Puss In Boots: on formalising Arm's Virtual Memory System Architecture (extended version)

Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, Nikos Nikoleris

Puss In Boots: on formalising Arm's Virtual Memory System Architecture (extended version)

Jade Alglave*†, Richard Grisenthwaite* Artem Khyzha*, Luc Maranget‡ and Nikos Nikoleris*
*ARM Ltd, Cambridge, UK
†University College of London, UK
‡Inria Paris, France

***Abstract*—We present our formalisation of Arm's Virtual Memory System Architecture (VMSA). This work has been developed with, and ratified by, Arm and its partners, and is now part of the Arm Architecture Reference Manual (Arm ARM). As part of this work, we uncovered subtleties in the definition of a feature of the VMSA called Enhanced Translation Synchronisation (ETS), which led Arm to deprecate ETS and replace it with a stronger feature called ETS2. Finally, we present our experimental validation methodology, which required extending KVM-unit-tests, a test harness for the Kernel Virtual Machine (KVM). The corresponding patches are currently being integrated into KVM. We used this infrastructure to run around 1300 VMSA litmus tests on a variety of Arm machines, thereby validating our model w.r.t. existing hardware. Our testing also uncovered infidelities to the definition of a feature called Translation Table Hardware Management, which led Arm to relax its architecture to accommodate those cases.**

I. Introduction

Virtual memory gives an abstract view of physical memory, which affords software more memory than might be physically available. The mapping from virtual addresses (VAs) to physical addresses (PAs) is the responsibility of the Memory Management Unit (MMU), and resides in a collection of page table entries, which Arm calls Translation Table Descriptors (TTDs). Translation Lookaside Buffers (TLBs) cache TTDs for faster address translation, but require synchronisation for ensuring coherency when updating TTDs, absence of which may lead to unintended out-of-order execution scenarios. To enable avoiding undesirable scenarios, we capture the ordering implications of address translation by extending Arm's application-level memory model [21]. Arm provides an informal record of intent in its Virtual Memory System Architecture (VMSA) [30, D8], and our work provides a formal basis for reasoning on the matter. In this paper, we hope to give the reader enough intuition for studying the model in depth.

We use the `cat` language [12] to write our formalisation, since the Arm application-level model [21] is written in `cat`. This also enables us to use the herd7 tool [9], a simulator for litmus tests under memory models written in `cat`. A litmus test is a small concurrent program with a specific initial state and a question about reachable final states. The herd7 tool answers the questions by determining final states reachable under a given `cat` model. We extended herd7 to run VMSA litmus tests. Thus, our VMSA model is an executable artefact, allowing user interaction to develop intuition.

We validated our model in two ways. First, by discussing and refining it with Arm and its partners, up to ratification and integration in the Arm Architecture Reference Manual (Arm ARM) [30]. Second, by testing it extensively on hardware. We extended the litmus7 testing tool [11], distributed [9] alongside herd7, to run VMSA litmus tests. This required extending KVM-unit-tests [2], a test harness for the Kernel Virtual Machine (KVM) [1], to run our tests as virtual machines.

a) Outline: Section II introduces Arm's application-level memory model. Section III gives an initial view of our VMSA extension: the semantics of memory accesses extended with TTD accesses and Faults as a part of address translation, and behaviours the model forbids in absence of synchronisation.

The VMSA extension has three main aspects, which we expose in turn. Section IV discusses the lack of ordering due to TLBs and means of restoring ordering when needed, as illustrated by a CopyOnWrite example from Linux, a behaviour validated by our model. Section V discusses the ordering implications of the Translation Table Hardware Management (TTHM) feature, which transfers software's responsibility to modify TTD permissions to hardware. Section VI discusses the stronger ordering guarantees of the Enhanced Translation Synchronisation (ETS) feature [31, pp. A2-78,79], which enables lightweight synchronisation for making an invalid TTD valid. We expose the subtleties we uncovered in the definition of ETS, which led Arm to deprecate it and replace with a new feature called ETS2 [30, pp. A2-84,85].

Section VII details our validation processes, both the work towards ratification by Arm and the experimental work. We discuss the extensions we made to the litmus7 tool and KVM-unit-tests. Our testing uncovered infidelities to a TTHM principle, which led Arm to relax the architecture to accommodate those cases. Section VIII discusses related works.

b) Companion material: We release a number of companion artefacts: the VMSA `cat` file itself, our litmus tests, the extensions to the herdtools distribution necessary to handle VMSA, the KVM-unit-tests patches necessary to run VMSA litmus tests on hardware and our testing logs [6].

Finally, an English transliteration of the VMSA `cat` file appears in Section B2.3 of the Arm ARM [30].

II. Arm's application-level memory model

The Arm memory model determines which concurrent executions are allowed by the Arm architecture. In this section,

```
AArch64 MP
{
0:X1=x; 1:X1=x;
0:X3=y; 1:X3=y;
x=0; y=0;
}
P0            | P1            ;
MOV W0,#1     | LDR W4,[X3]   ;
STR W0,[X1]   | LDR W5,[X1]   ;
MOV W2,#1     |               ;
STR W2,[X3]   |               ;
exists (1:X4=1 /\ 1:X5=0)
```

Fig. 1. A message passing litmus test—Allowed

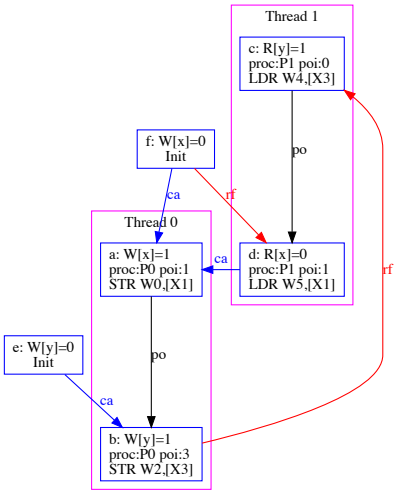


Fig. 2. An execution of the MP test given in Fig. 1

we give an overview of the application-level model, which focuses on concurrent executions of programs agnostic of virtual memory and not manipulating TTDs. We extend the scope in the subsequent section.

a) Litmus tests: We introduce the syntax of litmus tests and Arm's application-level memory model using a message passing example (MP) in Fig. 1.

The first section of a litmus test is the initial state in curly brackets, where the MP test lets registers X1 on P0 and P1 point to a location x, written 0:X1=x and 1:X1=x respectively, and registers X3 on P0 and P1 point to a location y. Locations x and y are initialised to 0.

The second section is the sequence of instructions on each thread. In MP, Thread P0 updates x to 1 via STR W0,[X1] and a flag y to 1 via STR W2,[X3]. Thread P1 reads the flag y via LDR W4,[X3] and x via LDR W5,[X1].

The last section, the exists clause, is an assertion about reachable final states. The exists clause of MP asks whether there is an execution where P1 reads the updated value of the flag y, and the outdated, initial value of x. As a convention, in exists clauses we use 64-bit registers like X4 rather than their lower 32-bit counterparts like W4.

We indicate answers to questions asked by exists clauses in captions of figures: "Required", if the final states of all MP executions allowed my the memory model satisfy the assertion, "Allowed", if the final states of some (but not all) do, or "Forbidden", otherwise. The MP test is Allowed, as evidenced by an allowed execution in Fig. 2, given as a graph whose nodes represent the Effects of the program instructions.

The Thread 0 box gives the semantics of P0: a write of x with value 1 (Effect a), in program order before (arrow po) a write of y with value 1 (Effect b). The Thread 1 box gives the semantics of P1: a read of y with value 1 (Effect c), in program order before (arrow po) a read of x with value 0 (Effect d). The values of the Reads c and d are specific to this execution and determined by the register values in the exists clause.

