt va) a va

consistent_lookup :: 'e set ⇒ 'e option ⇒ asid ⇒ vaddr ⇒ bool
consistent_lookup tp e a va ≡
lookup tp a va = Miss ∨ lookup tp a va = Hit (the e) ∧ no_fault e
```

The consistency condition combines internal consistency of the TLB and the PDC (no `Incon` results permitted), with external consistency, i.e. synchronicity with the current state of the mapped page table for this particular address. The side-condition for `Hit` models that the TLB and PDC do not provide negative results (`no_fault`), i.e. a `Hit` only models the presence, not the absence of a mapping.

We now provide our first abstraction of the MMU and its refinement theorem.

5.1 The Saturated MMU Model

In this abstraction, we remove nondeterminism and state change on memory reads by removing the TLB and PDC eviction and also by saturating them hierarchically with the mapped page table entries after every memory and MMU operation. This saturation eliminates the potential PDC lookup while resolving the virtual addresses: we first saturate the PDC with the mapped page table entries of the current state, and then from this saturated PDC we complete page table walks and saturate the TLB. This way the TLB contains all mapped entries for the current page table through the saturated PDC, and any TLB `Miss` implies a page table fault. One critical aspect of this saturation is that we propagate all inconsistencies of the PDC for the current state to the TLB, so that the TLB already captures the complete information provided by the PDC. This will allow us to mostly eliminate the PDC in this model already.

5.1.1 Memory Operations

We instantiate `mmu_translate` of type class `mmu` for a deterministic saturated MMU and name it `mmu_translate_sat`:

```
mmu_translate_sat va = do {
  pdc_tlb_refill;
  (a, tlb, pdc) ← read (ASID, TLBs);
  case lookup tlb a va of Miss ⇒ raise PAGE_FAULT
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒ return (va_to_pa va entry)
}
```

```
  pdc_tlb_refill = do {
    (m, rt, a) ← read (MEM, TTBR0, ASID);
    let pdes = ran (pdc_walk a m rt); entries = these (⋃v to_tlb pdes a m rt v)
    in update (λs. s⦇TLBs := TLBs s ∪ (entries, pdes)⦈)
  }
```

The call to `pdc_tlb_refill` at the beginning of `mmu_translate_sat` achieves the saturation mentioned above by adding all the mapped page directory entries and then all the mapped TLB entries for the current state and ASID. The function `to_tlb` (definition omitted) in `pdc_tlb_refill` takes a PDC and completes the page table walk for the given virtual address under the current ASID. It returns either a full walk or a fault, i.e. its results are the same as the TLB. The big-union term performs this walk-completion for all virtual addresses. In `mmu_translate_sat`, after the refill, we now only need a simple TLB lookup: `Incon` still leads to the same exception as before, `Miss` implies a page fault, since the TLB is saturated with the mapped entries, and `Hit` gives us the respective physical address.

We also adjust the definition of memory operations to preserve saturation for this model. The saturated instantiations are `mmu_write_sat` and `mmu_read_sat`:

```
  mmu_write_sat (val, va, sz) = do {
    pa ← mmu_translate_sat va;
    when_no_exc do { mem_write (val, pa, sz); pdc_tlb_refill }
  }

  mmu_read_sat (va, sz) = do { pa ← mmu_translate_sat va; mem_read (pa,
sz) }
```

In `mmu_write_sat` the two-stage TLB is refilled after the write operation to maintain saturation as this write could have changed a page table in memory. `mmu_read_sat` achieves the saturation implicitly through `mmu_translate_sat`, as reading from memory does not affect the state of page table.

5.1.2 MMU Operations

Similar to memory operations, we saturate the two-stage TLB after updating the page table root and ASID register with the mapped state of the page table under the active ASID. We saturate both stages of the TLB after the flush operations to make sure that, while inconsistencies are removed, saturation is maintained. The instantiations of MMU operations for this model are:

```
update_TTBR0_sat r = do { update (λs. s⦇TTBR0 := r⦈); pdc_tlb_refill }
update_ASID_sat a = do { update (λs. s⦇ASID := a⦈); pdc_tlb_refill }
```

```
flush_sat f = do {
  (case f of FlushTLB ⇒ update (λs. s⦇TLBs := (∅, ∅)⦈)
    | FlushVR vset ⇒ update (λs. s⦇TLBs := flush_vset (TLBs s) vset⦈)
    | FlushASID a ⇒ update (λs. s⦇TLBs := flush_asid (TLBs s) a⦈)
    | FlushAVR a vset ⇒ update (λs. s⦇TLBs := flush_asid_vset (TLBs s) a vset⦈));
  pdc_tlb_refill
}
```

5.1.3 Refinement Theorems

We now present refinement theorems between the nondeterministic and saturated MMU models. We define the refinement relation `tlb_rel_sat` as:

```
tlb_rel_sat s t ≡
let tlb_nondet = fst (TLBs s); pdc_nondet = snd (TLBs s);
   tlb_sat = fst (TLBs t); pdc_sat = snd (TLBs t)
in truncate s = truncate t ∧
  tlb_nondet ⊆ tlb_sat ∧ pdc_nondet ⊆ pdc_sat ∧ saturated t
```

where the notation `truncate s` means all fields of the extensible `state` record without the `tlb × pdc` extension. This relation demands that the states `s` and `t` differ only in their TLB and PDC content, that the nondeterministic TLB of state `s` has fewer entries than the saturated TLB of state `t`, and that the nondeterministic PDC of state `s` has fewer entries than the saturated PDC of state `t`. These subset relations provide a lookup order between the states `s` and `t`. The assertion `saturated t` demands that both stages remain saturated, i.e.

```
saturated t ≡ tlb_saturated t ∧ pdc_saturated t
tlb_saturated t ≡ ran (pt_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ fst (TLBs t)
pdc_saturated t ≡ ran (pdc_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ snd (TLBs t)
```

For presenting our refinement theorems below, we combine the evaluation of address translation, memory read and write functions for nondeterministic and saturated states into functions `mem_op` and `mem_op_sat` respectively. For example, `mem_op s` represents the evaluation of `mmu_translate`, `mmu_write` and `mmu_read` of Sect. 4.2 in the state `s` with a nondeterministic two-stage TLB. Similarly, MMU operations for flushing the TLB and for updating TTBR0 and ASID registers are combined into `mmu_op` and `mmu_op_sat` respectively.

Theorem 1 *The nondeterministic and saturated memory operations preserve the refinement relation and agree on the results, for consistent virtual addresses.*

$$\frac{\begin{array}{c}\texttt{mem\_op f s = (res, s')}\\ \texttt{mem\_op\_sat f t = (res', t')} \quad \texttt{consistent\_sat f t} \quad \texttt{tlb\_rel\_sat s t}\end{array}}{\texttt{res' = res} \wedge \texttt{tlb\_rel\_sat s' t'}}$$

where `consistent_sat` *ensures that the memory operation is for a consistent virtual address of the saturated two-stage TLB, using the* `consistent` *predicate from the beginning of Sect. 5.*

Proof For address translation, we observe that the nondeterministic two-stage TLB of state `s` is `va`-consistent given its subset relationship with the `va`-consistent two-stage TLB of state `t`. The lookup for virtual address `va` in both states `t` and `s` either produces a `Miss` or a `Hit`. When the saturated TLB of state `t` produces a `Miss` (implies a page table fault), the nondeterministic TLB of state `s` also must have a `Miss` and we are in the PDC lookup case. The nondeterministic PDC of state `s` then has to conform with the saturated PDC of state `t`, producing either a `Miss` or a `Hit` with a consistent page directory entry and completing the translation for the address `va` through a page table walk to eventually encounter a page table fault. In case of `Hit` with an entry in the saturated TLB of state `t`, the TLB of state `s` either agrees on the same entry with a `Hit`, or performs a consistent PDC lookup and page table walk. The refinement for memory read and write follows directly from the refinement of the address translation. □

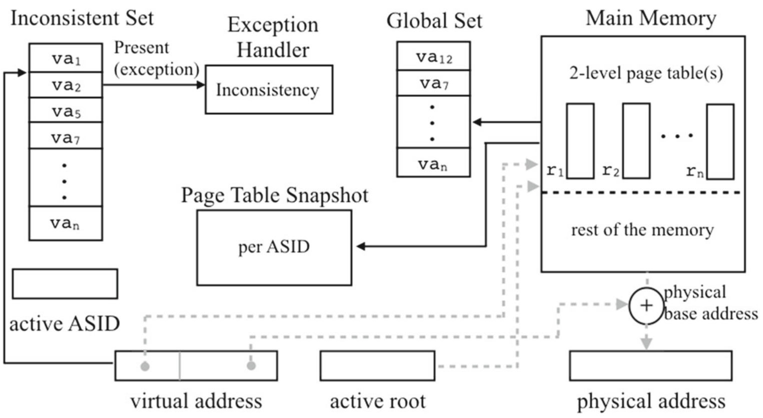

Fig. 4 ARMv7-style Memory Management Unit with Abstract TLB

Theorem 2 *The nondeterministic and saturated MMU operations preserve the refinement relation.*

$$\frac{\texttt{mmu\_op f s = ((), s')} \qquad \texttt{mmu\_op\_sat f t = ((), t')} \qquad \texttt{tlb\_rel\_sat s t}}{\texttt{tlb\_rel\_sat s' t'}}$$

Proof By unfolding definitions and set reasoning. □

With this we conclude our saturated MMU model and its refinement with the nondeterministic MMU.

5.2 The Abstract MMU

We now present the abstract MMU model of our refinement chain Fig. 3. We abstract the TLB lookup completely, and soundly model TLB functionality using a set of TLB-inconsistent virtual addresses. Address translation is performed directly using the page tables. For this MMU model, we do not extend the state with `tlb × pdc`, instead the two-stage TLB is modeled using these three components:

```
incon_set :: vaddr set
glb_set :: vaddr set
snapshot :: asid ⇒ vaddr set × (vaddr ⇒ pt_walk_typ)
```

Before presenting details, we briefly revisit the relationship between TLB and page tables: *the TLB caches entries from multiple page tables present in main memory under different ASIDs*. Now that we aim to completely abstract the TLB and aim to capture its caching functionality using only a set of addresses, we require a conservative estimate of what the TLB might remember from the time an ASID was last active. Essentially this is, for each ASID, a snapshot of the current page table state when that ASID was last active, modulo all addresses that were inconsistent at that time. This estimate then enables us to keep track of the inconsistencies for different ASIDs and also to detect new inconsistencies while switching between ASIDs.

Figure 4 gives an overview of the state space of our abstract MMU model. The `incon_set` stores inconsistent virtual addresses (both, global and non-global), and the `glb_set` keeps track of which addresses are globally mapped, irrespective of their consistency. The `snapshot` holds the state of the two-stage page table for all inactive ASIDs, using `pt_walk_typ`. The TLB-`snapshot` of the page table state modulo the inconsistent virtual addresses for every ASID is modeled as a pair `"vaddr set × (vaddr ⇒ pt_walk_typ)"`. The `snapshot` is used only in one situation: to detect inconsistent addresses while switching the ASID register. The ASID register is updated on a context switch between processes, and we use `snapshot` for a page table comparison: as for the rest of the model, no actual TLB lookup is involved.

We now explain the operations of the abstract MMU model.

5.2.1 Memory Operations

For translating a virtual address in the abstract two-stage TLB, we merely check its consistency using the `incon_set`, and subsequently translate it directly form the page table. The function `mmu_translate_abs` formalises this behaviour:

```
mmu_translate_abs va ≡ do {
  (m, rt, a, iset) ← read (MEM, TTBR0, ASID, incon_set);
  if va ∈ iset then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk a m rt va
      in if is_fault entry then raise PAGE_FAULT
         else return (va_to_pa va (the entry))
}
```

Note that the function `pt_walk` gives us end-to-end address translation, eliminating the PDC and simplifying the model. We can do so because we translate through the page table directly.

For the abstract memory write `mmu_write_abs`, we must figure out which new addresses might have become inconsistent for the two-stage TLB. For keeping track of such inconsistencies stemming from two different page table walks, we introduce a *less or equal* relation for the type `pt_walk_typ` as:

```
_ ≼ _ :: "pt_walk_typ ⇒ pt_walk_typ ⇒ bool"
walk ≼ walk′ ≡
walk = Fault ∨ walk = walk′ ∨
(∃pde. walk = Partial_Walk pde ∧ (∃te. walk′ = Full_Walk te pde))
```

In the above relation, the first two disjuncts imply that a `Fault` walk is smaller then a `Partial_Walk` and a `Full_Walk`. The third disjunct states that a `walk` is *less* than another `walk′` if `walk` is a `Partial_Walk` with a page directory entry `pde`, and `walk′` is a `Full_Walk` with the same `pde` and a TLB entry `te`. Using this *less or equal* relation, we can define a page table comparison `pt_comp` function as:

```
pt_comp :: "(vaddr ⇒ pt_walk_typ) ⇒ (vaddr ⇒ pt_walk_typ) ⇒ vaddr set"
pt_comp pt pt′ ≡ {va | ¬ pt va ≼ pt′ va}
```

For any given two page table walk functions `pt` and `pt′` of type `"vaddr ⇒ pt_walk_typ"`, this comparison gives us the set of unmapped and remapped virtual addresses, as well as the virtual addresses that result in the same end-to-end translation but have different page directory entries. The latter is important to catch inconsistencies that are only observable in the PDC in the concrete MMU model of Sect. 4.

Using the `pt_comp` function, the abstract memory write `mmu_write_abs` then is:

```
mmu_write_abs (val, va, sz) = do {
  (m, rt, a, iset, gset) ← read (MEM, TTBR0, ASID, incon_set, glb_set);
  pa ← mmu_translate_abs va;
  when_no_exc do {
    mem_write (val, pa, sz);
    m′ ← read MEM;
    update_iset (iset ∪ pt_comp (ptd_walk a m rt) (ptd_walk a m′ rt));
    update_gset (gset ∪ glb_vaddrs a m′ rt)
  }
}
```

glb_vaddrs a m rt ≡ $\bigcup_{e \in \texttt{glb\_entries (ran (pt\_walk a m rt))}}$ range_of e

The first step in the memory write is to resolve the virtual address va. On a successful translation, we (a) write to the physical memory, (b) compare the two-stage page table walks (ptd_walk) to figure out the potential TLB-inconsistencies, and (c) saturate the glb_set with the potentially new global addresses as a result of the memory write, using a function glb_vaddrs.

The global set reload is necessary to soundly model the inconsistencies while switching the ASID register as we will see later in this section. For the function glb_vaddrs presented above, we use the end-to-end page table walk function (pt_walk) instead of the two-stage ptd_walk, because global entries are inherently determined by the final address translation. The functions update_iset and update_gset in mmu_write_abs update the incon_set and glb_set of the state with the given arguments.

The abstract memory read mmu_read_abs is similar to the base-level model, we only use the new mmu_translate_abs instance.

```
  mmu_read_abs (va, sz) = do { pa ← mmu_translate_abs va; mem_read (pa,
sz) }
```

5.2.2 MMU Operations

Updating the page table root register in the abstract MMU is similar to a memory write, comparing page tables before and after. We define update_TTBR0_abs as:

```
  update_TTBR0_abs r = do {
    (m, rt, a, iset, gset) ← read (MEM, TTBR0, ASID, incon_set, glb_set);
    update (λs. s(|TTBR0 := r|));
    update_iset (iset ∪ pt_comp (ptd_walk a m rt) (ptd_walk a m r));
    update_gset (gset ∪ glb_vaddrs a m r)
  }
```

We use the comparison pt_comp function for two page tables with different roots, and also reload the glb_set with the updated global virtual addresses.

We now explain the instruction for updating the ASID register, it manipulates all three components of the abstract TLB model. The function update_ASID_abs is defined as:

```
  update_ASID_abs a′ = do {
    (m, rt, a, iset, gset, snp) ←
      read (MEM, TTBR0, ASID, incon_set, glb_set, snapshot);
    let snp′ = snp(a := (iset, ptd_walk a m rt));
    update_snp snp′ ; update (λs. s(|ASID := a′|));
    let glb_iset = iset ∩ gset; snp_iset = fst (snp′ a′);
       pt_iset = pt_comp (snd (snp′ a′)) (ptd_walk a′ m rt)
    in update_iset (glb_iset ∪ snp_iset ∪ pt_iset)
  }
```

The above function includes three main steps. We first store the incon_set and page table state (ptd_walk) of the active ASID to the snapshot. Next we update the ASID register to the given ASID `a′`. Finally for calculating the `incon_set` for the new ASID `a′`, we (a) find the globally inconsistent addresses by intersecting `incon_set` and `glb_set`, (b) retrieve the stored inconsistent addresses under ASID `a′` from the `snapshot`, and (c) compare the stored page table state of ASID `a′` with the active page table using `pt_comp`. The sequence of these steps is important: we update the `snapshot` for the previous ASID *before* updating the ASID register and calculating the `incon_set`, since the new ASID `a′` could be equal to the active ASID.

We now explain the flush operations for our abstract model.

