Relaxed virtual memory in Armv8-A (extended version)

Ben Simner[1] Alasdair Armstrong[1] Jean Pichon-Pharabod[2]
Christopher Pulte[1] Richard Grisenthwaite[3] Peter Sewell[1]

[1] University of Cambridge, UK first.last@cl.cam.ac.uk
[2] Aarhus University, Denmark jean.pichon@cs.au.dk
[3] Arm Ltd., UK

March 17, 2022

Abstract

Virtual memory is an essential mechanism for enforcing security boundaries, but its relaxed-memory concurrency semantics has not previously been investigated in detail. The concurrent systems code managing virtual memory has been left on an entirely informal basis, and OS and hypervisor verification has had to make major simplifying assumptions.

We explore the design space for relaxed virtual memory semantics in the Armv8-A architecture, to support future system-software verification. We identify many design questions, in discussion with Arm; develop a test suite, including use cases from the pKVM production hypervisor under development by Google; delimit the design space with axiomatic-style concurrency models; prove that under simple stable configurations our architectural model collapses to previous "user" models; develop tooling to compute allowed behaviours in the model integrated with the full Armv8-A ISA semantics; and develop a hardware test harness.

This lays out some of the main issues in relaxed virtual memory bringing these security-critical systems phenomena into the domain of programming-language semantics and verification with foundational architecture semantics.

This document is an extended version of a paper in ESOP 2022, with additional explanation and examples in the main body, and appendices detailing our litmus tests, models, proofs, and test results.

Contents

1 Introduction

Computing relies on virtual memory to enforce security boundaries: hypervisors and operating systems manage mappings from virtual to physical addresses to restrict access to physical memory and memory-mapped devices, and thereby to ensure that processes and virtual machines cannot interfere with each other, or with the parent OS or hypervisor. In a world with endemic use of memory-unsafe languages for critical infrastructure, and of hardware that does not enforce fine-grained protection, virtual memory is one of the few mechanisms one has to enforce strong security guarantees. This has driven interest in hypervisors and virtual machines, and it provides a compelling motivation for verification of the OS-kernel and hypervisor code that manages virtual memory to provide security.

However, any such verification requires a semantics for the protection mechanisms provided by the underlying hardware architecture. There are two major challenges in establishing such a semantics. First, there is its *sequential intricacy*: virtual memory is one of the most complex aspects of a modern general-purpose architecture. For 64-bit Armv8-A (AArch64) it is described in a 166-page chapter of the prose reference manual [13, Ch.D5] and includes a host of features and options. Second, and more fundamentally, there is its *relaxed memory behaviour*. Hardware implementations of virtual memory use in-memory representations of the virtual-to-physical address mappings, represented as hierarchical page tables. For performance, there are dedicated cache structures for commonly used mapping data, in Translation Lookaside Buffers (TLBs). Translations are used often – a single load instruction might need 40 or more page-table entries to translate its fetch and access addresses – but they are changed only rarely, and by systems code not user code. Architectures therefore require manual management of TLB caching, e.g. with specific instructions to invalidate old TLB entries that should no longer be used, instead of providing the simpler coherent memory abstraction that they do for normal accesses. All this gives rise to new relaxed-memory effects, with subtle constraints determining when translations are required or forbidden to read from specific writes to the page tables, and systems code has to handle these appropriately to provide the desired virtual-memory abstraction and its security properties.

Previous work has developed hand-written sequential semantics for some aspects of address translation in Arm [58, 60, 59, 61, 44, 38, 41] and x86 [34, 35, 29, 63], but these are at best lightly validated formalisations, and there is no well-validated relaxed-memory concurrency semantics of virtual memory. In the absence of that (and of proof techniques above it), previous OS and hypervisor verification work, e.g. on seL4, CertiKOS, KCore, Hyper-V, the PROSPER hypervisor, and SeKVM [25, 40, 37, 44, 11, 38, 43, 62] has had to make major simplifying assumptions, either assuming correctness of TLB management and a single-threaded setting (seL4), or assuming sequentially consistent concurrency with one of those hand-written sequential semantics, or assuming an extended notion of data-race-freedom (we return to the related work in §9).

We explore the design space for Armv8-A relaxed virtual memory semantics, to support future systems-software verification. We contribute:

- A description of the current Arm architectural intent as we understand it, and a set of design questions and issues arising from its relaxed virtual memory semantics (§3).
- A relaxed virtual memory test suite, comprising of a set of hand-written litmus tests which illustrate the aforementioned design questions and capture key use cases from pKVM, a production hypervisor under development by Google (§4).
- An axiomatic-style concurrency model for relaxed virtual memory in Armv8 (§5), which to the best of our knowledge and ability captures the architectural intent described in §3. We also define a weaker model, motivated by the properties pKVM relies on.
- We prove that, for stable injective page-tables, the first model collapses to the previous Armv8-A user-mode concurrency model (§6).

- We extend our Isla tool [15], enabling it to compute the allowed behaviours of virtual memory litmus tests with respect to arbitrary axiomatic models, using the authoritative Arm ASL definition of the intra-instruction semantics including pagetable walks (§7).
- We develop a test harness that lets us run virtual-memory litmus tests bare-metal, albeit currently only for Stage 1 tests, and report results from running these on hardware (§8).

We begin in §2 with an informal introduction to virtual memory in a simple sequential setting, to make this as self-contained as possible, but familiarity with virtual memory from a systems perspective, and with previous work on user-space relaxed memory, will be helpful.

Mainstream industrial architecture specifications evolve over many years, balancing hardware-implementation and systems-software concerns. Experience with "user" relaxed-memory concurrency has shown that the process of developing rigorous semantics for arbitrary code provides a useful third input into this process, leading one to ask questions which help clarify the architectural intent. The architects, hardware designers, and system-software authors typically have a deep understanding of the area, but there is usually not, *a priori*, a well-understood informal specification that just needs to be formalised; instead that needs to be iteratively and collaboratively developed. Our §3 is based on detailed discussion with the Arm Chief Architect (a co-author of this paper); the current Arm prose documentation [13]; discussion with the pKVM development team; and our experimental testing. To the best of our knowledge, our models provide a reasonable basis for software development and for verification, but this paper is surely not the last word on the subject, and it does not give an authoritative definition of the Armv8-A architecture. The history of relaxed-memory models shows that it typically takes multiple years, and gradual refinement of models, to converge on something reasonably stable for a production architecture or language, and even then they continue to change as new knowledge or features arise; with hindsight, few are definitive. Our goal here is rather to lay out some of the main issues, bringing this security-critical systems code into the domain of programming-language semantics and verification, above foundational architecture semantics.

This document is an extended version of a paper in ESOP 2022 [56], with additional explanation and examples in the main body, and appendices detailing our litmus tests (A), models (B), proofs (C), and test results (D). Further details are at `https://www.cl.cam.ac.uk/users/pes20/RelaxedVM-Arm/`.

Scope and non-goals

Our scope is Armv8-A virtual memory for the 64-bit (AArch64) architecture, aiming especially to support aspects relevant to hypervisors such as pKVM. Accordingly, we consider translation with multiple stages (for both hypervisor and OS), multiple levels, and the full Armv8-A intra-instruction semantics and translation walk behaviour (as defined by Arm in ASL and auto-translated to Sail [14]). Our models cover the Armv8-A ETS option as work in progress. We discuss some mixed-size aspects, but our models do not currently cover them. To keep things manageable, we do not consider hardware management of access flags or dirty bits, conflict aborts, FEAT_BBM, FEAT_CNP, FEAT_XS, the interactions between virtual memory and instruction-fetch, or all the relaxed behaviour of exceptions, and we handle only some of the many varieties of the TLBI instruction. We focus on the specification of the architecturally allowed envelope of functional behaviour, not on side-channel phenomena. We include some experimental testing, as a sanity check of our models, but our principal goal is to capture the architectural intent, and our principal validation is from discussion with Arm. Many of the issues should also be relevant to other architectures, but here we address only Armv8-A.

2 Background: A crash course on virtual memory

2.1 Virtualising addressing

In conventional computer systems, the underlying memory is indexed by *physical addresses* (PAs), as are memory-mapped devices. For a small microcontroller running trusted code, accessing resources directly via physical addresses may suffice. Larger systems rely heavily on virtual addressing: they interpose one or more layers of indirection between *virtual addresses* (VAs) used by instructions and the underlying physical addresses. This lets them:

1. partition resources among different programs, giving each access only to those it needs;
2. provide convenient numeric ranges of virtual addresses to each program; and
3. dynamically extend and change the mapping from virtual to physical addresses, e.g. to support copy-on-write or swapping, or shared buffers.

A simple system might have many processes managed by an operating system, each of which (including the OS) has a partial function that gives the physical address and permissions for the virtual addresses it can use, roughly:

$$\texttt{translate} : \texttt{VirtualAddress} \rightharpoonup \texttt{PhysicalAddress} \times 2^{\{\texttt{Read},\texttt{Write},\texttt{Execute}\}}$$

Typically each process would have access to a subset of the physical addresses (the range of its translate function), disjoint from those of the other processes and from that of the OS, while the OS would have sole access to its own working memory and also access to that of the processes. This is implemented with a combination of hardware and system software. The hardware memory management unit (MMU) automatically translates virtual to physical addresses when doing an access needed to execute an instruction. If the function is undefined, the instruction traps with a page fault; if it is defined but does not have the appropriate accesses, it traps with a permission fault; and if it is defined with the right permissions, the hardware performs the required access using the resulting physical address. The OS has to set up the translate functions, ensure that the appropriate function is used when switching to a new process, and handle those faults. In general translation functions are not necessarily injective, and includes not just access permissions (which can moreover vary between exception levels), but also additional fields for cacheability, shareability, security, contiguity, and other aspects which we elide for simplicity here.

2.2 The translation-table walk

The current translate function for execution is determined by a system register, a *translation table base register* or `TTBR`, that contains the physical address of a lookup-tree data structure in memory. The details of this structure are (in Armv8-A) highly configurable, e.g. for different page sizes, controlled by various system registers. In a common configuration used by Linux, it maps 4096-byte pages and has a tree up to four levels (0–3) deep. We assume this configuration for the remainder of this section.

Each node in the tree is a 4096-byte block of memory made up of 512 64-bit entries (called "descriptors" by Arm).

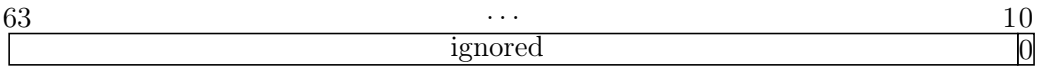
These descriptors are of various types, either: *invalid*, indicating that this part of the domain is unmapped; a *block* or *page* descriptor, defining a fixed-size mapping to a range of output addresses; or a *table* descriptor which points to another level of table for this part of the domain.

The least significant two bits of the descriptor define what type the descriptor is, and the other bits are partitioned into various fields depending on the type:

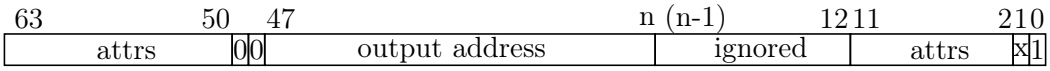
- Output address (OA): the page the final output (IPA or PA) address is in.

- Table pointer: a 4k-aligned pointer to the next-level translation table.
- Attrs: encoding of the access permissions, memory attributes, shareability, access bits and dirty flags.

Invalid descriptors:



Block or page descriptors:



Here n depends on how deep in the table this entry is: for a level 1 block descriptor $n == 30$, for a level 2 it is 21, and for level 3 it is 12. Note that bit 1 should be set when at level 3 (a page descriptor), otherwise it is 0 for block descriptors.

Table descriptors:

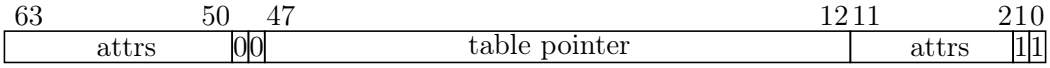
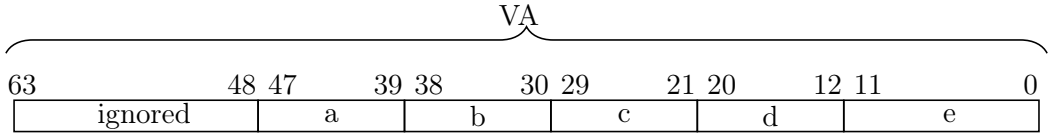


Table descriptors are allowed only at levels 0–2.

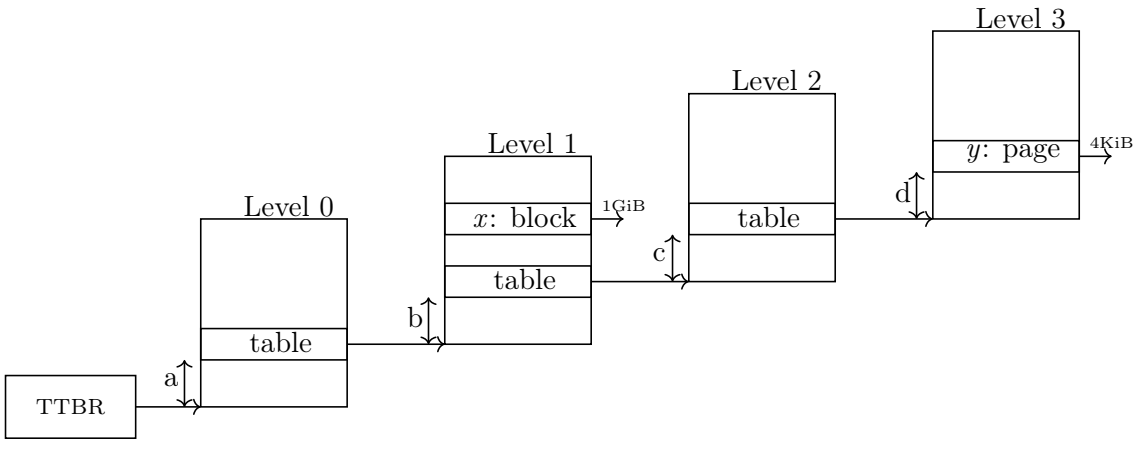
Arm's translation-table walk: The sequential behaviour of Arm's *translation-table walk* function is fully defined in the Arm ASL language.

The hardware walker first splits up the input virtual address into chunks: the upper 16 bits are typically ignored; fields *a-d* are used for indexing into the tables; and field *e* is added to the final result to get the physical address.



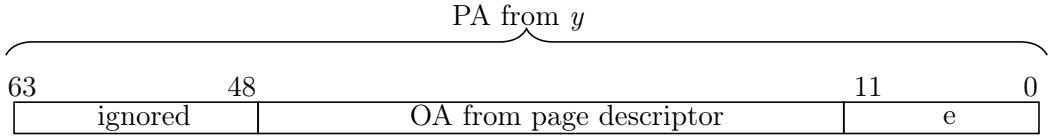
A pointer to the initial level of translation is obtained by reading the relevant translation table base register (`TTBR`). The fields `a-d` are then used to indirect into each table in turn, until a block (or page) mapping is found.

Each level of the tree maps a different size block. e.g. in a common configuration each level 1 entry maps a 1GiB region, each level 2 a 2MiB region, and each level 3 a single 4KiB page.

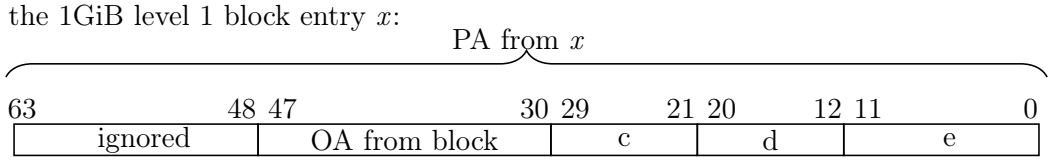


The final physical address is the output address field of the page or block mapping, with the remaining bits of the VA appended.