The arrows across boxes are the interactions between P0 and P1. The Read c of y by P1 takes its value from the update b made by P0, and this is depicted by the Reads-from (rf) arrow from Effect b to Effect c. The Read d of x by P1 takes its value from the initial state, which is depicted by a Reads-from (rf) arrow from the initial Write of x (Effect f) to the Read Effect d. The value read by Effect d is overwritten by the update a of x made by P0 (Effect a), which is depicted by the Coherence-after (ca) arrow from the Read d to the Write a.

b) Arm's application-level memory model: At a high level, a memory model determines which values a Read Effect can see. It does so by checking whether the Effects of an execution meet certain ordering requirements. Those requirements are defined on relations over Effects: *(a)* the Hardware-required-ordered-before relation [30, B2.3], e.g. local orderings that Arm hardware must enforce due to barriers or dependencies; and *(b)* the Observed-by relation [30, B2.3], i.e. interactions between threads such as the Reads-From or Coherence-after relations.

Together they form the Ordered-before relation. Informally, Ordered-before is the order in which Effects from a certain thread become visible to other threads, or in other words, externally visible. Formally, the External visibility requirement forbids executions with cycles in the Ordered-before relation.

The Arm memory model allows the MP test, but the "Forbidden" verdict could be achieved by a modification of it: e.g. with barrier instructions DMB ST between the stores on P0 and DMB LD between the loads on P1, or with a Release store STLR W2,[X3] in place of STR W2,[X3] and an Acquire load LDAR W4,[X3] in place of LDR W4,[X3]. In both cases, the extra synchronisation creates Hardware-Required-Ordered-Before relations between the Effects on each thread, forbidding their reordering. Furthermore, the interactions between P0 and P1 (viz, Reads-from from b to c and Coherence-after from d to a) contribute to Ordered-before by definition of the Arm memory model [30, B2.3]. Therefore the extra synchronisation creates a cycle in Ordered-before, violating the External visibility requirement, hence forbidding the MP behaviour.

III. Initial view of VMSA extension

The exists clause of the MP test can be interpreted as if x and y were PAs. Usually programs can only access PAs indirectly, through VAs that go through address translation. A

TTD prescribes how to translate a VA seen by software into a PA used by hardware, but only if a validity flag held in the TTD is set. Access to a VA with an invalid TTD raises a Fault.

We introduce the extention of the syntax of litmus tests for manipulating TTDs using the `MP-TTD` test in Fig. 3, featuring a variation of message passing seen in `MP`.

In the initial state of `MP-TTD`, we can now specify that `[TTD(x)]` holds the value `(valid:0,oa:PA(x))`, meaning that the TTD for VA `x` is invalid (has its `valid` bit set to `0`) and its output address is a symbolic value `PA(x)`.

In the body of `MP-TTD`, Thread `P0` updates the initially invalid `TTD(x)` via `STR X1,[X2]`, and `y` to 1 via `STR W7,[X8]`. Thread `P1` reads the flag via `LDR W7,[X8]` and attempts to read the VA `x` via `LDR W4,[X3]` at label `L0`).

In the `exists` clause of `MP-TTD`, we can now specify that an instruction is expected to fault. The predicate `Fault(P1:L0,x,MMU:Translation)` holds[1], if the instruction at label `L0` on Thread `P1` raises a fault due to an invalid `TTD(x)`, that is, an MMU Translation Fault. Overall, the `exists` clause of `MP-TTD` asks whether there is an execution where `P1` reads the updated flag as set by `P0`, and the outdated, initially invalid `TTD(x)`.

a) Extending instruction semantics: The instruction semantics of memory accesses requires extending to account for implicit memory accesses due to address translation and possibility of faulting. We illustrate key extensions for the `LDR` instruction in Fig. 4, and elaborate further in Section V.

Fig. 4 (i) gives the semantics of `LDR` without VMSA. We first obtain the location `x` by reading the source register `X2` (Effect c). We then obtain value 1 by reading from the location `x` (Effect a), and then write that value 1 into the target register `X1` (Effect d). The iico_data arrows depict the Intrinsic data dependencies due to Effect c passing on the location `x` to Effect a and Effect a passing on the value 1 to Effect d.

Fig. 4 (ii) gives the semantics of `LDR` with VMSA. We first obtain the VA `x` by reading the source register `X2` (Effect e). We then read from `TTD(x)` (Effect a)—Arm calls this an Implicit TTD Read Effect. Given that `TTD(x)` is valid, we obtain value 1 by reading from `PA(x)` (Effect b)—Arm calls this an Explicit Memory Read Effect. As before, we then write that value 1 into the target register `X1` (Effect g).

In Fig. 4 (ii), the Branching Effect f indicates that a decision has been made: depending on the validity of the TTD read by Effect a, we decide to access `PA(x)` as in Fig. 4 (ii), or to fault. Fig. 5 illustrates the faulting case. There we also read from `TTD(x)` (Effect a), but it is invalid, so after failing the validity check (Branching Effect e), we fault at Effect f.

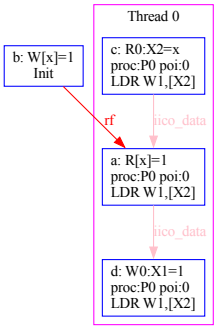
Lastly, notice the tr-ib arrows between the Implicit Read Effect a and the Explicit Memory Effect b in Fig. 4, and the Implicit Read Effect a and the Fault Effect b in Fig. 5. These depict a relation Arm calls Translation-Intrinsically-Before, which it considers as providing order: Explicit Memory Effects and Fault Effects are always Ordered-after Implicit TTD Reads of the corresponding address translations.

[1]Arm's special-purpose registers, the Fault Address Register `FAR_EL1` and the Exception Syndrome Register `ESR_EL1`, can be used to infer the information about the fault asserted by `Fault(P1:L0,x,MMU:Translation)`.

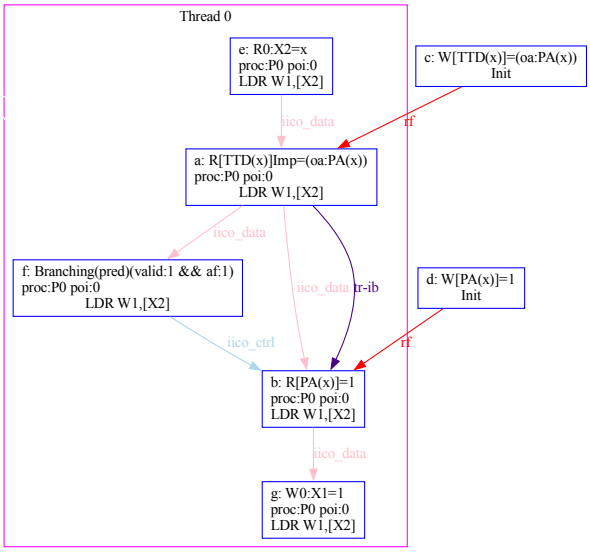
```
AArch64 MP-TTD
{
0:X2=TTD(x); 0:X1=(valid:1,oa:PA(x));
0:X8=y; 1:X8=y;
1:X3=x;
[TTD(x)]=(valid:0,oa:PA(x));
}
P0           | P1;
STR X1,[X2] | LDR W7,[X8]    ;
MOV W7,#1    | L0:LDR W4,[X3];
STR W7,[X8] |                ;
exists 1:X7=1
       /\ Fault(P1:L0,x,MMU:Translation)
```

Fig. 3. Message passing using TTDs—Allowed

```
AArch64 LDR
{
0:X2=x; 0:X1=0;
x=1;
}
P0          ;
LDR W1,[X2];
exists(0:X1=1)
```



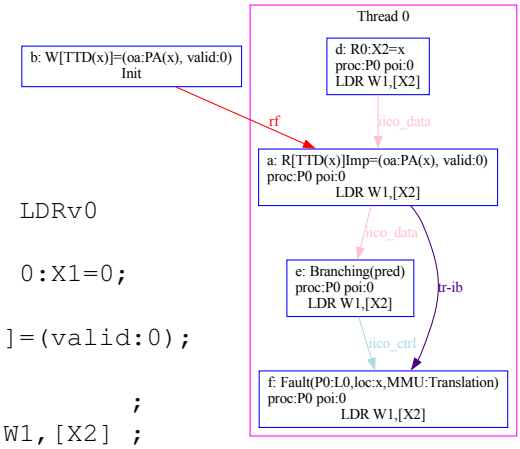
(i) Without VMSA



(ii) With VMSA

Fig. 4. Semantics of `LDR` without and with VMSA—Allowed

b) Extending memory model: Barriers and inter-thread observations affect memory accesses due to address translation differently from application-level ones. For example, the modification of the `MP-TTD` test, `MP-TTD+DMB.ST+DMB.LD` in Fig. 6, is not Forbidden, even though the `MP` test in Fig. 1




```
AArch64 LDRv0
{
0:X2=x; 0:X1=0;
x=1;
[TTD(x)]=(valid:0);
}
P0                ;
L0:LDR W1,[X2] ;
exists(Fault(P0:L0,x,MMU:Translation))
```