```
flush_abs f = do {
  (m, rt, a, iset, gset, snp) ←
    read (MEM, TTBR0, ASID, incon_set, glb_set, snapshot);
  case f of FlushTLB ⇒ upd_abs ∅ (glb_vaddrs a m rt) (λa. (∅, λv. Fault))
  | FlushVR vs ⇒
      upd_abs (iset - vs) (gset - vs ∪ glb_vaddrs a m rt) (fl_snp snp vs)
  | FlushASID a′ ⇒
      if a′ = a then update_iset (iset ∩ gset)
      else update_snp (snp(a′ := (∅, λv. Fault)))
  | FlushAVR a′ vs ⇒
      if a′ = a then update_iset (iset - (vs - gset))
      else let iset = fst (snp a′); pt = snd (snp a′)
          in update_snp (fl_asid_snp snp pt iset a′ vs)
}
```

```
fl_snp snp vs ≡
λa. (fst (snp a) - vs, λv. if v ∈ vs then Fault else snd (snp a) v)
```

```
fl_asid_snp snp pt is a′ vs ≡
λa. if a = a′ then (is - vs, λv. if v ∈ vs then Fault else pt v) else snp a
```

The `flush_abs` instruction simply removes the relevant virtual addresses from the `incon_set` and `glb_set`, and unmaps them from the relevant `snapshot`. Unmapping an address from the `snapshot` means to make its respective result a `Fault`, since flush instructions operate on both PDC and TLB. After flushing virtual addresses, we trivially saturate the `glb_set` to make sure it stays synchronised with memory. The function `upd_abs` updates the `incon_set`, `glb_set` and `snapshot` with the given arguments, the function `fl_snp` flushes the given snapshot for the given set of virtual addresses (i.e. reducing the stored `incon_set` and unmapping the stored page table state), and the function `fl_asid_snp` flushes the snapshot of the given ASID for the given set of virtual addresses.

5.2.3 Refinement Theorems

We now present the refinement between the saturated and abstract MMU models. The refinement relation provides the connection between the abstract (`incon_set`, `glb_set` and `snapshot`) and the saturated TLB and PDC. Again, the states `s` and `t` differ only in the TLB/PDC part:

```
tlb_rel_abs s t ≡
let tlb = fst (TLBs s); pdc = snd (TLBs s); a = ASID s; snp = snapshot t
in truncate s = truncate t ∧ saturated s ∧ incons s ⊆ incon_set t ∧
  globals s ⊆ global_set t ∧
  (∀a′ v. a′ ≠ a ⟶
            lookup (nglb_entries tlb) a′ v ≤ tlb_lookup_from snp a′ v ∧
            lookup (nglb_entries pdc) a′ v ≤ pdc_lookup_from snp a′ v)
```

The function `incons` models the inconsistent addresses under the *active* ASID in the saturated state `s`; the function `globals` models the global addresses:

```
incons s ≡
let tlb = fst (TLBs s); pdc = snd (TLBs s); a = ASID s; m = MEM s; rt = TTBR0 s
in {va | lookup tlb a va = Incon ∨ lookup pdc a va = Incon} ∪
   {va | (∃te. lookup tlb a va = Hit te ∧ is_fault (pt_walk a m rt va)) ∨
         (∃pe. lookup pdc a va = Hit pe) ∧ is_fault (pdc_walk a m rt va)}
```

```
globals s ≡
let tlb = fst (TLBs s); pdc = snd (TLBs s)
in (⋃e∈glb_entries tlb range_of e) ∪ (⋃e∈glb_entries pdc range_of e)
```

The subset relation between the `incons` addresses of `s` and the `incon_set` of `t` says that `t` covers all inconsistencies for the active ASID, hence guarantees safe executions. Similarly the `glb_set` of `t` conservatively approximates the `globals` addresses of state `s`. The last conjunct in `tlb_rel_abs` covers the *inactive* ASIDs: we assert that the snapshot covers everything the saturated TLB and PDC may remember from that ASID. We decode the `snapshot` into a two-stage TLB lookup as:

```
tlb_lookup_from snp a va ≡
let iset = fst (snp a); pt = snd (snp a)
in if va ∈ iset then Incon
   else case pt va of Full_Walk te pe ⇒ if asid_of te = None then Miss else Hit te
        | _ ⇒ Miss
```

```
pdc_lookup_from snp a va ≡
let iset = fst (snp a); pt = snd (snp a)
in if va ∈ iset then Incon
   else case pt va of Fault ⇒ Miss
        | Partial_Walk pe ⇒ if asid_of pe = None then Miss else Hit pe
        | Full_Walk te pe ⇒
            if asid_of te = None ∧ asid_of pe = None then Miss else Hit pe
```

The function `tlb_lookup_from` produces a TLB lookup from a snapshot `snp`. It results in `Incon` if the address `va` is in the inconsistent set of `snp` for the ASID `a`, otherwise `Fault` and global addresses are encoded to `Miss`, and an the ASID-specific entries to `Hit`. The function `pdc_lookup_from` does the same for a PDC lookup, with non-global `Partial_Walks` and non-global `Full_Walks` resulting in `Hit`.

For the theorem below, we again combine the operations `mmu_translate_abs`, `mmu_write_abs` and `mmu_read_abs` into the abstract TLB memory interface `mem_op_abs`, and the evaluation of `update_TTBR0_abs`, `update_ASID_abs` and `flush_abs` into `mmu_op_abs`.

Theorem 3 *The saturated and abstract MMU models preserve the refinement relation* `tlb_rel_abs`

$$\frac{\begin{array}{c}\texttt{mem\_op\_sat f s = (res, s')} \\ \texttt{mem\_op\_abs f t = (res', t')} \quad \texttt{consistent\_abs f t} \quad \texttt{tlb\_rel\_abs s t}\end{array}}{\texttt{res' = res} \wedge \texttt{tlb\_rel\_abs s' t'}}$$

$$\frac{\texttt{mmu\_op\_sat f s = ((), s')} \quad \texttt{mmu\_op\_abs f t = ((), t')} \quad \texttt{tlb\_rel\_abs s t}}{\texttt{tlb\_rel\_abs s' t'}}$$

where `consistent_abs` *ensures that the memory operation is for a consistent virtual address, i.e. the virtual address is not an element of the* `incon_set` *of state* `t`.

Proof We first explain how the refinement holds for address translation. According to the refinement relation `tlb_rel_abs`, the `incon_set` tracks the inconsistent virtual addresses for the active ASID in the saturated TLB. We are therefore in the `else` branch of `mmu_translate_abs`, and in either `Hit` or `Miss` branch of `mmu_translate_sat`. In both these cases, the results must agree because `saturated` and `tlb_rel_abs` imply that the `Hit` and `Miss` results represent precisely the walks we perform in `mmu_translate_abs`.

Memory reads preserve the refinement relation straightforwardly after address translation. Memory writes are interesting: we validate that the `pt_comp` function correctly tracks the resultant inconsistencies in the saturated TLB, and verify that the `glb_set` reload of the abstract model is sound. The last conjunct `tlb_rel_abs` is trivially true for memory writes, since all operations are under the active ASID. Refinement for updating the page table root register is similar to that of memory writes, as the `pt_comp` comparison is inherently the same. The flush refinement is mostly set reasoning to conclude that the relation is preserved after the respective `incon_set`, `glb_set` and `snapshot` updates.

The refinement proof for `update_ASID` is interesting: overall we establish that after updating the ASID register (a)the `incon_set` correctly models the inconsistencies of the saturated model, (b) the `glb_set` preserves the subset relation, and (c) the `snapshot` provides the lookup order for all inactive ASIDs.

We then look separately at global and non-global entries. Global entries are already covered by the `glb_set` and `incon_set`, and explicitly excluded from snapshots. This means `pt_comp` can deal with ASID-specific entries only. The correctness argument for `pt_comp` in the snapshot is then analogous to memory write. □

With this we conclude our abstract MMU model and its refinement with the saturated MMU.

5.3 Joining the Refinement Levels

We now join the refinement levels of Fig. 3 to show the soundness of our abstract model with respect to the base model. The refinement relation `tlb_rel` is:

```
tlb_rel r t ≡ ∃s. tlb_rel_sat r s ∧ tlb_rel_abs s t
```

where the states `r`, `s` and `t` are for the nondeterministic, saturated, and abstract TLB respectively. The functions `tlb_rel_sat` and `tlb_rel_abs` are the ones from Sects. 5.1 and 5.2.

Theorem 4 *Refinement between nondeterministic and abstract MMU model*

$$\frac{\texttt{mem\_op f r = (res, r')} \quad \texttt{mem\_op\_abs f t = (res', t')} \quad \texttt{consistent\_abs f t} \quad \texttt{tlb\_rel r t}}{\texttt{res = res'} \wedge \texttt{tlb\_rel r' t'}}$$

$$\frac{\texttt{mmu\_op f r = ((), r')} \quad \texttt{mmu\_op\_abs f t = ((), t')} \quad \texttt{tlb\_rel r t}}{\texttt{tlb\_rel r' t'}}$$

Proof By case analysis on the function `f` and using the respective refinement theorems, and observing that `consistent_abs` implies consistency for the lower TLB levels. □

```
datatype aexp =
    Const val
  | UnOp (val ⇒ val) aexp
  | BinOp (val ⇒ val ⇒ val) aexp aexp
  | HeapLookup aexp

datatype bexp =
    BConst bool
  | BComp (val ⇒ val ⇒ bool) aexp aexp
  | BBinOp (bool ⇒ bool ⇒ bool) bexp bexp
  | BNot bexp

datatype mode_t = Kernel | User
```

```
datatype com =
    SKIP
  | aexp := aexp
  | com ;; com
  | IF bexp THEN com ELSE com
  | WHILE bexp DO com
  | Flush flush_type
  | UpdateRoot aexp
  | UpdateASID asid
  | SetMode mode_t

type_synonym val = 32 word
```

Fig. 5 Syntax of the Heap based WHILE Language

Taken together, the refinement chain presented in this section means that a program logic on top of this model only has to keep track of inconsistent two-stage TLB addresses, and that TLB-consistent addresses can only be made inconsistent with changes to the page table, ASID, or TTBR0. TLB invalidation can be selective and can be deferred until we can no longer prove from other sources that we only access consistent mappings. In essence, the refinement hides low-level hardware TLB reasoning and provides a much simpler interface to the proof engineer.

6 Program Logic

This section presents a program logic for reasoning in the presence of cached address translation. We define the syntax of a simple Turing-complete heap language with TLB management primitives, and show the rules of program logic.

6.1 Syntax and Program State

Figure 5 shows the Isabelle data types for the abstract syntax of the language. Control structures are the standard SKIP, IF, WHILE and assignment, where assignment expects the left-hand side to evaluate to a heap address. In addition, we have specific privileged commands for flushing the TLB, updating the current page table root, the current ASID, and the processor mode. For simplicity, there are no local variables in this language, only the global heap. We identify values and pointers and admit arbitrary HOL functions for comparison, binary, and unary arithmetic expressions.

We now describe the program state for our example language and therefore also for our program logic. In the previous section we have developed an abstract and sound MMU model that keeps track of the set of TLB-inconsistent addresses and uses direct page table access for address translation. The program logic uses (morally) the same model, but there is still a break in logic: the TLB abstraction is on a machine-level ISA model; the program logic is for a higher-level language with explicit memory access, intended for languages such as C. The bridge between the two worlds would be a compiler correctness statement that takes the TLB into account. This may initially not sound straightforward: the high-level language makes fewer memory accesses visible than the low-level machine performs. In particular, a compiler will usually implement a stack for local variables, and memory areas for global variables, as well as for the code itself. These memory accesses are under address translation and might be relevant for TLB reasoning.

Sections 7 and 8 will show that we can ignore the TLB for kernel-level code, if we can assume that these memory areas (code, stack, globals) are statically known and that the compiler will not generate additional memory accesses outside these static areas. This is a reasonable assumption—otherwise kernel code could never be sure that privileged memory areas such as memory-mapped devices are not randomly overwritten by compiler-generated accesses. We will then have to prove that we never remove or change active mappings for these areas (adding new mappings for e.g. the stack would be fine). For user-level code, we will see that the issue becomes irrelevant.

The program state `p_state` of our language model is:

```
record p_state =
  heap :: "paddr ⇀ val"
  iset :: vaddr set
  gset :: vaddr set
  pt_snpshot :: asid ⇒ vaddr set × (vaddr ⇒ pt_walk_typ)
  root :: paddr
  asid :: asid
  mode :: mode_t
```

It has the following components: (a)the heap (physical memory), (b)the set of inconsistent virtual addresses (global, and under the active ASID), (c)the set of globally mapped virtual addresses, (d)the active page table root, (e)the active ASID, (f)the page tables snapshots for inactive ASIDs, with the last known ASID-specific page tables state modulo inconsistencies, and (g)the processor mode. The first of these is for traditional heap manipulation, the rest for keeping track of the abstract TLB interface. To simplify the language, we make it operate exclusively on type `val::32 word`, with physical memory as a partial function `heap:: "paddr ⇀ val"`.

6.2 Semantic Operations

This section presents the main semantic operations of the language. They describe the effects of memory accesses and the TLB operations on the state.

We interpret the values `val` of the language as virtual addresses, which means, memory read and write first undergo address translation. Both operations are sensitive to the current `mode` of the machine, since some mappings might be accessible in kernel mode only and lead to a page fault otherwise. On the ISA level of Sect. 4 this check would be performed on the permissions of the TLB entry in `mmu_translate`. Since our case study examples reason about the machine mode, we include this check here, and build it directly into the page table lookup, which means, in the program logic our interface to this formalisation is the end-to-end function `pt_lookup`, which takes a heap, a page table root, and the current mode, and yields a partial function from virtual address to physical address.

Adding a TLB to address translation only adds a check that the virtual address is not part of the `iset`:

```
phy_ad :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇀ paddr
 phy_ad IS hp rt m vp ≡ if vp ∉ IS then pt_lookup hp rt m vp else None
```

The memory read and write functions are then simply:

```
read :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇀ val
 read IS hp rt m vp ≡
 case phy_ad IS hp rt m vp of None ⇒ None | ⌊pp⌋ ⇒ load_value hp pp
```

```
write :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇒ val ⇀ heap
write IS hp rt m vp v ≡
case phy_ad IS hp rt m vp of None ⇒ None | ⌊pp⌋ ⇒ ⌊hp(pp ↦ v)⌋
```

Both functions first perform address translation, then access the physical heap. Read returns `None` when the translation failed, write returns a new heap if successful and `None` otherwise.

The effect of a write operation extends further than the heap. If the operation has modified the active page table, we may have to add new inconsistent addresses to the TLB `iset`, and new globally mapped addresses to the `gset`. As on the ISA level, for the `iset` reload, we compare the page table before and after:

```
incon_comp a hp hp′ rt rt′ = pt_comp (ptd_walk a hp rt) (ptd_walk a hp′ rt′)
```

where `a` is the current ASID. Again, as on the ISA-level, we need to update the global set after writes. The definition is identical to `glb_vaddrs` of Sect. 5.2, but takes a different heap type.

$$\texttt{global\_vaddrs a hp rt} \equiv \bigcup_{\texttt{e}\in\texttt{glb\_entries (ran (pt\_walk a hp rt))}} \texttt{range\_of e}$$

The effect of a write is then

```
heap_iset_gset_upd_s (pp ↦ v) ≡
let hp = heap s; hp′ = hp(pp ↦ v); rt = root s; a = asid s
in s⦇heap := hp′, iset := iset s ∪ incon_comp a hp hp′ rt rt,
     gset := gset s ∪ global_vaddrs a hp′ rt⦈
```

and the effect of a page table root update is

```
root_iset_gset_upd_s rt′ ≡
let rt = root s; hp = heap s; a = asid s
in s⦇root := rt′, iset := iset s ∪ incon_comp a hp hp rt rt′,
    gset := gset s ∪ global_vaddrs a hp rt′⦈
```

Changing the current ASID is again analogous to the abstract model on the ISA level, we merely access the program state instead of the ISA state.

