Program Verification in the Presence of Cached Address Translation

Hira Taqdees Syeda and Gerwin Klein

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

Abstract. Operating system (OS) kernels achieve isolation between user-level processes using multi-level page tables and translation lookaside buffers (TLBs). Controlling the TLB correctly is a fundamental security property — yet all large-scale formal OS verification projects leave correct functionality of the TLB as an assumption. We present a logic for reasoning about low-level programs in the presence of TLB address translation. We extract invariants and necessary conditions for correct TLB operation that mirror the informal reasoning of OS engineers. 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 and page table manipulations.

1 Introduction

We present a program logic in the interactive proof assistant Isabelle/HOL [15] for verifying programs in the presence of an ARMv7-style memory management unit (MMU), consisting of multi-level page tables and a translation lookaside buffer (TLB) for caching page table walks. This logic builds on our earlier work [17], a machine model with a sound abstraction of the ARMv7-style TLB. While program logics for reasoning in the presence of address translation exist [11], reasoning in the presence of a TLB has so far remained hard, and is left as an assumption in all large-scale operating system (OS) kernel verification projects such as seL4 [7] and CertiKOS [6].

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Since the TLB caches address translation, each of these operations may leave the TLB out of date w.r.t. the page table in memory, and the OS kernel must flush (or *invalidate*) the TLB before that lack of synchronisation can affect program execution.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 address space identifiers

(ASIDs) to only invalidate specific sets of entries. 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.

The main contribution of this paper apart from the logic itself and its soundness is to show that it 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 development of the logic, and the case study presented in the paper have led us to significantly extend the TLB model of our previous work [17], which provided soundness for memory operations, but not for ASID maintenance and page table root switches. The previous model would have required TLB invalidation where current kernel code (correctly) does not perform any. The case study that flushed out this deficiency is inspired by the seL4 kernel [7], and systematically covers all significant interactions with the TLB. In fact, we chose to model the ARMv7 TLB, because we aim to eventually integrate this logic with the existing seL4 proofs on ARM.

The logic is generic and can easily be adapted to, for instance, the shallow embedding the seL4 specifications 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 in Coq.

After related work in Sect. 2, we introduce the Isabelle/HOL notation we use in Sect. 3. Sect. 4 presents the syntax and semantic operations of a small example language, as well as the program logic. Sect. 5 shows the main reduction theorems that simplify reasoning, and Sect. 6 concludes with the case study examples. The corresponding Isabelle/HOL theories are available online [18].

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 [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, virtual memory access, and shared memory in Isabelle/HOL. We build directly on the abstract interface to page table encodings Kolanski developed, which makes our work independent of the precise page table format the architecture uses. Kolanski's model does not include the TLB and does not address TLB caching, consistency and invalidation, which we add here.

Nemati *et al.* [14] show the design, implementation and verification of a direct paging mechanism in a virtualization platform for ARMv7-A in HOL4 [16].

Similarly to others, they model the state parameters of the MMU, such as page table walks, but not the TLB or 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, which is based on ideas similar to the ones presented here. However, while other parts of this development are mechanised, this reduction theorem is not. As we will see in Sect. 4, the restriction to one ASID makes the model too conservative for usual OS code.

Barthe *et al.* [3] present an abstract TLB model including TLB flushes and invariants for enforcing isolation between guest operating systems, but stop short of a program logic and a proof that the abstraction is sound.

We build directly on our earlier work [17] which provides a detailed operational TLB model based on the ARM architecture manual [2] integrated with the ARM instruction set architecture (ISA) semantics of Fox and Myreen [5]. Reasoning directly about this detailed model is hard, because the TLB introduces non-determinism, because global state changes even on memory reads, and because it introduces new failure modes that need to be avoided. Our earlier work provides a tower of abstractions from this model, including soundness proof. The final abstraction is similar to the ideas by Kovalev [12] and Kolanski [10], but the case study in this paper shows that making efficient use of ASIDs requires additional complexity. We have therefore extended the existing tower of abstractions and soundness proofs and arrive at the TLB model we will show in Sect. 4.

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`. `Some` has an underspecified inverse called `the`, satisfying `the ⌊x⌋ = x`.

