This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

Puss In Boots: on formalizing Arm's Virtual Memory System Architecture

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

Abstract—We present our formalization 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. Additionally, we present our experimental validation methodology, which required extending KVM-unit-tests, a test harness for the Kernel Virtual Machine (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 uncovered infidelities to the definition of a feature called Translation Table Hardware Management, which led Arm to relax its architecture to accommodate those cases. Finally, as part of this work, we uncovered subtleties in the definition of a feature of the VMSA called Enhanced Translation Synchronization (ETS), which led Arm to deprecate ETS and replace it with a stronger feature called ETS2.*

Virtual memory is a mechanism giving an abstract view of memory, providing software with more memory than might be physically available and enforcing security boundaries. It is a result of cooperation between hardware and operating systems. The Memory Management Unit (MMU) translates virtual addresses (VAs) to physical addresses (PAs). The mappings from VAs to PAs are managed by the operating system and reside in a collection of page table entries that Arm calls Translation Table Descriptors (TTDs).

Chapter D8 of the Arm Architecture Reference Manual (Arm ARM) provides an informal record of intent of its Virtual Memory System Architecture (VMSA)[1]. We aim to provide a formal basis for reasoning about systems-level code managing TTDs by extending Arm's application-level memory model[2] with essential features of VMSA. Using our model, we illustrate the fact that Translation Lookaside Buffers do not provide any ordering by default, and how to restore ordering when needed—illustrated by a CopyOnWrite example from Linux. We also give an overview of a feature called Translation Table Hardware Management, whereby hardware updates TTDs itself.

As in the case for the Arm application-level model[2], we use the `cat` language[3] to write our formalization. This also enables us to use the herd7 tool[4], a simulator for litmus tests under memory models formalized in `cat`. A litmus test is a small concurrent program with a specific initial state and a question about its final states. The herd7 tool determines which final states can be reached from the initial state specified under that `cat` model. We extended herd7 to run VMSA litmus tests. Hence our VMSA model is an executable artifact, 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 ARM[1]. Second, by testing it extensively on hardware. We extended the litmus7 testing tool, distributed alongside herd7[4], to run VMSA litmus tests. This required extending KVM-unit-tests[5], a test harness for the Kernel Virtual Machine (KVM), to run our tests as virtual machines.

Supplementary materials

In the Appendix, we discuss the Enhanced Translation Synchronization (ETS) feature (as described in the version I.a. of the Arm ARM[1]), which allows lightweight synchronization in specific cases of managing TTDs. We expose the subtleties we uncovered in the definition of ETS, which led Arm to deprecate it in favour of a new feature called ETS2[1].

XXXX-XXX © IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

We release a number of companion artifacts: the VMSA `cat` file itself, our litmus tests, the extensions to the herdtools[6] distribution necessary to handle VMSA, the KVM-unit-tests patches necessary to run VMSA litmus tests on hardware and our testing logs[7], and an extended manuscript[8].

Finally, an English transliteration of the VMSA `cat` file appears in Section B2.3 of the Arm ARM[1].

VMSA-aware instruction semantics

We elaborate the instruction semantics for memory accesses in order to express various outcomes of address translation: a memory access to a VA first must induce an Implicit Memory Effect to a corresponding TTD in order to translate the VA into a PA. Each TTD holds a bit indicating its validity: if the bit is set, there is an Explicit Memory Effect to the PA (translated from its VA); otherwise, the instruction triggers an MMU Translation Fault. (Other kinds of faults may be triggered too; in particular, we discuss MMU Permission Faults in a later section.)

Figure 1 illustrates new Effects associated with the `LDR X1,[X2]` instruction. Its source register `X2` holds address `x`, initilized in memory with 1. Figure 1(i) gives the semantics of `LDR` without VMSA: we read the source register `X2` (Effect *a*), which gives us location `x`. We then read location `x` (Effect *b*), which gives us value 1, and then write that value 1 into the target register `X1` (Effect *c*). The Intrinsic data dependency arrows *a* $\xrightarrow{\text{iico\_data}}$ *b* and *b* $\xrightarrow{\text{iico\_data}}$ *c* depict that Effect *a* hands over some data (viz, the location `x`) to Effect *b* and that Effect *b* provides its value to Effect *c*.

Figure 1(ii) gives the semantics of `LDR` with VMSA: instead of reading `x` immediately after obtaining `x` from register `X2`, we instead read `TTD(x)` (Effect *b*)—Arm calls this an Implicit TTD Read Effect. In this case `TTD(x)` is valid, so from the Effect *b* we proceed to the Read *d* of `PA(x)`—Arm calls this an Explicit Memory Read Effect. The Read *d* gets the value 1 from memory, which is then written into `X1` (Effect *e*).