For example, if the VA was translated using the 4KiB page entry y from above:



Or for the 1GiB level 1 block entry x:



Note that, as mentioned before, the architecture is highly configurable and the above diagrams give just a common configuration. Various system registers allows the user to configure: the size of the input addresses (e.g. 48 bit, 52 bit); the number of translation table levels (e.g. 3, 4, 5); the size of a page (e.g. 4K, 16K, 64K), and much more that we elide here for brevity.

2.3 Multiple stages of translation

The above suffices for an operating system isolating multiple processes from each other, but one often wants to isolate multiple operating systems (or other guests), managed by a hypervisor. To support this, the architecture provides a second layer of indirection: instead of going straight from virtual to physical addresses, with a single *stage* of mapping controlled by the OS, one can have two stages, with the OS managing a Stage 1 table which maps virtual addresses to an *intermediate physical addresses* (IPAs), composed with a hypervisor-managed Stage 2 table, mapping IPAs to PAs. The full translation composes the two, intersecting their permissions.

$$\texttt{translate\_stage1} : \texttt{VirtualAddress} \rightharpoonup \texttt{IPA} \times 2^{\{\texttt{Read},\texttt{Write},\texttt{Execute}\}}$$
$$\texttt{translate\_stage2} : \texttt{IPA} \rightharpoonup \texttt{PhysicalAddress} \times 2^{\{\texttt{Read},\texttt{Write},\texttt{Execute}\}}$$

Armv8-A has various *exception levels* (ELs), determined by the `PSTATE.CurrentEL` register, including EL0 (for user processes), EL1 (for OSs or other guests), and EL2 (for a hypervisor). These each have associated translation-table base registers:

- `TTBR0_EL1`: contains a pointer (IPA) to the Stage 1 table for EL1&0, lower VA range (process addresses), producing IPAs, controlled by OS at EL1
- `TTBR1_EL1`: contains a pointer (IPA) to the Stage 1 table for EL1&0, upper VA range (OS kernel addresses), producing IPAs, controlled by OS at EL1
- `VTTBR_EL2`: contains a pointer (PA) to the Stage 2 table (second stage for IPAs translated at EL1&0), producing PAs, controlled by hypervisor at EL2
- `TTBR0_EL2`: contains a pointer (PA) to the single-stage table for EL2 (hypervisor's own addresses), producing PAs, controlled by hypervisor at EL2

Each hardware thread has its own base registers (and other system registers), and so different hardware threads can be using different address spaces (for example, for different processes) at the same time.

2.4 Caching translations in TLBs

A naive hardware implementation of address translation would need many translation memory reads – with four levels, up to 24 with both stages enabled, for every instruction-fetch, read, or write. This would have unacceptable performance, so processors have specialised caches for translation-table wal k reads called *translation lookaside buffers* (or TLBs). Under normal operation the TLBs are invisible to user code, but systems code has to manage them explicitly, to change which translation table is currently in use (e.g. when context switching), or to make changes to the tables for one process or guest. Without correct management a TLB could hold incorrect (stale) data, breaking the protection that the address translation is intended to provide.

The architecture supports explicit TLB maintenance with various flavours of the `TLBI` instruction (TLB invalidate), to invalidate old entries for specific ranges of virtual or intermediate-physical addresses, or even whole ASIDs or VMIDs at once. The *memory management unit* (MMU) is responsible for performing these translations. It does this by looking at the TLB and, if the TLB does not contain an entry for the given address (called a *miss*), it performs the translation table walk function as described earlier and caches the result in the TLB (a *fill*).

TLB maintenance and TLB misses are expensive, and one would not want the cost of TLB invalidation on every context switch, so the architecture provides *address space identifiers* (ASIDs). The translation table base registers include an ASID in addition to the table base address, and when translation data is cached in a TLB it is tagged with the current ASID, giving the illusion of separate TLBs per ASID, and allowing switching from one to another without TLB maintenance. Eventually the system will need to reclaim and reuse a previously used ASID, and then TLB maintenance is required to clean that ASID's old entries. There are similar identifiers for Stage 2 intermediate physical memory, known as virtual-machine identifiers or VMIDs.

3 Concurrency architecture design questions

Now we will introduce the main concurrency architecture design questions that arise for Armv8-A virtual memory, within the scope laid out in the introduction. As usual, the architecture has to define an envelope of behaviour that provides the guarantees needed by software, while admitting the relaxed behaviour of the microarchitectural techniques necessary for performance. That means we have to discuss both, including just enough microarchitecture to understand the possible programmer-visible behaviour, before we abstract it in the semantic models we give in §5. The discussion includes points of several kinds: some that are clear in the current Arm documentation, some where Arm have a change in flight, some that are not documented but where the semantics is (after discussion) obviously constrained by existing hardware or software practice, and some where there is a tentative Arm intent but it is not yet fixed upon; our modelling raised a number of questions of the latter two. To make this as coherent as possible, we discuss all these in a logical order, laying out the design principles. We have developed a suite comprised of 119 hand-written Isla-compatible virtual-memory litmus tests that illustrate the issues, but to keep this concise we just give the main ideas here. For ease of reference, we give the actual tests in App. A, with links in the margin. As a sample, we explain one pKVM test in detail in §4.

3.1 Coherence with respect to physical or virtual addresses

For normal memory accesses, the most fundamental guarantee that architectures provide is *coherence*: in any execution, for each memory location, there is a total order of the accesses to that location, consistent with the program order of each thread, with reads reading from the most recent write in that order. Hardware implementations provide this, despite their elaborate cache hierarchies and out-of-order pipelines, by coherent cache protocols and pipeline hazard checking, identifying and restarting instructions when possible coherence violations are detected.

Previous work on relaxed-memory semantics for architectures has taken virtual addresses as primitive, implicitly considering only execution with well-formed, constant, and injective address translation mappings.

Now, we have to consider whether coherence is with respect to virtual or physical addresses, for non-injective mappings. For Arm, coherence is w.r.t. physical addresses [13, D5.11.1 (p2812)]. This means that if two virtual addresses alias to the same physical address, then (still assuming well-formed and constant translation): a load from one virtual address cannot ignore a program-order (po) previous store to the other; and a load from one virtual address can have its value forwarded from a store to the other, and similarly on a speculative branch. A.2.1.1 A.2.1.2 A.2.1.3 A.2.3.4 A.2.2.1

3.2 Relaxed behaviour from TLB caching

There are two main aspects of the concurrency semantics of virtual memory: the relaxed behaviour arising directly from TLB caching, and the relaxed behaviour of the *not-from-TLB* (*non-TLB*) memory accesses for translation reads that read from memory or by forwarding from po-previous writes, and that might supply TLB cache fills. We discuss them in this and the following subsection respectively.

What can be cached: The MMU can cache information from successful translations, and also from translations that result in permission faults, but it is architecturally forbidden from caching information from attempted translations that result in translation faults.This ensures that the handlers of those faults do not need to do TLB maintenance to remove the faulting entry [13, D5.8.1 (p2780)], and makes the potential behaviour for page-table updates from invalid-to-valid and valid-to-any quite different, as we shall see.

TLB implementations might cache any combination of individual page-table entries and partial or complete translations, e.g. from the virtual address and context to the physical address of the last-level page. Conceptually, however, we can simply view a TLB as containing a set of cached page-table-entry writes (i.e., writes that have been read from for a translation), including at least:

- the context information of the translation: the VMID, ASID (or a "global indicator"), , and the originating exception level;
- the virtual address, intermediate physical address, and/or physical address of the translation;
- the translation stage and level at which the write was used;
- the system register values used in the translation (those which can be cached); and
- for an entry used for a Stage 1 translation, whether it has been invalidated at both stages.

That additional information allows the various TLBI instructions to target specific entries. A translation walk can arbitrarily use either a cached write (if one exists) or do a non-TLB read, either from memory or by forwarding from a po-previous write, for any stage or level. A.4.1.1 A.4.1.4 A.7.2.1

Caching of multiple entries for the same virtual address and context: High-performance hardware implementations may have elaborate TLB structures, including multiple "micro TLBs" per thread. These can be seen as a conceptual single per-thread TLB that can hold zero, one, or more entries for each combination of input address and the other information above. If zero, a translation will necessarily read from memory (with ordering constrained as discussed below). If one or more, a translation may use any of those entries or read from memory (and the write read from might or might not be cached). However, in some cases multiple entries constitute a *break-before-make* failure, leading to relatively unconstrained behaviour; we return to this below. A.3.4.4

When can page-table entries be cached: Any memory read by a translation can be cached. Any thread can spontaneously do a translation for any virtual address at any program point, with respect to its context at that point (though this interacts with the system-register write/read semantics). Spontaneous translations model hardware prefetching, speculative execution, and branch prediction. They mean that, in the absence of cache maintenance, translations may use TLB entries from arbitrarily old writes. Additionally, any thread may do a spontaneous translation at any point using the configuration from any exception level higher than the current one, but not for lower levels. Preventing spontaneous walks at lower EL is essential, as during an EL2 hypervisor switch between VMs, the EL1 control registers will be in an inconsistent state. Allowing spontaneous walks at higher EL models arbitrary interrupts to the higher level and then doing a spontaneous walk there. A.4.1.1

Each virtual-memory access by a thread involves a non-spontaneous translation which is constrained by the normal inter-instruction constraints on out-of-order and speculative execution by the thread. These constraints are especially important in order to understand when a translation must fault: as invalid entries cannot be cached, a translation that gives rise to such a fault must be at least in part from a non-TLB read, subject to these ordering constraints. A.3.4.11

Coherence of translations: Due to the TLB caching as described above translations of the same virtual address by the same thread need not see a coherent view of page-table memory. This is in sharp contrast to normal accesses, but analogous to instruction-fetch reads [57] and reads from persistent memory [51]. A.3.4.4

Removing cached entries: TLBs may spontaneously forget any cached information at any point. To *ensure* that a cached entry is removed, software must ensure that it will not be spontaneously re-cached. It can do this with a write of an invalid entry and then a DSB instruction (data synchronization barrier) to ensure that it is visible across the system, followed by a TLBI. A.4.1.1 A.3.4.8

Break-before-make failures: When changing an existing translation mapping, from one valid entry to another valid entry, Arm require in many cases the use of a *break-before-make (BBM)* sequence: breaking the old mapping with a write of an invalid entry; a DSB to ensure that is visible across the system; and a broadcast TLBI to invalidate any cached entries for all relevant threads; a DSB to wait for the TLBI to finish; then making the new mapping with a write of the new entry, and additional synchronisation to ensure that it is visible to translations (specifically, to translation-walk non-TLB reads). The current Arm text [13, D5.10.1 (p2795)] identifies six cases of page-table updates that without such a sequence constitute *BBM failures*, and gives very severe architectural consequences: failures of coherency, single-copy atomicity, ordering, or uniprocessor semantics. Note that these consequences are architecturally allowed if there could exist a break-before-make-failure change to the translation tables for some virtual address, irrespective of whether the program architecturally accesses it. A.4.1.3 A.6.1.1 A.6.1.2 A.6.1.3

This severity is because, in some of the six cases, hardware implementations could give rather arbitrary behaviour, e.g. an amalgamation of old and new entries. From a software point of view, it seems that one must treat such cases more-or-less as fatal errors. This is analogous to the Data-race-free-or-catch-fire semantics underlying the C/C++ relaxed memory model [4, 33, 22, 20], in which any program with a consistent execution that includes a race between nonatomic accesses is deemed to have undefined behaviour, and the C/C++ standards do not constrain implementation behaviour for such programs in any way. This makes many potential litmus tests that change between valid entries uninteresting, as they simply exhibit BBM failures (though changes of permissions do not necessitate a BBM sequence).

However, for a processor architecture that supports virtualisation, one cannot regard BBM failures as allowing completely arbitrary behaviour for the entire machine: if one guest virtual machine (at EL1) changes one of its own translation mappings without correctly following the

BBM sequence, either mistakenly or maliciously, that should not impact security of the hypervisor (at EL2) or other guests. Instead, one has to bound the arbitrary behaviour to that virtual machine, allowing arbitrary memory and register accesses that are possible within its context. In our exhaustively executable semantics, to keep litmus-test executions finite, we currently simply detect BBM failures; we do not explicitly model that arbitrary behaviour.

In reality, these six BBM failure cases include some where hardware may give such weakly constrained behaviour and others where, because coherence is over physical addresses and the mapping may be temporarily indeterminate, software might see well-defined but nondeterministic or surprising results. These were architected as a guide for system software to produce predictable behaviour, and future versions of the architecture might refine this.

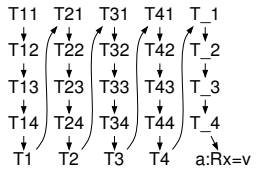
When a hypervisor installs a new guest, it has to be able to reset to a clean state, in case a previous guest has failed to follow the BBM sequence (which in general the hypervisor cannot know). It can do so with a TLBI covering all the previous guest's processes address space. There seems to be no need or support for finer-grain cleanup.

3.3 Relaxed behaviour of translation-walk non-TLB reads

Now we turn to the semantics of translation-walk *non-TLB* reads, those that are satisfied from memory or by forwarding, not from a TLB, and which might then be cached in a TLB. This matters especially when one knows that there are no relevant cached TLB entries, e.g. when an invalid entry has been written and a TLBI performed, so one knows that translation walks will do such a non-TLB read. A.3.4.5 A.3.4.2

Ordering among the translation-walk reads of an access: Each translation-table walk for a virtual-memory access can involve many memory reads, one for each level of the table for each stage of translation.

The diagram on the right is an example walk, where each `Tn` is read of level `n` of the Stage 1 table. Each of those Stage 1 reads must first be translated to get the PA (as the table contains IPAs) and so each `Tnk` is a read of level `k` of the Stage 2 table for the address of the Stage 1 table at level `n`. Once the full Stage 1 walk has been completed the final output IPA must be translated to the final PA, and those are the final 4 `T_n` reads, of the Stage 2 table at level `n`. The reads are ordered one after another in the order they appear in the ASL walk function. This ordering must be respected by hardware as software relies on it when building the tables bottom-up.



For example, if one starts with a Level 2 invalid entry, one might first create a Level 3 table (then a barrier to keep writes ordered). In other words, the architecture has to prohibit value speculation of page table entries. A.7.2.2 A.7.2.1

Dependencies into translation-walk non-TLB reads: Address dependencies into a memory-access instruction in classic "user" models are now explainable as dataflow dependencies to the translation reads of those accesses, as the address has to be available before a walk can start. These are virtual-address dataflow dependencies (contrasting with physical-address coherence). A.3.3.5 A.8.1.14

Translation-walk non-TLB reads from non-speculative same-thread writes:

PO-past A translation-walk non-TLB read might read from a po-previous page-table-entry write, but it is only guaranteed to see such a write if there is enough intervening synchronisation. Arm have recently introduced *Enhanced Translation Synchronization* (ETS), optional in Armv8.0 and mandatory from Armv8.7. Armv8-A implementations without ETS require both a DSB, to make the write visible to translation-walk non-TLB reads, and an ISB, to ensure that any translations for later instructions that were done out-of-order, before the write, are restarted.

With ETS, only the DSB is required for a translation-walk non-TLB read to definitely see the write, though one might still need an ISB if the new translation enables new instruction fetch. A.3.4.6

Because invalid entries cannot be cached, this means that if an entry is initially invalid, A.7.1.2 then after a write of a valid entry and a DSB;ISB/DSB, translations will use that valid entry. However, the DSB;ISB/DSB does not remove cached entries, so an initially valid entry might be cached by a spontaneous walk, so even after a write (of an invalid or non-BBM-failure valid entry) and a DSB;ISB/DSB, the old entry could still be used by translations. One would need a TLBI sequence to remove old cached entries, which we return to below.

