Reasoning about Translation Lookaside Buffers

Syeda Hira Taqdees and Gerwin Klein

Data61, CSIRO, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
{Hira.Syeda,Gerwin.Klein}@data61.csiro.au

Abstract

The main security mechanism for enforcing memory isolation in operating systems is provided by page tables. The hardware-implemented Translation Lookaside Buffer (TLB) caches these, and therefore the TLB and its consistency with memory are security critical for OS kernels, including formally verified kernels such as seL4. If performance is paramount, this consistency can be subtle to achieve; yet, all major formally verified kernels currently leave the TLB as an assumption.

In this paper, we present a formal model of the Memory Management Unit (MMU) for the ARM architecture which includes the TLB, its maintenance operations, and its derived properties. We integrate this specification into the Cambridge ARM model. We derive sufficient conditions for TLB consistency, and we abstract away the functional details of the MMU for simpler reasoning about executions in the presence of cached address translation, including complete and partial walks.

1 Introduction

In this paper we construct a formal model of an ARM-style memory management unit (MMU), consisting of multi-level page tables and translation lookaside buffers (TLBs) for complete and partial page table walks in the interactive proof assistant Isabelle/HOL [15]. We integrate this model with the extensive, well-validated Cambridge instruction set architecture model for ARM [5], and use it as the basis for reasoning about executions in the presence of a TLB and page tables, without the complexity direct naive reasoning about the model would entail.

The motivation for developing a low-level memory model for program verification is that TLBs are caches for page tables, and operating systems use page tables as the main mechanism for enforcing memory isolation. Page tables encode the address translation from virtual to physical addresses. In most widely deployed architectures they are hardware-defined data structures that reside in main memory, typically as two- to four-level sparse lookup tables. Without further help, this central mechanism is slow: main memory is already significantly slower than the CPU, and traversing a page table can cost up to five memory accesses. The TLB caches such lookups, and significantly reduces the number of such memory accesses. It is therefore the TLB that is the main security mechanism the OS relies on. This is also true for verified OS kernels such as seL4 [7] and CertiKOS [6], which do reason about page table structures and their content, but currently ignore the TLB, assuming it is used correctly. What could go wrong? When a page table structure is changed by the OS, changing the translation for a set of virtual addresses, the TLB may still contain the old translation and provide incorrect results, violating the isolation the OS is trying to achieve. Consistency can be re-established by the OS using TLB maintenance operations to *invalidate* the corresponding TLB entry or the entire TLB, forcing it to reload its contents from the page table in main memory. TLB invalidation is expensive, and OS developers work hard to delay the operation and to make it as specific as possible, using additional TLB features such as address space identifiers (ASIDs)

to only invalidate specific sets of entries. Getting this right is subtle, and deserves support by the hardware model used to reason about the program.

In order to verify programs that run in a virtual memory environment, we develop a model of an ARMv7-style MMU including TLB and page tables, and integrate it with the Cambridge ARM model [5]. This establishes the ground truth of how the hardware behaves in the presence of a TLB. We chose ARM, because we aim to eventually integrate this method with the existing seL4 proofs [7] on ARM. To demonstrate the idea of the logic and reasoning method, we model the features of the TLB that make reasoning complex, but do not yet provide full fidelity of all modes and flags present in the ARMv7 architecture. We then extend this model to also cover partially cached page table walks as they occur in the x86 and ARMv7-A architectures.

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. The main contribution of this paper is to show how we can reduce this complexity using data refinement and arrive at a model 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.

After presenting related work in Sect. 2, we introduce Isabelle/HOL notation in Sect. 3. We describe our formal model of the MMU, the TLB, and its maintenance operations for the ARMv7 architecture, as well as the integration with the Cambridge ARM model in Sect. 4. Sect. 5 presents the series of step-wise data refinements that each simplify reasoning. We then extend this framework in Sect. 6 to the ARMv7-A architecture which provides a dedicated cache for partial page table walks, and show how it reduces to almost the same abstract model.

2 Related Work

As a cache, 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, most OS kernel verification work so far has left correct TLB behaviour as an assumption. This includes the OS kernel verification work in seL4 [7,8] and CertiKOS [6], which both do reason about page table structures, but omit the TLB. Similarly, Daum *et al.* [4] reason about user-level programs on top of seL4, including page tables, but not about the TLB.

Kolanski *et al.* [9–11] develop an extension of separation logic to formally reason about page tables, direct physical memory access, virtual memory access, and shared memory in the Isabelle/HOL theorem prover. However, they stop short of the TLB and do not address TLB caching, consistency and invalidation.

Nemati *et al.* [14] show the design, implementation and verification of a direct paging mechanism in a virtualization platform for ARMv7-A in the HOL4 theorem prover. They model the state parameters of the MMU, such as page table walks, but do not reason about the TLB or the effect of its maintenance operations.

Kovalev [12] and Alkassar *et al.* [1] do provide a TLB model, in particular a model of the Intel x64 TLB including selected maintenance operations and partial walks. Kovalev [12] states a reduction theorem for page table walks in ASID 0 for a specific hypervisor setup. However, while the result of the abstraction is used in mechanised reasoning about a hypervisor, the reduction theorem itself is not mechanised in a theorem prover. It is also aimed specifically at the correctness of TLB virtualisation in this hypervisor, not at reasoning about programs under TLBs in general.

Barthe *et al.* [3] present an abstract TLB model including TLB flushes and invariants for enforcing isolation between guest operating systems. Our model provides a similarly general TLB abstraction, but grounds it by refinement proof in a detailed operational model.

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 `range f` is the set of values returned by function `f`, i.e. `range f = {y | ∃x. f x = y}`. 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`.