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`. Function update is written `f(x := y)` where `f::'a ⇒ 'b`, `x::'a` and `y::'b`, `f(x ↦ y)` stands for `f (x:= Some y)`. We model the program 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 `heap s` denotes the value of the `heap` field of `s`, and `s⦇heap := id⦈` will update `heap` of `s` to be the identity function `id`.

```
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 asid = 8 word
type_synonym val = 32 word

datatype flush_type = flushTLB        | flushvarange (val set)
                    | flushASID asid | flushASIDvarange asid (val set)
```

Fig. 1. Syntax of the heap based WHILE language.

4 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, introduce the abstract TLB and memory model the language works on, and show the rules of program logic.

4.1 Syntax and Program State

Fig. 1 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. The Flush operation has a number of variants: invalidate all entries, invalidate by virtual address or by virtual address/ASID pair, and invalidate an entire ASID [2, Chapter B3].

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.

Fig. 2 illustrates the program state. It consists of the heap (physical memory), the set of inconsistent virtual addresses, the active page table root, the active ASID, the last known page table state for all inactive ASIDs (page tables snapshot), and the processor mode.

The first of these is for traditional heap manipulation, the rest for keeping track of the TLB. This state model is similar to the TLB-relevant machine state in our previous work on the ISA level [17], but it is not the same. The main idea of this previous work was that it is sufficient to keep track of the addresses on which the TLB and the in-memory page table may disagree. The logic presented in the present paper made case studies feasible and showed that this is

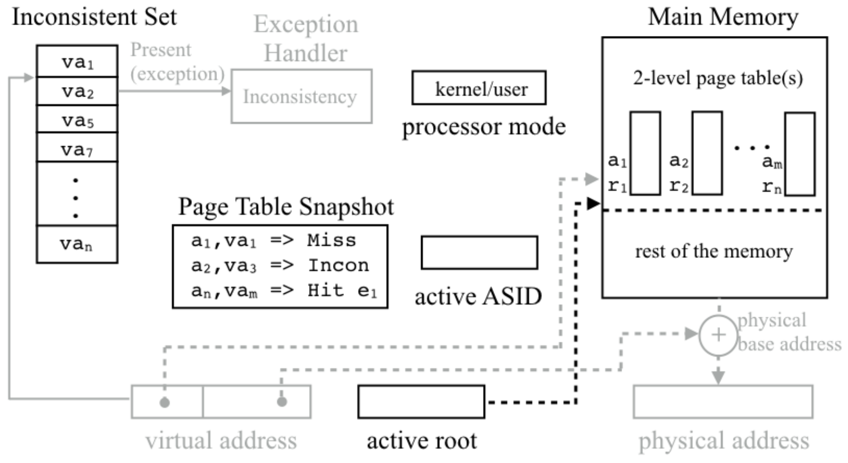

Fig. 2. Abstracted TLB Memory Model

not sufficient: to soundly model the effect of the `UpdateASID` command without requiring unnecessary flushes, this new model keeps track of 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. An ASID becomes active when the `UpdateASID` command updates the corresponding machine register. In addition, this model also relaxes which addresses become inconsistent when page tables are modified. Sect. 4.2 will provide more details.

Using the types `vaddr` and `paddr` for virtual and physical addresses from Kolanski's page table interface [11] we can declare the following:

```
datatype lookup_type = Miss | Incon | Hit tlb_entry
type_synonym iset = vaddr set
type_synonym ptable_snapshot = asid ⇒ vaddr ⇒ lookup_type
```

where it suffices for this paper to see `tlb_entry` as the result of a page table walk (see [17] for details).

With these, the record `state` has the components `heap :: paddr ⇀ val`, `iset :: iset`, `pt_snpshot :: ptable_snapshot`, `root :: paddr`, `asid :: asid`, and `mode :: mode_t`. Most of these are straightforward. The `iset` is the set of TLB-inconsistent addresses, and the snapshot is a function from ASID `a` to address `va` to `lookup_type`, where `Miss` encodes that the snapshot has no information for `va`, `Incon` encodes that `va` should not be used, and `Hit` is the result of the page table walk for `va` when `a` was last active.