In Figure 1(ii), the Branching Effect *c* indicates that a decision has been made: depending on the TTD that Effect *b* reads is valid, the instruction proceeds to access `PA(x)` as in Figure 1(ii), or to fault. In contrast, Figure 1(iii) illustrates the case of invalid `TTD(x)`, so there after reading `TTD(x)` at Effect *b*, and checking its validity at Branching Effect *c*, we fault at Effect *d*.

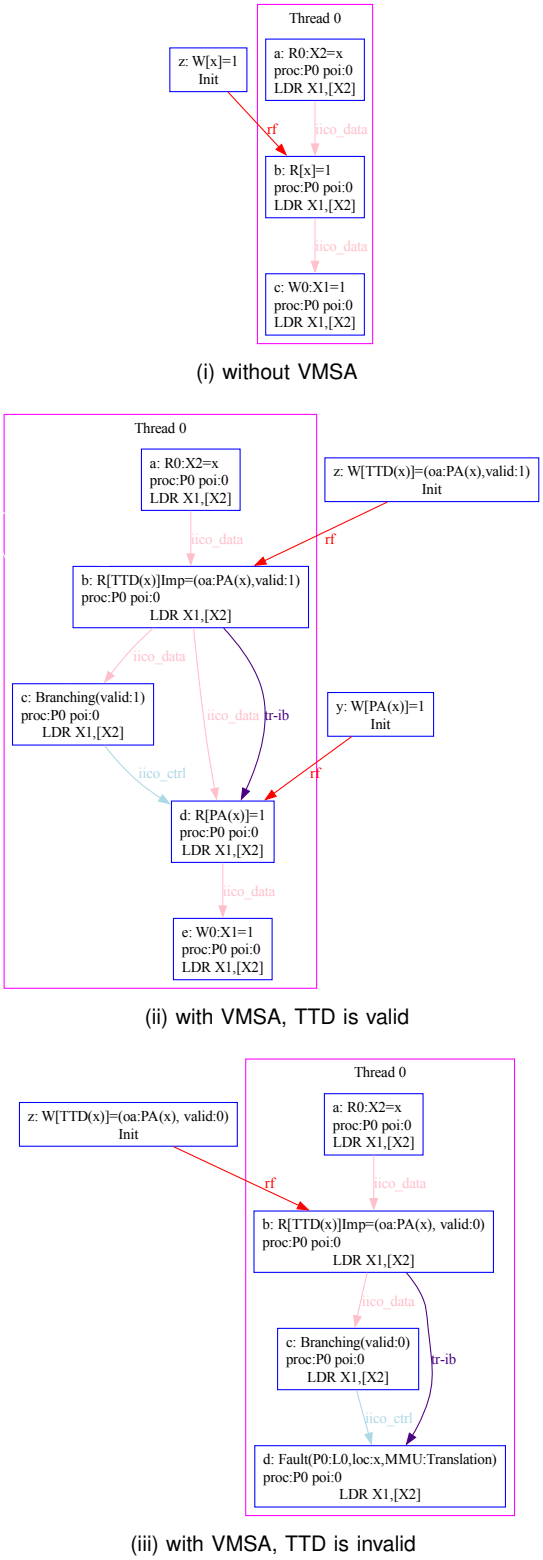


FIGURE 1. Instruction semantics with and without VMSA

Application-level memory model

We use litmus tests to illustrate the memory model, as they can encode questions about the memory or-

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

```
AArch64 coRR
{
0:X2=x; 1:X2=x;
int64_t x=1;
}
P0           | P1          ;
LDR X0,[X2]  | MOV X0,#2   ;
LDR X1,[X2]  | STR X0,[X2];
exists(0:X0=2 /\ 0:X1=1)
```

(i) A load hazarding example—Forbidden

```
AArch64 coRR-Translation
{
[TTD(x)]=(valid:0);
0:X1=(oa:PA(x),valid:1);
0:X2=TTD(x); 1:X3=x;
}
P0           | P1                ;
STR X1,[X2]  | MOV X4,#1         ;
             | L1: STR X4,[X3]  ;
             | L0: STR X4,[X3]  ;
exists ~Fault(P1:L1,x) /\
  Fault(P1:L0,x,MMU:Translation)
```