PO-future The Armv8-A architecture allows load-store reordering, but it does not allow writes to become visible to other threads while they are still speculative. In the same vein, translation-walk non-TLB reads cannot read from po-later page-table-entry writes [13, D5.2.5 (p2683)]. Before the po-earlier translation is complete, one cannot know that it is not going to fault, so the later write has to be considered speculative. This prevents a thread-local self-satisfying translation cycle, analogous to the prevention of load-store cycles with dependencies. A.3.4.1

PO-present On the margin, can a translation-walk non-TLB read for a write access see that write, or a distinct write from the same instruction? The second case could arise from a store-pair or misaligned store that does two writes, with one to a page-table-entry that could be used by the other, though real code would typically not do this intentionally. This is explicitly allowed by the current architecture text [13, D5.2.5 (p2683)]. However that text does not specify whether the translations for those two writes could *both* read from the other, a self-satisfying translation cycle where the writes write each others translations. In general such self-satsifying cycles give rise to *thin air* behaviours and are universally forbidden by the architecture.

Translation-walk non-TLB reads from speculative same-thread writes: Speculative execution requires translation walks, which might result in additional page-table entries being cached, but in most cases this is indistinguishable from the effects of a non-speculative spontaneous walk. However, one has to ask whether a translation-walk non-TLB read can see a po-previous write that is still speculative, e.g. while both instructions follow an as-yet-unresolved conditional branch. It is clear that the result of such a walk should not be persistently cached, or made visible to other threads (via a shared TLB), while it remains speculative. Moreover, such translations could lead to arbitrary reads of read-sensitive device locations, which one normally relies on the MMU to prevent. The conclusion is therefore that this must be forbidden. A.3.5.1

Translation-walk non-TLB reads from same-thread writes, forbidden past (same-thread TLBI completion): To remove an existing mapping on a single thread, one needs first to write an invalid entry, then a DSB to ensure that has reached memory and thus is visible to translation-walk non-TLB reads (to prevent spontaneous re-caching), then a TLBI to invalidate any cached entries, then a DSB to wait for TLBI completion. Without ETS, one also needs an ISB to ensure that po-later translations that have been done early are restarted. With ETS, the ISB is not always necessary, though might still be needed for its instruction-cache effects if the change of mapping affects instruction fetch. After all that, an attempted access by that thread is guaranteed to fault. A.4.1.1 A.4.1.2

Translation-walk non-TLB reads from other-thread writes, guaranteed past, initially invalid: Now consider when a translation-walk non-TLB read is guaranteed to see a write by another thread of a new entry, assuming that the entry was previously invalid and any cached entries for it invalidated. Consider a two-thread message-passing case, where a producer P0 writes a new valid page table entry (`pte_valid`), then has some ordering before a write of a flag, while a consumer P1 reads the flag, then has some ordering before an access `Rx` or `Wx` that needs that entry for a translation `Tx` of virtual address `x`. A.4.1.4 A.3.3.6 A.3.3.5

P0	P1

`a:W pte(x)=pte_valid`
`<Producer ordering>`
`b:W flag=1`
	`c:R flag=1`
	`<Receiver ordering>`
	`d:Tx, for a Rx or Wx`

On some Armv8-A implementations that do not support ETS, some "obvious" combinations of ordering on P0 and P1 could lead to an abort of the translation of (d), which some OS software would find difficult to handle. This was the main motivation for ETS: implementations without it can have weak behaviour, requiring strong synchronisation to prevent the abort, while with ETS the architecture is stronger, requiring only weaker ordering to prevent the abort.

Without ETS, two combinations of ordering are architected as sufficient to ensure that the translation (d) sees the new valid entry:

1. P0 has any ordered-before relationship, and P1 has DSB+ISB. A.3.3.2
2. P0 has DSB; TLBI; DSB, and P1 has any ordered-before relationship. A.3.3.13

In Case 1, the message-passing is enough to ensure the write (a) is in main memory, the P1 ISB ensures that any out-of-order translation of (d) is restarted, and the P1 DSB keeps the read (c) and that ISB in order. In Case 2, the first DSB ensures the write is visible to all threads, the TLBI (broadcast, for the virtual address `x`) invalidates any older cached entry on P1, and the second DSB waits for that TLBI to be complete, after which any new translation on P1 will have to see the new entry. However, it appears that the probability of an unhandleable abort in practice, where one usually does not have these operations immediately adjacent, and where in many cases the abort could be handled, has been judged low enough that OS code is not necessarily using either of these.

With ETS, the architecture says [13, D5.2.5,p2683] that *"if a memory access RW1 is Ordered-before a second memory access RW2, then RW1 is also Ordered-before any translation table walk generated by RW2 that generates a Translation fault, Address size fault, or Access flag fault."* Microarchitecturally, the intuition here is that with ETS any translation done while speculative that leads to such a fault will have to be reconfirmed as faulting when execution is no longer speculative, so an early faulting translation of (d) would have to be restarted after the ordered-before edges have ensured that (a) is visible. However, in the case that the RW2 instruction faults, there is no read or write event, and if the fault is a translation fault, there is no physical address. One therefore has to ask what the meaning of ordered-before edges into RW2 is, especially for the parts of ordered-before dependent on physical addresses, such as coherence. The conclusion is that this should be only the non-physical-address parts of ordered-before into RW2, and in modelling one needs a "ghost" event to properly record what the dependencies would have been if it had succeeded. Note that this includes ordered-before to RW2 that ends with a data dependency into a write, even though that data would not normally be necessary for the translation.

Even with ETS, one might need an ISB on P1 if the new translation affects instruction fetch.

Translation-walk non-TLB reads from other-thread writes, guaranteed past, initially valid (other-thread TLBI completion): The following test has a read-only mapping for some physical address that is updated with a new writeable mapping to the same physical address, followed by a message-pass to another thread that attempts to write. There is no requirement for break-before-make here, as the output address has not changed, but TLB maintenance is required to ensure that the new writeable entry is guaranteed to be used by later translation-reads.

P0	P1

`STR pte_writeable,[pte(`x`)]`
`DSB SY`
`TLBI VAAE1IS,[page(`x`)]`
`DSB SY`
`MOV X7,#1`
`STR X7,[`y`]`
	`LDR X0,[`y`]`
	`DMB SY`
	`MOV X1,#1`
	`L0:`
	`STR X1,[`x`]`

Forbid: `1:X0=1 & permission_fault(L0,`x`)?`	

Arm forbid the outcome where the `STR` faults due to a permission check. This is because the `TLBI` only completes once all instructions using any old translations which would be invalidated by the TLBI, on all other threads that the TLBI affects, have also completed, and the following `DSB` waits for that (the same-thread case is different; see §3.3). In practice this means that A.4.1.6

once the `TLBI` completes, one of the following holds: either the final `STR` has not performed its translation of x yet and will be required to see the writeable mapping for its page table entry (pte); or the `STR` has translated using the new writeable mapping; or the `STR` has already translated using the old read-only mapping, in which case we know that the `STR` has finished and performed its write, since the `TLBI` could not complete while it was still in-progress. In that case if the `STR` has completed, then so must have the locally-ordered-before `LDR`, and that must have read 0. This explanation also covers the make-after-break case above, for non-ETS Case 2.

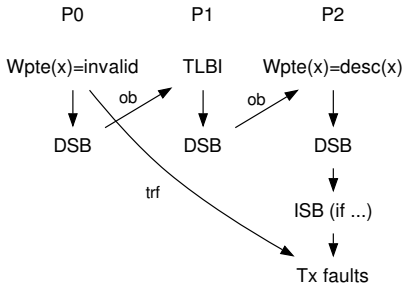
This is reflected in text to be included in future versions of the Arm ARM: *A TLB maintenance operation [without nXS] generated by a TLB maintenance instruction is finished for a PE when:*

1. *all memory accesses generated by that PE using in-scope old translation information are complete.*
2. *all memory accesses RWx generated by that PE are complete. RWx is the set of all memory accesses generated by instructions for that PE that appear in program order before an instruction (I1) executed by that PE where:*
   (a) *I1 uses the in-scope old translation information, and*
   (b) *the use of the in-scope old translation information generates a synchronous data abort, and*
   (c) *if I1 did not generate an abort from use of the in-scope old translation information, I1 would generate a memory access that RWx would be locally-ordered-before.*

Translation-walk reads from same- and other-thread writes, forbidden past (break-before-make): Now we can finally return to the break-before-make sequence. Normal reads cannot read from the coherence-predecessors of the most coherence-recent write that is visible to them, but translation reads can read old (non-invalid) values from a TLB. To prevent this, and to ensure that a translation read sees a new page-table entry, one has to both ensure that any old TLB entries are invalidated, with a suitable TLBI, and that the new entry is visible to translation-walk non-TLB reads.

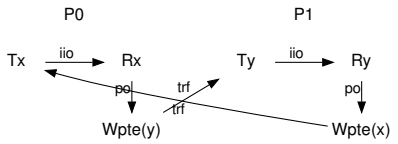
Armv8-A says [13, D5.10.1 (p2795)] *"A break-before-make sequence on changing from an old translation table entry to a new translation table entry requires the following steps: (1) Replace the old translation table entry with an invalid entry, and execute a DSB instruction. (2) Invalidate the translation table entry with a broadcast TLB invalidation instruction, and execute a DSB instruction to ensure the completion of that invalidation. (3) Write the new translation table entry, and execute a DSB instruction to ensure that the new entry is visible."*.

Typically the write of an invalid entry and TLBI would be on the same thread, but more generally, any shape as below should be forbidden, where `Tx` is a translation-walk read for an access of `x` and the `trf` relation shows the page-table write it reads from. In other words, the sequence ensures that the write of the invalid entry, and of any co-predecessor writes, are hidden behind the new page-table entry as far as new translations are concerned. Here the P0 DSB and P0-to-P1 `ob` ensure the P0 write has propagated to memory before the P1 TLBI starts; the P1 DSB waits for that TLBI to have finished on all threads; the P1-to-P2 `ob` ensures that has happened before the new page-table-entry write starts; and the DSB ensures the new write has reached memory and so is visible to translation before subsequent instructions. The P2 ISB is needed if on non-ETS hardware, to force restarts of any out-of-order translations for po-later instructions, or (on any hardware) if P2=P1, to ensure any later translations on the TLBI thread are restarted, or if the new mapping affects instruction fetch. A.3.4.8



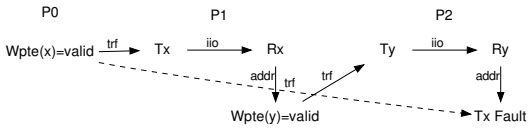
This generalisation seems necessary, as a TLBI might be performed by a virtual CPU at EL1 which is interrupted and rescheduled by an EL2 hypervisor. One should be able to rely on the hypervisor doing a DSB on the same hardware thread as part of the context switch, and that has to suffice. It is sound because the DSBs and TLBI are all broadcast, though note that the DSBs and TLBI are all broadcast, though note that the DSB waiting for TLBI completion has to be on the same hardware thread as it.

Translation-walk non-TLB reads from other-thread writes, forbidden future: Above we saw that translation-walk non-TLB reads should not read from po-later writes. How should that be generalised to multiple threads? For the simplest example, consider the translation version of the LB test on the right, in which two threads translation-read from each other's po-future (`iio` relates translation reads to their accesses). Standard LB shapes for normal accesses without dependencies are allowed in Armv8-A, but this example should be forbidden: until each translation is done, one cannot know that the first instruction on each thread will not abort, so one could not make the po-later write visible to the other thread without inter-thread roll-back. In other words, the possibility of translation aborts creates ordering rather like a control dependency from translation reads to po-later writes. A.3.3.16



Multicopy atomicity of translation-walk non-TLB reads: The ARMv7 and early Armv8-A architectures for normal accesses were *non-multicopy-atomic*: a write could become visible to some other threads before becoming visible to all threads, broadly similar in this respect to the IBM POWER architecture [1, 53]. This is one of the most fundamental choices for a relaxed memory model. In 2017 Arm revised their Armv8-A architecture to be *multicopy-atomic* (*other multicopy-atomic*, or OMCA, in their terminology), a considerable simplification [49, 12]. However, there was no consideration at the time of whether this should also apply to the visibility of writes by translation-walk non-TLB reads, or of the force of the ARM statement that *a translation table walk is considered to be a separate observer* [13, D5.10.2 (p2808)]. A.8.1.14 A.8.1.15 A.8.1.7

For example, consider the following translation-read analogue of the classic WRC+addrs test, which would be forbidden in OMCA Armv8-A for normal reads. Suppose one has ETS, the last-level page-table entries for `x` and `y` are initially invalid and not cached in any TLB, P0 writes a valid entry for `x`, P1 does a translation that sees that entry and then (via an address dependency) writes a valid entry for `y`, then P2 does a translation that sees that entry and then (via an address dependency) tries a translation for `x`, is that last guaranteed to see the valid entry instead of faulting? This might be exhibited by a microarchitecture with a shared TLB between P0 and P1 (e.g. if they are SMT threads on the same core, or have a shared TLB for a subcluster). The tentative Arm conclusion is that this should be forbidden, to avoid software issues with unexpected aborts similar to those motivating ETS.



Now consider the above translation version of LB, generalising from po-future writes to other ob-future writes. For transitive combinations of reads-from and dependencies, it should clearly still be forbidden, to avoid needing inter-thread roll-back, but for `ob` including coherence edges (`coe`) one can imagine that a translate read could see a write before the coherence relationships are established, analogous to the weakness of coherence in the Power non-MCA model.

Discussion of these and other cases with Arm led to the tentative conclusion for Armv8-A that translation-walk non-TLB reads (like normal reads) do not see any non-OMCA behaviour. In other words, there is no programmer-visible caching observable to some non-singleton subsets of threads' translations but not others.

3.4 Further issues

Our discussions with Arm identified and clarified various other architectural choices, though we do not discuss them fully here, and our models do not cover them at present. To give a flavour: (1) Misaligned or load/store-pair instructions give rise to multiple accesses, which might be to different pages. Each has their own translation; not ordered w.r.t. each other, and with no prioritisation of faults between them. As noted in §3.3, one might translate-read from the other, but not both simultaneously. (2) Normal registers act like a per-thread sequential memory, with reads reading from the most recent po-previous write, but the system registers that control translations can have more relaxed behaviour, requiring `ISB`s to enforce sequential behaviour. (3) The architecture requires, and OSs rely on, the fact that turning on the MMU does not need TLB maintenance. However, in a two-stage world, if Stage 1 is off, one is still using the TLB for Stage 2, so entries do get added to the TLB. When one later turns on Stage 1, it is essential that the entries added from those earlier Stage 2 translations are not used, so one has to regard them as from a 257'th ASID.

4 Virtual memory in the pKVM production hypervisor

Protected KVM, or pKVM [30, 27, 2], is currently being developed by Google to provide a common hypervisor for Android, to provide improved compartmentalisation by a small trusted computing base (TCB) between the Linux kernel and other services. pKVM is built as a component of Linux. During boot, the Linux kernel hands over control of EL2 to the pKVM code, which constructs a memory map for itself and a Stage 2 memory map to encapsulate the Linux kernel. The Linux kernel thereafter runs only at EL1 (managing EL1&0 Stage 1 memory maps for itself and for user processes), as the *principal guest*, also known as the *host* (not to be confused with the host hardware). Other services can run as other guests, which are protected from the kernel and vice versa. The kernel remains responsible for scheduling, but context switching and inter-guest communication is done by hypervisor calls to the pKVM code at EL2. This gives us an ideal setting in which to examine the management of virtual memory by production code for Armv8-A relaxed-memory-concurrency, with both one and two stages of translation (for EL2 and EL1&0 respectively). The pKVM codebase is small, so it is feasible to examine all uses of TLB management, and we benefit from discussions with the pKVM development team. We have manually abstracted the main pKVM relaxed-virtual-memory scenarios into 14 tests.