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 [5] 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 a pure functional model of computation with side effects. 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 ⇒ ('s, 'a) state_monad
return a ≡ λs. (a, s)

bind :: ('s, 'a) state_monad ⇒ ('a ⇒ ('s, 'b) state_monad) ⇒ ('s, 'b) state_monad
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 occasionally write `bind f g` as `f ≫= g` and use the `do` syntax for longer computations.

```
f ≫= g ≡ do { x ← f; g x }
```

For fetching and updating a particular parameter from the state, the Cambridge ARM model uses the `read_state` and `update_state` functions (sometimes also called `gets` and `puts`):

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

We abbreviate multiple `read_state` calls into tuple notation, e.g. `(a,b) ← read_state (f,g)`.

4 A Formal MMU Model for ARM-style architectures

In this section, we present an operational model of ARMv7-style address translation including the TLB in Isabelle/HOL. We will later extend this model with an ARMv7-A-style intermediate translation cache in Sect. 6. 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 TLBs 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; c.f. [2, Chapter B3]) and a two-level page table structure. The location of the root of the page table structure in main memory is determined by a hardware register, the translation table root register TTBR0.

The hardware memory management unit (MMU) of ARMv7 consists of a TLB that caches entries and the machinery required for resolving address translations using the page table from main memory when needed. For page table operations, we reuse Kolanski's existing ARM page table model [11], and integrate it with the TLB formalisation that we build up in this section to form a model of the MMU.

The ARM architecture manual [2] describes the 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 directly on the architecture manual: we specify TLB lookup, TLB reloading, as well as maintenance operations for invalidating (evicting) outdated entries either by a tag called address space identifier (ASID), or by address in the current ASID, or by address globally for all ASIDs. Together with Kolanski's ARM page table model, this will then form the basis for specifying the semantics of the memory reads and writes that we eventually want to reason about.

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 modelling the addresses of an ARMv7-style machine, 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.

4.1 Formal Model of the TLB

We now present the operational model of the TLB itself and its maintenance operations. The next section will then develop this into a full MMU model and integrate it with memory reads and writes in the Cambridge ARM semantics.

The state of a TLB is straightforward: it is merely a set of TLB entries, where a TLB entry consists of an ASID tag, a virtual base address, a physical base address, and potentially a set of 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. To keep the presentation small, we show only two of these in the paper, one with 20 bit base addresses for small pages and one with 12 bit for sections. Formally:

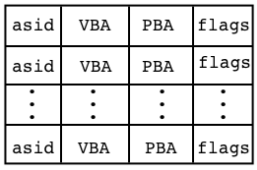

Figure 1: An Abstraction of TLB

```
type_synonym tlb = tlb_entry set

datatype tlb_entry   =   EntrySmall asid (20 word) (20 word option) flags
                     |   EntrySection asid (12 word) (12 word option) flags
```

where the type `asid` is an abbreviation for `8 word`.

To accommodate our later refinements of this model in Sect. 5, we generalised the physical base address in this definition from `20` or `12 word` to `20` or `12 word option`. In the TLB base model, these will always be entries with `Some`.

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

```
datatype lookup_type = Miss | Incon | Hit tlb_entry
```

These results are: either there is no corresponding entry and the TLB needs to be refilled (`Miss`), or there is more than one matching entry and the TLB is inconsistent (`Incon`), or there is exactly one correct result (`Hit`).

We say a TLB entry *matches* a pair of ASID `a` and virtual address `va` when the entry has the same ASID `a` and the top bits of `va` equal the virtual base address of the entry. Let `entry_range e` be the set of addresses matched by TLB entry `e`, and `asid_of e` the ASID of `e`, then we can define the `lookup` operation as

```
entry_set :: tlb ⇒ asid ⇒ 32 word ⇒ tlb_entry set
entry_set t a va = {e ∈ t | va ∈ entry_range e ∧ a = asid_of e}

lookup :: tlb_entry set ⇒ 8 word ⇒ 32 word ⇒ lookup_type
lookup t a va ≡
  let S = entry_set t a va
  in if S = ∅ then Miss
     else if ∃x. S = {x} then Hit (the_elem S) else Incon
```

where `the_elem {x} = x`.

Any result `Hit e` for a given `vaddr va` can be translated directly into a `paddr pa` by replacing the high bits of `va` with the 12-bit or 20-bit physical base address stored in `e`.

The TLB reload operation simply adds entries to the set, i.e. there is nothing specific to formalise. The TLB maintenance operations remove entries from the set:

```
selective_invalidation :: tlb ⇒ asid ⇒ 32 word ⇒ tlb
selective_invalidation t a va = t - entry_set t a va

asid_invalidation :: tlb ⇒ asid ⇒ tlb
asid_invalidation t a ≡ t - {e ∈ t | asid_of e = a}

va_invalidation :: tlb ⇒ 32 word ⇒ tlb
va_invalidation t va ≡ t - {e ∈ t | va ∈ entry_range e}
```

The operation `selective_invalidation` selectively removes the entry or entries covering the given virtual address and ASID. This may do nothing if there is no such entry, may remove exactly one entry, or all (inconsistent) entries that match. Similarly, `asid_invalidation` and `va_invalidation` remove all entries that match a particular ASID (ignoring the address) or all entries that match a particular virtual address (ignoring ASIDs), respectively.

These operational definitions have the expected declarative properties. For instance:

Lemma 1. *All invalidation operations produce subsets of the original TLB.*

```
selective_invalidation t a va ⊆ t
asid_invalidation t a ⊆ t
va_invalidation t va ⊆ t
```

Proof. By unfolding definitions. □

We can also rely on the fact that they remove inconsistent entries, and therefore are guaranteed to produce the safe `Miss` result for the relevant queries:

Lemma 2. *All invalidation instructions produce* `Miss` *for the corresponding* `lookup`.

```
lookup (selective_invalidation t a va) a va = Miss
lookup (asid_invalidation t a) a va = Miss
lookup (va_sel_invalidation t va) a va = Miss
```

Proof. By unfolding definitions. □

It is harder to characterise which entries are guaranteed to remain in the TLB, because each entry covers a whole set of virtual addresses, depending on its size. However, since the TLB is free to evict entries at any point, we cannot rely on the presence of specific entries anyway. One basic property does hold in Isabelle:

Lemma 3. *ASID invalidation affects only the ASID that is being invalidated.*

$$\mathtt{a}_1 \neq \mathtt{a}_2 \wedge \mathtt{lookup\ t\ a}_2\ \mathtt{va = Hit\ e} \Longrightarrow \mathtt{lookup\ (asid\_invalidation\ t\ a}_1\mathtt{)\ a}_2\ \mathtt{va = Hit\ e}$$

Proof. Also by unfolding definitions. □

This covers the base model of the TLB itself. We have defined its state and the basic TLB operations.