```
new_snp s ≡
let a = asid s; hp = heap s; rt = root s
in (pt_snpshot s)(a := (iset s, λv. ptd_walk a hp rt v))
```

```
snp_incon a s ≡
let snp = new_snp s; iset = iset s; gset = gset s; hp = heap s; rt = root s;
   snp_incon = fst (snp a); glb_incon = iset ∩ gset;
   pt_incon = pt_comp (snd (snp a)) (ptd_walk a hp rt)
in snp_incon ∪ glb_incon ∪ pt_incon
```

```
asid_iset_snp_upd_s a ≡
s⦇asid := a, iset := snp_incon a s, pt_snpshot := new_snp s⦈
```

The final set of semantic effects are flush operations. The functions

`flush_iset :: flush_type ⇒ iset ⇒ asid ⇒ iset` and
`flush_gset :: flush_type ⇒ gset ⇒ asid ⇒ iset` and
`flush_snpshot :: flush_type ⇒ pt_snpshot ⇒ asid ⇒ pt_snpshot`

$$\{\!|P|\!\}\ \texttt{SKIP}\ \{\!|P|\!\} \qquad \frac{\{\!|P|\!\}\ c\ \{\!|Q|\!\} \qquad P' \longrightarrow P}{\{\!|P'|\!\}\ c\ \{\!|Q|\!\}}$$

$$\frac{\{\!|P \wedge \langle b\rangle|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|P \wedge \neg\langle b\rangle|\!\}\ c_2\ \{\!|Q|\!\}}{\{\!|P \wedge \langle\langle b\rangle\rangle|\!\}\ \texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \{\!|Q|\!\}}$$

$$\frac{\{\!|P \wedge \langle b\rangle|\!\}\ c\ \{\!|P|\!\} \qquad P \longrightarrow \langle\langle b\rangle\rangle}{\{\!|P|\!\}\ \texttt{WHILE}\ b\ \texttt{DO}\ c\ \{\!|P \wedge \neg\langle b\rangle|\!\}} \qquad \frac{\{\!|P|\!\}\ c_1\ \{\!|Q|\!\} \qquad \{\!|Q|\!\}\ c_2\ \{\!|R|\!\}}{\{\!|P|\!\}\ c_1;;\ c_2\ \{\!|R|\!\}}$$

Fig. 6 Hoare Logic Rules for Standard Commands

simply remove the relevant entries from the `iset`, `gset` and set them to `Fault` in the `pt_snpshot` depending on the specific flush instruction. The effect of the flush instruction on the state is:

```
    iset_gset_snp_upd_s f ≡
    let is = iset s; gs = gset s; snp = pt_snpshot s; a = asid s;
hp = heap s;
       rt = root s
    in s(|iset := flush_iset f is gs a, gset := flush_gset f gs a hp rt,
        pt_snpshot := flush_snpshot f snp a|)
```

6.3 Logic Rules

With the syntax and the semantic operations of the previous sections it is straightforward to define an operational semantics for the language. We briefly summarise the salient points before we focus on the rules of the program logic.

The semantics of arithmetic and Boolean expressions, $[\![A]\!]$ `s` and $[\![B]\!]_b$ `s`, are partial functions from program `state` to `val` and `bool`, respectively. While the rest is standard, `HeapLookup` goes through virtual memory:

```
    ⟦HeapLookup vp⟧ s =
    (case ⟦vp⟧ s of None ⇒ None
     | ⌊v⌋ ⇒ read (iset s) (heap s) (root s) (mode s) (Addr v))
```

For commands, we write `(c, s) ⇒ s'` for *command* `c` *executed in state* `s` *terminates in state* `s'`, where `s'` is of type `state option` with `None` indicating failure. More details about the semantics can be found at [27].

Our Hoare triples are partial for termination, but demand absence of failure.

$$\{\!|P|\!\}\ c\ \{\!|Q|\!\} \equiv \forall s\ s'.\ (c,\ s) \Rightarrow s' \wedge P\ s \longrightarrow (\exists r.\ s' = \lfloor r\rfloor \wedge Q\ r)$$

Figures 6 and 7 show the rules of the program logic. Their soundness derives directly from the operational semantics. Figure 6 summarises the rules for traditional commands such as `SKIP`, `WHILE`, etc. and Fig. 7 gives the rules for the commands that interact with the TLB. We note that the traditional rules are completely standard, as intended. We write $\langle\langle b\rangle\rangle$ `s` to denote that "$[\![b]\!]_b$ `s` $\neq$ `None`": the precondition in the `IF` and `WHILE` rules must be strong enough for failure free evaluation of `b`. The rules in Fig. 7 are in weakest-precondition form. They have a generic postcondition `P` and the weakest precondition that will establish `P`. We will now explain them.

The assignment rule requires that the expressions `l` and `r` evaluate without failure. The assignment succeeds if the virtual address `vp` is *consistent* in the current state (`vp` $\notin$ `iset s`) and `vp` is *mapped* (`Addr vp` $\hookrightarrow_s$ `pp`), where

```
{|λs. [[l]] s = ⌊vp⌋ ∧ [[r]] s = ⌊v⌋ ∧ vp ∉ iset s ∧ Addr vp ↪ₛ pp ∧
     P (heap_iset_gset_updₛ (pp ↦ v))|}
l ::= r {|P|}
{|λs. mode s = Kernel ∧ [[rte]] s = ⌊rt⌋ ∧ P root_iset_gset_updₛ Addr rt|}
UpdateRoot rte {|P|}
{|λs. mode s = Kernel ∧
     P s(|asid := a, iset := snp_incon a s, pt_snpshot := new_snp s|)|}
UpdateASID a {|P|}
{|λs. mode s = Kernel ∧ P iset_gset_snp_updₛ f|} Flush f {|P|}
{|λs. mode s = Kernel ∧ P (s(|mode := flg|))|} SetMode flg {|P|}
```

Fig. 7 Hoare logic rules for commands with TLB effects

`vp ↪ₛ pp = (phy_ad (iset s) (heap s) (root s) (mode s) vp = ⌊pp⌋)`

The effect of the assignment is the heap, `iset` and `gset` update `heap_iset_gset_upd` (from Sect. 6.2). The rule for the command `UpdateRoot`, only available in kernel mode, updates the current page table root to the value of the expression `rte` and the state by `root_iset_gset_upd`. The `UpdateASID` command, also only available in kernel mode, sets the new ASID `a`, increases the `iset` using `snp_incon`, and records a page table snapshot for the old ASID using `new_snp`. Finally, `Flush` is the instruction that the makes the `iset` smaller, and removes mappings in the snapshots of inactive ASIDs, using `iset_gset_snp_upd`.

7 Safe Set

This section introduces a reduction theorem that restricts and simplifies the assignment rule, which is the most frequent reasoning step in any usual program. The general assignment rule reasons about a) consistency of the target address in the current state b) valid address translation, and c) potential update of the `iset` and the `gset`.

The rule explicitly mentions page table walks, which means the proof engineer has to discharge page table obligations even if the memory write has nothing to do with page tables. This is not what systems programmers do. They instead establish invariants under which most of the code can be reasoned about without awareness of the TLB or page tables.

Given a TLB-consistent set of virtual addresses, this set can only become unsafe to write to when we change one of the page table mappings that translate the addresses in this set. If none of these are contained in the set, any write to the set is safe, even if it may change other mappings and increase the TLB `iset`. To formalise this notion, we re-use another function from Kolanski's page table interface [15]: `ptable_trace`. It takes a heap, a root, and a virtual address `va`, and returns the set of physical addresses visited in the page table walk for `va`. Memory writes outside the `ptable_trace` for `va` will not change the outcome of the walk for `va`. Generalising this notion to a set of virtual addresses, we define

`ptrace_set V s = (⋃v∈V ptable_trace (heap s) (root s) v)`

where `f ‘ V` applies `f` to all elements of the set `V`. The `ptrace_set V` gives us the set of physical addresses that encode the translation for the virtual addresses in `V`. We can now define what a *safe set* is:

`safe_set V s ≡ ∀va∈V. va ∈ 𝒞 s ∧ (∃p. va ↪ₛ p ∧ p ∉ ptrace_set V s)`

where `𝒞 s ≡ {va | va ∉ iset s}`. In words, a set `V` is a safe set in state `s` iff all addresses `va ∈ V` are consistent in the current state, if they map to a physical address `p`, and if that address is not part of the page table encoding for any of the addresses in `V`.

Our first observation is that once a set V is a safe set, assignments in V can no longer make it unsafe, and the safe set property will remain invariant:

Theorem 5 *Any write to the safe set will preserve the safe set. Formally:*