To give a flavour for these we will explore just a few of the most fundamental for hypervisor control of translation tables: `__pkvm_init` performs first-time per-CPU initialisation of pKVM, which includes setting up the `vmemmap`, a large array storing a struct entry for each physical page in memory with ownership information; `__kvm_vcpu_run` switches to a different guest; `__kvm_flush_vm_context` flushes all entries in all TLB caches that relate to Stage 1 translations; `__kvm_tlb_flush_vmid_ipa` flushes entries in caches that relate to a particular guest for a given address; `__kvm_tlb_flush_vmid` flushes all entries in caches that relate to a particular guest; `__kvm_flush_cpu_context` flushes all entries in caches at EL1 for a given guest; `__pkvm_cpu_set_vector` sets the vector base address for handling exceptions; `__pkvm_create_mappings` creates entries in the Stage 1 page table that pKVM uses when it executes; and `__pkvm_prot_finalize` which enables Stage 2 translations during boot.

In the remainder of this section we describe a selection of litmus tests extracted from the current pKVM source code [2] as of Dec 2021.

4.1 Switching to another guest

The most basic task that pKVM is in charge of is the actual swapping from one vCPU's context to another to execute a particular VM on the physical CPU. Deciding *when* and to *which* vCPU to switch is done by the underlying host Linux guest. pKVM's role is to safely save the current

guest's state, and load the EL1 state for the target vCPU, and manage the Stage 2 translation tables for the vCPU's VM.

Aside from the fundamental shape of switching from one VM to another on the same core, there are a few interesting cases to consider:

- Switching from one vCPU in a VM to another vCPU in the same VM.
- Switching to a new vCPU, re-using a previous VMID.
- Re-using an old VMID with a concurrently executing vCPU on another core.

When the host guest Linux kernel wishes to perform the switch from one vCPU to another, it writes to some general-purpose registers and then performs an `HVC` instruction (in this case, to call the `__kvm_vcpu_run` hypercall).

4.1.0.1 pKVM.vcpu_run : In the simplest case where pKVM is just switching from one vCPU to another vCPU in a different VM, pKVM restores the per-CPU register state and sets the `VTTBR` with the new VMID. So long as the two vCPUs are using disjoint VMIDs there is no requirement for TLB maintenance.

This test, `pKVM.vcpu_run`, is below, with typesetting generated from the TOML input format

AArch64 pKVM.vcpu_run

Page table setup:

```
option default_tables = false;

virtual x;
physical pa1 pa2;
intermediate ipa1 ipa2;

s1table hyp_map 0x200000 {
  identity 0x1000 with code;
  x ↦ invalid;
}
s1table vm1_stage1 0x2C0000 {
  x ↦ ipa1;
}
s1table vm2_stage1 0x300000 {
  x ↦ ipa2;
}
s2table vm1_stage2 0x240000 {
  ipa1 ↦ pa1;
  ipa2 ↦ invalid;
  s1table vm1_stage1;
}
s2table vm2_stage2 0x280000 {
  ipa1 ↦ invalid;
  ipa2 ↦ pa2;
  s1table vm2_stage1;
}
*pa1 = 1;
*pa2 = 2;
```

Initial state:

```
PSTATE.EL=0b01
R1=x
R2=ttbr(base=vm2_stage1,asid=0x00)
R3=ttbr(vmid=0x0002,base=vm2_stage2)
R5=x
TTBR0_EL1=ttbr(asid=0x00,base=vm1_stage1)
TTBR0_EL2=ttbr(asid=0x00,base=hyp_map)
VBAR_EL2=0x1000
VTTBR_EL2=ttbr(base=vm1_stage2,vmid=0x0001)
```

Thread 0

```
MOV X0,#0
// in guest
LDR X0,[X1]
HVC #0x0
MOV X4,#0
// in guest
LDR X4,[X5]
```

thread0 el2 handler

```
0x1400:
MRS X13, ESR_EL2
// read ESR_EL2.EC
UBFX X13, X13, #26, #5
// if EC==HVC (010110)
SUBS X13, X13, 0b010110
// Branch to HVC handle
CBZ X13, 2f
// EC==DABT
1:
MOV X4,#0
MRS X20,ELR_EL2
ADD X20,X20,#4
MSR ELR_EL2,X20
ERET
// EC=HVC
2:
// kvm/hyp/sysreg-sr.h:96
MSR TTBR0_EL1,X2
// include/asm/kvm_mmu.h:276
MSR VTTBR_EL2,X3
// kvm/hyp/nvhe/host.S
ERET
```

Final state: `0:R0=1 & ~0:R4=2`

of our Isla tool (§7). Here there is a single physical CPU, initially running a virtual machine VM1, with VMID `0x0001`, at EL1. The section on the left defines the initial and all potential states of the page tables, and any other memory state. This test sets up separate translation tables for pKVM at EL2 (which has just a single stage) and for two VMs (each with two stages, Stage 2 controlled by pKVM and Stage 1 controlled by the VM). pKVM's own mapping `hyp_map` maps its code. VM1's own Stage 1 mapping `vm1_stage1` maps virtual address `x` to `ipa1`, and the initial pKVM-managed Stage 2 mapping `vm1_stage2` maps that `ipa1` to `pa1`, which implicitly

initially holds 0. These page tables are described concisely by a small declarative language we developed, determining the page-table memory (here ∼30k) required for the Armv8-A page-table walks.

The top-right block gives the initial Thread 0 register values, including the various page-table base registers. The bottom-right blocks give the code of the test both the guest (EL1) code and the (simplified) hypervisor code.

This test is written in an end-to-end style, where the test will start at EL1, which performs some actions in the guest, before reaching an HVC instruction which performs a pKVM hypercall, before finally returning to the EL1 code and continuing after the HVC. Just as a full execution of guests and hypervisor together would. The final state can then make assertions about the state before and after any hypercalls, by inspecting the results of any guest operations.

The key assembly lines are annotated with the pKVM source line numbers they correspond to. To switch to run another virtual machine VM2, with VMID `0x0002`, on this same physical CPU, pKVM changes `VTTBR_EL2` to the new `vm2_stage2` mapping and, as part of the context-switch register-file changes, restores `TTBR0_EL1` to the VM2's own Stage 1 mapping `vm2_stage1`. The code then executes an `ERET` ("exception-return") instruction to return to EL1, and then tries to read `x`. The test includes a final assertion of the relaxed outcome that register `x0=1` and `x4!=2`, which would occur if the second `ldr`'s translation used the old VM1 mapping instead of VM2's mapping. In this case that should be forbidden.

Other tests capture more elaborate scenarios. For example, currently the host kernel manages VMIDs and assigns each VM its own VMID. If the host runs out of VMIDs to allocate to new vCPUs, it currently revokes all previously allocated VMIDs and re-allocates from the beginning, during which pKVM has to ensure that any old vCPUs' translations using that VMID are expelled from any TLBs (`pKVM.vcpu_run.update_vmid`). If there is a concurrently executing vCPU using that VMID, that vCPU must be paused until after the new VMID generation (and hence any required TLB maintenance), before continuing with the freshly allocated VMID (`pKVM.vcpu_run.update_vmid.concurrent`).

For another example, for pKVM to maintain the illusion that each vCPU is on its own core, the per-core state must be cleaned between running different vCPUs, including ensuring that translations for one vCPU are not cached and visible to another, even if they happen to be in the same VM (and using the same VMID) (`pKVM.vcpu_run.same_vm`).

The most basic task that pKVM is in charge of is the actual swapping from one vCPU's context to another to execute a particular VM on the physical CPU. Deciding *when* and *to which* vCPU to switch is done by the underlying host Linux guest. pKVM's role is to safely save off the current guest's state, and load the EL1 state for the target vCPU, and manage the Stage 2 translation tables for the vCPU's VM.

Aside from the fundamental shape of switching from one VM to another on the same core, there are a few interesting cases to consider:

- Switching from one vCPU in a VM to another vCPU in the same VM.
- Switching to a new vCPU, re-using a previous VMID.
- Re-using an old VMID with a concurrently executing vCPU on another core.

When the host guest Linux kernel wishes to perform the switch from one vCPU to another, it writes to some general-purpose registers and then performs an `HVC` instruction (in this case, to call the `__kvm_vcpu_run` hypercall).

4.1.0.2 pKVM.vcpu_run.update_vmid :

Since pKVM's host Linux kernel is responsible for all scheduling, it is responsible for the selection of VMIDs. Typically, each VM is given its own VMID, and all vCPUs within that VM share that VMID. By giving distinct VMs different VMIDs, the host Linux kernel can switch between vCPUs in different VMs without requiring TLB maintenance, as was seen in the previous test. Eventually, the host kernel runs out of VMIDs to allocate, either from the naive incremental-allocation or because the number of VMs is larger than the available VMID space (8 or 16 bits in the worst case). Currently, pKVM's host Linux kernel will simply revoke all previously allocated VMIDs, and when pKVM goes to switch to a vCPU for a VM without an allocated VMID, it will be allocated there and then.

The `pKVM.vcpu_run.update_vmid` test in Figure 1 is the simplest case, where the new vCPU is the only vCPU of that VM that is running, and no other physical CPUs are executing. Here, we have two VMs and their associated tables in the initial state: `vm1_stage2` is the root of the Stage 2 table for the first VM, and `vm2_stage2` is the root of the Stage 2 table for the second VM. Initially, the `VTTBR` points to VM1's table, with VMID 1. When switching to VM2's table, with the same VMID, the host Linux kernel begins a sequence of calls to the hypervisor: first it must flush the CPU context, to make sure any old cached translations for the old VMID are removed. From there, pKVM can perform the switch to the new vCPU by restoring the EL1 system register state, then pointing the `VTTBR` to the new table with the new ASID, before doing an exception-return to the guest. If the guest then accesses a location, it should be guaranteed to use the new Stage 2 mapping with its own Stage 1 tables. Figure 1 contains the code listing and initial conditions for this test; execution begins at EL2 with `vm1`'s vCPU state, with VMID 1. The test performs the pKVM `vcpu_run` sequence to clean up the TLB, switch to `vm2`'s EL1 state and then switching to `vm2`'s Stage 2 mapping with VMID 1 (the same as what `vm1` had been using) before returning to the guest. The test then asserts that the guest's access uses `vm2`'s own restored Stage 1 translation tables, and does not see the old Stage 2 entries of `vm1`.

Note that this test *begins* with TLB invalidation. To really capture the full sequence that is microarchitecturally interesting, the test should really begin from execution at EL1 inside the guest allowing Stage 2 TLB fills. The test beginning at EL2 from a clean machine state would not give the TLB time to actually fill with stale entries from `vm1`'s Stage 2 translation tables in an operational model. Also note that, currently, Isla does not produce candidates with re-ordering of system register reads, and so there are no 'bad' candidate executions to show from the Isla-generated executions currently. This is work-in-progress to find the correct semantics for such re-orderings.

AArch64 pKVM.vcpu_run.update_vmid

Page table setup:

```
option default_tables = false;
physical pa1 pa2;
intermediate ipa1 ipa2;

s1table hyp_map 0x200000 {
    x ↦ invalid;
}

s2table vm1_stage2 0x300000 {
    ipa1 ↦ pa1;
    ipa2 ↦ invalid;

    s1table vm1_stage1 0x280000 {
        x ↦ ipa1;
    }
}

s2table vm2_stage2 0x380000 {
    ipa1 ↦ invalid;
    ipa2 ↦ pa2;

    s1table vm2_stage1 0x2C0000 {
        x ↦ ipa2;
    }
}

*pa2 = 1;
```

Initial state:

```
R1=ttbr(base=vm2_stage2,vmid=0x0001)
PSTATE.SP=0b1
TTBR0_EL1=ttbr(base=vm1_stage1,asid=0x0000)
R3=x
VBAR_EL2=0x1000
VTTBR_EL2=ttbr(vmid=0x0001,base=vm1_stage2)
PSTATE.EL=0b10
TTBR0_EL2=ttbr(asid=0x0000,base=hyp_map)
ELR_EL2=L0:
SPSR_EL2=0b00101
R0=ttbr(base=vm2_stage1,asid=0x0000)
```

Thread 0

```
// kvm/hyp/nvhe/tlb.c:145
dsb ishst
// kvm/hyp/nvhe/tlb.c:146
tlbi alle1is
// kvm/hyp/nvhe/tlb.c:160
dsb sy
// kvm/hyp/sysreg-sr.h:96
MSR TTBR0_EL1,X0
// include/asm/kvm_mmu.h:276
MSR VTTBR_EL2,X1
// kvm/hyp/nvhe/host.S
ERET
L0:
// in guest
LDR X2,[X3]
```

thread0 el2 handler

```
0x1200:
mov x2, #0
// data abort preferred-return-address is itself
// so jump to next instr instead
mrs x20,elr_el2
add x20,x20,#4
msr elr_el2,x20
eret
```

Final state: `0:R2=0`

Figure 1

4.1.0.3 pKVM.vcpu_run.update_vmid.concurrent :

The concurrent case of the previous `pKVM.vcpu_run.update_vmid` test is critical: while switching from one vCPU to another on the same core is typically a thread-local event, care must be taken in the case where another CPU is executing a VM whose current VMID should be revoked, as in Fig. 2.

In this case, pKVM must interrupt the other core, so that it is not concurrently executing while the new VMIDs are being allocated; otherwise, it might pollute the address space of the VM that gets allocated that VMID.

pKVM does this by sending an IPI to all the other cores to break out of their current vCPUs. When it does this, each CPU will attempt to switch back to the host, and, in doing so, will be forced to take a lock when acquiring the VMID to use. This lock prevents those CPUs from executing at EL1, and the architecture prevents the hardware from performing TLB fills while at EL2. These guarantees ensure that while VMIDs are being re-allocated and TLB maintenance performed on the original core, no stale entries can find their way back into the TLB.