4.2 Semantic Operations

This section presents the main semantic operations of the language. They describe the effects of memory accesses and the new 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. To decode page tables, we reuse Kolanski's existing ARM page table formalisation [11], extended with this additional access control behaviour. Our interface to this formalisation is the 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. With this, we can formalise address translation, read, and write under a TLB.

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 va ≡ if va ∉ IS then pt_lookup hp rt m va else None
```

The memory read and write functions are then simply:

```
read :: iset ⇒ heap ⇒ root ⇒ mode_t ⇒ vaddr ⇀ val
read IS hp rt m va ≡ phy_ad IS hp rt m va ▷ load_value hp

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

where `x ▷ g ≡ case x of None ⇒ None | ⌊y⌋ ⇒ g y`. 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 modified the active page table, we may have to add new addresses to the TLB `iset`. For this, we compare the page table before and after:

```
pt_comp wlk wlk' =
{va | ¬ is_fault (wlk va) ∧ ¬ is_fault (wlk' va) ∧ wlk va ≠ wlk' va ∨
      ¬ is_fault (wlk va) ∧ is_fault (wlk' va)}

incon_comp a hp hp' rt rt' = pt_comp (pt_walk a hp rt) (pt_walk a hp' rt')
```

where `a` is the current ASID and `pt_walk` is a version of `pt_lookup` that returns more information, including whether the walk resulted in a page fault (missing mapping). We compare the results of page table walks in a heap `hp` from a root `rt` with walks in a different, updated heap `hp'` and potentially different root `rt'`. For heap writes, the root will be the same, and for root updates, the heaps will be the same. Two scenarios might add inconsistent entries: changing an existing mapping (first disjunct), or removing an existing mapping (second disjunct). Note that a single heap write can affect multiple mappings at once, for instance when it changes the pointer to an entire page table level. It is the effect of this comparison that OS engineers reason about informally when they compute which addresses need to be flushed from the TLB. We will show examples in Sect. 6.

The effect of a write is then

```
heap_iset_update_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|)
```

and the effect of a page table root update is

```
root_iset_update_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'|)
```

For changing the current ASID, we will make use of the page table snapshots to determine which addresses become inconsistent, and we need to update the snapshot for the ASID we are switching away from.

```
lift_pt walk ≡ λva. if is_fault (walk va) then Miss else Hit (walk va)
to_incon V walk ≡ λva. if addr_val va ∈ V then Incon else walk va
snap_pt s = to_incon (ℐ𝒞 s) (lift_pt (pt_walk (asid s) (heap s) (root s)))
new_snp s = (pt_snpshot s)(asid s := snap_pt s)
```

where ℐ𝒞 `s ≡ {vp | Addr vp ∈ iset s}` and `addr_val (Addr a) = a` and `Addr` is the constructor for addresses.

Taking a snapshot is taking the `pt_walk`s in the current state, marking all unmapped entries as `Miss`, and everything in the `iset` as `Incon`, and then storing that function under the current ASID in `new_snp`.

```
snp_comp a snp walk ≡ {va | snp a va ≠ Miss ∧ snp a va ≠ Hit (walk va)}
snp_incon a s ≡ snp_comp a (new_snp s) (pt_walk a (heap s) (root s))
```

Determining the `iset` for the new ASID `a` compares the `Hit` entries in the snapshot for `a` with the current `pt_walk`. We use `new_snp s` instead of `pt_snpshot s`, because `a` could also be the current ASID. The `UpdateASID` command then executes

```
asid_pt_snpshot_update_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_snpshot :: flush_type ⇒ pt_snpshot ⇒ asid ⇒ pt_snpshot`

simply remove the relevant entries from the `iset`, and set them to `Miss` in the `pt_snpshot` respectively. The flush instruction does both simultaneously:

```
iset_pt_snpshot_update_s f ≡
let is = iset s; snp = pt_snpshot s; a = asid s
in s(|iset := flush_iset f is a, pt_snpshot := flush_snpshot f snp a|)
```

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

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

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

Fig. 3. Hoare Logic rules for standard commands.

4.3 Hoare Logic

With the syntax and the semantic operations of the previous sections it is straightforward to define an operational semantics for the language. We omit the details here and only 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 and omitted here, `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 [18].

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 3 and 4 show the rules of the program logic. Their soundness derives directly from the operational semantics. Fig. 3 summarises the rules for traditional commands such as `SKIP`, `WHILE`, etc. and Fig. 4 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. 4 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 ($vp \notin \mathcal{IC}\ s$) and `vp` is *mapped* ($\texttt{Addr vp} \hookrightarrow_s \texttt{pp}$), where

$$vp \hookrightarrow_s pp = (\texttt{phy\_ad (iset s) (heap s) (root s) (mode s) vp} = \lfloor pp\rfloor)$$

The effect of the assignment is the heap and `iset` update `heap_iset_update` we described in Sect. 4.2.

$$
\begin{array}{l}
\{\!|\lambda \mathtt{s}.\ [\![\mathtt{l}]\!]\ \mathtt{s} = \lfloor \mathtt{vp} \rfloor \wedge [\![\mathtt{r}]\!]\ \mathtt{s} = \lfloor \mathtt{v} \rfloor \wedge \mathtt{vp} \notin \mathcal{IC}\ \mathtt{s} \wedge \mathtt{Addr\ vp} \hookrightarrow_s \mathtt{pp} \wedge \\
\quad \mathtt{P\ (heap\_iset\_update}_s\ \mathtt{(pp} \mapsto \mathtt{v))}|\!\} \\
\mathtt{l\ ::=\ r}\ \{\!|\mathtt{P}|\!\}
\end{array}
$$

$$
\begin{array}{l}
\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge [\![\mathtt{rte}]\!]\ \mathtt{s} = \lfloor \mathtt{rt} \rfloor \wedge \mathtt{P\ (root\_iset\_update}_s\ \mathtt{Addr\ rt)}|\!\} \\
\mathtt{UpdateRoot\ rte}\ \{\!|\mathtt{P}|\!\}
\end{array}
$$

$$\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge \mathtt{P\ (asid\_pt\_snpshot\_update}_s\ \mathtt{a)}|\!\}\ \mathtt{UpdateASID\ a}\ \{\!|\mathtt{P}|\!\}$$

$$\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge \mathtt{P\ (iset\_pt\_snpshot\_update}_s\ \mathtt{f)}|\!\}\ \mathtt{Flush\ f}\ \{\!|\mathtt{P}|\!\}$$

$$\{\!|\lambda \mathtt{s}.\ \mathtt{mode\ s = Kernel} \wedge \mathtt{P\ (s(\!|mode := flg|\!))}|\!\}\ \mathtt{SetMode\ flg}\ \{\!|\mathtt{P}|\!\}$$

Fig. 4. Hoare Logic rules for commands with TLB effects.

The rule for the command `UpdateRoot`, only available in kernel mode, updates the current page table root to the value of the expression `rte`. The effect is modelled by `root_iset_update` defined in Sect. 4.2.

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_asid_map_update` from Sect. 4.2.

4.4 Discussion

Our previous work [17] motivated an abstract TLB model that we have extended and refined here. The program logic uses the (morally) 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.

Sect. 5 and 6 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 logic could be made slightly more precise by distinguishing between situations that must always be avoided, such as using inconsistent TLB entries, and page faults, which can be recoverable by executing a page fault handler. In kernel-level code, page faults are usually unwanted as modelled here, in user-level code they will usually be recoverable. We omit the distinction here for simplicity. Page fault handlers could for instance be modelled as exceptions in the logic.

In summary, we have so far provided a Hoare logic for reasoning about programs in the presence of cached address translation. The model as shown is specific to the ARMv7 architecture, but should generalise readily to similar architectures, since it uses an abstract interface for page table encoding. So far, reasoning is possible, and is at the right level of abstraction for code that manipulates page tables, but it is not yet convenient for code that does not interfere with virtual memory mappings or even runs in user mode.

5 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`. 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 [11]: `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

$$\texttt{ptrace\_set V s = }\bigcup\texttt{ptable\_trace (heap s) (root s) ` V}$$

where `f ‘ V` applies `f` to all elements of the set `V`, and $\bigcup$ is the union of a set of sets. 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:

$$\texttt{safe\_set V s} \equiv \forall \texttt{va}\in\texttt{V. va} \in \mathcal{C}\ \texttt{s} \wedge (\exists \texttt{p. va} \hookrightarrow_s \texttt{p} \wedge \texttt{p} \notin \texttt{ptrace\_set V s})$$

where $\mathcal{C}$ `s` $\equiv$ `{va | va` $\notin$ `iset s}`. In words, a set `V` is a safe set in state `s` iff all addresses `va` $\in$ `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 1 *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. See lemma `safe_set_preserved` in [18].

Our previous work [17] already contains a corresponding theorem for the concrete machine model. The following theorem is new. It develops the concept further into a simpler assignment rule where it is sufficient to check that the address is part of the safe set. We know with Theorem 1 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 2 *In the assignment rule, it is sufficient to check the static safe set instead of the dynamic inconsistency set $\mathcal{IC}$.*