(ii) A TLBUncacheable example—Allowed

```
AArch64 CopyOnWrite
{
int64_t x=1; int64_t y=0;
[TTD(x)]=(oa:PA(x),valid:1);
[TTD(z)]=(oa:PA(x),valid:1); 2:X7=(valid:0);
0:X2=x; 1:X2=x; 2:X2=x; 2:X4=y; 2:X10=z;
2:X6=TTD(x); 2:X8=TTD(y);
}
P0           | P1           | P2               ;
                            (*unmap VA x      *)
             |              | STR X7,[X6]      ;
                            (*TLB maintenance*)
             |              | DSB ISH          ;
             |              | LSR X3,X2,#12    ;
             |              | TLBI VAAE1IS,X3;
             |              | DSB ISH          ;
                            (*copy data via z*)
             |              | LDR X5,[X10]     ;
             |              | STR X5,[X4]      ;
             |              | DMB ISH          ;
                            (*map x to PA(y) *)
LDR X0,[X2]| MOV X0,#2     | LDR X1,[X8]      ;
LDR X1,[X2]| STR X0,[X2]   | STR X1,[X6]      ;
exists 0:X0=2 /\ 0:X1=1
```

(iii) A `CopyOnWrite` example from Linux—Forbidden

FIGURE 2. Hazarding examples with and without the VMSA context

dering rules. Figure 2(i) presents a `coRR` litmus test illustrating a classic load hazarding behavior. Threads `P0` and `P1` represent instructions executing on the same Processing Element. The initial state in curly brackets indicates that the register `X2` on both `P0` and `P1` holds the address of a location `x`, written `0:X2=x` and `1:X2=x` respectively), and that `x` is initialized to 1, written `int64_t x=1`. Thread `P0` reads from `x` twice (via `LDR X0,[X2]` and `LDR X1,[X2]`). Thread `P1` updates `x` to 2 (via `STR X0,[X2]`). The `exists` clause is an assertion asking if there is an execution of this program where `P0` first reads the updated value of `x` and then the initial one.

A litmus test can be put in correspondence with its concurrent executions, which are graphs whose nodes represent Effects of the instructions in the test such that the memory ordering of those may affect the final state of memory locations and registers. Consider an execution satisfying the `exists` clause, and let *a* denote Explicit Read Effect of `LDR X0,[X2]`, *b* denote Explicit Read Effect of `LDR X0,[X2]`, *c* denote Explicit Write Effect of `STR X0,[X2]` and *d* denote Initial Write Effect initializing `x`. The following relationships between them hold:

- $c \xrightarrow{\text{rf}} a$ (*a* Reads-from *c*), i.e., *a* reads the value written by *c*.
- $d \xrightarrow{\text{rf}} b$ (*b* Reads-from *d*), i.e., *b* reads the initial value of the location *x*.
- $d \xrightarrow{\text{ca}} c$ (*c* is Coherence-after *d*), i.e., *c* overwrites the initial value of the location *x*.
- $b \xrightarrow{\text{ca}} c$ (*c* is Coherence-after *b*), i.e., *c* overwrites the value *b* reads.
- $a \xrightarrow{\text{po}} b$ (*a* is program-order-before *b*), i.e., the loads generating *a* and *b* are in program order.

The Arm application-level memory model determines which concurrent program executions are allowed by the Arm architecture. To this end, it restricts which values a Read Effect may see in a concurrent execution, and does so by ensuring consistency of observations with ordering implications (such as Reads-from or Coherence-after) for the Explicit Effects in an execution with the orderings enforced by hardware (such as due to barriers and dependencies). Together these two kinds of orderings form the Ordered-before relation on Effects, which represents the order in which Effects become visible to other threads, or in other words, externally visible. Formally, the External visibility

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

requirement determines an execution as forbidden if it has cycles in the Ordered-before relation, and allowed otherwise.

The Arm memory model forbids the `coRR` test, meaning that it forbids all executions satisfying its `exists` clause. When $a \xrightarrow{po} b \xrightarrow{ca} c$, a is considered Ordered-before c (as captured by Explicitly-Hazard-ordered-before relation). However, $c \xrightarrow{rf} a$ is an observation meaning that c is Ordered-before a. Thus, the execution induces a cycle in Ordered-before, and therefore violates the External visibility requirement of Arm's memory model.

Address translation in presence of concurrent modifications of TTDs

Handling TTDs efficiently necessitates Translation Lookaside Buffers (TLBs): special caches to store VA-to-PA translations. The effects of caching in TLBs can be managed by system-level software operating at a more privileged level that Arm calls Exception Level 1 (EL1); if not managed correctly, they may become visible to application-level software (EL0) and undermine the virtual memory properties. Our formalization captures the ordering guarantees of the Arm architecture and enables checking sufficiency of TLB maintenance.