```
⦃λs. safe_set V s ∧ (∃vp v. ⟦lval⟧ s = ⌊vp⌋ ∧ ⟦rval⟧ s = ⌊v⌋ ∧ Addr vp ∈ V)⦄
lval ::= rval ⦃λs. safe_set V s⦄
```

Proof We apply the weakest-precondition rule for assignment and reason that a write to a mapped virtual address `vp` from the set `V` does not resolve to the page table trace of set `V`, and therefore will not change any page table entries for the set `V`. Hence none of the addresses in `V` will be added to the inconsistent set in the `incon_comp` update, and if they were consistent before, they will be consistent afterwards. While we might have changed other mappings, the trace of the mappings for `V` has not changed, and so all conditions of `safe_set` are still satisfied. □

We know with Theorem 5 that the safe set will remain invariant, so we could now ignore the `iset` completely, but since the proof engineer might want to keep track of it for other purposes, we still record it in the rule. However, in contrast to the general assignment rule, if the post condition does not mention the TLB, now neither will the precondition.

Theorem 6 *In the assignment rule, it is sufficient to check the static safe set instead of the dynamic inconsistency set* `iset`*.*

```
⦃λs. (∃vp v. ⟦lval⟧ s = ⌊vp⌋ ∧ ⟦rval⟧ s = ⌊v⌋ ∧ Addr vp ∈ V ∧
      Q (heap_iset_gset_upd_s (the_phy_ad vp s ↦ v))) ∧ safe_set V s⦄
lval ::= rval ⦃Q⦄
```

where `the_phy_ad vp s ≡ the (pt_lookup (heap s) (root s) (mode s) (Addr vp))`

Proof Follows directly from the definition of `safe_set` and the assignment rule. □

For code that is not interested in TLB effects, i.e. outside context switching and page table manipulations, this rule enables proof engineers to reason as if no TLB was present. The majority of OS and user-level code satisfies this condition. The rule still mentions address translation, but the translation is now static within `V`, i.e. can be computed once. The reduction to checking a static set of addresses also give us justification that compilers do not introduce additional complexity into reasoning under the TLB, they merely add addresses that need to be part of this safe set, e.g. the area of virtual memory that contains code, stack, and globals.

8 Case Studies

In this section, we apply the program logic and its reduction theorems to the main scenarios where TLB effects are relevant. These are: kernel-level code without TLB or page table manipulations, standard user-level code, context-switching, and page table manipulations. Of these, page table manipulations turn out to be the least interesting, their example theorems are available in our theory files [27], while we present the rest here.

The case study uses the seL4 microkernel as inspiration to distill out code sequences for a toy kernel that manages page tables and the TLB, and prevents users from accessing these, as well as other kernel data structures, directly. It maintains a set of page tables, typically one per user, potentially shared. This setting applies to all major protected-mode OS kernels, e.g. Linux, Windows, MacOS, as well as microkernels. While simplified, the case study aims to be realistic in demonstrating popular techniques for avoiding TLB flushes, such as ASIDs, and uses a so-called kernel window to reduce page tables switches. The kernel window is a set of virtual addresses, unavailable to the user, backed by kernel mappings with permissions that make them available only in kernel mode.[2]

As is customary, the mappings for this kernel window are global and constant, and each user-level page table that the kernel maintains has a number of known kernel mapping entries which reside at the same position in the page table encoding. This gives us a ready candidate for safe-set reasoning about kernel code: all addresses in the kernel window minus the addresses that are used to encode the kernel mappings in page table data structures.

Since the aim is to show reasoning principles, not to prove correctness of a particular kernel, the examples below use two-level ARMv7 page tables with a simple concrete encoding, and a specific layout. The encoding and layout should generalise readily to larger settings. In addition to the page tables (one per user) that are stored in the kernel window, we assume the existence of one further kernel data structure: a map `root_map` from page table roots to the ASID for the user of this page table. A real OS kernel might maintain these as part of a larger data structure. We ignore the details here, and use them only to formulate basic invariants the kernel must maintain.

The main invariants we use in this example are (a) all kernel data structures reside in physical kernel memory, (b) they do not overlap, (c) page table roots are aligned to page directory size, (d) the current ASID is associated correctly with the current page table root, (e) all page tables contain the kernel mappings, (f) the kernel mappings are global and static, (g) the user mappings are always non-global, (h) no page table contains mappings that allows user mode to resolve to physical kernel memory, and (i) the mapping from page table roots to ASIDs is injective. The following two properties are true for most of the execution of the system, but are invalidated temporarily: (j) The kernel window minus the entries that encode kernel mappings is a safe set. This property only holds in kernel mode. (k) The ASID snapshots agree with the page table for that ASID/user. This property is invalidated for a specific ASID between page table manipulations and flush instructions.

Formally:

```
mmu_layout s ≡
kdata_area s ⊆ kphy_mem ∧ non_overlap (kdata s) ∧ aligned (roots s) ∧
partial_inj (root_map s) ∧ root_map s (root s) = ⌊asid s⌋ ∧ umappings s ∧
kmappings s ∧ sat_glb_set s
```

The memory area of the kernel data structures is the union of the footprint of all static data structures plus the footprint of all page tables. The memory area of a page table starting at root `rt` is the set of all addresses that can be produced by a `ptable_trace`.

```
pt_area s rt ≡ ⋃v ptable_trace (heap s) rt v
kdata s ≡ map (pt_area s) (root_log s) @ [rt_map_area]
kdata_area s ≡ ⋃ (set (kdata s))
```

The definitions of non-overlapping and aligned page table roots are:

[2] This is the technique attacked by Meltdown [17]. Since hardware manufacturers are in the process of fixing this flaw, we present the more interesting setting instead of the less complex and slower scenario with a separate kernel address space.

```
non_overlap [] = True
non_overlap (x · xs) = (x ∩ ⋃ (set xs) = ∅ ∧ non_overlap xs)
```

```
aligned rts ≡ ∀rt∈rts. addr_val rt && mask 14 = 0
```

where `mask n = (1 << n) - 1`, 2^{14} is the size of page directories, and `&&` is bit-wise and. We define partial injectivity for correctly associating ASIDs with page table roots as:

```
partial_inj f ≡ ∀x y. x ≠ y ⟶ f x ≠ f y ∨ f x = None ∧ f y = None
```

The restriction on user mappings is easily phrased with our previous address translation predicates, where `roots s = set (root_log s)`, `root_log` is a list of page table roots with `root_map s r ≠ None`, and `set` turns a list into a set.

```
umappings s ≡
∀rt∈roots s. ∀va pa. pt_lookup (heap s) rt User va = ⌊pa⌋
                     ⟶ pa ∉ kphy_mem ∧ non_glb (heap s) rt va
```

The presence of kernel mappings is more technical. We spare the reader the details of the formal page table encoding, but note that it represents a constant offset translation, such that for all virtual addresses `va` in the kernel window, we get `Addr va` $\hookrightarrow_s$ `Addr (va - offset)` for a constant `offset`, i.e. the outcome of the translation is easily described statically. This is a simple yet realistic setup, similar to what e.g. seL4 uses. Finally, the assertion `sat_glb_set` (definition omitted) in `mmu_layout` states that the global set of the abstract TLB is equal to the globally mapped virtual addresses of the active page table, which is also equal to that of all page tables in the system. Both definitions can be found online [27].

To avoid flushing the TLB, we maintain for most of the execution the additional invariant that the TLB is fully consistent for all ASIDs that we might switch to, and that for each ASID the TLB snapshot agrees with the page table that we *would* switch to for that ASID. This means, if there were page table modifications for a user we are about to switch to, we assume that the corresponding flush has already happened. Since the property is not valid for all ASIDs between page table modifications and flush, we provide a set of ASIDs as argument to exclude. We assert ASID consistency as:

```
asids_consistent S s ≡ assigned_asids_consistent S s ∧ gset_consistent s
```

```
assigned_asids_consistent S s ≡
∀r a. let is = fst (pt_snpshot s a); pt = snd (pt_snpshot s a);
          wlk = ptd_walk a (heap s) r
      in root_map s r = ⌊a⌋ ∧ a ∉ S ∪ {asid s} ⟶ is ∪ pt_comp pt wlk = ∅
```