Fig. 5. Semantics of `LDR` when `valid` is 0—Allowed

```
AArch64 MP-TTD+DMB.ST+DMB.LD
{
0:X2=TTD(x); 0:X1=(valid:1,oa:PA(x));
0:X8=y; 1:X8=y;
1:X3=x;
[TTD(x)]=(valid:0,oa:PA(x));
}
P0           | P1;
STR X1,[X2]  | LDR W7,[X8]     ;
DMB ST       | DMB LD          ;
MOV W7,#1    | L0: LDR W4,[X3];
STR W7,[X8]  |                 ;
exists (1:X7=1
        /\ Fault(P1:L0,x,MMU:Translation))
```

Fig. 6. Insufficient synchronisation for `MP-TTD`—Allowed

with the same modification is. Our VMSA model extends Hardware-required-before and Observed-by relations of Arm's application-level memory model to apply to Implicit TTD Effects and Fault Effects, but accounts for the differences.

The `VIS01-load` test in Fig. 7 illustrates Implicit TTD Reads exhibiting ordering similar to Explicit Memory Effects. It is Forbidden, because the Implicit TTD Read of the `LDR` is Hardware-required-ordered-before the write to `TTD(x)` by the `STR` later in program order, and must not observe it.

The `MP+rfi-addr+dmb.ld` test in Fig. 8 and the `MP-TTD+rfi-addr+dmb.ld` test in Fig. 9 illustrate the only case of Implicit TTD Reads having stronger ordering than Explicit Memory Effects. These message passing tests both order the load of the flag `y` ahead of the load of the data with a `DMB LD`. The `exists` clause of each test reduces to whether there is an order between the stores on `P0`, and it is only in the `MP-TTD+rfi-addr+dmb.ld` test that there is—due to a Implicit TTD Read.

In `MP+rfi-addr+dmb.ld`, the store of the data (`STR W0,[X1]`) is not ordered before the store of the flag (`STR W4,[X5,W3,SXTW]`). Indeed, although `LDR W2,[X1]`

```
AArch64 VIS01-load
{
0:X1=x; 0:X2=0;
0:X4=TTD(x); 0:X3=(oa:PA(y));
y=1;
[TTD(x)]=(oa:PA(x));
}
P0           ;
LDR W2,[X1] ;
STR X3,[X4] ;
exists(0:X2=1)
```

Fig. 7. Visibility of TTD updates—Forbidden

```
AArch64 MP+rfi-addr+dmb.ld
{
0:X1=x; 1:X1=x;
0:X5=y; 1:X5=y;
}
P0                    | P1          ;
MOV W0,#1             | LDR W0,[X5];
STR W0,[X1]           | DMB LD      ;
LDR W2,[X1]           | LDR W2,[X1];
EOR W3,W2,W2          |             ;
MOV W4,#1             |             ;
STR W4,[X5,W3,SXTW] |             ;
exists (0:X2=1 /\ 1:X0=1 /\ 1:X2=0)
```

Fig. 8. No Explicit order via Internal Reads-from—Allowed

```
AArch64 MP-TTD+rfi-addr+dmb.ld
{
0:X11=TTD(x); 1:X11=TTD(x);
0:X5=y; 1:X5=y;
0:X10=(oa:PA(x), valid:1);
0:X1=x;
[TTD(x)]=(oa:PA(x),valid:0);
x=1;
}
P0                    | P1             ;
STR X10,[X11]         | LDR W4,[X5]    ;
LDR W2,[X1]           | DMB LD         ;
EOR W3,W2,W2          | LDR X10,[X11] ;
MOV W4,#1             |                ;
STR W4,[X5,W3,SXTW] |                ;
exists(1:X4=1 /\ 1:X10=(oa:PA(x),valid:0))
```

Fig. 9. Reads-from orders Implicit TTD Effects—Forbidden

reads the data just written by the former store and creates an address dependency into the latter, Reads-from relation between Explicit Effects on the same thread does not contribute to the Observed-by relation. Intuitively, the new value of `x` could be in a buffer private to `P0` and visible to `P0` only.

In `MP-TTD+rfi-addr+dmb.ld`, the store of the data (`STR X10,[X11]`), which is `TTD(x)` in this case, is ordered before the store of the flag (`STR W4,[X5,W3,SXTW]`).

Indeed, an Implicit TTD Read of `TTD(x)`, which is due to `LDR W2,[X1]`, reads the TTD value just written by the former store, and uses the outcome of the Explicit Read of the load to create an address dependency into the latter. This internal Reads-from relation contributes to the Observed-by relation due to ending on an Implicit TTD Read. Arm considers thread's translation table walks generating Implicit TTD Reads a "separate observer" [30, D8.2.6,R_TGRMW] from the thread itself. The associated Reads-from relations are externally visible and have ordering implications akin to Reads-from relations due to Explicit Memory Effects of different threads that also contribute to the Observed-by relation in the Arm application-level memory model.

IV. TLB MAINTENANCE

Translation Lookaside Buffers (TLBs) are caches that store recently used TTDs for faster address translation. Caching in TLBs is visible to system-level software, and TLBs must be managed at that level, which Arm calls Exception Level 1 (EL1). TTDs change infrequently, so it is pragmatic for hardware not to enforce coherence of TLBs. Therefore, unlike the application-level model, where Coherence-after orders Explicit Memory Effects and contributes to the Observed-by relation (as discussed in Section II), in the VMSA extension Coherence-after does not always order Implicit TTD Effects. Some TTDs are never cached in TLBs, e.g. invalid ones, and are always subject to ordering due to Coherence-after, while the TTDs that can be cached require using an instruction called `TLBI` to ensure that TLBs are coherently up to date.

Part of the complexity of modelling VMSA is in capturing the ordering implications of `TLBI`. We illustrate this complexity by first presenting load hazarding in an application-level example, and then explain its implications for VMSA in a case study of the `CopyOnWrite` example.

a) Load hazarding: The `coRR` test in Fig. 10 exposes load hazarding. Thread `P0` reads `x` twice, while thread `P1` updates `x` from 1 to 2. Arm forbids `P0` from first seeing the updated value and then the outdated value due to the Hazard-ordered-before relation, which is included in Ordered-before and defined as follows [30, B2.3]. *An Effect E1 is Hazard-ordered-before an effect E2 if all of the following apply:*

- *E1 is an Explicit Read Memory Effect R1.*
- *R1 appears in program-order before an Explicit Read Memory Effect R3*
- *R1 and R3 access the same Location.*
- *R1 and E2 are from different Observers.*
- *R3 is Coherence-before E2.*
- *E2 is an Explicit Write Memory Effect W2.*

Looking at the execution in Fig. 10: the Read a is Hazard-ordered-before the Write c (using a as E1, c as E2 and the Read b as R3 in the definition). Moreover the Read a takes its value from the update of `x` by `P1` and hence a Reads-from c, which forms a cycle in Ordered-before and therefore violates the External visibility requirement of Arm's memory model.

```
AArch64 coRR
{
0:X2=x; 1:X2=x;
int x=1;
}
P0           | P1           ;
LDR W0,[X2]  | MOV W0,#2    ;
LDR W1,[X2]  | STR W0,[X2];
exists(0:X0=2 /\ 0:X1=1)
```