AArch64 pKVM.vcpu_run.update_vmid.concurrent

Page table setup:

```
option default_tables = false;
physical pa1 pa2 pa_ipi pa_kvm_vmid_lock;
intermediate ipa1 ipa2;

s1table hyp_map 0x200000  {
    identity 0x1000 with code;
    x ↦ invalid;
    ipi ↦ pa_ipi;
    kvm_vmid_lock ↦ pa_kvm_vmid_lock;
}

s2table vm1_stage2 0x300000  {
    ipa1 ↦ pa1;
    ipa1 ?-> invalid;
    ipa2 ↦ invalid;
    ipa2 ?-> pa2;

    s1table vm1_stage1 0x280000 {
        x ↦ ipa1;
    }
}

s2table vm2_stage2 0x380000  {
    ipa1 ↦ invalid;
    ipa1 ?-> pa1;
    ipa2 ↦ pa2;
    ipa2 ?-> invalid;

    s1table vm2_stage1 0x2C0000 {
        x ↦ ipa2;
    }
}

*pa2 = 1;
*pa_kvm_vmid_lock = 1;
```

Initial state:

```
0:R7=kvm_vmid_lock
0:PSTATE.SP=0b1
0:R6=0b0
0:R2=0b1
0:TTBR0_EL2=ttbr(asid=0x0000,base=hyp_map)
0:PSTATE.EL=0b10
0:R3=ipi
0:R5=ipi
0:R0=0b1
0:R1=kvm_vmid_lock
1:TTBR0_EL2=ttbr(base=hyp_map,asid=0x0000)
1:TTBR0_EL1=ttbr(base=vm1_stage1,asid=0x0000)
1:VBAR_EL2=0x1000
1:VTTBR_EL2=ttbr(base=vm1_stage2,vmid=0x0001)
1:R1=x
1:R3=x
1:PSTATE.EL=0b01
1:R6=0b10
1:R10=ttbr(asid=0x0000,base=vm2_stage1)
1:R9=kvm_vmid_lock
1:R5=ipi
1:R7=ipi
1:R11=ttbr(vmid=0x0001,base=vm2_stage2)
```

Thread 0

```
// kvm/arm.c:551 force_vm_exit(cpu_all_mask);
STR X2,[X3]
// kvm/arm.c:551 force_vm_exit(cpu_all_mask);
LDR X4,[X5]
// kvm/hyp/nvhe/tlb.c:145
dsb ishst
// kvm/hyp/nvhe/tlb.c:146
tlbi alle1is
// kvm/hyp/nvhe/tlb.c:160
dsb sy
// kvm/arm.c:567 spin_unlock(&kvm_vmid_lock);
STR X6,[X7]
```

Thread 1

```
// in guest, read X
LDR X0,[X1]
DSB SY
ISB
// fake IPI
HVC #0
// try again
LDR X2,[X3]
```

thread1 el2 handler

```
0x1400:
// smb recieve
LDR X4,[X5]
// smb reply
STR X6,[X7]
// kvm/hyp/include/nvhe/spinlock.h
LDAR X8,[X9]
// kvm/hyp/sysreg-sr.h:96
MSR TTBR0_EL1,X10
// include/asm/kvm_mmu.h:276
MSR VTTBR_EL2,X11
// kvm/hyp/nvhe/host.S
ERET
```

Final state: `0:R4=2 & 1:R0=0 & 1:R4=1 & 1:R8=0 & 1:R2=0`

Figure 2

4.1.0.4 pKVM.vcpu_run.same_vm :

For another example, for pKVM to maintain the illusion that each vCPU is on its own core, the per-core state must be cleaned between running different vCPUs, including ensuring that translations for one vCPU are not cached and visible to another, even if they happen to be in the same VM (and using the same VMID) (`pKVM.vcpu_run.same_vm` in Figure 3).

AArch64 pKVM.vcpu_run.same_vm

Page table setup:

```
option default_tables = false;
physical pa1 pa2;
intermediate ipa1 ipa2;

s1table hyp_map 0x200000  {
    identity 0x1000 with code;
    x ↦ invalid;
}

s2table vm1_stage2 0x300000  {
    ipa1 ↦ pa1;
    ipa1 ?-> invalid;
    ipa2 ↦ invalid;
    ipa2 ?-> pa2;

    s1table vm1_stage1 0x260000 {
        x ↦ ipa1;
    }
}

s2table vm2_stage2 0x380000  {
    ipa1 ↦ invalid;
    ipa1 ?-> pa1;
    ipa2 ↦ pa2;
    ipa2 ?-> invalid;

    s1table vm2_stage1 0x2C0000 {
        x ↦ ipa2;
    }
}

*pa2 = 1;
```

Initial state:

```
R3=x
R5=ttbr(base=0b0,vmid=0x0000)
TTBR0_EL1=ttbr(base=vm1_stage1,asid=0x0000)
SPSR_EL2=0b00101
TTBR0_EL2=ttbr(asid=0x0000,base=hyp_map)
ELR_EL2=L0:
R1=ttbr(vmid=0x0001,base=vm2_stage2)
R0=ttbr(asid=0x0000,base=vm2_stage1)
R4=ttbr(base=0b0,vmid=0x0001)
VBAR_EL2=0x1000
VTTBR_EL2=ttbr(vmid=0x0001,base=vm1_stage2)
PSTATE.EL=0b10
```

Thread 0

```
// arm64/include/asm/kvm_mmu.h:276
MSR VTTBR_EL2,X4
// kvm/hyp/nvhe/tlb.c:43
isb
// kvm/hyp/nvhe/tlb.c:135
tlbi vmalle1
// kvm/hyp/nvhe/tlb.c:137
dsb nsh
// kvm/hyp/nvhe/tlb.c:138
isb
// arm64/include/asm/kvm_mmu.h:276
MSR VTTBR_EL2,X5
// kvm/hyp/nvhe/tlb.c:52
isb
// kvm/hyp/sysreg-sr.h:96
MSR TTBR0_EL1,X0
// include/asm/kvm_mmu.h:276
MSR VTTBR_EL2,X1
// kvm/hyp/nvhe/host.S
ERET
L0:
// in guest
LDR X2,[X3]
```

thread0 el2 handler

```
0x1200:
mov x2, #0
// data abort preferred-return-address is itself
// so jump to next instr instead
mrs x20,elr_el2
add x20,x20,#4
msr elr_el2,x20
eret
```

Final state: `0:R2=0`

Figure 3

4.2 Data Aborts

As mentioned earlier, a vCPU accessing a location that it does not have permissions for or which is not mapped, results in an Abort which is handled at EL2.

4.2.0.1 pKVM.host_handle_trap.stage2_idmap.l3 : If the vCPU accesses a location which is currently un-mapped, but should be mapped on-demand, then pKVM will make a new mapping for that vCPU, install it into its VM's Stage 2 translation tables, and return to the vCPU to re-try the access (Figures 4 and 5). This action typically requires no TLB invalidation, as the previously unmapped entries could not have been stored in any TLB.

An added complexity here is that if pKVM wishes to map only a single page then it *must* install this mapping with a Level 3 entry. If the mapping is currently invalid at Level 2 or Level 1 then care must be taken to not create entries out-of-order. pKVM manages this by producing a fresh table of invalid entries first, then installing that new table into the translation tables, and recursing down until it reaches the level it needs to install the valid mapping at. Otherwise, a concurrent translation could see the new table entries before the leaf entries had been installed into the table.

AArch64 pKVM.host_handle_trap.stage2_idmap.l3

Page table setup:

```
option default_tables = false;
physical pa1;
intermediate ipa1;

s2table vm_stage2 0x260000 {
    ipa1 ↦ invalid;
    ipa1 ?-> pa1;

    s1table host_s1 0x2C0000 {
        x ↦ ipa1;
    }
}

s1table hyp_map 0x200000 {
    x ↦ invalid;
    s1table host_s1;
    s2table vm_stage2;
    identity 0x1000 with code;
}

*pa1 = 1;
```

Initial state:

```
R1=x
R12=pte3(ipa1,vm_stage2)
R3=x
TTBR0_EL1=ttbr(asid=0x0000,base=host_s1)
TTBR0_EL2=ttbr(base=hyp_map,asid=0x0000)
R11=mkdesc3(oa=pa1)
R10=0b0
PSTATE.EL=0b01
VTTBR_EL2=ttbr(base=vm_stage2,vmid=0x0000)
VBAR_EL2=0x1000
```

Thread 0

```
LDR X0,[X1]
LDR X2,[X3]
```

thread0 el2 handler

```
0x1400:
// count number of exceptions
add x10,x10,#1
// remember which IPA failed
mrs x9,hpfar_el2
lsl x9,x9,#8
// pkvm code
stlr x11,[x12]
dsb ishst
1:
// return to next instruction
// pKVM doesn't really do this, it just tries the same instr again
// but without this the test can loop forever ...
mrs x20,elr_el2
add x20,x20,#4
msr elr_el2,x20
// return from handle_trap
eret
```

Final state: `0:R10=2`

Figure 4: `pKVM.host_handle_trap.stage2_idmap.l3`: code listing

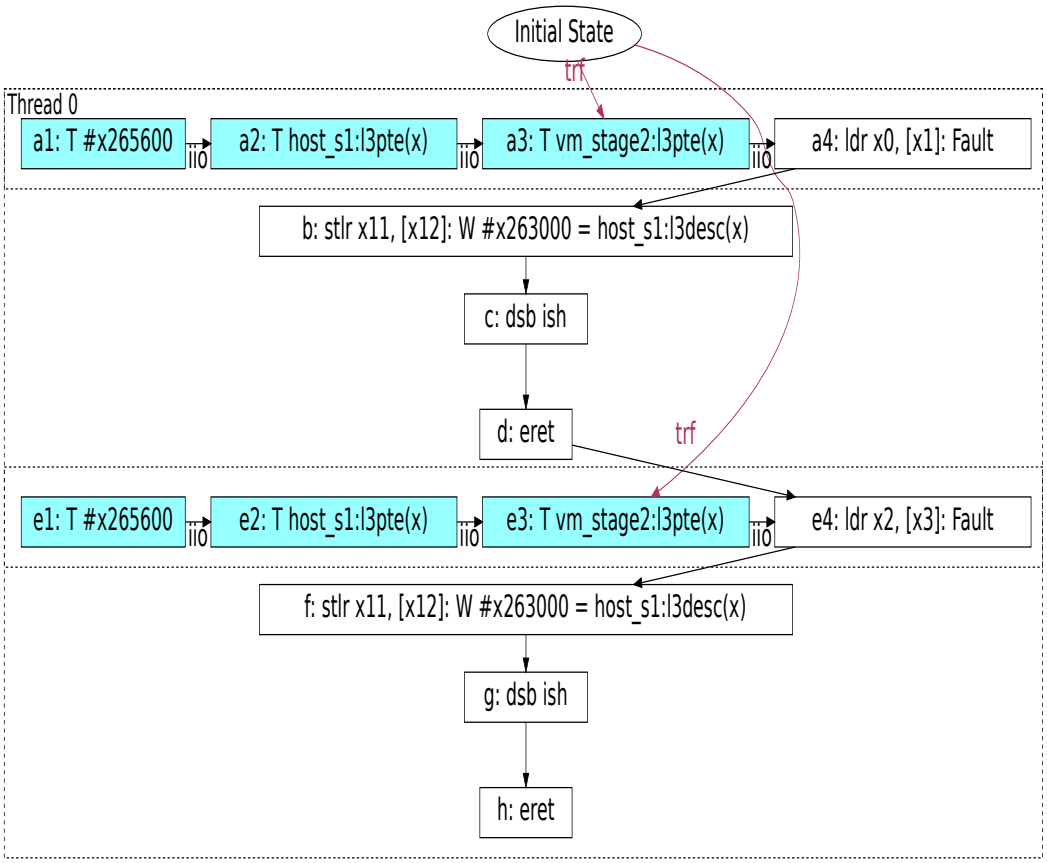


Figure 5: `pKVM.host_handle_trap.stage2_idmap.l3`: forbidden candidate execution

4.2.0.2 pKVM.host_handle_trap.stage2_idmap.already_exists : If the vCPU tries to access a location that is mapped but that it lacks the necessary permissions for, then there may be an existing entry already mapping that location. Here, the process is much more delicate. Because another vCPU in the same VM may be concurrently accessing the same physical locations mapped by the shared Stage 2 table, pKVM cannot assume its own internal locks are sufficient to prevent race conditions. Therefore, this is one case where pKVM is *required* to use the break-before-make sequence as described by §3.2.

In the following code (Figures 6 and 7), the initial state starts out with `x` mapped at level 3, but without read permissions. The first load will fault and pKVM will naturally map a writeable mapping on-demand, following the break-before-make sequence invalidating the entry by writing zero (event `b`) before writing a new valid descriptor (event `m`). After returning to the guest, we ask whether loads of `x` are allowed to fault at Stage 2.

AArch64 pKVM.host_handle_trap.stage2_idmap.l3.already_exists

Page table setup:

```
option default_tables = false;
physical pa1 pa2;
intermediate ipa1;

s2table vm_stage2 0x260000  {
    ipa1 ↦ pa1 with [AP=0b00] and default;
    ipa1 ?-> invalid;
    ipa1 ?-> pa2;

    s1table host_s1 0x2C0000 {
        x ↦ ipa1;
    }
}

s1table hyp_map 0x200000  {
    x ↦ invalid;
    s2table vm_stage2;
    identity 0x1000 with code;
}

*pa1 = 1;
*pa2 = 2;
```

Initial state:

```
R1=x
R16=ttbr(vmid=0x001,base=vm_stage2)
R17=mkdesc3(oa=pa2)
R0=0b0
PSTATE.EL=0b01
R18=pte3(ipa1,vm_stage2)
VTTBR_EL2=ttbr(vmid=0x001,base=vm_stage2)
R10=0b0
R15=page(ipa1)
R13=pte3(ipa1,vm_stage2)
R14=ttbr(vmid=0x001,base=0b0)
R12=0b0
R3=x
TTBR0_EL1=ttbr(asid=0x000,base=host_s1)
VBAR_EL2=0x1000
TTBR0_EL2=ttbr(asid=0x000,base=hyp_map)
```

Thread 0

```
STR X0,[X1]
LDR X2,[X3]
```

thread0 el2 handler

```
0x1400:
// count number of exceptions
add x10,x10,#1
// remember which IPA failed
mrs x9,hpfar_el2
// really pKVM would do a read of the pagetable to tell if it needs to update
// we elide that code from the test and just skip on the second fault
cmp x10,#2
b.eq 1f
// pkvm code
// pgtable.c:573
STR X12,[X13]
// tlb.c:63
DSB ISH
// tlb.c:66
MSR VTTBR_EL2,X14
// tlb.c:66
ISB
// tlb.c:74
TLBI IPAS2E1IS,X15
// tlb.c:82
DSB ISH
// tlb.c:83
TLBI VMALLE1IS
// tlb.c:84
DSB ISH
// tlb.c:85
ISB
// tlb.c:109
MSR VTTBR_EL2,X16
// tlb.c:109
ISB
STLR X17,[x18]
DSB ISHST
// return to next instruction
// pKVM doesn't really do this, it just tries the same instr again
// but without this the test can loop forever ...
1:
mrs x20,elr_el2
add x20,x20,#4
msr elr_el2,x20
// return from handle_trap
eret
```

Final state: `0:R10=2` 0:R2=1|

Figure 6: `pKVM.host_handle_trap.stage2_idmap.l3.already_exists`: code listing

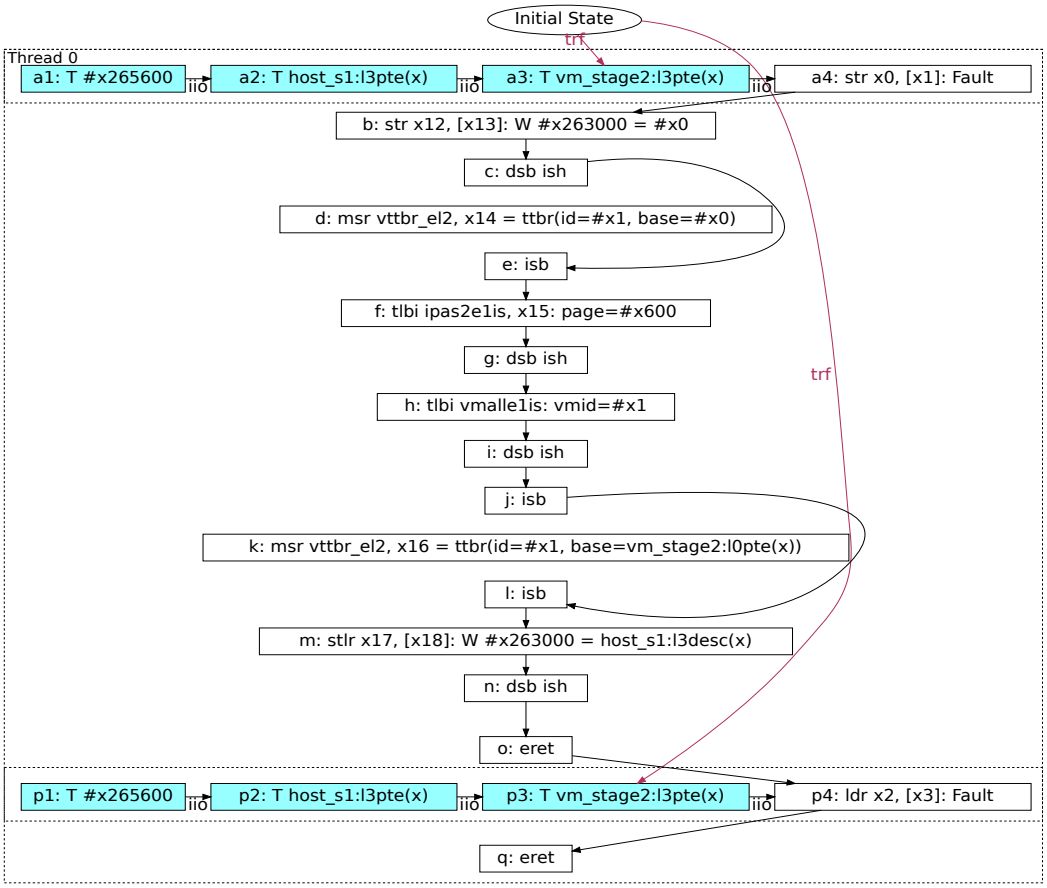


Figure 7: `pKVM.host_handle_trap.stage2_idmap.l3.already_exists`: forbidden candidate execution

4.2.0.3 pKVM.host_handle_trap.stage2_idmap.change_block_size : The complexity of the previous scenario is compounded by the fact that pKVM may wish to map a larger region (higher in the table) than is currently mapped, and, without `FEAT_BBM`, this adds extra break-before-make requirements.