```
gset_consistent s ≡
∀r a. let is = fst (pt_snpshot s a); pt = snd (pt_snpshot s a);
          wlk = ptd_walk a (heap s) r
      in r ∈ roots s ⟶ gset s ∩ (is ∪ pt_comp pt wlk) = ∅
```

With `assigned_asids_consistent` we ensure that for all known page tables and ASIDs, unless they are explicitly excluded or they are current ASID, the `pt_comp` between snapshot and memory as well as the corresponding set of inconsistent addresses are empty. Similarly, the assertion `gset_consistent` ensures the TLB-consistency of the globally mapped virtual addresses for all the page table present in the memory. Note that this predicate does not have exclusions.

This concludes the formalisation of the necessary kernel invariants.

8.1 User Execution

The simplest of the reduction theorems is user-level execution: when the kernel has switched to user mode, the `iset` should be empty for the current ASID, and since the user cannot perform any actions that add addresses to this set, it will remain empty. Most actions that have any effect on the `iset` are explicitly privileged, i.e. unavailable in user mode. Only assignments could possibly have an adverse effect.

The following theorem shows that they do not, and that any arbitrary assignment in user mode will preserve not only this property of the `iset`, but, almost trivially, also all kernel invariants. In that sense it is a simple demonstration of the separation that virtual memory achieves between kernel and user processes.

Theorem 7 *When the kernel invariants hold, we are in user mode, and the* `iset` *is empty, then these three conditions are preserved, and the heap is updated as expected. We assume that the address the left-hand side resolves to is mapped.*

```
⦃λs. mmu_layout s ∧ mode s = User ∧
       iset s = ∅ ∧ ⟦lval⟧ s = ⌊vp⌋ ∧ ⟦rval⟧ s = ⌊v⌋ ∧ Addr vp ↪s p⦄
lval ::= rval
⦃λs. mmu_layout s ∧ mode s = User ∧ iset s = ∅ ∧ heap s p = ⌊v⌋⦄
```

Proof See lemma `user_safe_assignment` in the theory files [27]. □

The essence of the rule above is the same as Kolanski's assignment rule [15] without TLB. The invariant part of the rule could be moved to the definition of validity and be hidden from the user completely. Like Kolanski, we still had to assume that the address `vp` is mapped, because we do not distinguish between recoverable page faults and program failure. In the settings we are interested in, we aim to avoid page faults. In a setting with dynamically mapped pages, e.g. by a page fault handler, the logic can be extended to take this conditional execution into account, for instance using an exception mechanism or a conditional jump. In that case, the condition that addresses are mapped can be dropped, and we arrive at a standard Hoare logic assignment rule.

8.2 Kernel Execution

User execution boils down to standard reasoning. We can show that kernel execution without virtual memory modifications does as well.

As mentioned in Sect. 7, the safe set for kernel execution is the entire kernel window, i.e. the virtual addresses that are mapped by the global mappings, minus the addresses of the page table entries that encode these global mappings. Since we will need to re-establish this set every time we switch to a different page table, and it is always safe to reduce the safe set, we not only remove the kernel window encoding in the *current* page table, but also that of of all *other* page tables the kernel might switch to and call this set `kernel_safe`.

Since we fixed the global mappings in `mmu_layout`, we can give a short, closed form of translation for addresses in `kernel_safe`: `k_phy_ad vp = Addr vp - offset`. With these, we can formulate a theorem for assignments in kernel mode that do not touch any of the virtual memory data structures, i.e. when the write does not take place in any of the addresses covered by `kdata`.

Theorem 8 *If the* `mmu_layout` *invariants hold, we are in kernel mode, and we are performing a write in the kernel safe set that does not touch any MMU-relevant data structures, then the* `mmu_layout` *invariants are preserved and the effect is a simple heap update with known constant address translation.*

```
⦃λs. mmu_layout s ∧ mode s = Kernel ∧ safe_set (kernel_safe s) s ∧
     asids_consistent ∅ s ∧ ⟦lval⟧ s = ⌊vp⌋ ∧ ⟦rval⟧ s = ⌊v⌋ ∧
     Addr vp ∈ kernel_safe s ∧ k_phy_ad vp ∉ kdata_area s⦄
lval ::= rval
⦃λs. mmu_layout s ∧ mode s = Kernel ∧ safe_set (kernel_safe s) s ∧
     asids_consistent ∅ s ∧ heap s (k_phy_ad vp) = ⌊v⌋⦄
```

Proof See lemma `kernel_safe_assignemnt` in the theory files [27]. □

This lemma covers kernel code that is uninteresting for the purposes of the MMU and TLB, which is the majority of code in a normal kernel. The Isabelle theories [27] also contain examples for page table modifications. The main difference to this theorem is that, while the write still happens in the safe set, and the safe set is preserved, there are now inconsistent addresses that need to be flushed before we return to user mode. These could be for the active page table, but also for an inactive page table, where the need for flushing is observed in the `asids_consistent` invariant.

8.3 Context Switch

We have so far shown reduction theorems for simpler reasoning when nothing interesting happens to the TLB. This section is the opposite: context switching. There are many ways for the OS to implement context switching—our example shows one where we change to a new address space, i.e. a new page table and ASID, without flushing the TLB, establishing the conditions of Theorem 7 for user-level reasoning. Switching page table roots without flushing is non-trivial, and the ARM architecture manual [3, Chapter B3.10] even gives a specific sequence of instructions to achieve this. The manual uses this sequence, because speculative execution might otherwise contaminate the new ASID with mappings from the old page table, i.e. the TLB might still contain entries from the previous user. Theorem 9 below shows that our model is conservative for speculative execution, but precise enough so we can reason about this sequence and see why it is safe.

The recommended sequence switches to a new user-level page table and ASID by using a reserved ASID (in this case 0). It first switches to this reserved ASID, then sets the new page table root, then switches to the ASID for that root, before it switches to user mode.

Theorem 9 *The context switch sequence to a new ASID* `a` *and new page table root* `r` *preserves the* `mmu_layout` *and ASID snapshot consistency invariants and establishes the conditions for user-level reasoning, provided that the TLB has no inconsistent addresses at this point, that the reserved ASID 0 is not used for any user page table, and that that* `r` *is a known page table associated with ASID* `a`.