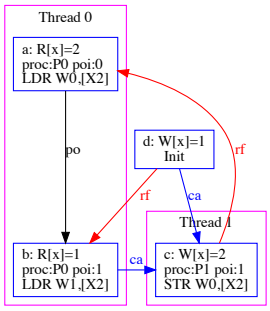


Fig. 10. Load hazarding behaviour—Forbidden

```
AArch64 CopyOnWrite
{
int x=1; int y=0; int z=0;
[TTD(x)]=(oa:PA(x),valid:1);
[TTD(z)]=(oa:PA(x)); 2:X7=(oa:PA(x),valid:0);
2:X6=TTD(x); 2:X8=TTD(y);
0:X2=x; 1:X2=x; 2:X2=x; 2:X4=y; 2:X10=z;
}
P0           | P1           | P2                ;
                            (*invalidates x*)
             |              | STR X7,[X6]       ;
                            (*TLB maintenance*)
             |              | DSB ISH           ;
             |              | LSR X3,X2,12      ;
             |              | TLBI VAAE1IS,X3   ;
             |              | DSB ISH           ;
                            (*memcopy via z*)
             |              | LDR W5,[X10]      ;
             |              | STR W5,[X4]       ;
             |              | DMB ISH           ;
                            (*overwrites TTD(x)*)
LDR W0,[X2]  | MOV W0,#2    | LDR X1,[X8]       ;
LDR W1,[X2]  | STR W0,[X2]  | STR X1,[X6]       ;
exists 0:X0=2 /\ 0:X1=1
```

Fig. 11. A `CopyOnWrite` example from Linux—Forbidden

b) CopyOnWrite: Although the distinction between virtual and physical memory is apparent at the system level (at EL1), it must not be at the application level (at EL0). Hence, the `coRR` test must remain Forbidden regardless of TTD manipulations performed by an operating system (OS). We illustrate this using the `CopyOnWrite` test in Fig. 11, which is an excerpt of the Linux kernel given to us by its maintainers featuring a typical TTD manipulation sequence.

Threads `P0` and `P1` communicate exactly as `coRR` in Fig. 10 and represent user-level instructions. Thread `P2` represents the OS manipulating TTDs: `P2` remaps `VA(x)` to a different PA, and a load hazard on `P0` is made impossible by careful TLB maintenance on `P2`, which we explain next.

Initially `TTD(x)` is valid and points to `PA(x)`. `TTD(z)` points to `PA(x)`: `z` is an alias of `x`. Thread `P2` performs a *break-before-make* sequence over VA `x`. Step-by-step, `P2`:

- invalidates `x` by setting `valid` to 0 in `TTD(x)`;

- uses a first `DSB ISH` instruction for ordering;
- computes a Logical Shift Right `LSR` of 12 bits over the VA `x`, i.e. the key for searching the TLB-cached value of `TTD(x)`;
- uses a `TLBI` instruction;
- uses a second `DSB ISH` instruction for ordering;
- makes a copy of `PA(x)` (accessed via VA `z`) into `y`;
- uses a `DMB` barrier instruction for ordering;
- overwrites `TTD(x)` with `TTD(y)`.

c) The DMB barrier: The `DMB` on `P2` ensures that the `STR W5,[X4]` and the `LDR X1,[X8]` are not re-ordered, as per the Arm application-level memory model [21]. Hence the `DMB` ensures that when `P2` makes a copy of the value of `PA(x)` (accessed via the VA `z`) into `y` and overwrites the TTD of `x` with the TTD of `y`, then the copy is done ahead of the overwrite.

d) The TLB maintenance sequence: Fig. 12 shows an execution of `CopyOnWrite` satisfying its `exists` clause. To see why, we map the instructions of the test in Fig. 11 to the Effects and relations of Fig. 12. As the full execution graph is large, we only show selected Effects: on `P0`, the Implicit TTD Read Effects a and c generated by the first load and second load respectively; on `P2`, the `TTD` Write Effects e and k generated by the `STR X7,[X6]` and the `STR X1,[X6]` respectively; and the `TLBI` Effect ev31. In this execution:

- $o \xrightarrow{\text{rf}} c$ (c Reads-from o), i.e. the Implicit TTD Read Effect c of `LDR W1,[X2]` on `P0` reads its value from the initial Write to `TTD(x)` represented by Effect o.
- $k \xrightarrow{\text{rf}} a$ (a Reads-from k), i.e. the Implicit TTD Read Effect a of `LDR W0,[X2]` on `P0` reads the value written by the Explicit TTD Write Effect k of `STR X1,[X6]` on `P2`.
- $a \xrightarrow{\text{po-va-loc}} c$, i.e. `LDR W0,[X2]` on `P0` is in program order before `LDR W1,[X2]` and to the same VA.
- $c \xrightarrow{\text{ca}} e$, ($e$ is Coherence-after c), i.e. the value read by c is invalidated by Effect e of `STR X7,[X6]` on `P2`, since by convention the initial Write o that c Reads-from is Ordered-before other Writes to `TTD(x)` such as e.
- $e \xrightarrow{\text{dsb.ish}}$ ev31, i.e. e is Ordered-before ev31 thanks to `DSB ISH` between the two corresponding instructions, the invalidation `STR X7,[X6]` and `TLBI`.
- ev31 $\xrightarrow{\text{dsb.ish}} k$, i.e. the `TLBI` Effect ev31 is Ordered-before the overwrite k made by `P2` thanks to `DSB ISH` between the two corresponding instructions, `TLBI` and `STR X1,[X6]`.
- $c \xrightarrow{\text{TTD-read-ordered-before}} k$, i.e. the Implicit TTD Read c of `LDR W1,[X2]` is Ordered-before the overwrite k.

e) Hardware intuition for TTD-read-ordered-before: From a hardware point of view, the `TLBI` on `P2` sends a message to `P0`, which is ordered arbitrarily w.r.t. the Implicit TTD Reads a and c. The TLBI-before relation denotes this arbitrary order: formally, we enumerate all possible pairs of `TLBI` Effects and Implicit Reads within the scope of the `TLBI`. If the `TLBI` message is inserted before the Implicit

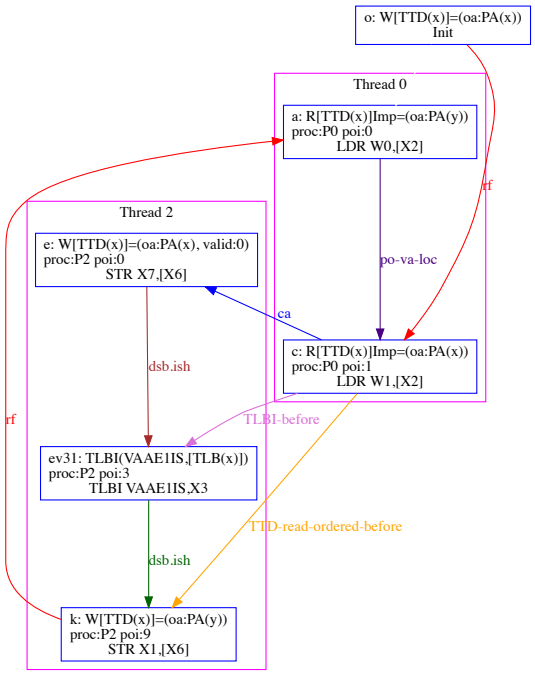


Fig. 12. Forbidden execution of `CopyOnWrite` (Fig. 11)

Read c, then ev31 is TLBI-before c, otherwise c is TLBI-before ev31.

The first `DSB ISH` on `P2` ensures that only an execution where c is TLBI-before ev31 (as shown in Fig. 12) is possible. An execution where this arbitrary order goes the other way, ev31 TLBI-before c, has a cycle forbidden by the External Visibility requirement: ev31 $\xrightarrow{\text{TLBI-before}} c \xrightarrow{\text{ca}} e \xrightarrow{\text{dsb.ish}}$ ev31.

When c is TLBI-before ev31, the `TLBI` message ev31 waits until c has completed. `P0` sends a message back to `P2` to enable the completion of the second `DSB`. The overwrite k is in program order after the second `DSB`, and therefore dsb.ish-after the `TLBI`. Consequently, c is TTD-read-ordered-before k.

f) TLBI contributes to External Visibility: Unlike in the `coRR` test, the hazardous Read Effects a and c are Implicit, so `CopyOnWrite` is not forbidden by Hazard-ordered-before.