```
⦃λs. (∃vp v. ⟦lval⟧ s = ⌊vp⌋ ∧ ⟦rval⟧ s = ⌊v⌋ ∧ Addr vp ∈ V ∧
       Q (heap_iset_update_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. See lemma `weak_pre_write` in [18].

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 global variables.

6 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, so we only summarise them, while we present the rest in more detail.

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 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. [1] It is the combination of ASID use, context switching, and flush avoidance that led us to adjust our previous model [17] for this case study.

As is customary, the mappings for this kernel window are 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) the current ASID is associated correctly with the current page table root, (d) all page tables contain the kernel mappings, (e) no page table contains mappings that allows user mode to resolve to physical kernel memory, and (f) 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: (g) The kernel window minus the entries that encode kernel mappings is a safe set. This property only holds in kernel mode. (h) 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:

[1] This is the technique attacked by Meltdown [13]. Since hardware manufacturers are promising to fix this major flaw, we present the more interesting setting instead of the less complex and slower scenario with a separate kernel address space.

```
mmu_layout s ≡
kernel_data_area s ⊆ kernel_phy_mem ∧ non_overlapping (kernel_data s) ∧
root_map s (root s) = ⌊asid s⌋ ∧ kernel_mappings s ∧
user_mappings s ∧ partial_inj (root_map s)
```

where we define partial injectivity 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.

```
user_mappings s ≡
∀rt∈roots s.
   ∀va pa. pt_lookup (heap s) rt User va = ⌊pa⌋ ⟶ pa ∉ kernel_phy_mem
```

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.

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 ≡ ⋃ptable_trace (heap s) rt ‘ UNIV
kernel_data s ≡ map (pt_area s) (root_log s) @ [rt_map_area]
kernel_data_area s ≡ ⋃set (kernel_data s)
```

The definition of non-overlapping is:

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

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. If this set is empty, we will omit the argument in the notation.

```
asids_consistent S s ≡
∀r a. root_map s r = ⌊a⌋ ∧ a ∉ S ∪ {asid s} ⟶
      (∀v. pt_snpshot s a v = Miss ∨
           pt_snpshot s a v = Hit (pt_walk a (heap s) r v))
```

This concludes the formalisation of the necessary kernel invariants.

6.1 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 adds 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 3 *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.*

$$\{\!|\lambda \texttt{s. mmu\_layout s} \wedge \texttt{mode s = User} \wedge$$
$$\mathcal{IC}\ \texttt{s} = \emptyset \wedge [\![\texttt{lval}]\!]\ \texttt{s} = \lfloor\texttt{vp}\rfloor \wedge [\![\texttt{rval}]\!]\ \texttt{s} = \lfloor\texttt{v}\rfloor \wedge \texttt{Addr vp} \hookrightarrow_s \texttt{p}|\!\}$$
$$\texttt{lval ::= rval}$$
$$\{\!|\lambda \texttt{s. mmu\_layout s} \wedge \texttt{mode s = User} \wedge \mathcal{IC}\ \texttt{s} = \emptyset \wedge \texttt{heap s p} = \lfloor\texttt{v}\rfloor|\!\}$$

Proof. See lemma `user_safe_assignment` in [18].

The essence of the rule above is the same as Kolanski's assignment rule [11] 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.

6.2 Kernel Execution

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

As mentioned in Sect. 5, 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 `kernel_data`.

Theorem 4 *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 ∉ kernel_data_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 [18].

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 [18] 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.

6.3 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 3 for user-level reasoning.

Switching page table roots without flushing is non-trivial, and the ARM architecture manual [2, 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 5 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. A real kernel would at this point also restore registers, which we omit.

Theorem 5 *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 ∧
     𝓘𝓒 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 ∧ 𝓘𝓒 s = ∅ ∧ mode s = User ∧ asids_consistent s⦄
```