In the Arm memory model, the Coherence-after relation provides order for Explicit Memory Effects—as a consequence of cache coherence. As TTDs change infrequently, it is pragmatic for hardware not to enforce coherence of TLBs. Architecturally, some translations cannot be cached in a TLB (e.g., if they are invalid); Arm calls TTDs corresponding to those TLBUncacheable. We model this nuance in our formalization by letting Coherence-after from Implicit TTD Effects contribute to Ordered-before only if they are TLBUncacheable or if they are affected by a `TLBI` instruction.

Consider the `coRR-Translation` test in Figure 2(ii). Initially, `TTD(x)` is not valid. Thread `P0` overwrites it with `(oa:PA(x),valid:1)`, i.e., a valid TTD with an output address `PA(x)`. Let the corresponding Explicit Write Effect be a, and Implicit TTD Read Effects of stores at lines `L1` and `L0` be b and c, respectively. The `exists` clause considers an execution where the `STR` at line `L1` does not trigger a fault and the `STR` at line `L0` does. To satisfy that, Implicit TTD Read Effect c must read the initially invalid `TTD(x)`, so it is TLBUncacheable. Overall, we have $a \xrightarrow{rf} b$ and $c \xrightarrow{ca} a$ both contributing to Ordered-before. However, in absence of synchronization for Effects b and c, there is no cycle in this execution, so it is allowed.

The missing order can be enforced in three ways. Firstly, a combination of `DSB ISH` and `ISB` barriers can be used on Thread `P1`. These may be costly and undesirable in otherwise VMSA-agnostic application-level code. Secondly, Arm provides an instruction called `TLBI` that ensures TLBs are not out-of-date and enforces synchronization w.r.t. concurrent address translations. The next example illustrates our modelling of `TLBI`. Thirdly, and specifically for the TLBUncacheable cases like in `coRR-Translation`, Arm provides a feature of the architecture called *Enhanced Translation Synchronization (ETS)* offering stronger ordering guarantees. Please refer to the supplementary materials for further detail.

CopyOnWrite example from Linux

A typical TTD manipulation sequence appears in Figure 2(iii), a `CopyOnWrite` excerpt of the Linux kernel given to us by its maintainers. Threads `P0` and `P1`, as well as the `exists` clause of the test, coincide with the `coRR` example. They represent user-level instructions occurring in a wider context of the OS manipulating TTDs: Thread `P2` remaps VA `x` to a different PA and copies the data over. To provide the virtual memory abstraction to Threads `P0` and `P1` in presence of concurrent modification of TTDs, Thread `P2` employs a so called *break-before-make* sequence:

- `STR X7,[X6]` overwrites `TTD(x)`, initialized as mapping the VA `x` to `PA(x)`, with an invalid TTD, which makes it TLBUncacheable;
- `DSB ISH` is a strong barrier that orders `STR X7,[X6]` ahead of `TLBI VAAE1IS,X3`;
- `LSR X3,X2,#12` performs a logical shift right to the VA `x`, removing 12 bits identifying the address within the page, but leaving the other bits identifying `TTD(x)`;
- `TLBI VAAE1IS,X3` invalidates out-of-date values of `TTD(x)` cached in TLB;
- `DSB ISH` orders `TLBI VAAE1IS,X3` ahead of the subsequent Explicit Memory Effects;
- `LDR X5,[X10]` reads from `PA(x)` using an alias VA `z`, whose `TTD(z)` maps it to `PA(x)`;
- `STR X5,[X4]` writes the just-read value into VA `y`, whose `TTD(y)` maps it to `PA(y)`;
- `DMB ISH`, which is a barrier to ensure the data is copied over ahead of the subsequent Explicit Memory Effects;
- `LDR X1,[X8]` and `STR X1,[X6]` copy `TTD(y)` into `TTD(x)`, mapping VA `x` to `PA(y)`.

The `CopyOnWrite` test is forbidden thanks to the break-before-make sequence: it ensures the TLB maintenance on Thread P2 not only invalidates out-of-date TLB entries, but also synchronizes with Threads

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

P0 and P1. At the hardware implementation level, the `TLBI` instruction performs a synchronizing broadcast: it sends a message to other threads and only proceeds to executing after acknowledging a response that active uses of affected address translations are resolved. We model this by enumerating every pair of a TLBI Effect and an Implicit TTD Read Effect in the same invalidation scope, and considering two executions for the two orders in which the Effects could be ordered. The TLBI-before relation denotes this communication order and enables reasoning formally as follows.