Forbidding `CopyOnWrite` requires extra synchronisation: the `TLBI` instruction on `P2` ensures that if the Read a sees the value of `TTD(x)` overwritten by `STR X1,[X6]`, then the Read c cannot see the initial state for `TTD(x)` any longer.

Formally, `TLBI` creates a cycle forbidden by External Visibility: $k \xrightarrow{\text{rf}} a \xrightarrow{\text{po-va-loc}} c \xrightarrow{\text{TTD-read-ordered-before}} k$.

V. Translation Table Hardware Management

The Arm architecture associates read and write permissions of a VA with two bits of a TTD called Access Flag and Dirty Bit, respectively. Managing those can be a responsibility of system-level software, but with Translation Table Hardware Management (TTHM), a feature of the Arm architecture introduced in v8.1, the responsibility can be transferred to hardware. To this end, two bits in the special Translation Control Register, `TCR_ELx.HA` for the Access Flag and

```
AArch64 STRdb0dbm1
TTHM=HA HD
{
0:X2=x;
[TTD(x)]=(db:0,dbm:1);
}
P0                ;
MOV W0,#1         ;
L0: STR W0,[X2]   ;
exists(x=1 /\
  [TTD(x)]=(db:1,dbm:1) /\
  ~Fault(P0:L0,x,MMU:Permission))
```

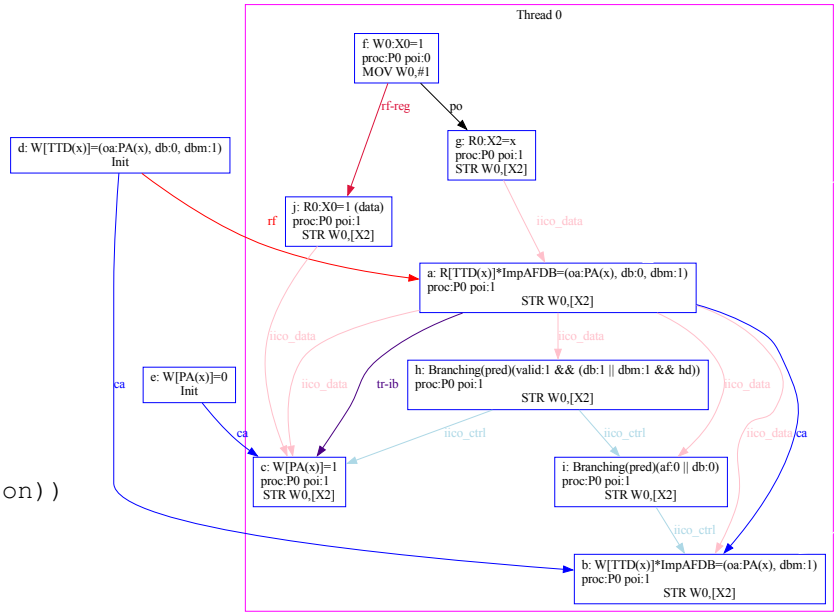


Fig. 13. Semantics of `STR` when `db` is 0 and `dbm` is 1—Required

`TCR_ELx.HD` for the Dirty Bit, enable hardware management of permissions in TTDs.

We model permissions by letting a Translation Table Descriptor `TTD(x)` to be a tuple `(oa, valid, af, db, dbm)`, whose default value is `(oa:PA(x), valid:1, af:1, db:1, dbm:0)`. (For brevity, in litmus tests we tend to omit fields with default values from initial states or `exists` clauses.) The output address `oa` is the PA corresponding to the VA `x`. We have seen the effect of the `valid` bit in Section III. The Access Flag is modelled by the `af` bit, and the Dirty Bit by `db`. The Dirty Bit Management bit `dbm` indicates if hardware management of the Dirty Bit is on.

Permissions are considered in the instruction semantics with the following three cases.

Case 1: no hardware management. With TTHM implemented and `TCR_ELx.HA,HD=0,0`, or without TTHM:

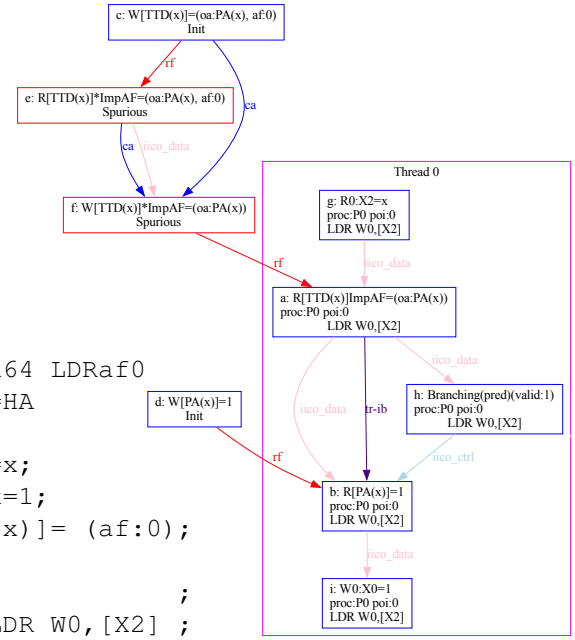
- A load or store with `af` set to 0 raises a Permission Fault;
- A store to `x` where `TTD(x)` has `db` set to 0 raises a Permission Fault.

Case 2: hardware management of Access Flag only. With TTHM implemented and `TCR_ELx.HA=1`:

- A load or store with `af` set to 0 results in the MMU updating `af` to 1, and continuing without a Fault;
- A store with `db` set to 0 raises a Permission Fault.

Case 3: hardware management of Access Flag and Dirty Bit. With TTHM is implemented and `TCR_ELx.HA,HD=1,1`:

- A load or store with `af` set to 0 results in the MMU updating `af` to 1, and continuing without a Fault;
- A store with `db` and `dbm` set to 0 raises a Permission Fault;
- A store with `db` set to 0 and `dbm` set to 1 results in the MMU updating `db` to 1, and continuing without a Fault.




```
AArch64 LDRaf0
TTHM=HA
{
0:X2=x;
int x=1;
[TTD(x)]= (af:0);
}
P0                ;
L0: LDR W0,[X2]   ;
exists(0:X0=1 /\ [TTD(x)]=(af:1) /\
  ~Fault(P0:L0,x,MMU:Permission))
```

Fig. 14. Semantics of `LDR` when the Access Flag is 0—Required

a) Metadata in litmus tests: Litmus tests can now specify if TTHM is implemented and if `TCR_ELx.HA,HD` are set to 1 (by default they are not). Thus, in the test in Fig. 14, `TTHM=HA` indicates that TTHM is implemented and that `TCR_ELx.HA` is set to 1. Initially `x` is 1, and `TTD(x)` has `af` at 0. At line `L0` thread `P0` attempts a load of `x`. The `exists` clause asks if register `X0` holds the value 1, `TTD(x)` has `af` set to 1 and there was no Fault at line `L0` relative to `x`. In the execution

```
AArch64 coWW-HD
TTHM=HA HD
{
0:X1=TTD(x); 0:X0=(oa:PA(x),db:0,dbm:1);
0:X3=x;
[TTD(x)]=(oa:PA(x),db:0,dbm:0);
}
P0              ;
STR X0,[X1]     ;
MOV W2,#1       ;
L0: STR W2,[X3] ;
exists(~Fault(P0:L0,x,MMU:Permission) /\
  [TTD(x)]=(oa:PA(x),db:0,dbm:1))
```

Fig. 15. Hardware Updates are Read-Modify-Writes—Forbidden

```
AArch64 STRva-STRttd
TTHM=HA HD
{
0:X0=x; 0:X1=1;
1:X3=(oa:PA(b),db:0,dbm:1); 1:X2=TTD(x);
int a=0; int b=0;
[TTD(x)]=(oa:PA(a),db:0,dbm:1);
}
P0          | P1          ;
STR W1,[X0] | STR X3,[X2];
exists(b=1 /\ TTD(x)=(oa:PA(b),db:0,dbm:1))
```

Fig. 16. Coherence-after from Hardware Update—Forbidden

graph, also in Fig. 14, we see the atomic pair (e, f) updating the af bit to 1. This represents the hardware management of the Access Flag.