4.2 From TLB to MMU and the ARM Cambridge Semantics

Using this operational TLB model, we now develop an MMU model based on the ARM architecture manual [2] and integrate it with the instruction set architecture (ISA) semantics by Fox and Myreen [5]. 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 will keep `MEM` as the basic model for physical memory, but we 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 will then change all read and write instructions that access main memory to not access physical memory directly, but to go through the TLB and address translation first. The existing Cambridge ARM 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 can concentrate our work there.

Since our plan for Sect. 5 is to provide a series of these models that differ in the details of TLB operation, 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 two type classes 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 Cambridge ARM 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 [13] to extend `state` with the type `tlb` which will contain the TLB hardware state we modelled in the previous section.

The main interface for the rest of the model to the MMU is the address translation function, which we wrap up in its own type class `mmu`:

```
class mmu =
  fixes mmu_translate :: vaddr ⇒ 'a state_scheme ⇒ paddr × 'a state_scheme
```

where `'a state_scheme` are the potential extensions of the existing record type `state`.

In the bottom-most TLB and MMU models in our later refinement chain, this function has the following instantiation, which we explain below.

```
mmu_translate va = do {
  update_state (λs. s(|tlb := tlb s - tlb_evict s|));
  (mem, asid, ttbr0, tlb) ← read_state (MEM, ASID, TTBR0, tlb);
  case lookup tlb asid (addr_val va) of
  Miss ⇒
    let entry = pt_walk asid mem ttbr0 va
    in if is_fault entry then raise PAGE_FAULT
       else do {
             update_state (λs. s(|tlb := tlb ∪ {entry}|));
             return (va_to_pa va entry)
           }
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```

The function `mmu_translate` first evicts an underspecified set of entries from the TLB. 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 effect for the refinement theorems later is the same.

The next step in `mmu_translate` after reading out the hardware state is to do a 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 e`, we check whether the entry encodes a page fault (using `is_fault`). In the ground-truth, base-level model, this will never trigger because the hardware TLB will only be reloaded with valid page translations, but later in our refinement chain we will relax that condition. If we encounter such a page fault, we raise the corresponding page fault exception, otherwise we translate `e` 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 is `Miss`, we perform a page table walk using the function `pt_walk` starting from `TTBR0`. We omit its definition here. It is constructed from Kolanski's page table model [11] using his functions `get_pde` and `get_pte` to find the respective page table entries and to encode the result in the corresponding `tlb_entry` format with the current ASID.

If the result of the page table walk is a page fault, we raise this fault, which will cause the machine to jump to the appropriate exception handler. If the result of the walk is a particular mapping entry `e`, we perform a TLB reload by adding this entry to the TLB, and execute address translation as in the `Hit` case.

On top of class `mmu`, which specifies the translation interface, we can now build read and write instructions that use virtual instead of physical addresses. We wrap these up in their own

[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.

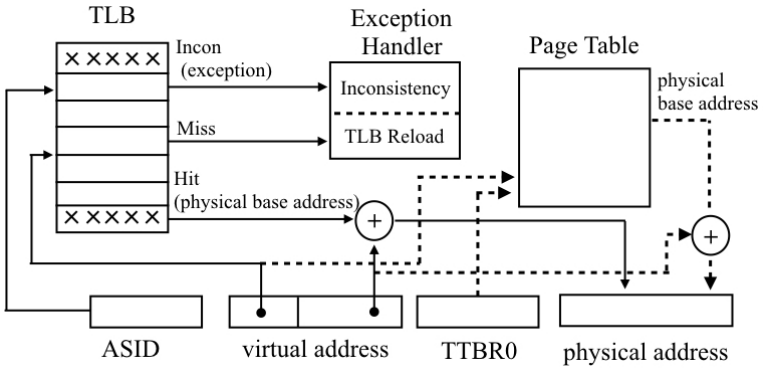

Figure 2: ARM-style Memory Management Unit

type class `mmu_op`:

```
class mmu_op = mmu +
 fixes mmu_write :: bool list × vaddr × nat ⇒ 'a state_scheme ⇒ unit × 'a state_scheme
 fixes mmu_read :: vaddr × nat ⇒ 'a state_scheme ⇒ bool list × 'a state_scheme
```

The interface for the values being read and written in the ARM model is via `bool list` instead of machine words directly, which we keep here, and the `nat` paramenter indicates how many bytes to read/write, e.g. one byte, a word, a double word, etc.

Reusing the original functions `mem_write` and `mem_read` for physical memory, the instances for the base-level TLB and MMU model are then 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_state 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.

By redirecting all other memory-related functions in the ARM model to go through this interface, we arrive at a full operational model that supports address translation and TLB.

The purpose of this paper is not to provide a fully detailed formalisation that is validated to comprehensively conform with existing hardware, but to present the main ideas on how to simplify reasoning in the presence of a TLB. Despite this focus, we have validated the model by executing a number of instructions in the theorem prover, manually checking consistency with the expected behaviour. A full formalisation would need a more extensive test suite in the spirit of Fox and Myreen [5].

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 TLB and page table lookup mechanisms, and an adjustment of the subsequent memory operations to include the address translation layer.

5 TLB 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 TLB, but the model is hard to reason about directly. From Sect. 4, we see that a TLB introduces:

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

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 non-determinism, and a correctly operated TLB 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.

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