Consider executions satisfying the `exists` clause. Let *a* and *c* denote the Implicit TTD Reads of `LDR X0,[X2]` and `LDR X1,[X2]`; let *e* and *k* denote the Explicit Writes of `STR X7,[X6]` and `STR X1,[X6]`; and *z* denote the Effect of `TLBI VAAE1IS,X3`. Then we have:

- $a \xrightarrow{\text{po-va-loc}} c$, i.e. `LDR X0,[X2]` is to the same VA and in program order before `LDR X1,[X2]`.
- $k \xrightarrow{\text{rf}} a$, i.e. the Implicit TTD Read Effect *a* of `LDR X0,[X2]` on `P0` reads-from the Explicit Write Effect *k* of `STR X1,[X6]` on `P2`.
- $c \xrightarrow{\text{ca}} e$, i.e. *c* reads the initial value of TTD overwritten by Explicit Write Effect *e*.
- $e \xrightarrow{\text{DSB-ob}} z$, i.e. *e* is Ordered-before *z* thanks to `DSB ISH` between `STR X7,[X6]` and `TLBI`.

The first execution has $z \xrightarrow{\text{TLBI-before}} c$ denoting that the `TLBI` message arrives before the Implicit TTD Read *c*. This creates a cycle forbidden by the External Visibility requirement: $z \xrightarrow{\text{TLBI-before}} c \xrightarrow{\text{ca}} e \xrightarrow{\text{DSB-ob}} z$. Therefore, the execution is forbidden.

The second execution has $c \xrightarrow{\text{TLBI-before}} z$ denoting that the `TLBI` message does not arrive before the Implicit TTD Read *c*—and so it ensures *c* and the Implicit TTD Read Effects to the same VA earlier in program order (such as *a*) result in a Explicit Memory or a Fault Effect before Thread P2 proceeds past the second `DSB` ensuring completion of the `TLBI`, in particular, before Explicit Write Effect *k* of a subsequent in program order `STR X1,[X6]`. We display this causal chain in the execution using a TLBI-ob relation, according to which $c \xrightarrow{\text{TLBI-ob}} k$ contributes to Ordered-before and, since $a \xrightarrow{\text{po-va-loc}} c$ holds, so does $a \xrightarrow{\text{TLBI-ob}} k$. This way `TLBI` creates a cycle forbidden by External Visibility: $k \xrightarrow{\text{rf}} a \xrightarrow{\text{po-va-loc}} c \xrightarrow{\text{TLBI-ob}} k$. Therefore, the execution is forbidden.

Translation Table Hardware Management

Translation Table Hardware Management (TTHM) is a feature of the Arm architecture introduced in v8.1, to let hardware update permissions in TTDs and relieve software from this responsibility. Under TTHM the hardware is expected to update bits of a TTD called Access Flag and Dirty Bit, which represent read and write permissions on TTDs respectively. Depending on whether two bits are set in the special Translation Control Register, `TCR_ELx.HA` for the Access Flag and `TCR_ELx.HD` for the Dirty Bit, the hardware manages TTD permissions differently.

We model a TTD value as a tuple `(oa, valid, af, db, dbm)` of an output address `oa`, a validity bit `valid` an Access Flag bit `af`, a Dirty Bit `db`, and a Dirty Bit Management bit `dbm` indicating if Hardware Updates of the Dirty Bit are enabled. In litmus tests, our convention is that the default value of `TTD(x)` is `(oa:PA(x), valid:1, af:1, db:1, dbm:0)`, and that fields take default values whenever omitted.

We outline several cases of how our formalization extends the instruction semantics with permission checks in addition to previously exposed validity checks.

When `TCR_ELx.{HA,HD} == {0,0}` or TTHM is not implemented:

- If `TTD(x)` has `af` unset, a memory access to VA `x` raises a Permission Fault;
- If `TTD(x)` has `db` unset, a store to VA `x` raises a Permission Fault. Software is expected to manage permissions with store or atomic instructions to `TTD(x)`.

When `TCR_ELx.{HA,HD} == {1,0}`:

- If `TTD(x)` has `af` unset, a memory access to VA `x` continues without a Fault and the MMU sets `af`;
- If `TTD(x)` has `db` unset, a store to VA `x` raises a Permission Fault.