In the test in Fig. 13, TTHM=HA HD indicates that TTHM is implemented and that TCR_ELx.HD is 1. Initially x is 0, db is 0 and dbm is 1. At line L0 thread P0 attempts a store of 1 to x. In the exists clause, we ask if x holds the value 1, db is set to 1 and there was no Fault at L0 relative to x. In the execution graph, also in Fig. 13, we see the atomic pair (a, b) updating db to 1. This represents the hardware management of the Dirty Bit.

b) Spontaneous hardware updates: With the hardware management of the Access Flag, the Arm architecture allows for spontaneous setting of Access Flags, e.g, as a side effect of squashed speculative execution of an instruction. We model this as a *spontaneous hardware update* occurring at any point in the execution. Thus, on the graph in Fig. 14, the Hardware Update f of af occurs outside of the Thread P0. In contrast, Dirty Bits are not set spontaneously, and certain OSes rely on the precise ordering guarantees for those. In Fig. 13, the Hardware Update k of db in is Ordered-after the Branching Effect i, or in other words, after the TTD has been checked and a decision has been made by the MMU. This discrepancy is documented in the Arm ARM, in [30, D8.4.5,R_LHQRX] for af and [30, D8.4.6,R_DYCFD] for db.

```
AArch64 MP-TTD+DSB-TLBI-DSB+DMB.LD
{
0:X2=TTD(x); 0:X1=(valid:1,oa:PA(x));
0:X3=x; 1:X3=x;
0:X8=y; 1:X8=y;
[TTD(x)]=(valid:0,oa:PA(x));
}
P0              | P1;
STR X1,[X2]     | LDR W7,[X8]   ;
DSB ISH         | DMB LD        ;
LSR X9,X3,#12   |L0:            ;
TLBI VAAE1IS,X9 | LDR W4,[X3]   ;
DSB ISH         |               ;
MOV W7,#1       |               ;
STR W7,[X8]     |               ;
exists (1:X7=1
        /\ Fault(P1:L0,x,MMU:Translation))
```

Fig. 17. Unnecessary TLB maintenance—Forbidden

c) Hardware Updates must be Read-Modify-Writes: Test coWW-HD in Fig. 15 shows that Hardware Updates are Read-Modify-Writes. This ensures that the TTD Read which provokes the Hardware Update (e.g. when db is seen to be 0) cannot take its value from a Write which is Coherence-after the Hardware Update.

d) Coherence for Hardware Updates: Hardware Updates exhibit familiar coherence orderings, unlike Implicit TTD Read Effects that generally require TLB maintenance to restore coherence.

Fig. 16 shows that Coherence-after from a Hardware Update to a TTD Write counts as an ordering (the opposite ordering holds as well but is not shown for brevity). In this test, TTD(x) initially points to PA(a), with db set to 0. The STR on P0 aims to update x with value 1, and the STR on P1 aims to change the output address of TTD(x) to point to PA(b), also with db set to 0. It must not be the case that P1 can make TTD(x) point to PA(b) if the Hardware Update of db by P0 is Coherence-before the Explicit TTD Write by P1.

VI. Enhanced Translation Synchronisation

Enhanced Translation Synchronisation (ETS) is a feature of the Arm architecture aimed at allowing lighter synchronisation in system-level code for ensuring order between TTD updates and concurrent uses of address translations: suitably for patterns of updating TTDs that cannot be cached to those that can, hardware would provide ordering w.r.t. Implicit TTD Effects without requiring ISB.

We illustrate the motivation for ETS using the MP-TTD test in Fig. 3 and showing two approaches to synchronising the update of TTD(x) on P0 and the Implicit TTD Read of the load from x on P1—in absence of ETS.

Firstly, the MP-TTD+DSB-TLBI-DSB+DMB.LD test in Fig. 17. It uses a DMB LD barrier on P1, but other approaches to local ordering of Explicit Read Effects of loads could be used equivalently, e.g., an address dependency.

```
AArch64 MP-TTD+DMB.ST+DSB-ISB
{
0:X2=TTD(x); 0:X1=(valid:1,oa:PA(x));
1:X3=x;
0:X8=y; 1:X8=y;
[TTD(x)}=(valid:0,oa:PA(x));
}
P0              | P1;
STR X1,[X2]     | LDR W7,[X8]    ;
DMB ST          | DSB SY         ;
MOV W7,#1       | ISB            ;
STR W7,[X8]     | L0: LDR W4,[X3];
exists (1:X7=1
        /\ Fault(P1:L0,x,MMU:Translation))
```

Fig. 18. Canonical synchronisation pre-ETS—Forbidden

```
AArch64 MP-TTD+DMB.ST+lobF
{
0:X2=TTD(x); 0:X1=(valid:1,oa:PA(x));
0:X8=y; 1:X8=y;
1:X3=x;
[TTD(x)]=(valid:0,oa:PA(x));
}
P0           | P1              ;
STR X1,[X2]  | LDR W4,[X8]     ;
DMB ST       | L0: STLR W4,[X3];
MOV W7,#1    |                 ;
STR W7,[X8]  |                 ;
exists (1:X4=1
        /\ Fault(P1:L0,x,MMU:Translation))
```

Fig. 19. Ordered-before a Fault?—Allowed w/o ETS or ETS2

Crucially, it uses a TLB maintenance sequence on `P0`. Like in the `CopyOnWrite` test in Section IV, the `TLBI` on `P0` ensures order w.r.t. a concurrent Implicit TTD Read on `P1`. Note that `TTD(x)` is initially invalid and, therefore, not cached in a TLB. Indeed, the Arm ARM states that *Translation table entries that generate a Translation fault, an Address size fault, or an Access flag fault are never cached in a TLB* [30, D8.12, R_XCLRD] and calls these Fault Effects TLBUncacheable [30, pp. B2-169]. Therefore, the store to `TTD(x)` on `P0` does not require TLB maintenance to invalidate a cached TLB entry; `TLBI` is only used as a synchronisation primitive, which is both very costly and counter-intuitive.

Secondly, consider the `MP-TTD+DMB.ST+DSB-ISB` test in Fig. 18. It uses a `DMB ST` barrier on `P0`. Crucially, it uses `DSB SY` and `ISB` barriers on `P1`. Note that this approach places the burden of synchronisation on threads concurrent with the TTD manipulation. Feedback from Arm partners indicated that requiring an `ISB` was both costly and confusing.

ETS was intended to ensure that in the `MP-TTD` test, any approach to local ordering of Explicit Read Effects (as, e.g., in the `MP-TTD+DMB.ST+DMB.LD` test in Fig. 6 that would be forbidden with ETS) orders the Explicit Read Effect of the load from `y` before the Implicit TTD Read Effect of the load from `x` on `P1`.

a) From ETS to ETS2: ETS intended to extend the local ordering requirement w.r.t. a faulting instruction to its Implicit TTD Read. Our efforts towards modeling ETS uncovered subtle special cases that led to a revision of the feature.

Consider the `MP-TTD+DMB.ST+lobF` test in Fig. 19, which follows the pattern of `MP-TTD`, but on `P1` instead of a load from `x` it performs a store release (`STLR W4,[X3]`). Generally speaking, the store release ensures local ordering between the Explicit Effects of the two stores on `P1`. However, in the execution of interest in the `MP-TTD+DMB.ST+lobF` test, the `STLR` faults. This leads to a question: does `STLR` have the release semantics even when the instruction faults, or only when it does not fault?