```
⦃λs. mmu_layout s ∧ asids_consistent ∅ s ∧ mode s = Kernel ∧
     iset s = ∅ ∧ 0 ∉ ran (root_map s) ∧ root_map s (Addr r) = ⌊a⌋⦄
UpdateASID 0;; UpdateRoot (Const r);; UpdateASID a;; SetMode User
⦃λs. mmu_layout s ∧ iset s = ∅ ∧ mode s = User ∧ asids_consistent ∅ s⦄
```

Proof See lemma `context_switch_invariants` in the theory files [27]. □

For compiler correctness, we would additionally need to know that ASID 0 does not have inconsistent entries for the code and data areas of the kernel, which is maintained if ASID 0 is used only in the way above. To make this more explicit, we could add a static set to the program logic for code and data that must always be consistency, and the condition `asids_consistent` would maintain that at least the global kernel mappings are consistent in ASID 0.

This concludes the case study examples for our logic. We have seen that we can reason about user code, kernel code, and context switching code, each at their appropriate level of abstraction.

8.4 Application to seL4

We have not yet applied the reasoning framework presented here to the verification of the seL4 microkernel. Below are the steps that one would have to undertake to do so. The functional correctness proof of seL4 is itself a stack of multiple refinement layers.

1. Choose one or more refinement layers to apply the model to.
2. Integrate virtual memory translation into that layer and derive from existing invariants that kernel memory behaves like plain memory.
3. Implement the abstract TLB in the seL4 machine interface.
4. Force the current program logic of the chosen seL4 refinement layer to provide failure on memory accesses according to the model shown here.
5. Formulate and prove the necessary invariants to show absence of these failures, based on the TLB invariants shown in the case study presented here.

In more detail, the first step in this chain is the one with most freedom and likely the largest impact on the effort of the integration of TLB behaviour into the seL4 model. At the current state of investigation, it looks like the best layer to introduce TLB behaviour is not on the machine or even C code level, but in the abstract specification of seL4. This is because this layer still models all relevant memory accesses, and at the same time is the simplest one to reason about, providing the richest set of invariants and properties of the behaviour of the kernel. In terms of refinement soundness, the abstraction level does not matter, because when all layers are composed, failure neither refines nor is refined by non-failure. The greatest fidelity would be achieved at the machine code level, since that would additionally include any effects the compiler might introduce. To transport TLB behaviour to this level, it may still be easiest to prove absence of TLB failures on the abstract level and encode their absence by using refinement assertions on the more concrete layers. These are assertions that can be assumed when reasoning purely on that level or below, but must be proved during refinement.

The second step, integration of virtual memory, is largely orthogonal to TLB behaviour, but is a prerequisite for using it. We estimate that this will cause the largest effort, about 5 person months, on the currently 1-million-line proof base of seL4, since it affects the semantics of every memory access. To show that the memory model essentially remains the same as before would require either the application of existing page table invariants to memory accesses that previously needed no further preconditions or would require a separate refinement layer that provides the simpler memory model to the rest of the kernel in the style of CertiKOS. The latter is complex in seL4 since establishing the invariants necessary for correct global mappings is interwoven with authority management, which is one of the more advanced parts of seL4. Overall, this task is not hard, but technically involved. The necessary safety invariants are already proved, but the model and proof would have to be rearranged.

Step 3, implementing the abstract TLB state in the seL4 machine interface, should be straightforward.

Step 4, adding TLB failure semantics to memory accesses, should also be straightforward, given the thin formal interface for accessing kernel memory that the rest of the proof uses. It will simply lead to failure in more cases.

Step 5, proving absence of these failures, will require the addition of invariants that mention the TLB state, similar to the ones presented in this paper. Since changes to this state are few and local in seL4, the additional proof effort should be low. We estimate about 1 person month for this part.

9 Summary

We have presented a program logic for reasoning about low-level OS code in the presence of cached address translation, with a multi-stage TLB, ASIDs, and global entries. The model and case study use the ARMv7 architecture, but our interface to page table encodings is generic and should apply to all architectures with conventional multi-level page tables. The details of TLB maintenance may differ between architectures, i.e. Intel x86 does not require an explicit TLB flush on context switch, but the ideas of the model should again transfer readily.

The model can also capture the effect of problematic hardware, such as the recent Meltdown attack [17] which exploits the fact that permission bits of TLB entries are not checked during speculative execution on some platforms, and uses a cache side channel to thereby make kernel-only TLB mappings readable to user space. To conservatively formalise the effect of this attack, one could change the model to ignore read restrictions in TLB entries. A system that can be proved safe under that conservative model, will then be safe under Meltdown.

We currently do not treat locked (pinned) TLB entries—the main effect would be explicitly allowing inconsistency between the TLB and the page table, with the TLB taking preference. Our logic does not address concurrency aspects—they are orthogonal. In a multi-core setting, each core has its own TLB which reads from global memory. Modifying a page table that is active on another core is almost never safe, unless the change merely adds new mappings or the change happens in the same safe set style presented here, where the execution on all cores must adhere to the intersection of all safe sets.

Weak memory and caches do have an interaction point with the TLB, because page table walks are subject to both and caches can be either virtually or physically indexed. We expect our safe set reasoning to transfer directly, requiring cache flushes and/or barrier instructions in addition to TLB flushes. We leave a cache formalisation for future work.

The strength of the model and logic is its simplicity, which took multiple iterations to achieve, finding a balance between abstraction soundness, not too complex reasoning, and not too much conservatism for allowing optimisations and idioms used in real OS code, resulting in a program logic that feels familiar to proof engineers. The logic allows us to prove reduction theorems that mirror the informal reasoning OS engineers perform when they write kernel code. It also allows us to drop into a simpler setting when we reason about code that does not affect virtual memory mappings. In these cases, we only need to show that memory accesses are within a set of safe addresses. Our work shows that reasoning in the presence of a TLB does not need to be significantly more onerous than without.

References

1. Achermann, R., Humbel, L., Cock, D., Roscoe, T.: Physical addressing on real hardware in Isabelle/HOL. In: Proceedings of the 9th International Conference on Interactive Theorem Proving, pp. 1–19 (2018)
2. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB virtualization implemented in C. In: Proceedings of Verified Software: Theories, Tools and Experiments 2012, vol. 7152 of Lecture Notes in Computer Science, Philadelphia, pp. 209–224, PA, USA (Jan 2012)
3. ARM Ltd.: ARM Architecture Reference Manual, ARM v7-A and ARM v7-R, April 2008. ARM DDI 0406B
4. ARM Ltd.: ARM Cortex-A15 MPCore Processor Technical Reference Manual, June 2013. ARM DDI 0438I
5. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Cache-leakage resilient OS isolation in an idealized model of virtualization. In: Proceedings of the 25th IEEE Computer Security Foundations Symposium, pp. 186–197 (2012)
6. Daum, M., Billing, N., Klein, G.: Concerned with the unprivileged: user programs in kernel refinement. Form. Asp. Comput. **26**(6), 1205–1229 (2014)
7. Feiertag, R.J., Neumann, P.G.: The foundations of a provably secure operating system (PSOS). In: AFIPS Conference Proceedings. 1979 National Computer Conference, pp. 329–334, New York, NY, USA, June 1979
8. Fox, A., Myreen, M.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Proceedings of the 1st International Conference on Interactive Theorem Proving, volume 6172 of Lecture Notes in Computer Science, pp. 243–258, Edinburgh, UK (July 2010)
9. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X. (Newman), Weng, S.-C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 595–608 (2015)
10. Gu, R., Shao, Z., Chen, H., Wu, X., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: USENIX Symposium on Operating Systems Design and Implementation, pp. 653–669, Savannah, GA, USA (Nov 2016)
11. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2:1–2:70 (2014)
12. 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: ACM Symposium on Operating Systems Principles, pp. 207–220, Big Sky, MT, USA (Oct 2009)
13. Kolanski, R.: A logic for virtual memory. In: Systems Software Verification, pp. 61–77, Sydney, Australia (July 2008)
14. Kolanski, R.: Verification of Programs in Virtual Memory Using Separation Logic. PhD thesis, UNSW, Sydney, Australia, (July 2011). Available from publications page at http://ts.data61.csiro.au/
15. Kolanski, R., Klein, G.: Types, maps and separation logic. In: International Conference on Theorem Proving in Higher Order Logics, pp. 276–292, Munich, Germany (Aug 2009)
16. Kovalev, M.: TLB Virtualization in the Context of Hypervisor Verification. PhD thesis, Saarland University, Saarbrücken, Germany (2013)
17. Lipp, M., Schwartz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: reading kernel memory from user space. In: USENIX Security Symposium, Baltimore, MD, USA (Aug 2018)
18. Lutsyk, P.: Correctness of Multi-core Processors with Operating System Support. PhD thesis, Saarland University, Saarbrücken, Germany (2018)
19. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in Higher-Order Logic. In: Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics, volume 1479 of Lecture Notes in Computer Science, pp. 349–366, Canberra, Australia (Sept 1998)
20. Nemati, H., Dam, M., Guanciale, R., Do, V., Vahidi, A.: Trustworthy memory isolation of Linux on embedded devices. In: Trust and Trustworthy Computing, pp. 125–142 (Aug 2015)
21. Nemati, H., Guanciale, R., Dam, M.: Trustworthy virtualization of the ARMv7 memory subsystem. In: Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science, vol. 8939 of Lecture Notes in Computer Science, pp. 578–589, Pec pod Sněžkou, Czech Republic (Jan 2015)
22. Neumann, P.G., Boyer, R.S., Feiertag, R.J., Levitt, K.N., Robinson, L.: A Provably Secure Operating System: The System, Its Applications, and Proofs, 2nd Edn. Technical Report CSL-116, Computer Science Laboratory, SRI International, Menlo Park, CA, USA (May 1980)
23. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, vol. 2283 of Lecture Notes in Computer Science (2002)

24. Slind, K., Norrish, M.: A brief overview of HOL4. In: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, pp. 28–32 (Feb 2008)
25. Syeda, H.T., Klein, G.: Reasoning about translation lookaside buffers. In: Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, vol. 46 of EPiC Series in Computing, pp. 490–508 (2017)
26. Syeda, H.T., Klein, G.: Program verification in the presence of cached address translation. In: Proceedings of the 9th International Conference on Interactive Theorem Proving, volume 10895 of Lecture Notes in Computer Science, pp. 542–559, Oxford, UK (July 2018)
27. Syeda, H.T., Klein, G.: Isabelle/HOL program logic for cached address translation. https://github.com/SEL4PROJ/tlb/tree/jar19 (April 2019)