When `TCR_ELx.{HA,HD} == {1,1}`:

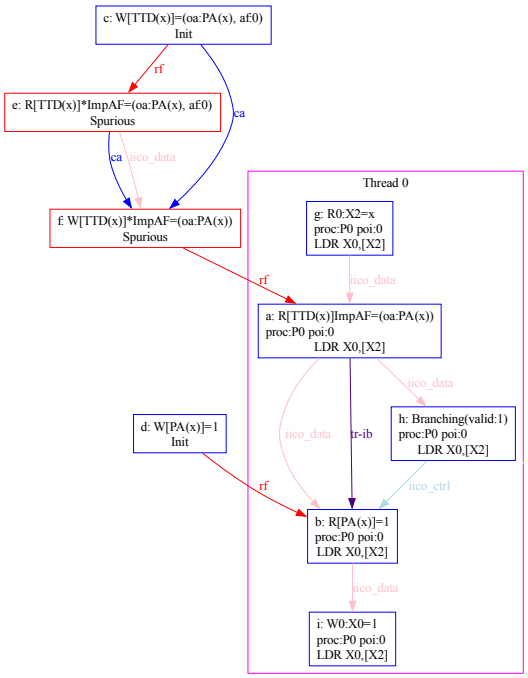
- If `TTD(x)` has `af` unset, a memory access to VA `x` continues without a Fault and the MMU sets `af`;
- If `TTD(x)` has `db` and `dbm` unset, a store to VA `x` raises a Permission Fault;
- If `TTD(x)` has `db` unset, but `dbm` set, a store to VA `x` continues without a Fault and the MMU sets `db`.

To enable studying the listed behaviors, litmus tests may specify if the `TCR_ELx.{HA,HD}` bits are set (by default they are not).

The test in Figure 3(i) specifies that TTHM is implemented and that `TCR_ELx.HA==1` using keywords `TTHM=HA`. Initially `x` is 1, and `TTD(x)` has `af` unset.

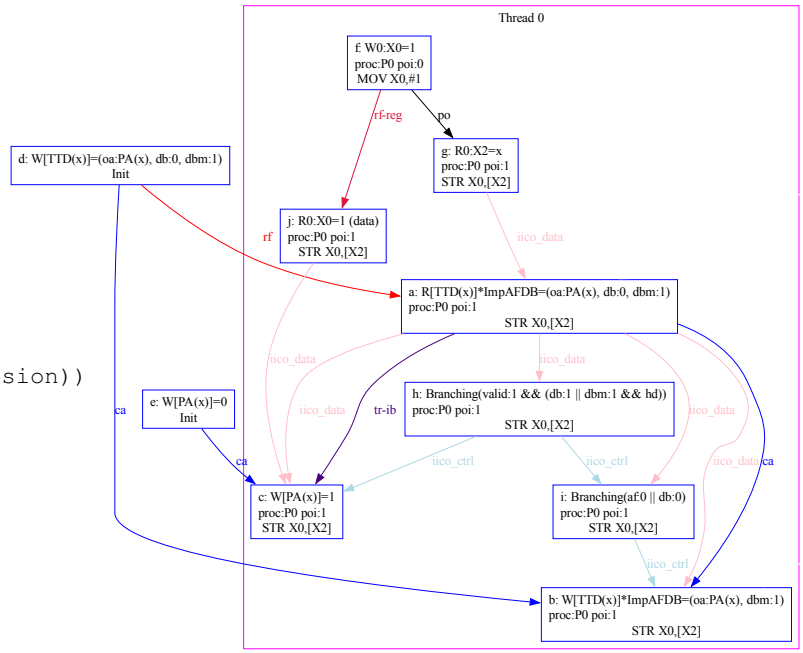
This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

```
AArch64 LDRaf0
TTHM=HA
{
0:X2=x;
int64_t x=1;
[TTD(x)]= (af:0);
}
P0                ;
L0: LDR X0,[X2] ;
exists 0:X0=1 /\ [TTD(x)]=(af:1)
/\ ~Fault(P0:L0,x,MMU:Permission)
```



(i) `LDR` with the Access Flag unset

```
AArch64 STRdb0dbm1
TTHM=HA HD
{
0:X2=x;
[TTD(x)]=(db:0,dbm:1);
}
P0                ;
MOV X0,#1         ;
L0: STR X0,[X2] ;
exists(x=1 /\
  [TTD(x)]=(db:1,dbm:1) /\
  ~Fault(P0:L0,x,MMU:Permission))
```



(ii) `STR` with the Dirty Bit unset and Dirty Bit Management on

FIGURE 3. Instruction semantics with TTHM

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

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 is no Fault at line L0 relative to x. In the execution, also in Figure 3(i), we see the atomic pair (*e*, *f*) setting the af bit. This represents the hardware management of the Access Flag.