```
consistent mem asid ttbr0 tlb va ≡
  lookup tlb asid (addr_val va) = Hit (pt_walk asid mem ttbr0 va) ∨
  lookup tlb asid (addr_val va) = Miss
```

This condition combines internal consistency (no `Incon` results permitted), with external consistency, i.e. synchronicity with the current state of the page table for this particular address.

5.1 Determinism

With this in mind, 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: `l ≤ l' ≡ l = Miss ∨ l' = l ∨ l' = Incon`, and we can prove monotonicity

Lemma 4. `t ⊆ t' ⟹ lookup t a v ≤ lookup t' a v`

Proof. By case distinction and unfolding the definitions. □

We can use this 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.

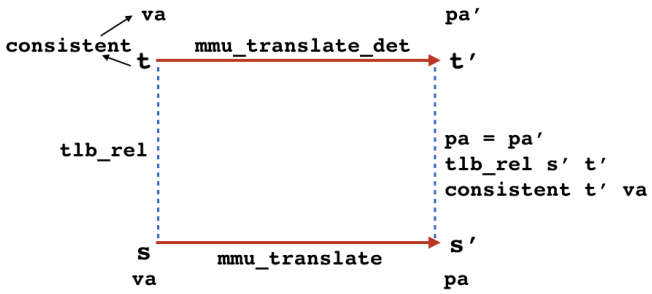

Figure 3: Data Refinement between Non-Deterministic and Deterministic MMUs

This means, we can use our observation above by noting that, instead of a TLB that non-deterministically evicts entries, we can use a TLB that *never* evicts entries, unless explicitly instructed. If we can prove a program safe with this larger TLB, it will also be safe with the smaller TLB. We can prove this fact by instantiating `mmu_translate` for a deterministic version in which the TLB does not evict entries and then proving refinement. We name this instantiation `mmu_translate_det`. We do not repeat the definition here; the only difference to `mmu_translate` from Sect. 4.2 is the missing `tlb_evict` line.

Theorem 1. *Assuming that two states* `s` *and* `t` *have the refinement relationship*

$$\texttt{tlb\_rel s t} \equiv \texttt{truncate s} = \texttt{truncate t} \wedge \texttt{tlb s} \subseteq \texttt{tlb t}$$

where the notation `truncate s` *means all fields of the extensible* `state` *record without the* `tlb` *extension. That is, the states* `s` *and* `t` *differ only in the contents of the TLB, and the TLB of* `s` *contains fewer entries. If the TLB of* `t` *is consistent w.r.t. lookups in* `va`*, then the address translation of a virtual address* `va` *performed using* `mmu_translate` *in* `s` *is the same as the one performed by* `mmu_translate_det` *in* `t`*. Moreover, the resultant final states retain the relationship* `tlb_rel` *and the TLBs remain consistent w.r.t.* `va`*. Figure 3 depicts this theorem as a diagram. Formally:*

$$\frac{\texttt{mmu\_translate va s} = \texttt{(pa, s')} \qquad \texttt{mmu\_translate\_det va t} = \texttt{(pa', t')} \qquad \texttt{consistent t va} \qquad \texttt{tlb\_rel s t}}{\texttt{pa'} = \texttt{pa} \wedge \texttt{consistent t' va} \wedge \texttt{tlb\_rel s' t'}}$$

Proof. We observe that the abstract TLB in state `t` is consistent for `va`, that is, a lookup for `va` will either produce `Miss` or `Hit`. Given the subset relationship and Lemma 4, we get that either both TLBs produce the same `Hit e`, or both walk the page table (with the same result, since the states only differ in TLB content), or that `t` produces a `Hit`, but `s` walks the page table. Since `t` is consistent for `va`, the result of the walk has to agree with the `Hit`. □

This step removes nondeterminism from the model and is sound for executions in which the larger TLB never triggers an inconsistency. The definitions of the memory write and read operations remain unchanged compared to the base model, but they now pick up the new `mmu_translate_det` instance of the `mmu` class. Since Theorem 1 says that `mmu_translate_det` and `mmu_translate` return the same results, memory write and read behaviour is trivially equal to the base model.

5.2 Invariance

As the next step, we eliminate TLB state change for memory reads. We note that the presence of an inconsistent entry is not dangerous yet, only *using* the inconsistent entry is. This means, for every memory transaction we can add to the TLB *all* entries that the current page table produces. As we have seen in the previous step, this is sound because we add more entries that are consistent with the page table, and inconsistency with older entries is not dangerous yet. This will give us a TLB that is always saturated with entries. On read operations, the state will not change, because the set of page table results before and after reading is the same. On write operations outside the page table we have the same — only on writes to the page table we will get a state change in the TLB, which is what we should expect.

Formally, we instantiate `mmu_translate` in this model such that the TLB always remains saturated i.e. whenever it accesses memory it reloads the entire page table. We name the operation `mmu_translate_sat`:

```
mmu_translate_sat va = do {
  tlb_refill;
  (tlb, asid) ← read_state (tlb, ASID);
  case lookup tlb asid (addr_val va) of
  Miss ⇒ raise IMPLEMENTATION_DEFINED
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}