In general, pKVM will itself perform a translation table walk. On the way down, it will look for the entry to be replaced, invalidate it, and perform TLB maintenance, ensuring that all entries from old leaf entries below it are cleaned away, but additionally that any old stale Stage 1 translations are invalidated, before it replaces the entry.

Figures 8 and 9 contain the code listing and the interesting candidate execution diagram (as generated by Isla). We consider a scenario where there are two locations, `x` and `y` which are in the same 2 MiB region of memory but are mapped by different Level 3 entries. If the Level 3 table contains all valid entries except for one entry for `x`, then on updating the entry for `x` to be valid, if the host kernel maps the whole 2M region, then pKVM will invalidate the 2M entry before writing a new 2M block entry.

This in effect 'promotes' the set of 4K mappings into a single 2M mapping. pKVM then frees the child table to be re-allocated later.

pKVM may want to *remove* a mapping for an IPA. This is very similar to the previously described break-before-make scenario, but without the final make. pKVM just has to ensure that the old IPA mapping is invalidated, and the necessary TLB maintenance is performed. We do not include this test at present.

AArch64 pKVM.host_handle_trap.stage2_idmap.change_block_size

Page table setup:

```
option default_tables = false;
physical pa1 pa2;
intermediate ipa1 ipa2;

s2table vm_stage2 0x260000  {
    ipa1 ↦ invalid at level 3;
    ipa2 ↦ pa2 at level 3;
    ipa1 ?-> pa1 at level 3;

    ipa1 ?-> invalid at level 2;
    ipa2 ?-> invalid at level 2;
    ipa2 ?-> pa2 at level 2;

    s1table host_s1 0x2C0000 {
        x ↦ ipa1;
        y ↦ ipa2;
    }
};

s1table hyp_map 0x200000  {
    x ↦ invalid;
    s2table vm_stage2;
    identity 0x1000 with code;
}

*pa1 = 1;
*pa2 = 2;
```

Initial state:

```
R15=ipa1
R13=pte2(ipa1,vm_stage2)
R17=mkdesc2(oa=pa2)
VTTBR_EL2=ttbr(vmid=0x0001,base=vm_stage2)
R12=0b0
R16=ttbr(vmid=0x001,base=vm_stage2)
R3=y
TTBR0_EL2=ttbr(base=hyp_map,asid=0x0000)
TTBR0_EL1=ttbr(base=host_s1,asid=0x0000)
R18=pte2(x,vm_stage2)
R14=ttbr(base=0b0,vmid=0x001)
R1=x
R10=0b0
VBAR_EL2=0x1000
PSTATE.EL=0b01
```

Thread 0

```
// in guest
MOV X0,#0
LDR X0,[X1]
MOV X2,#0
LDR X2,[X3]
```

thread0 el2 handler

```
0x1400:
// count number of exceptions
add x10,x10,#1
// remember which IPA failed
mrs x9,hpfar_el2
// on second fault just exit test
cmp x10,#2
b.eq 1f
// then pkvm code
0:
// pgtable.c:181
STR X12,[X13]
// tlb.c:116
DSB ISH
// tlb.c:119
MSR VTTBR_EL2,X14
// tlb.c:119
ISB
// tlb.c:121
TLBI vmalls12e1is
// tlb.c:122
DSB ISH
// tlb.c:123
ISB
// tlb.c:125
MSR VTTBR_EL2,X16
// tlb.c:125
ISB
STLR X17,[x18]
DSB ISHST
1:
// return to next instruction
// pKVM doesn't really do this, it just tries the same instr again
// but without this the test can loop forever ...
mrs x20,elr_el2
add x20,x20,#4
msr elr_el2,x20
// return from handle_trap
eret
```

Final state: `0:R10=2` 0:R2=1

Figure 8: `pKVM.host_handle_trap.stage2_idmap.change_block_size`: code listing

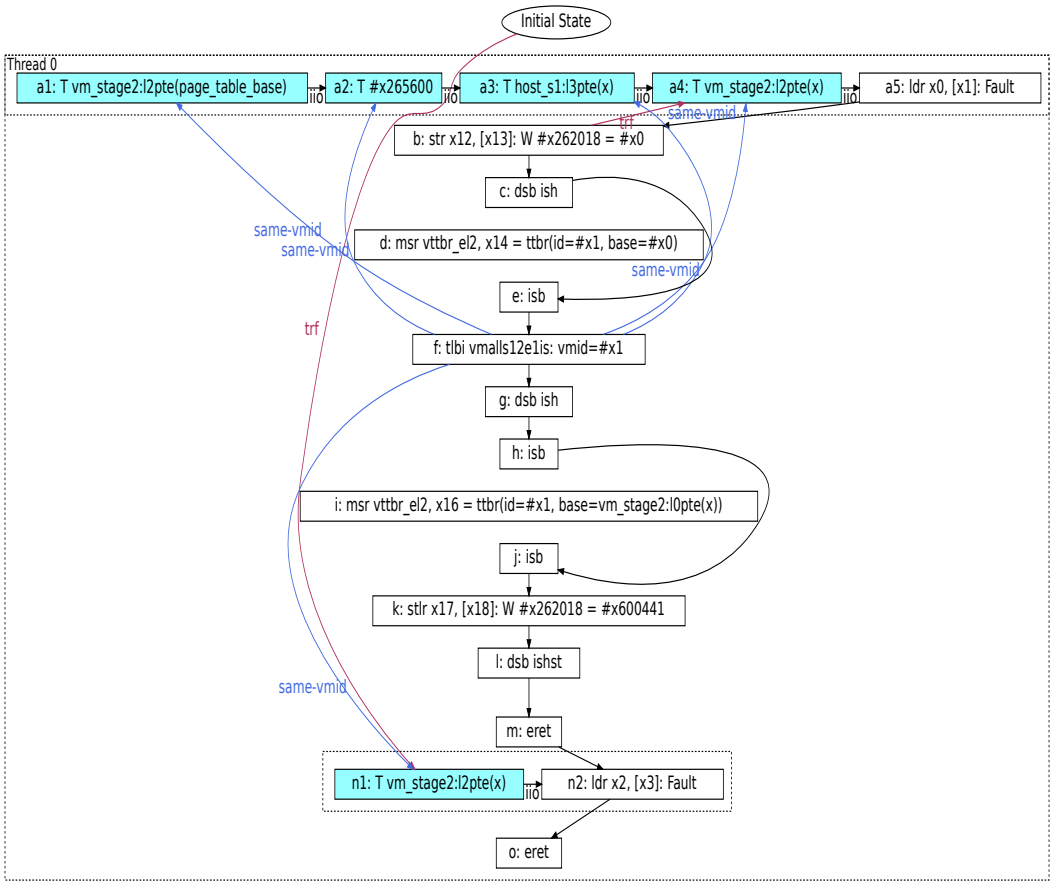


Figure 9: `pKVM.host_handle_trap.stage2_idmap.change_block_size`: forbidden candidate execution

4.3 Initialisation

During execution, and especially at initial start-up of pKVM, it will be required to create its own Stage 1 translation tables.

Currently, this only happens on a single core, so there are no interesting concurrent cases.

In the future, however, it is expected that pKVM may need to dynamically map some parts of memory, as the hypervisor gains richer features. For example, it may need to map some shared page between guests, or between pKVM and a guest, in which to pass messages.

4.3.0.1 pKVM.switch_to_new_table :

When pKVM first starts, it is using translation tables set up by Linux, so one of the first things pKVM does is to create its own tables and switch to them.

Usually, switching from using one translation table to another happens at a higher exception level, and then the new one is not used until the return to the lower exception level. The current case is more complicated, as pKVM has to change its own translation tables *while it is executing*. If pKVM only had to change the `TTBR`, then this would not be a problem, but there are many system registers involved in configuring the translation tables (the `TCR`, `MAIR`, and so on), and these registers cannot all be updated 'atomically'.

To maintain the atomicity, pKVM switches the `TTBR` to a page of memory where the code it is executing is identity-mapped, then it disables translation (disabling the MMU), before updating all of the required system registers (including the new `TTBR`), before re-enabling the MMU.

Figure 10 contains the code for a litmus test that tries to capture the core of this process. This `pKVM.switch_to_new_table` test is an EL2 Stage 1 test with two tables, `hyp_pgtable` and `new_hyp_pgtable`, in memory. The code disables the MMU (by writing to the appropriate field of the SCTLR), invalidates all of the old cached TLB entries for EL2, updates all the system registers (only the `TTBR` is included here) and then re-enables the MMU. We check that the final load in the hypervisor after the switch has happened reads using the new state.

This test has been cut down for brevity to remove the writes of other system registers, which would be present in a full execution of pKVM; ideally, we would have a test that included those too, and a final state which ensured that the load used a translation using all of the new register values.

Note that the model presented in Section 5 does not currently contain axioms for when the MMU is disabled, but the semantics seem clear, and we do not see any impediment in extending the model to handle it fully.

AArch64 pKVM.switch_to_new_table

Page table setup:

```
option default_tables = false;
physical pa1;

s1table hyp_pgtable 0x200000  {
    x ↦ invalid at level 3;
    x ?-> pa1;
    identity 0x2000 with code;
}

s1table new_hyp_pgtable 0x240000  {
    x ↦ pa1 at level 3;
    x ?-> invalid;
    identity 0x2000 with code;
}

*pa1 = 1;
```

Initial state:

```
R6=x
R4=ttbr(asid=0x0000,base=new_hyp_pgtable)
VBAR_EL2=0x2000
PSTATE.SP=0b1
PSTATE.EL=0b10
TTBR0_EL2=ttbr(base=hyp_pgtable,asid=0x0000)
```

Thread 0

```
// hyp-init.S:247
mrs x2, sctlr_el2
bic x3, x2, #1
msr sctlr_el2, x3
isb
tlbi alle2
msr ttbr0_el2, x4
msr sctlr_el2, x2
isb
ldr x5,[x6]
```

thread0 el2 handler

```
0x2200:
mov x5, #0
```

Final state: `0:R2=0`

Figure 10: pKVM.switch_to_new_table

4.3.0.2 pKVM.create_hyp_mappings.inv.l2 :

Constructing new translation tables is done incrementally, starting from a single zero'd page of memory as the root table, and then performing a manual translation table walk on insertion to locate an entry to insert into.

In the case where there is no Level 3 table to install into, pKVM first creates a Level 2 table and installs that, before writing the new valid Level 3 entry. To avoid break-before-make violations here, pKVM always ensures the table is zeroed before inserting it into the table.

The `pKVM.create_hyp_mappings.inv.l2` test, in Figures 11 and 12, gives the case where pKVM is trying to create a new mapping for itself for a 4K page block. This block mapping *must* be installed as a Level 3 entry, as installing it any higher would end up mapping more than 4K of memory. Initially, `x` is translated using an invalid Level 2 entry in `hyp_pgtable`. The table `hyp_pgtable_new` contains the new Level 3 table which starts life zeroed (all invalid); The test then sets the Level 2 entry in `hyp_pgtable` to point to the new table, and then updates the Level 3 (leaf) entry with a valid descriptor.

Interestingly, we note that pKVM sets these entries with a store-release; although there seems to be no relaxed-virtual-memory reason why. Given that pKVM is well locked, it is not clear why making these writes store-releases helps.

AArch64 pKVM.create_hyp_mappings.inv.l2

Page table setup:

```
option default_tables = false;
physical pa1;

s1table hyp_pgtable_new 0x280000 {
    x ↦ invalid at level 3;
    x ?-> pa1 at level 3;
}

s1table hyp_pgtable 0x200000  {
    x ↦ invalid at level 2;
    x ?-> table(0x283000) at level 2;
    identity 0x1000 with code;
    s1table hyp_pgtable_new;
}

*pa1 = 1;
```

Initial state:

```
R0=mkdesc2(table=0x283000)
R2=mkdesc3(oa=pa1)
PSTATE.SP=0b1
R1=pte2(x,hyp_pgtable)
R5=x
PSTATE.EL=0b10
TTBR0_EL2=ttbr(asid=0x0000,base=hyp_pgtable)
VBAR_EL2=0x1000
R3=bvor(0x283000,offset(level=3,va=x))
```

Thread 0

```
STLR X0,[X1]
STLR X2,[X3]
DSB SY
ISB
L0:
LDR X4,[X5]
```

thread0 el2 handler

```
0x1200:
mov x2, #0
mrs x20,ELR_EL2
add x20,x20,#4
msr ELR_EL2,x20
eret
```

Final state: `0:R2=0`

Figure 11: `pKVM.create_hyp_mappings.inv.l2`: Code listing

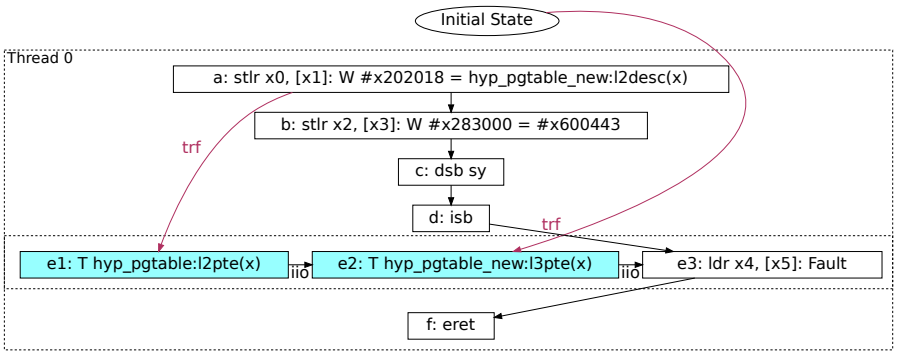


Figure 12: `pKVM.create_hyp_mappings.inv.l2`: (forbidden) candidate execution

4.3.0.3 pKVM.create_hyp_mappings.inv.l3 :

pKVM can set entries at any level of the table, assuming they are initially invalid. The following test is a variation on the previous, where the Level 3 table is already created, but contains an invalid entry at Level 3.

As before, we will map a 4K region, and so it must go at Level 3.