The Arm architecture permits *spontaneous* hardware updates to Access Flags, that is, without any ordering relation to corresponding memory accesses. In Figure 3(i), our formalization does not require that the Hardware Update *f* of af be Ordered-after any other Effect besides the Read *e* of the TTD.

The test in Figure 3(ii) specifies that TTHM is implemented and that TCR_ELx.HD == 1 using keywords TTHM=HA HD. Initially x is 0, db is unset and dbm is set. At line L0 thread P0 attempts a store of 1 to x. The exists clause asks if x holds the value 1, db is set and there is no Fault at L0 relative to x. In the execution, also in Figure 3(ii), we see the atomic pair (*a*, *b*) setting the db bit. This represents the hardware management of the Dirty Bit.

The Arm architecture does not permit spontaneous hardware updates to Dirty Bits, as certain OSes rely on that. In Figure 3(ii), our formalization requires that the Hardware Update *k* of db be 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[1] (clause R_LHQRX in D8.4.5 for af, and clause R_DYCFD in D8.4.6 for db).

Validating our model

We validated the Arm VMSA memory model by discussing with Arm architects and Arm partners, and by testing extensively against existing hardware.

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 Figures 1 and 3 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 behavior, both from hardware implementation and software usage point of views.

Third, after having developed an initial model which matched the recorded intent, we discussed the definitions themselves. Much reorganization 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 Chapter B2.3 of the Arm Architecture Reference Manual[1], and released in the herdtools distribution[4].

Testing against a variety of Arm hardware

We tested the Arm VMSA model extensively against existing hardware[7]. 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[4] 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 EL2: 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 and MacOS HVF—using QEMU. The source code generated by litmus7 uses the library functions provided by KVM-unit-tests to setup the system, get pointers to TTDs and install fault handlers as necessary. For the purposes of this work, we added support for configurable translation granule for the arm64 target in KVM-unit-tests[5,6].

Our testing uncovered infidelities to the principle that the Dirty Bit must not be updated spontaneously by hardware. More precisely, we observed 32 contradicting tests; Figure 4 shows one of them.

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

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

an operating system. Our work is open-source and we have already worked with the community to upstream the first series of patches to KVM-unit-tests[5,6].

Interestingly, we observed tests contradicting the Dirty Bit principle (e.g. Figure 4) in our bare-metal infrastructure. Some designs were confirmed to allow such behaviors, and relaxations to the architecture were introduced (see clause R_DYCFD in Chapter D8.5[1]).

Related works

Previous work on application-level memory models, including the Arm application-level memory model[2], paved the way for the methodology that we use here: writing executable models validated both by discussion with architects and testing.

A model of virtual memory called x86t_elt[9] extends the x86-TSO memory model in a style similar to ours. Empirical validation of x86t_elt remains open and could be approached with our testing methodology.

A VRM verification framework[10] for system-level software accounts 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 VRM hold on Arm hardware.

The work of Simner et al.[11] is closest to ours. It presents a VMSA extension to the Arm application-level memory model (without Hardware Updates) written in pseudo `cat`. To validate their model, the authors have developed tools similar to ours. For simulation, they rely on Isla[12], which builds the semantics of Arm instructions from their machine-readable description given in the Arm ARM[1]. 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[12], but herd7 would necessitate more work by hand to handle radical architecture evolution transcribed through ASL code.

For testing, the custom tool of Simner et al.[11] and our extended litmus7 tool offer similar functionalities: including manipulation of TTDs, custom fault handlers, EL0/EL1 transitions in both directions and possibility of bare-metal execution. However, the simulator and testing tool of Simner et al.[11] 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 Simner et al.[11] 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 Simner et al.[11] differs from ours: it is both stronger on certain tests (e.g., `coRR-Translation` in Fig. 2(ii)) and weaker on others (e.g., `CopyOnWrite` in Fig. 2(iii)).

```
AArch64 STRva-SWPttd
TTHM=HA HD
{
0:X2=x;
1:X4=TTD(x);
1:X6=(oa:PA(x),valid:0,db:0,dbm:1);
[TTD(x)]=(db:0,dbm:1);
}
P0              | P1              ;
MOV X1,#1       | SWP X6,X8,[X4]  ;
L0:STR X1,[X2]  |                 ;
exists (1:X8=(oa:PA(x), dbm:1)
/\ [TTD(x)]=(oa:PA(x),valid:0,db:0,dbm:1)
/\ [x]=0 /\ Fault(P0:L0,x,MMU:Translation))
```

FIGURE 4. Dirty Bit set spontaneously—Forbidden?