tlb_refill = do {
  (asid, mem, ttbr0, tlb) ← read_state (ASID, MEM, TTBR0, tlb);
  let tlb' = tlb ∪ range (pt_walk asid mem ttbr0);
  update_state (λs. s(|tlb := tlb'|))
}
```

The call to `tlb_refill` at the beginning of `mmu_translate_sat` achieves the saturation mentioned above by adding the `range` of the `pt_walk` function to the TLB for the current state and ASID.

Note that by using the full range of the `pt_walk` function we have benefited from the `option` type in the physical base address field of TLB entries. For addresses that the page table specifies a page fault for, the physical base address will be `None`. In the model this means we have reduced external and internal inconsistency conditions to one condition: our saturated TLBs always encode the full state of the page table, and even removing a mapping from the table will now lead to (internal) TLB inconsistency, one entry showing `Some` and another `None` for the same virtual address.

In this saturated TLB `mmu_translate_sat` then performs a standard lookup. `Incon` results still lead to the same exception as before. `Miss` results cannot be triggered any more, since the TLB is saturated, and `Hit` results are the same as in the previous models.

Theorem 2. *With the refinement relation*

`tlb_rel_sat s t ≡ truncate s = truncate t ∧ tlb s ⊆ tlb t ∧ saturated t`

where `saturated t ≡ range (pt_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ tlb t`*, we get data refinement between the original* `mmu_translate` *and* `mmu_translate_sat`*:*

$$\frac{\texttt{mmu\_translate va s = (pa, s')} \qquad \texttt{mmu\_translate\_sat va t = (pa', t')} \qquad \texttt{consistent t va} \qquad \texttt{tlb\_rel\_sat s t}}{\texttt{pa' = pa} \wedge \texttt{consistent t' va} \wedge \texttt{tlb\_rel\_sat s' t'}}$$

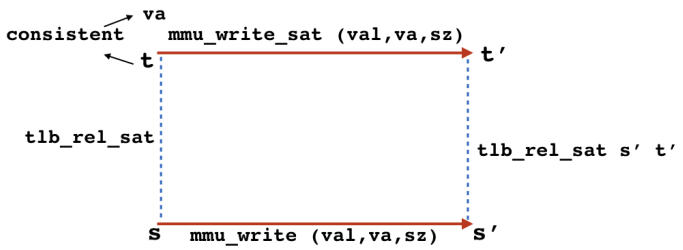

Figure 4: Data Refinement between Memory Write Functions using Non-Deterministic and Saturated MMUs

This means, we still get the same address translation results, and preserve consistency, as well as the refinement relation, including saturation.

Proof. Essentially the same argument as before, observing that entries stemming from `pt_walk` cannot make a `va`-consistent entry inconsistent. □

For this model, we do not only need to change `mmu_translate`, but also the write operation to preserve saturation. The new instantiations for saturated TLBs 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); tlb_refill }
}

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

Similarly, the TLB invalidation operations now need to include a global TLB refill to preserve saturation. They will still remove old entries that in this model would lead to internal instead of external inconsistency.

Since these operations have now changed, it is worth explicitly stating refinement for the write operation. Figure 4 shows the corresponding diagram.

Theorem 3. *Memory writes preserve the TLB refinement relation, including saturation.*

$$\frac{\texttt{mmu\_write\_sat (val, va, sz) t = ((), t')} \quad \texttt{mmu\_write (val, va, sz) s = ((), s')} \quad \texttt{tlb\_rel\_sat s t} \quad \texttt{consistent t va}}{\texttt{tlb\_rel\_sat s' t'}}$$

Proof. Follows directly from the refinement result on `mmu_translate`. □

It is important to note here that consistency of the TLB cannot be preserved in Theorem 3, since memory writes can change the page table. This means, re-establishing consistency on writes will be an obligation on the proof engineer, not an automatic invariant that is provided by the model. This mirrors the reasoning OS developers do mentally.

Consistency is established either by reasoning that the write was not to a page table, by using appropriate invalidation instructions, or by reasoning that if a page table was changed, the change was unrelated to the address that is about to be accessed.

For memory *reads*, as planned, the TLB state remains unchanged, eliminating one of the major difficulties in reasoning about the TLB.

Theorem 4. *In saturated states, memory reads do not change the TLB.*

$$\frac{\texttt{mmu\_read\_sat (va, sz) s = (val, t)} \qquad \texttt{saturated s}}{\texttt{tlb t = tlb s}}$$

Proof. By observing that memory reads do not change the state and that a saturated TLB already contains all current page table entries. □

A simple optimisation to this model would be to not update the TLB for *every* memory write, but only for writes to the current page table structure or the page table root register. This immediately produces a reduction result: if the current page table structure is not writeable, and if the execution mode is unprivileged, i.e. the page table root register cannot be changed, then we know that no memory transaction will change the saturated TLB state, and we can therefore reason about a much simpler model without TLB and with fixed address translation. This is what user-level execution expects: users should not need to worry about the presence or absence of a TLB. The following theorem encapsulates the conditions for this reduction.

Theorem 5. *Memory writes that do not change the page table content leave the saturated TLB constant, preserving consistency and saturation.*

$$\frac{\begin{array}{c}\texttt{mmu\_write\_sat (val, va, sz) s = ((), s')} \\ \forall \texttt{va. pt\_walk (ASID s) (MEM s) (TTBR0 s) va = pt\_walk (ASID s') (MEM s') (TTBR0 s') va} \\ \texttt{consistent s v} \qquad \texttt{saturated s}\end{array}}{\texttt{tlb s' = tlb s} \wedge \texttt{consistent s' v} \wedge \texttt{saturated s'}}$$

Proof. The condition that all `pt_walk` outcomes remain the same after the memory write directly implies that the `range (pt_walk asid mem ttbr0)` term in `tlb_refill` remains the same, and since the TLB is already saturated, the TLB refill has no effect. □

In summary, reasoning about the TLB has become much more tractable in this model. Inconsistency is reduced to internal inconsistency only, and non-determinism as well as unnecessary state change are removed. For a program logic on top of this model it would suffice to guarantee the absence of inconsistencies, and to treat page faults the same way a program logic for standard address translation would, e.g. as in Kolanski's work [10].

5.3 Essence

This leads us to the last refinement step, where we abstract the saturated TLBs to the extent that no actual TLB `lookup` is required: the functionality of the TLB can be captured completely by only keeping record of those virtual address/ASID pairs that are inconsistent in the TLB with the current page table. It is then enough to perform address translation using the page table only.

For this last abstraction, we extend the record type `state` not with `tlb`, but with `incon_set` of type `(asid × 32 word) set` and instantiate `mmu_translate` of type class `mmu` to:

```
mmu_translate_set va = do {
  (mem, asid, ttbr0, incon_set) ← read_state (MEM, ASID, TTBR0, incon_set);
  if (asid, addr_val va) ∈ incon_set then raise IMPLEMENTATION_DEFINED
  else let entry = pt_walk asid mem ttbr0 va
       in if is_fault entry then raise PAGE_FAULT else return (va_to_pa va entry)
}
```

Note that this address translation contains no state change at all any more, apart from potentially raising exceptions.

With this definition, we get the following theorem.

Theorem 6. *Let* `tlb_rel_abs` *denote the refinement relation*

```
tlb_rel_abs s t ≡ truncate s = truncate t ∧ asid_va_incon (tlb s) ⊆ incon_set t
```

where `asid_va_incon tlb ≡ {(asid, va) | lookup tlb asid va = Incon}` *constructs the set of TLB-inconsistent addresses in the saturated TLB. Then the functions* `mmu_translate_sat` *and* `mmu_translate_set` *preserve this relation and yield the same result. Formally:*

```
mmu_translate_sat va s = (pa, s')     mmu_translate_set va t = (pa', t')
 saturated s     (ASID t, addr_val va) ∉ incon_set t     tlb_rel_abs s t
──────────────────────────────────────────────────────────────────────
                    pa = pa' ∧ tlb_rel_abs s' t'
```

Proof. According to the refinement relation, `incon_set` tracks the inconsistent entries in the saturated TLB. We are therefore in the `else` branch of `mmu_translate_set` and in the `Hit` case of `mmu_translate_sat`. The results must agree, because `saturated` says that the `Hit` results represent precisely the walks we perform in `mmu_translate_set`. □

As in the previous step, the memory access instantiations have to change. For `mmu_write_set`, we must figure out which new addresses might have become inconsistent. We do this by comparing the page tables before and after the physical write operation: all addresses for which the result of a page table walk is different after the write operation are potentially unsafe. For `mmu_read_set`, the definition is similar to the base-level model, we only use the new `mmu_translate_set` instance.

```
mmu_write_set (val, va, sz) = do {
  (ttbr0, asid, mem, incon_set) ← read_state (TTBR0, ASID, MEM, incon_set);
  pa ← mmu_translate_set va;
  when_no_exc do {
    mem_write (val, pa, sz);
    mem' ← read_state MEM;
    let new_incon = ptable_comp asid mem mem' ttbr0;
    update_state (λs. s(|incon_set := incon_set ∪ new_incon|))
  }
}

ptable_comp asid m m' ttbr0 =
{asid} × {va | pt_walk asid m ttbr0 (Addr va) ≠ pt_walk asid m' ttbr0 (Addr va)}

mmu_read_set (va, sz) = do { pa ← mmu_translate_set va; mem_read (pa, sz) }
```

While for `mmu_translate_set` and `mmu_read_set` it is now obvious in this model that the entire state remains constant if there is no translation exception, and, with Theorem 6 also that memory reads return the correct result, the `mmu_write_set` operation is interesting enough for its own refinement theorem:

Theorem 7. *Memory writes for TLB-consistent addresses preserve the refinement relation* `tlb_rel_abs` *while the* `saturated` *invariant holds.*

```
mmu_write_set (val, va, sz) t = ((), t')     mmu_write_sat (val, va, sz) s = ((), s')
       (ASID t, addr_val va) ∉ incon_set t     saturated s     tlb_rel_abs s t
──────────────────────────────────────────────────────────────────────────────
                              tlb_rel_abs s' t'
```

Proof. First, we observe that, with Theorem 6, the TLB lookup produces the same result on each abstraction level, and therefore the two physical write operations produce the same memory state. Second, we need to establish that `incon_set` correctly tracks which entries in the saturated TLB have become inconsistent. These are the entries with those addresses for which `pt_walk` now yields a different result, which is precisely what `ptable_comp` computes. □

For unprivileged user-level code, we have already established that we can reduce to a model without TLB and with fixed address translation. For privileged OS-level code, address translation is usually fixed for the OS code itself and all locations it accesses. In this case, the TLB will always return these fixed mappings, and cannot become inconsistent since they never change. The following theorem ties this down more precisely.

Theorem 8. *Let* `SA` *be a set of consistent, safe addresses as follows.*