AArch64 pKVM.create_hyp_mappings.inv.l3

Page table setup:

```
option default_tables = false;

physical pa1;

s1table hyp_pgtable 0x200000  {
    x ↦ invalid at level 3;
    x ?-> pa1;
    identity 0x1000 with code;
}

*pa1 = 1;
```

Initial state:

```
R0=mkdesc3(oa=pa1)
TTBR0_EL2=ttbr(base=hyp_pgtable,asid=0x0000)
R3=x
VBAR_EL2=0x1000
PSTATE.EL=0b10
R1=pte3(x,hyp_pgtable)
```

Thread 0

```
STLR X0,[X1]
DSB SY
ISB
L0:
LDR X2,[X3]
```

thread0 el2 handler

```
0x1000:
mov x2, #0
```

Final state: `0:R2=0`

Figure 13: `pKVM.create_hyp_mappings.inv.l3`: Code listing

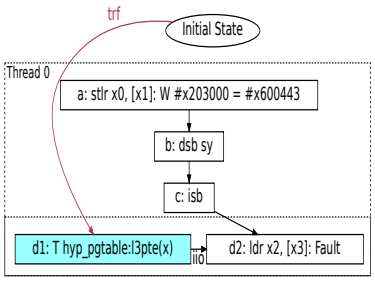


Figure 14: `pKVM.create_hyp_mappings.inv.l3`: (forbidden) candidate execution

5 Model

We now define a semantic model for Armv8-A relaxed virtual memory that, to the best of our knowledge, captures the Arm architectural intent for the scope laid out in §1 and discussed in §3, including Stage 1 and Stage 2 translation-table walks and the required TLB maintenance. For some important questions, most notably for multi-copy atomicity, the Arm intent is currently tentative, so it is not possible to be more definitive. To capture just the synchronization required for "simple" software such as pKVM to work correctly we also give a *weaker* model (in App. B): instead of trying to exactly capture the architecture or the behaviour of hardware, it has individual axioms for each behaviour that such software needs to rely on. This gives an over-approximation to the architecture, which we prove sound with respect to the model given in this section. The two models together delimit the design space as we understand it.

In §3 and §4 we described the design issues in microarchitectural terms, discussing the behaviour of TLB caching and translation-walk non-TLB reads, along with the needs of system software. We now abstract from microarchitecture: instead of explicitly modelling TLBs we include a translation-read event for each read performed by the architected translation-table walker, and make those reads read-from writes in the execution (so there are no special 'pagetable write' events). We give the model in an axiomatic Herd-like [9] style, as an extension to the base Armv8-A semantics [26, 49, 13]. In principle it would be desirable to also have equivalent abstract-microarchitectural operational models, as for base Armv8-A [49, 48] but with explicit TLBs for each thread and events for reading from and into the TLB. However, address translation introduces many more events to litmus-test executions, which would make them harder to explore exhaustively, and a proof of equivalence would be a major undertaking, so we leave this to future work.

The base Armv8-A axiomatic model is defined as a predicate over *candidate executions*, each of which is a graph with various events (reads, writes, barriers) and relations over them, notably the per-thread program order `po`, the per-location coherence order `co`, the reads-from relation `rf` from writes to reads, the `addr`, `data`, and `ctrl`-dependency subsets of `po`, and others. These candidates may be arbitrarily inconsistent graphs, possibly containing executions that can never happen.

The model is then a per-candidate consistency check consisting of two parts: that the graph corresponds to *some* execution consistent with the underlying ISA, but with arbitrary memory reads and writes; and a global consistency check over those reads and writes which enforces memory consistency properties such as *coherence*.

The base memory consistency model is essentially the conjunction of two acyclicity checks with an emptiness check for atomics: an `external` (inter-thread) acyclicity property, effectively stating that the execution must respect some total order of events hitting the shared memory, constrained by the derived ordered-before (`ob`) relation; and an `internal` acyclicity property, enforcing per-location coherence; and an `atomic` axiom for atomic and exclusive operations.

As usual in Herd-style models, relations are suffixed `e` or `i` to restrict to their inter-thread or intra-thread parts. The Herd concrete syntax for relational algebra uses `[X]` for the identity on a set `X`, `;` for composition, `~` for complement, `|` and `&` for union and intersection, and $*$ for product. To extend this base memory consistency model to the world with translations and TLBs we add translation data to events, including virtual, intermediate physical, and physical addresses (as determined by the translation regime).

We add the following events and relations:

- **`T`** for reads originating from architected translation-table walks.
  These roughly correspond to the actual satisfaction from memory which with TLBs may happen very early.

- **`TLBI`** events for each `TLBI` instruction, with a single such event per `TLBI` instruction, corresponding to the `TLBI` being completed on all relevant cores.

- **TE** and **ERET** events for taking and returning from an exception (these might not correspond to changes in exception level).
- **MSR** events for writes to relevant system registers, such as the TTBR.
- **DSB** events for DSB instructions.
- trf, tfr relations as analogues to rf and fr but for translation-read events (**T**s).
- iio relation ("intra-instruction order") which relates events of the same instruction in the order they occur during execution of that instruction's intra-instruction semantics as defined by the Arm ASL.
- same-va, same-ipa, same-pa relations which relate events whose virtual, intermediate physical or physical address of the associated explicit memory access are the same.
- same-va-page, same-ipa-page, same-pa-page which relate events whose associated explicit memory events are in the same page (e.g. 4KiB chunk) of the virtual, intermediate physical or physical address space.
- same-asid, same-vmid relates events for which translations for the associated memory event are using the same ASID or VMID.

In addition we modify some existing events and relations:

- **R**, **W** events are now to a physical location.
- loc and co both relate events which are to the same physical address.
- addr which is derived from a new tdata relation, which relates the event which provide the input address for a translation.
- We re-arrange the barrier events into a hierarchy which includes DSBs, see Figure 17.

For convenience we define new event sets: **C** for all cache-maintenance operations (DC, IC, and TLBI instructions); **T_f** for all translation-read events which read a descriptor which causes a translation fault; **W_inv** for all the write events which write an invalid descriptor; **Stage1** and **Stage2** for the **T** events which originate from the respective stage of translation; **ContextChange** for all context-changing events (such as writes to translation-controlling system registers); and **CSE** for all context-synchronizing events (taking and returning from exceptions and ISB).

5.1 Strong model

The model is given in full in Figure 15 with auxiliary definitions of the tlb-affects and barrier hierarchy given in Figures {16,17}.

Its basic form is very similar to previous multicopy-atomic Armv8-A models. It still has external, internal, and atomic axioms, to which we add a translation-internal axiom for ensuring translations do not read from po-later writes.

Most of the changes to the model are in the external axiom, where we add several relations to ordered-before (ob): iio orders the intra-instruction events as ordered by the ASL; tob ("translation ordered-before") ensures the order arising from the act of translation itself is respected; obtlbi orders translates and their explicit memory events with TLBIs which affect these translations; and ctxob ("context ordered-before") orders events which must come before some context-changing operation or after some context-synchronizing operation. We also add a generalised coherence-order relation, wco, an existentially quantified total order expressing when TLBIs complete w.r.t. writes.

```
let tlb-affects =
  see Figure 16

let TLB_barrier =
  ([TLBI] ; tlb-affects ; [T] ; tfr ; [W])^-1
  & wco

let maybe_TLB_cached =
  ([T] ; trf^-1 ; wco ; [TLBI-S1]) & tlb-affects^-1

let tcache1 = [T & Stage1] ; tfr ; TLB_barrier
let tcache2 = [T & Stage2] ; tfr ; TLB_barrier

let speculative =
    ctrl
  | addr; po
  | [T] ; instruction-order
(* translation-ordered-before *)
let tob =
    [T_f] ; tfre
  | ([T_f] ; tfri)
    & (po ; [DSB.SY] ; instruction-order)^-1
  | [T] ; iio ; [R|W] ; po ; [W]
  | speculative ; trfi
(* observed by *)
let obs = rfe | fr | wco
  | trfe
(* ordered-before TLBI and translate *)
let obtlbi_translate =
    tcache1
  | tcache2
    & (iio^-1 ; [T & Stage1] ; trf^-1 ; wco^-1)
  | (tcache2 ; wco? ; [TLBI-S1])
    & (iio^-1 ; [T & Stage1] ; maybe_TLB_cached)

(* ordered-before TLBI *)
let obtlbi =
    obtlbi_translate
  | [R|W|Fault] ; iio^-1 ; (obtlbi_translate & ext)
    ; [TLBI]

(* context-change ordered-before *)
let ctxob =
    speculative ; [MSR]
  | [CSE] ; instruction-order
  | [ContextChange] ; po ; [CSE]
  | speculative ; [CSE]
  | po ; [ERET] ; instruction-order ; [T]

(* ordered-before a translation fault *)
let obfault =
    data ; [Fault & IsFromW]
  | speculative ; [Fault & IsFromW]
  | [dmbst] ; po ; [Fault & IsFromW]
  | [dmbld] ; po ; [Fault & (IsFromW|IsFromR)]
  | [A|Q] ; po ; [Fault & (IsFromW | IsFromR)]
  | [R|W] ; po ; [Fault & IsFromW & IsReleaseW]

(* ETS-ordered-before *)
let obETS =
    (obfault ; [Fault]) ; iio^-1 ; [T_f]
  | ([TLBI] ; po ; [dsb] ; instruction-order ; [T])
    & tlb-affects

(* dependency-ordered-before *)
let dob =
    addr | data
  | speculative ; [W]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | data); trfi

(* atomic-ordered-before *)
let aob = rmw
  | [range(rmw)]; rfi; [A | Q]

(* barrier-ordered-before *)
let bob = [R] ; po ; [dmbld]
  | [W] ; po ; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb] ; po

(* Ordered-before *)
let ob = (obs | dob | aob | bob
  | iio | tob | obtlbi | ctxob | obfault | obETS)^+

(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal
(* External visibility requirement *)
irreflexive ob as external
(* Atomic requirement *)
empty rmw & (fre; coe) as atomic
(* Writes cannot forward to po-future translates *)
acyclic (po-pa | trfi) as translation-internal
```

Figure 15: Strong Model (with baseline Armv8-A model parts in gray)

```
let tlb_might_affect =
    [ TLBI-S1 & ~TLBI-S2 &  TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-va-page & same-asid & same-vmid)
    ; [T & Stage1]
  | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-asid & same-vmid) ; [T & Stage1]
  | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  & ~TLBI-ASID & TLBI-VMID] ; same-vmid ; [T & Stage1]
  | [~TLBI-S1 &  TLBI-S2 &  TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; (same-ipa-page & same-vmid) ; [T &
    Stage2]
  | [~TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid ; [T & Stage2]
  | [ TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid ; [T]
  | ( TLBI-S1 &           ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T & Stage1)
  | (           TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T & Stage2)

let tlb-affects =
    [TLBI-IS] ; tlb_might_affect
  | ([~TLBI-IS] ; tlb_might_affect) & int
```

Figure 16: The `tlb-affects` relation.

Coherence: By making `loc` (and therefore `rf` and `co`) relate events with the same physical addresses, we get coherence over physical addresses rather than virtual, and all the previously allowed shapes are also allowed when there is aliasing with different virtual addresses. Coherence

```
let dsbsy = DSB.ISH | DSB.SY | DSB.NSH
let dsbst = dsbsy | DSB.ST | DSB.ISHST | DSB.NSHST
let dsbld = dsbsy | DSB.LD | DSB.ISHLD | DSB.NSHLD
let dsbnsh = DSB.NSH
let dmbsy = dsbsy | DMB.SY
let dmbst = dmbsy | dsbst | DMB.ST | DSB.ST | DSB.ISHST | DSB.NSHST
let dmbld = dmbsy | dsbld | DMB.LD | DSB.ISHLD | DSB.NSHLD
let dmb = dmbsy | dmbst | dmbld
let dsb = dsbsy | dsbst | dsbld
```

Figure 17: Barrier definitions.
Note we do not distinguish between Inner-Shareable and Full-System barriers

of writes to translation tables is expressed in two places: including `trfe` in `obs` captures the fact that translation-table reads from memory microarchitecturally come from the 'flat' coherent storage subsystem, and so the writes that they read from must have been propagated before the translation happened; and the `translation-internal` axiom forbids forwarding against program-order. Note that including only `trfe` allows forwarding locally (a `trfi` edge), and including `(addr|data);trfi` in `dob` ensures those forwarded writes cannot form bad self-satisfying cycles.

TLB maintenance and break-before-make: The `obtlbi` relation ensures that instructions whose translations read from writes which are "hidden" by some `TLBI` instruction are ordered before the completion of that `TLBI`. This is achieved by the two clauses of `obtlbi`: the first clause ensures the translation-before-TLBI ordering is preserved, and the second clause orders the explicit memory access of any such instruction with the same `TLBI` as the first clause. To do this, the model computes the set of writes which are in effect "barriered" by a given `TLBI` instruction, by looking at all translations in the execution, and if any translation reads-from a write which is before a `TLBI`, we then get `TLB_barrier` between them. The `tcache1` and `tcache2` relations then simply relate translations which read from coherence-predecessors of any of those writes with their respective barriering `TLBI`.

To accurately match up each of the various `TLBI` instructions with the translations they may affect, we define a `tlb-affects` relation which relates `TLBI` events with the `T` events they are relevant to. Its definition uses sets `TLBI-VA`, `TLBI-ASID`, `TLBI-IPA`, `TLBI-VMID`, and `TLBI-ALL` for each of the categories of `TLBI` instruction. Note that some instructions can fall into multiple categories, such as `TLBI VAE1` which is in `TLBI-VA` for the specified virtual address, `TLBI-ASID`, as the register input contains an ASID to perform the invalidation for, and also `TLBI-VMID` as the invalidations only affect translations in the same VM.

We add `obtlbi_translate` to relate those translations to `TLBI`s which invalidate the writes they read from. For Stage 1 translations we can simply order any Stage 1 translation before any `TLBI` which would `tlb_affect` this translation where the translation reads from a write which is ordered-before than the `TLBI`. However, for Stage 2 translations this is not sufficient. Recall that microarchitecturally the TLB could store whole virtual-to-physical mappings, and so a Stage 2 translation-read is only ordered after the `TLBI`s which remove not only any Stage 2 mappings but also those that would remove the combined Stage 1 and Stage 2 mappings. For a Stage 2 translation whose previous Stage 1 walk only read from writes newer than the `TLBI` then the Stage 2 invalidation is sufficient. But where any of the reads read-from a write older than the `TLBI`, a cached virtual-to-physical mapping could exist and Stage 1 invalidation is required, hence the Stage 2 translation is ordered after the *Stage 1* invalidation.

Translation-table-walk reading from memory: As noted in §3.3, a translation which results in a translation fault must read from memory or be forwarded from program-order earlier instructions, and those memory reads behave *multi-copy atomically*. In general the only time the model can guarantee that such a memory read happens is when the read results in a translation fault, since entries that result in a translation fault cannot be stored in the TLB (§3.2). The

model captures this succinctly by including `[T_f];tfr` in `ob`.

In general, a translation-read is ordered after the write which it reads from, as captured by the inclusion of the `trfe` edge in `ob`; this is strong enough to ensure that TLB fills and faulting memory walks pull values out of the memory system in a coherent way, but still weak enough to allow *other*-multi-copy-atomic behaviour such as forwarding.

As discussed in §3.3, a `DSB` ensures that writes are propagated out to memory. For translations this amounts to ensuring that a faulting translation cannot read-from something older than a po-previous `DSB`-barriered write, as captured by the last edge in `tob` which says that a `tfri` edge from such a faulting translation must not have an interposing `DSB`.

Note that the absence of the full `tfr` relation in `ob` for non-faulting translations intentionally allows some incoherence, in essence allowing a translation-read to "ignore" a newer write.

Context-changing operations: In general, the sequential semantics takes care of the context, such as current base register and system register state, for us. The `ctxob` relation simply ensures that such context-changing operations cannot be taken speculatively, and that context-synchronization ensures that all po-previous context-changing operations are ordered-before po-later translations.