Conclusion and Limitations

We provide a formalization of the Arm Virtual Memory System Architecture, as a machine-readable and executable model written in `cat`[3,4]. This model has been validated extensively against hardware, and ratified by Arm—it now appears in Chapter B2.3 of the Arm Architecture Reference Manual[1].

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.

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).

Despite these limitations, our work has already helped clarify Arm's intent for the Enhanced Translation Synchronization feature. Furthermore, our testing infrastructure uncovered exceptions to architectural prin-

This article has been accepted for publication in IEEE Micro. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MM.2024.3422668

ciples about the Dirty Bit with and without virtualization. This led Arm to relax the cases with virtualization, and investigate how to accommodate the cases without virtualization. This exemplifies how recording the architectural intent in a formal model helps bridging the gap between the intent and its hardware interpretations.

REFERENCES

1. Arm Ltd, "Arm Architecture Reference Manual (for A-profile architecture)", Version K.a. https://developer.arm.com/documentation/ddi0487/ka/ (URL)
2. Will Deacon. "herd: Update aarch64.cat to align with the armv8 memory model." https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7#diff-0461c726950c4454a08bd97fbfd49252 (URL)
3. Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory". *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014. (Journal)
4. Jade Alglave and Luc Maranget. "The diy7 tool suite". http://diy.inria.fr/. (URL)
5. "KVM Unit Tests". https://www.linux-kvm.org/page/KVM-unit-tests (URL)
6. Jade Alglave, "Initial formalisation of the VMSA". https://github.com/herd/herdtools7/commit/c3f916c21c2265dd09bac6eb0c0e6b8bfc97ebd4 (URL)
7. "AArch64 VMSA experiment report". https://diy.inria.fr/vmsa/. (URL)
8. Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. "Puss In Boots: on formalizing Arm's Virtual Memory System Architecture". *Extended version.* https://hal.science/hal-04567296v1 ((Unpublished manuscript))
9. Naorin Hossain, Caroline Trippel, and Margaret Martonosi. "TransForm: Formally Specifying Transistency Models and Synthesizing Enhanced Litmus Tests". *International Symposium on Computer Architecture (ISCA)*, 2020. (Conference proceedings)
10. 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". *Symposium on Operating Systems Principles (SOSP)*, 2021.(Conference proceedings)
11. Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. "Relaxed virtual memory in Armv8-A". *European Symposium on Programming (ESOP)*, 2022. (Conference proceedings)
12. Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. "Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models". *Computer Aided Verification (CAV)*, 2021. (Conference proceedings)

Jade Alglave is a Fellow at Arm (Cambridge, England CB1 9NJ, UK) and a Professor of Computer Science at University College London (England WC1E 6EA, UK). Her research interests include concurrency, semantics and formal specifications. Alglave received her PhD in Computer Science from INRIA and Universite Paris-Diderot in France. She is a Fellow of the Royal Academy of Engineering. Contact her at `jade.alglave@arm.com` or `j.alglave@ucl.ac.uk`.

Richard Grisenthwaite is Chief Architect at Arm (Cambridge, England CB1 9NJ, UK). His research interests include the Arm computer architecture, computer security and formalization of memory ordering. Grisenthwaite received his BA in Electrical & Information Sciences from University of Cambridge. He is a member of the IEEE. Contact him at `richard.grisenthwaite@arm.com`.

Artem Khyzha is a Principle Modelling Engineer at Arm (Cambridge, England CB1 9NJ, UK). His research interests include memory consistency models and multicore programming. Khyzha received a joint PhD degree in Software and Systems from IMDEA Software Institute and Technical University of Madrid. Contact him at `artem.khyzha@arm.com`.

Luc Maranget is a researcher at Inria at Paris, France. His research interests include formal models, concurrency, functional programming. Maranget received his PhD in Computer Science from University Paris 7. Contact him at `luc.maranget@inria.fr`.

Nikos Nikoleris Nikos Nikoleris is a Senior Principal Engineer at Arm (Cambridge, England CB1 9NJ, UK). His research interests include concurrency and system architecture. Nikoleris received his PhD in Computer Science from Uppsala University in Sweden. Contact him at `nikos.nikoleris@arm.com`.