```
consistent_set SA s ≡ ∀va∈SA. (ASID s, addr_val va) ∉ incon_set s ∧
                                ¬ is_fault (pt_walk (ASID s) (MEM s) (TTBR0 s) va)
```

Then any memory write to one of these safe addresses will leave all safe addresses consistent in the same sense, if the write does not change their page table mappings.

$$\frac{\texttt{mmu\_write\_set (val, va, sz) s = ((), s')} \quad \texttt{va} \in \texttt{SA} \quad \texttt{consistent\_set SA s} \quad \forall \texttt{v}\in\texttt{SA. pt\_walk (ASID s) (MEM s) (TTBR0 s) v = pt\_walk (ASID s') (MEM s') (TTBR0 s') v}}{\texttt{consistent\_set SA s'}}$$

Proof. The consistency condition ensures that the TLB lookup is a page table walk without faults, and the page mapping condition ensures that the safe addresses are not part of the `ptable_comp` term in the `incon_set` update of the `mmu_write` operation. □

That means, if we prove that each OS memory access remains within a safe set of addresses and that the page table mappings for this set never change, execution is safe and does not need to reason about the TLB. For seL4 for instance, this property is already proved as part of its reasoning about page tables without the TLB.

Taken together, the proofs [16] in the refinement chain presented in this paper mean that a program logic on top of this model only has to keep track of and check for inconsistent TLB entries, and that TLB entries 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 chain in this paper hides the low-level hardware TLB reasoning and provides a much simpler interface to the proof engineer, including reduction theorems to the common cases of pure user-level execution and fixed, safe OS mappings.

6 Caching Partial Walks

Our TLB model so far always caches full page table walks. The ARMv7-A architecture adds a separate cache for partial page table walks, similar to x86. In this section, we show how our model can be extended to cover such partial walks. For 2-level page tables, this means there is one additional cache for storing page directory entries, the Page Directory Cache (PDC). Figure 5 represents the the role of the PDC in address translation. The formalisation idea remains the same: we extend the `state` record with the a type `tlb` × `pdc`, define base-level, non-deterministic address translation and memory operations, and then prove a refinement chain up towards the model of Sect. 5.3. This time, we will skip one step in the chain, and directly prove refinement between saturated and non-deterministic TLB/PDC. We then introduce one new step, in which we observe that the saturated TLB/PDC combination can be replaced by just a saturated TLB. The refinement step to the most abstract model in Sect. 5.3 then follows naturally and can be reused.

Formally, a PDC is a set of page directory entries:

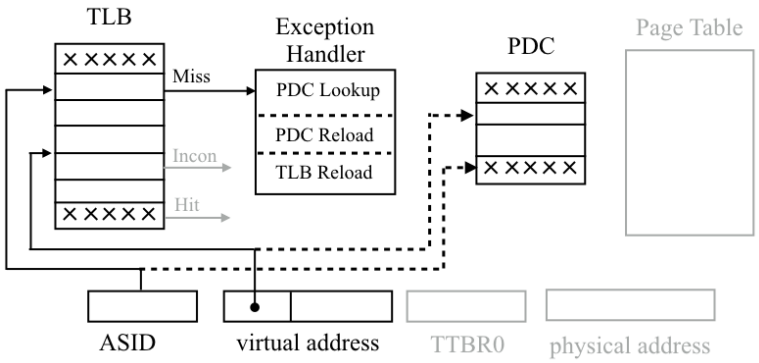

Figure 5: Intermediate Translation Table Cache in ARM-style MMU

```
type_synonym pdc = pde set
datatype pde = EntryPDE asid (12 word) (32 word option) flags
```

An `EntryPDE` provides the 32 bit physical address of the page directory entry for an ASID and a virtual address. Analogously to the TLB lookup function `lookup`, we can define a PDC lookup `lookup_pdc` (details omitted), with the return type:

```
datatype lookup_pdc_type = Miss_pde | Incon_pde | Hit_pde pde
```

For the base-level address translation function, we then instantiate `mmu_translate` as follows.

```
mmu_translate_part va = do {
  update_state (λs. s(|tlb_pdc := tlb_pdc s - tlb_evict s|));
  (asid, tlb, pdc) ← read_state (ASID, tlb_pdc);
  case lookup tlb asid (addr_val va) of Miss ⇒ pdc_lookup_reload va
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}

pdc_lookup_reload va = do {
  (asid, tlb, pdc) ← read_state (ASID, tlb_pdc);
  case lookup_pde pdc asid (addr_val va) of
  Miss_pde ⇒ tlb_pdc_reload_trans va
  | Incon_pde ⇒ raise IMPLEMENTATION_DEFINED
  | Hit_pde pde ⇒ tlb_reload_trans pde va
}
```

After evicting an underspecified set of entries from the PDC and the TLB, the function `mmu_translate_part` performs the TLB lookup. If the result is `Hit` or `Incon`, we proceed as in the old model. If the result is `Miss`, we perform a PDC lookup and potential reload instead: if `lookup_pdc` results in `Hit_pde e`, we use the function `tlb_reload_trans` to complete the partial translation for address `va` from `pde` and store the result in the TLB. If the result is `Miss_pde`, we use `tlb_pdc_reload_trans` to perform the full address translation, and reload both, PDC and TLB. If the result was `Incon_pde`, the machine raises an unrecoverable exception and halts. We omit the formal definitions of `tlb_reload_trans` and `tlb_pdc_reload_trans` here for space reasons; they are analogous to the simpler model with just one TLB.

From this, we can start the refinement chain. We abstract `mmu_translate_part` to a saturated model to achieve determinism and state-invariance for memory operations at consistent addresses. We saturate both, the PDC and the TLB, as below.

```
mmu_translate_part_sat va = do {
  pdc_tlb_refill;
  (asid, tlb, pdc) ← read_state (ASID, tlb_pdc);
  case lookup tlb asid (addr_val va) of
  Miss ⇒ raise IMPLEMENTATION_DEFINED
  | Incon ⇒ raise IMPLEMENTATION_DEFINED
  | Hit entry ⇒
      if is_fault entry then raise PAGE_FAULT
      else return (va_to_pa va entry)
}
```

```
pdc_tlb_refill = do {
  (mem, asid, ttbr0) ← read_state (MEM, ASID, TTBR0);
  let all_pdes = range (pde_walk asid mem ttbr0);
     all_tlbes = ⋃range (tlb_pde_walk asid all_pdes mem ttbr0)
  in update_state (λs. s(|tlb_pdc := tlb_pdc s ∪ (all_tlbes, all_pdes)|))
}
```

Similar to `tlb_refill`, the call to `pdc_tlb_refill` at the beginning achieves the saturation of the PDC and the TLB. We preserve the caching hierarchy by first loading all current page directory entries into the PDC using `pde_walk`, and then using this saturated PDC to load all completed address translations into the TLB with `tlb_pde_walk`. We then prove the first step in the refinement chain: this hierarchical saturation retains address translation and consistency.

Theorem 9. *With the refinement relation*

```
tlb_rel_sat' s t ≡
truncate s = truncate t ∧ tlb s ⊆ tlb t ∧ pdc s ⊆ pdc t ∧ saturated' t
```

and