We clarified this question with Arm: Arm intended the release semantics and, more generally, any local ordering to apply to faulting instructions and, hence, the `MP-TTD+DMB.ST+lobF` test to be forbidden with ETS. Despite clarifying the intent for ETS, it was felt by Arm that ETS deserved to be strengthened to avoid any confusion. Thus, ETS was deprecated in favour of ETS2, which provides the ordering desired on `P1` in the `MP-TTD` test without any synchronisation at all.

```
AArch64 coRR-Permission
{
0:X2=TTD(x); 0:X1=(oa:PA(x),db:1,valid:1);
1:X3=x;
[TTD(x)]=(db:0, valid:1);
}
P0           | P1              ;
STR X1,[X2]  | MOV W4,#1       ;
             | L1: STR W4,[X3] ;
             | L0: STR W4,[X3] ;
exists (~Fault(P1:L1,x)
        /\ Fault(P1:L0,x,MMU:Permission))
```

Fig. 20. ETS2 does not affect TLBCacheable Faults—Allowed

Our VMSA model and the Arm ARM define the ordering due to ETS2 as follows [30, B2.3]: *If ETS2 is implemented, E1 is an Explicit Memory Effect, E2 is an Implicit TTD Read and all of the following applies:*

- *E1 is program-order-before a TLBUncacheable Effect E3*
- *E2 is Translation-intrinsically-before E3*

Then E1 is Ordered-before E2

b) ETS2 and non-TLBUncacheable Faults: Although ETS2 provides stronger ordering than ETS, it still only applies to TLBUncacheable Faults, notably excluding Permission Faults. This permits designs with one TLB per store pipeline. As a consequence, the `coRR-Permission` test in Fig. 20 is allowed under ETS2. In contrast, the `coRR-Translation` test in Fig. 21 is forbidden under ETS2. This is because the Permission Fault in the `coRR-Permission` test is not TLBUncacheable, as opposed to the Translation Fault in `coRR-Translation`.

```
AArch64 coRR-Translation
{
0:X2=TTD(x); 0:X1=(oa:PA(x),valid:1);
1:X3=x;
[TTD(x)]=(valid:0);
}
P0            | P1                ;
STR X1,[X2]   | MOV W4,#1         ;
              | L1: STR W4, [X3]  ;
              | L0: STR W4, [X3]  ;
exists(~Fault(P1:L1,x) /\
  Fault(P1:L0,x,MMU:Translation))
```

Fig. 21. ETS2 on TLBUncacheable Faults—Forbidden w/ ETS2

VII. Validating our model

We validated the Arm VMSA memory model in two ways:

- by discussing with Arm architects and Arm partners
- by testing extensively against existing hardware

a) Ratification by Arm and partners: We validated our model via 3-year long in depth discussions with a forum made of Arm architects and partners. First, we developed and validated the instruction semantics, including in presence of hardware management: for example, the execution graphs in Figs. 4, 5, 14, and 13 were discussed and refined in that forum.

Second, we recorded the architectural intent by studying litmus tests (e.g. the ones in this paper) and discussing their expected behaviour, from the perspectives of both hardware implementation and software usage.

Third, after having developed an initial model which matched the recorded intent, we discussed the definitions themselves. Much reorganisation and redefinition happened at that stage: one difficulty was devising definitions which would not only make the model behave as intended, but also would be transliterated into English in an acceptable manner, and would convey relatable intuition to both hardware and software folks.

Fourth, we reviewed this revamped model with Arm and its partners from start to finish, and eventually ratified it. It was then incorporated into the Arm Architecture Reference Manual [30, B2.3], and released in the herdtools distribution [9].

b) Testing against a variety of Arm hardware: We tested the Arm VMSA model extensively against existing hardware. To do so, we have put together a suite of about 1300 litmus tests that exercise different aspects of the new semantics, such as the ones shown in this paper. Most of those tests have also been discussed during model design to validate intent.

For testing we use the open source tool litmus7 [11] to produce binaries that we can run on hardware. The litmus7 tool takes litmus tests as input and generates `C` with inline AArch64 assembly source code. The source code is compiled and linked to provide executable binaries that run at EL0.

We extended litmus7 to support running litmus tests at EL1 or higher exception levels: that is where TTD features used by many litmus tests of interest can be controlled. Hence, we extended litmus7 to build binaries that can run on top of a virtual machine—we handle Linux KVM [1] and MacOS HVF [3]—using QEMU [4]. The source code generated by litmus7 uses the library functions provided by KVMunit-tests to setup the system, get pointers to TTDs and install fault handers as necessary. For the purposes of this work, we added support for configurable translation granule for the arm64 target in KVM-unit-tests [6].

```
AArch64 STRva-SWPttd
TTHM=HA HD
{
0:X2=x;
1:X4=TTD(x);
1:X6=(oa:PA(x),db:0,dbm:1,valid:0);
[TTD(x)=(db:0,dbm:1);
}
P0             | P1              ;
MOV W1,#1      | SWP X6,X8,[X4]  ;
L0:STR W1,[X2] |                 ;
exists (1:X8=(oa:PA(x), dbm:1) /\
  [TTD(x)]=(oa:PA(x), valid:0, dbm:1, db:0) /\
  [x]=0 /\ Fault(P0:L0,x,MMU:Translation))
```

Fig. 22. Dirty Bit set spontaneously—Forbidden?

Our testing uncovered infidelities to the principle that the Dirty Bit should not be updated spontaneously by hardware, as discussed in Section V. More precisely, we observed 32 contradicting tests; Fig. 22 shows one of them.

Initially it was hypothesised that we could see this result due to interactions with virtualisation in our testing setup. This led to two significant next steps: *(a)* Arm relaxed the architecture to allow those behaviours in the virtualised case [30, 1st bullet of R_DYCFD]; and *(b)* we developed a bare-metal testing infrastructure as well, to remove noisy interactions with virtualisation and confirm the hypothesis.

To that aim, we extended KVM-unit-tests and added support for building binaries that run as Extensible Firmware Interface (EFI) applications. An EFI application is typically used for booting or interfacing with an operating system.

Advanced Configuration and Power Interface (ACPI) and Device Tree (DT) are two methods used by system-level software (e.g. KVM-unit-tests) to discover information about the system. DT used to be the preferred way for Arm systems; it requires a file with the device tree information (FDT). ACPI removes the need for providing an FDT, but is only supported for Arm systems that boot through EFI. To make the process of running litmus tests as EFI applications as simple as possible, we added support for ACPI in KVM-unit-tests. Our work is open-source and we have already worked with the community to upstream the first series of patches to KVM-unit-tests [6].

Our 1300 tests ran for at least 160 million times both in bare-metal and virtual modes, on a Cavium ThunderX2 CN9975 (2 sockets, 28 Vulcan cores per socket, 4 threads per core); two Arm N1SDP (4 Neoverse-N1 cores); an Apple M1 Pro (8 Firestorm and 2 Icestorm cores); a HiSilicon Kunpeng 920-6426 (2 sockets, 48 TaiShan v110 cores per socket); an

ODROID C2 (4 A53 cores); a RaspberryPi 4 Model B (4 A72 cores); an ODROID N2+ (2 A53 cores, 4 A73 cores).

Interestingly, the tests which contradict the Dirty Bit principle (e.g. in Fig. 22) were observed in our bare-metal infrastructure as well. Some designs were confirmed to allow such behaviours and we are currently investigating with Arm how to relax the architecture in those cases too.

VIII. Related works

Previous work on application-level memory models (e.g. for x86 [36], IBM Power [10], [35], NVidia [7], [33], Arm [12], [23], and RISC-V [5]) paved the way for the methodology that we use here: writing executable models validated both by discussion with architects and testing. Our work naturally builds on the Arm application-level memory model [21].

The works of [24] and [8] aim to extend the Arm application-level model with mixed-size accesses semantics—the latter has been ratified by Arm and integrated in the Arm ARM. The work of [39] aims to extend the Arm application-level model with instruction-fetch semantics. The work of [20] studies transactional memory for Arm. The work of [19] studies persistent memory for x86 and Arm. The work of [34] studies non-temporal memory accesses for x86.

Several works study virtual memory for x86 or Arm [13], [15]–[18], [22], [25]–[27], [29], [40]–[42], [44], but with no account of the weak memory models of those architecture.

The work of [28] presents a model of virtual memory called x86t_elt, which extends the x86-TSO memory model [37] in a similar style to ours. Empirical validation of x86t_elt remains open and could be approached with our testing methodology.