Proof. See lemma `context_switch_invariants` in [18].

Our previous ISA-level model [17] without ASID page table snapshots was not strong enough to admit this theorem without flushing the TLB. In particular, the fact that the TLB does not contain entries for the ASID we are switching to that are inconsistent with the current page table at that point would either be lost (making it unsound) or over-approximated (requiring a flush).

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, 'uninteresting' kernel code, and kernel code that manipulates paging structures, each at their appropriate level of abstraction.

7 Summary

We have presented a program logic for reasoning about low-level OS code in the presence of cached address translation.

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 defective hardware, such as the recent Meltdown attack [13] 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, should then be safe under Meltdown.

We currently do not treat global locked (pinned) TLB entries, and the TLB in this version of the logic does not cache partial page table walks (as in e.g.

ARMv7-A). Our previous work does cover partial walks — the main influence on the model is that the update of the `iset` becomes slightly more conservative. Pinned TLB entries would have the effect of 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. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB virtualization implemented in C. In: VSTTE 2012. LNCS, vol. 7152, pp. 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. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Cache-leakage resilient OS isolation in an idealized model of virtualization. In: 25th CSF. pp. 186–197 (2012)
4. Daum, M., Billing, N., Klein, G.: Concerned with the unprivileged: User programs in kernel refinement. Formal Aspects Comput. **26**(6), 1205–1229 (Oct 2014)
5. Fox, A., Myreen, M.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: 1st ITP. LNCS, vol. 6172, pp. 243–258. Edinburgh, UK (Jul 2010)
6. Gu, L., Vaynberg, A., Ford, B., Shao, Z., Costanzo, D.: CertiKOS: A certified kernel for secure cloud computing. In: 2nd APSys (2011)
7. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. Trans. Comp. Syst. **32**(1), 2:1–2:70 (Feb 2014)

8. 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: SOSP. pp. 207–220. Big Sky, MT, USA (Oct 2009)
9. Kolanski, R.: A logic for virtual memory. In: SSV. pp. 61–77. Sydney, Australia (Jul 2008)
10. Kolanski, R.: Verification of Programs in Virtual Memory Using Separation Logic. Ph.D. thesis, UNSW, Sydney, Australia (Jul 2011), available from publications page at `http://ts.data61.csiro.au/`
11. Kolanski, R., Klein, G.: Types, maps and separation logic. In: TPHOLs. pp. 276–292. Munich, Germany (Aug 2009)
12. Kovalev, M.: TLB Virtualization in the Context of Hypervisor Verification. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2013)
13. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown. ArXiv e-prints **1801.01207** (Jan 2018)
14. Nemati, H., Guanciale, R., Dam, M.: Trustworthy virtualization of the ARMv7 memory subsystem. In: 41st SOFSEM. LNCS, vol. 8939, pp. 578–589. Pec pod Sněžkou, Czech Republic (Jan 2015)
15. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283 (2002)
16. Slind, K., Norrish, M.: A brief overview of HOL4. In: 21st TPHOLs. pp. 28–32 (Feb 2008)
17. Syeda, H.T., Klein, G.: Reasoning about translation lookaside buffers. In: 21st LPAR. EPiC Series in Computing, vol. 46, pp. 490–508 (2017)
18. Syeda, H.T., Klein, G., Kolanski, R.: Isabelle/HOL program logic for cached address translation. `https://github.com/SEL4PROJ/tlb/tree/ITP18`, DOI 10.5281/zenodo.1246933 (Jan 2018)