```
saturated' t ≡
range (pt_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ tlb t ∧
range (pde_walk (ASID t) (MEM t) (TTBR0 t)) ⊆ pdc t
```

we get refinement between the original `mmu_translate_part` *and* `mmu_translate_part_sat`*:*

$$\frac{\texttt{mmu\_translate\_part va s = (pa, s')} \qquad \texttt{mmu\_translate\_part\_sat va t = (pa', t')} \qquad \texttt{consistent' t va} \qquad \texttt{tlb\_rel\_sat' s t}}{\texttt{pa' = pa} \wedge \texttt{consistent' t' va} \wedge \texttt{tlb\_rel\_sat' s' t'}}$$

where `consistent' t va` *ensures the absence of inconsistent entries for* `va` *in the PDC and the TLB in state* `t`*. This means, we get the same address translation results, and preserve consistency, as well as the refinement relation, including saturation.*

Proof. We observe that entries stemming from `pde_walk` and `pt_walk` cannot make a `va`-consistent entry inconsistent, and that the PDC is part of the changed definition of `consistent`. □

We can now proceed to the next refinement step and replace `mmu_translate_part_sat` with `mmu_translate_sat`, which contains no PDC, only a saturated TLB.

Theorem 10. *With the refinement relation*

$$\texttt{tlb\_rel\_sat''\ s\ t} \equiv \texttt{truncate\ s} = \texttt{truncate\ t} \wedge \texttt{tlb\ s} \subseteq \texttt{tlb\ t} \wedge \texttt{saturated'\ s} \wedge \texttt{saturated\ t}$$

we get data refinement between the original `mmu_translate_part_sat` *and* `mmu_translate_sat`*:*

$$\frac{\begin{array}{c}\texttt{mmu\_translate\_part\_sat\ va\ s} = \texttt{(pa, s')} \\ \texttt{mmu\_translate\_sat\ va\ t} = \texttt{(pa', t')} \qquad \texttt{consistent\ t\ va} \qquad \texttt{tlb\_rel\_sat''\ s\ t}\end{array}}{\texttt{pa'} = \texttt{pa} \wedge \texttt{consistent\ t'\ va} \wedge \texttt{tlb\_rel\_sat''\ s'\ t'}}$$

Again, we get the same address translation results, and preserve consistency, as well as the refinement relation, including saturation.

Proof. Observing that a saturated PDC is transparent to a saturated TLB in the same state. □

Taken together, these two refinement steps have eliminated the need to reason about the PDC, and the last step of Sect. 5.3 in the refinement chain composes directly. Note that because of the PDC, more potentially different entries are being reloaded into the TLB at write operations. Formally, the function `ptable_comp` from Sect. 5.3 now not only needs to talk about full page table walks that might have changed result, but also about partial walks on the page directory level. This is the only effect that adding the additional cache has at the most abstract level.

The conditions for safe memory read and write operations in the presence of cached partial walks are analogous to those in Sect. 5. We omit the details here, but the interested reader can find them in the full Isabelle formalisation [16].

7 Conclusions and Future Work

We have shown a formal ARM-style TLB model that we used as ground truth to develop a much simpler model for reasoning that is deterministic, and where normal memory accesses do not change further global state. It also lets us reason about cached partial walks and memory accesses that change page tables, giving us sufficient conditions for establishing safe execution. We established soundness of this model by constructing "less safe" TLB models, and then giving sufficient conditions for showing that even these less safe models lead to safe execution.

The model presented here does not have full fidelity for any specific ARM architecture version, but shows the principles to be applied for constructing such a model. If the intent is to reason on the ISA directly, a useful next step would be to lift the refinement theorems for the memory interface we have shown to the entire ISA model. This is a mostly mechanical exercise, since the refinement theorems show equality for the effect of the memory operations on the state the rest of the ARM model cares about. If the intent is to reason about higher-level languages, we have laid the groundwork for compiler correctness in the presence of a TLB and the main reduction needed for a program logic: we know we only have to keep track of and avoid TLB-inconsistent addresses. All other low-level TLB complexity can be abstracted away.

Acknowledgments We would like to thank Rafal Kolanski for discussions about and insights into the design of this TLB model. We would also like to thank the anonymous reviewer who motivated us to add a formalisation of partial page table walks.

References

[1] Eyad Alkassar, Ernie Cohen, Mikhail Kovalev, and Wolfgang J. Paul. Verification of TLB virtualization implemented in C. In *VSTTE 2012*, volume 7152 of *LNCS*, pages 209–224, Philadelphia, PA, USA, Jan 2012.

[2] ARM Ltd. *ARM Architecture Reference Manual, ARM v7-A and ARM v7-R*, Apr 2008. ARM DDI 0406B.

[3] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *25th CSF*, pages 186–197, 2012.

[4] Matthias Daum, Nelson Billing, and Gerwin Klein. Concerned with the unprivileged: User programs in kernel refinement. *Formal Aspects Comput.*, 26(6):1205–1229, Oct 2014.

[5] Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *1st ITP*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, Jul 2010.

[6] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: A certified kernel for secure cloud computing. In *2nd APSys*, 2011.

[7] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009.

[9] Rafal Kolanski. A logic for virtual memory. In *SSV*, pages 61–77, Sydney, Australia, Jul 2008.

[10] Rafal Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, UNSW, Sydney, Australia, Jul 2011. Available from publications page at `http://ts.data61.csiro.au/`.

[11] Rafal Kolanski and Gerwin Klein. Types, maps and separation logic. In *TPHOLs*, pages 276–292, Munich, Germany, Aug 2009.

[12] Mikhail Kovalev. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, Germany, 2013.

[13] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In *11th TPHOLs*, volume 1479 of *LNCS*, pages 349–366, Canberra, Australia, Sep 1998.

[14] Hamed Nemati, Roberto Guanciale, and Mads Dam. Trustworthy virtualization of the ARMv7 memory subsystem. In *41st SOFSEM*, volume 8939 of *LNCS*, pages 578–589, Pec pod Sněžkou, Czech Republic, Jan 2015.

[15] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] Syeda Hira Taqdees, Gerwin Klein, and Rafal Kolanski. Isabelle/HOL TLB model for ARMv7-A. `https://github.com/SEL4PROJ/tlb/tree/LPAR-21`, 2017.