The work of [43] proposes a verification framework for system-level software, accounting for virtual memory and Arm's application-level memory model. It abstracts away the semantics of Arm VMSA by relying on a discipline called wDRF, shown to be followed by an implementation of Linux KVM. Our model provides a step towards proving that the wDRF conditions assumed by [43] hold on Arm hardware.

The work of [32] extends the NVidia model [33] with a notion of proxy to formalise memory accesses going through different coherence paths. A proxy is metadata associating a memory access to a VA and a coherence path (which seems related to an Arm Observer). However, as stated in [32]: *(a)* the NVidia memory model is concerned with the ordering between accesses with different VAs, but not with VA modifications or TLB maintenance; *(b)* there is no pre-established expectation from software, such as the `CopyOnWrite` example of Fig. 11. Hence [32] is at liberty to propose a new barrier to order different proxies, whilst we needed to uncover the semantics of pre-existing barrier and TLB maintenance instructions and validate these against architects' intent, software expectations and existing hardware implementations.

The work closest to ours is in [38], which presents a VMSA extension of the Arm application-level memory model (without Hardware Updates) written in pseudo `cat`.

To validate their model, the authors of [38] have developed tools similar to ours. For simulation, [38] relies on Isla [14], which builds the semantics of Arm instructions from their machine-readable description given in the Arm ARM [30]. Meanwhile herd7 relies on hand-coded semantics devised with and ratified by Arm and its partners. It appears that herd7 is faster than Isla by one order of magnitude [14], but herd7 would necessitate more work by hand to handle radical architecture evolution transcribed through ASL code.

For testing, the custom tool of [38] and the extended litmus7 tool we propose offer similar functionalities, including manipulation of TTDs, custom fault handlers, EL0/EL1 transitions in both directions and possibility of bare-metal execution. But the simulator and testing tool of [38] consume litmus tests in different formats; whereas all our tools consume the same format and produce outputs that we can compare automatically. This guarantees the identity of simulated and executed tests by design, and partly explains that our test base, and as a result our testing campaigns, are significantly larger.

To compare the model of [38] to ours, we implemented it in `cat` to execute it with herd7, and some amount of guess work as to certain relations, undefined in the paper (e.g. trf, IsFrom{W,R}, instruction-order, po-pa) proved necessary. Under those assumptions, the model of [38] differs from ours: it is both stronger on certain tests (e.g. `MP-TTD+DMB.ST+DMB.LD` in Fig. 6, `MP-TTD+DMB.ST+lobF` in Fig. 19 and `coRR-Translation` in Fig. 21, all observed on hardware) and weaker on others (e.g. `MP-TTD+rfi-addr+dmb.ld` in Fig. 9, `CopyOnWrite` in Fig. 11, and `MP-TTD+DSB-TLBI-DSB+DMB.LD` in Fig. 17).

IX. Conclusion and Limitations

We provide a formalisation of the Arm Virtual Memory System Architecture, as a machine-readable and executable model written in `cat` [9], [12]. This model has been validated extensively against hardware, and ratified by Arm—it now appears in the Arm Architecture Reference Manual [30, B2.3].

Our work does not address the matter of Stage 2 translation, which can be used to ensure that a VM can only see resources allocated to it, and not the resources allocated to other VMs or the hypervisor. Instead, our model goes from VA to PA directly, as opposed to through an intermediate address (which Arm calls IPA), as a two-stage translation would. Our work also does not model levels of translation table walks. The Arm architecture chooses a radix tree of three levels as a data structure for a translation table, so translating a VA requires traversing the tree to the leaf nodes. Our VMSA model focuses only on the last step of the traversals. We envision modelling levels and stages of translation with multiple Implicit TTD Reads per translation, and leave that out of scope of this work.

Furthermore, our instruction semantics assumes that accesses have naturally aligned addresses and do not cross page boundaries. Accesses crossing a page boundary for instance may necessitate two translations (viz, Implicit TTD Reads).

The model developed with the aforementioned assumptions has proven successfull at capturing essential specifi-

cation requirements of Arm and Arm partners, which led to its ratification by Arm. Despite the limitations, our work has already helped clarify Arm's intent for the Enhanced Translation Synchronisation feature. Furthermore, our testing infrastructure uncovered exceptions to architectural principles about the Dirty Bit with and without virtualisation. This led Arm to relax the cases with virtualisation, and investigate how to accommodate the cases without virtualisation. This exemplifies how recording the architectural intent in a formal model helps bridging the gap between the intent and its hardware interpretations.

REFERENCES

[1] Kernel Virtual Machine (KVM). https://www.linux-kvm.org/page/Main_Page.

[2] KVM Unit Tests. https://www.linux-kvm.org/page/KVM-unit-tests.

[3] MacOS HVF. https://developer.apple.com/documentation/hypervisor.

[4] QEMU. https://www.qemu.org.

[5] The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. Document Version 20181221- Public-Review-draft.

[6] Vmsa experiment report. https://diy.inria.fr/vmsa/.

[7] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 577–591. ACM, 2015.

[8] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2):8:1–8:54, 2021.

[9] Jade Alglave and Luc Maranget. The diy7 tool suite. http://diy.inria.fr/, 2011–present.

[10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in Weak Memory Models. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2010.

[11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 41–44. Springer, 2011.

[12] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. volume 36, pages 7:1–7:74, New York, NY, USA, July 2014. ACM.

[13] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for armv8-a, risc-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019.

[14] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2021.

[15] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Jesús Mauricio Chimento, and Carlos Luna. Formally verified implementation of an idealized model of virtualization. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 45–63. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[16] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In Michael J. Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2011.

[17] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 186–197. IEEE Computer Society, 2012.

[18] Gilles Barthe, César Kunz, and Jorge Luis Sacchini. Certified reasoning in memory hierarchies. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2008.

[19] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. Revamping hardware persistency models: view-based and axiomatic persistency models for intel-x86 and armv8. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 16–31. ACM, 2021.

[20] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, power, arm, and C++. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019.

[21] Will Deacon. herd: Update aarch64.cat to align with the armv8 memory model. https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7#diff-0461c726950c4454a08bd97fbfd49252.

[22] Ulan Degenbaev, Wolfgang J. Paul, and Norbert Schirmer. Pervasive theory of memory. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2009.

[23] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 608–621, New York, NY, USA, 2016. ACM.

[24] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: Arm, power, c/c++11, and SC. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 429–442. ACM, 2017.

[25] Shilpi Goel. *Formal verification of application and system programs based on a validated x86 ISA model*. PhD thesis. UT Austin, 2016.

[26] Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In Michael G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, editors, *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, pages 173–209. Springer, 2017.

[27] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for linux on ARM. *J. Comput. Secur.*, 24(6):793–837, 2016.

[28] Naorin Hossain, Caroline Trippel, and Margaret Martonosi. Transform: Formally specifying transistency models and synthesizing enhanced litmus tests. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 874–887. IEEE, 2020.

[29] Rafal Kolanski. Verification of programs in virtual memory using separation logic. PhD thesis.

[30] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. https://developer.arm.com/documentation/ddi0487/ja/?lang=en.

[31] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. https://developer.arm.com/documentation/ddi0487/ia/?lang=en.

[32] Daniel Lustig, Simon Cooksey, and Olivier Giroux. Mixed-proxy extensions for the NVIDIA PTX memory consistency model: industrial

product. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 1058–1070. ACM, 2022.

[33] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 257–270. ACM, 2019.

[34] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022.

[35] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186. ACM, 2011.

[36] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 379–391. ACM, 2009.

[37] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[38] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of the 31st European Symposium on Programming, ESOP 2022*, April 2022.

[39] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Armv8-a system semantics: Instruction fetch in relaxed architectures. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 626–655. Springer, 2020.

[40] Hira Taqdees Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 490–508. EasyChair, 2017.

[41] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation, 2018.

[42] Hira Taqdees Syeda and Gerwin Klein. Formal reasoning under cached address translation. *J. Autom. Reason.*, 64(5):911–945, 2020.

[43] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 866–881. ACM, 2021.

[44] Hendrik Tews, Marcus Völp, and Tjark Weber. Formal memory models for the verification of low-level operating-system code. *J. Autom. Reason.*, 42(2-4):189–227, 2009.