Detecting BBM Violations: As discussed in §3.2, we do not model in detail the bounded-catch-fire semantics that currently architecturally results from a missing break-before-make sequence, as that would make it hard to enumerate possible litmus-test executions. Instead, because what one normally wants to know for litmus tests is that a test does not exhibit a BBM failure, we conservatively detect the existence of such violations and flag them for the user. This is achieved through a per-candidate-execute predicate, written in SMT, which looks for a situation which *could* be a break-before-make violation. It does this by asserting that there does not exist a pair of writes which conflict such that there is no interposing break-and-`TLBI` sequence. This approach is slightly over-approximate, as it might look for two writes that technically conflict even if they (for other reasons) are not used at the same time. This means that while we support programs that switch from one page table to another, we do not support programs that garbage collect page-table memory and then repurpose it.

ETS: We discussed the Armv8-A optional ETS feature, providing additional ordering strength for translations, in §3.3,3.3. The intuition is that the model would have *ghost* events in the event an instruction faults, to represent the explicit read or write which would have happened had the instruction not faulted. The model would then have to compute a special variant of `ob` including such dependencies, but without the physical-address-dependent relations such as `loc`, `rf` and `co`. Then any edge in the version of `ob` with the ghost events would become an edge in the real `ob` but attached to the faulting translation. To capture this, our model produces fault events which have the correct dependencies (and fault information) and the model orders the fault event with respect to program-order previous events which would have ordered and place those into `ob`. To achieve this, we manually insert all edges from the syntactic subsets (those edges which do not rely on `loc`) from `bob` and `dob` into a `obfault` relation. We use this to build an `obETS` relation which then orders translations that result in a translation fault after anything the fault is ordered-after.

An additional complexity here is for thread-local behaviours of `TLBI` instructions. With ETS one does not require context synchronization to see the effect of a `TLBI` thread-locally. Our `obETS` covers this with its second clause which orders translations from instructions `po`-after a subsequent `DSB` as happening after any `TLBI` which affected that translation.

Reclaimation of pagetable memory: There may be cases where the memory being used to store a translation table may become unreachable by any TTBR and properly cleaned from the TLBs. In practice this means the memory can now be reclaimed and re-purposed.

Allowing this is work-in-progress but the model as presented here does not support it and our break-before-make-violation detection predicate will assume that this is a break-before-make violation.

5.2 Weak Model

Relaxed memory model design for hardware architectures has to resolve a three-way tension between providing enough strength for software (forbidding enough behaviours so that code works as desired without needing excessive synchronisation), weakness for hardware (rendering desirable microarchitectural optimisations sound), and simplicity. For "user" concurrency, one has to accommodate the broad space of concurrent code in the wild, which is hard to map, but systems concurrency is managed by much smaller bodies of code, in more specific ways. This makes it interesting to explore models which are as weak as possible, subject to the constraints from system software such as pKVM.

We define such a model by capturing just the requirements we identified from pKVM usage (§4), expressing them as additional axioms over the Armv8-A base model: coherence over physical memory; no self-satisfying translations or translations using speculative writes; 'breaking' a translation with an invalidation and a broadcast `TLBI` ensuring that all cores have finished using that translation before the `TLBI` returns; writing a new entry to a broken page without `TLB` maintenance; and changing translation tables and context without TLB maintenance.

This weak model uses the same candidates and auxiliary definitions as the strong, but instead of including extra edges in `ob` we impose new axioms for each of those behaviours. For example, for the 'break' part of Stage 1 break-before-make we have:

```
empty ([W] ; co ; [W_invalid] ; ob ; [dsb.sy] ; po
  ; ([TLBI-S1] ; po ; [dsb.sy] ; ob ; [CSE] ; instruction-order ; [T]) & tlb_affects
  ) & trf
  as brk2
```

This forbids the case where a write is read-from by a translation-table-walk when there is an interposing break and Stage 1 TLB-invalidation sequence, which would 'hide' that write from future translations. Note the `ob` edges allow the sequence to be split over multiple threads in the context-switching scheduler case described in §3.3.

Note that the pKVM developers believe that pKVM does not rely on the ETS feature, and so the weak model does not include ETS.

6 Metatheory: relationships between models

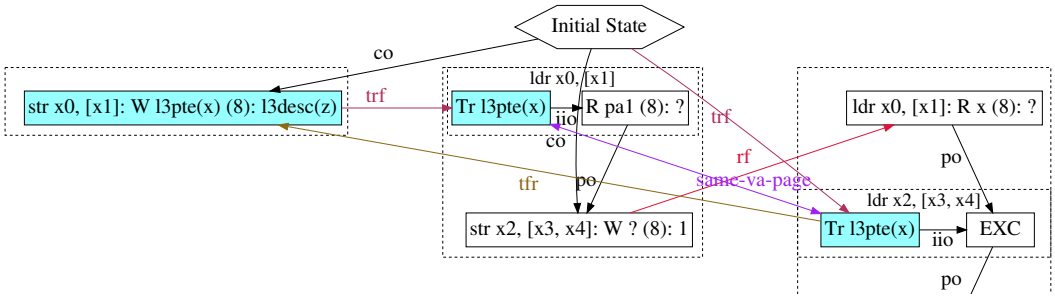
The virtual memory mechanisms are complex in both sequential and concurrent ways, as we have seen, but they are intended to let system software provide a relatively simple abstraction to higher-level code. As first steps towards establishing this, and as sanity checks of our models, we prove three theorems: that for static injectively-mapped address spaces, any execution which is consistent in the model with translation, erasing translation events gives an execution that is consistent in the original Armv8-A model without translation; that for any consistent execution in the original Armv8-A model, there is a corresponding consistent execution in our extended model with translations; and that our weak model is a sound over-approximation of our full translation model, i.e., that for any consistent execution in our full translation model, that same execution is consistent in the weak translation model. Details are in App. C.

7 Isla-based model evaluation

Making relaxed-memory semantics exhaustively executable is essential for exploring their behaviour on examples [67, 54, 53, 20, 9, 36, 66, 23, 64, 49, 57]. Handling relaxed virtual memory

brings several new challenges. First, even just the sequential definition of Armv8-A address translation, with the page-table walk and its options, is remarkably intricate, defined in thousands of lines of Arm's ASL instruction description language. Manually reimplementing a simplified version would be error-prone and incomplete, so we instead build on our Isla tool [15], which integrates the full 123,000 line Armv8-A ISA semantics (as defined by Arm in ASL and automatically translated into Sail [14]), with SMT-based tooling to evaluate tests w.r.t. axiomatic concurrency models. Previously Isla supported only "user" models, expressed in a language based on relational-algebra similar to the Cat language of Herd [9]. The integration with a full ISA semantics led us to raise several of the questions of §3, e.g. relating to system registers and mixed-size effects, which would not arise in a more idealised setting. The second main challenge is the combinatorics.

Previous litmus tests typically involved only a few abstract memory locations and events, but even simple virtual memory tests require 30kB of page tables, each "user" memory access might have 24 or more page-table accesses, and each 64-bit descriptor may be represented by a symbolic value representing all possible states that descriptor can be in. To avoid overwhelming the SMT solver during symbolic execution, the formula representing each symbolic descriptor is created dynamically when read. When encoding the final SMT problem that decides whether a candidate execution is allowed, we ensure that only the parts of the page tables actually used by that candidate execution are included. We also implemented a model-specific optimization that removes irrelevant translation events which cannot affect the result of the test, improving performance by a factor of 13 on average, and up to 90 times for some tests. Third, we had to provide a convenient way to express the page table configuration for each test, with the declarative language of which we saw a small part on the left-hand side of the §4 test.



A good user interface is essential. Above, we show an Isla-generated execution for a WRC test like that of §3.3, showing how uninteresting translation events can be suppressed in the output to avoid overwhelming noise.

The main result is that, in the strong model, all 119 litmus tests and 14 pKVM tests are allowed or forbidden as intended, based on our discussion with Arm of their architectural intent, except two pKVM tests which time out. Additionally, we tested that the weak model never forbids any test allowed by the strong model. The tool performance is eminently usable in practice: most tests take around 1 minute, and the full set of litmus tests can be run in less than 2 hours CPU time, on a 36-core Intel Xeon Gold 6240. Details are in App. D.

A further key property is that for ordinary relaxed-memory litmus tests which do not involve virtual memory, our model should give the same results as the published Armv8-A [26, 49, 13] axiomatic memory model. To validate this (and our tools) we test our strong model on an existing library of tests, comparing to reference results from Herd and RMEM [31]. We ran an additional 1927 such litmus tests, which all returned the expected results.

8 Experimental testing of hardware

Experimental investigation of hardware implementation behaviour, and experimental validation of models with respect to that, is one important input to the development of practically relevant relaxed memory semantics [24, 54, 3, 8]. However, almost all that work has focussed on "user"

concurrency, with litmus tests that could be run as user processes under a normal OS, and that could easily iterate tests over arrays. Experimental testing of virtual memory behaviour is considerably more challenging, as one needs to run code at higher privilege levels, including exception handlers, and manipulate the page tables that a normal OS and/or hypervisor would be depending on. When we started this work, that was not supported by litmus, so we have developed a litmus-like test harness for running virtual-memory tests bare-metal or in KVM. Currently it runs Stage 1 tests only; for Stage 2 tests some adaption to run code at EL2 is still needed. The harness can be found at `https://github.com/rems-project/system-litmus-harness`. At present this and Isla use different test formats, so we have some tests manually written in both.

We ran tests on three devices: a Raspberry Pi 3 (Arm A53), a Raspberry Pi 4 (Arm A72), and an AWS `m6g.metal` (AWS Graviton2, claiming to be an A72). Our experimental data suggests that all are multi-copy atomic with respect to translation-table-walks, respect coherence over physical locations, correctly perform TLB maintenance, and do not disagree with the tests presented here except for one behaviour: we sometimes observe anomalous results with respect to writes not being made globally visible to translation-table-walks (a `DSB` not sufficing); this is currently under discussion with Arm. Full results are in App. D.

Further testing on other platforms would be desirable, but our emphasis in this work is principally on exploring the design space and capturing the architectural intent, and the main validation is from discussion with the Arm Chief Architect, who ultimately is responsible for determining what the architecture is. In this context, experimental data serves mainly to provide reassurance that some envisaged architecture strength is not invalidated by extant hardware implementations.

9 Related work

There is extensive previous work on "user" relaxed-memory semantics of modern architectures, but very little extending this to cover systems aspects such as virtual memory. We build on the approaches established in "user" models for x86, IBM Power, Arm, and RISC-V, combining executable-as-test-oracle models, discussion with architects, and experimental testing [54, 5, 7, 47, 55, 53, 21, 52, 46, 9, 36, 31, 32, 49, 65].

Arm publish a machine-readable version of their Armv8-A relaxed memory model [45], in the Cat language of the Herd7 tool [6], but that model does not currently cover the relaxed virtual-memory semantics. Independent work in progress by Alglave et al. is similarly aiming to characterise this, and to update Arm's published model in due course, but with complementary scope to the current paper: including hardware updates of access and dirty bits, but without integration with the full ASL/Sail instruction semantics and its multiple levels and stages of translation. Both have been informed by discussion with senior Arm staff, and one would hope to synthesise the understanding in future. Hossain et al. [39] develop an "estimated" model for virtual memory in x86 (which has a much less relaxed base semantics) in a broadly similar axiomatic style. Tao et al. [62] axiomatise six conditions for *weak data-race-freedom* that should be satisfied by Armv8-A kernel code that uses virtual memory in simple ways, and an extension of Promising-Arm [50] that effectively builds in these conditions; they extend the sequential verification of the SeKVM hypervisor by Li et al. [43] to show it satisfies these conditions. The paper does not attempt to characterise the exact guarantees provided by the Armv8-A architecture, or discuss the issues of our §3. A foundational model such as our §5 would let one ground such results on the actual architecture. Simner et al. [57] study relaxed instruction-fetch semantics.

Several works give non-relaxed-memory semantics for Arm or x86 address translation, more or less simplified and with or without TLBs: Bauereiss [14], Goel et al. [34, 35], Syeda and Klein [58, 60, 59, 61], Degenbaev [29] (used for verification of a hypervisor shadow pagetable implementation [42, 28, 11, 10]), Barthe et al. [19, 17, 18, 16], Tews et al. [63], Kolanski [41],

and Guanciale et al. [38].

10 Acknowledgments

We thank Arm Ltd. for its support of Simner's PhD and the wider project of which this is part. We thank the Google pKVM development team, especially Will Deacon, Quentin Perret, Andrew Scull, Andrew Walbran, and Serban Constantinescu, for discussions on pKVM, and the Google Project Oak team, Ben Laurie, Hong-Seok Kim, and Sarah de Haas, for their support. We thank Luc Maranget for comments on a draft.

This work was partially funded by an Arm/EPSRC iCASE PhD studentship (Simner), Arm Limited, Google, ERC Advanced Grant (AdG) 789108 ELVER, and the UK Government Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694).

A VMSA litmus tests

This appendix gives the main Armv8-A virtual-memory-systems-architecture litmus tests that we have developed, systematically exploring the design space.

It is structured into subsections, with each building upon the tests of the previous and expanding the architectural scope of the tests. Each subsection is divided into subsubsections for each *shape*. Each shape may have many variations, e.g. with different choices of dependencies or barriers or cache maintenance instructions. §A.1 explains the test format, then subsequent sections describe test shapes in detail:

§A.2 explores coherence over physical and virtual addresses.

§A.3 gives tests which create new simple mappings for previously unused pages.

§A.4 considers unmapping in-use pages and the TLB invalidation requirements.

§A.5 considers extended questions about the operation of the `TLBI` instruction.

§A.6 gives tests which swap one translation for another, and the required break-before-make sequence.

§A.7 considers ordering within a single translation-table walk.

§A.8 considers for multi-copy atomicity.

§A.9 gives address-space tests.

Throughout, unless otherwise stated, the tests apply to both Stage 1 and Stage 2 translations, and for all exception levels, and memory is by default *normal* and *cacheable*.

The Isla-generated results can be found alongside each test, but we also include a table in App. D with all results.

A.1 Test Format

In this document the tests are given in a consistent format. Each test is given in three parts:

- The test listing.
- Execution witness diagram.
- Isla output.

A.1.1 Naming Convention

Throughout this document we will use a standard convention for names.

Each test name is of the following format:

```
TestName ::= ExtendedShape ("+" ThreadEdges)+
ThreadEdges ::= EDGE | ThreadEdges "-" ThreadEdges
ExtendedShape ::= SHAPE ("." ["Tf"|"T"|"R"|"Rpte"])* (".EL1")? (".inv")?
```

For tests that are completely new shapes, those shapes have their own names:

- ROT ("Re-ordered translations", §A.7.2.1)
- RBS ("Read broken secret", §A.4.1.6)
- BBM ("Break-before-make", §A.6.1.1)
- etc

Otherwise the underlying shape is just one of the original 'data memory' shapes:

- MP
- SB
- LB
- etc

To produce the full name we take the shape and expand it out to include an extra `R` for each read, for example `MP` becomes `MP.RR` as there are two reads. The `R`s represent the loads that happen with each thread appearing as a block with the `R`s within the block following program order. Then each `R` can be replaced with either, `T` (a successful translation), `T_f` (a translation which results in a fault), a `Rpte` (a load of the pagetable entry itself) or remain a `R` (a data memory load)

For example, `MP.RpteT.inv+dsb-isb` represents an `MP`-shaped test where the first load on the receiver thread is replaced with a load of the pagetable and the second read is a translation (which succeeds) reading from the (initially invalid) initial state, where there is a `DSB SY ; ISB` between the two instructions on the receiving thread. See MP.RpteT.inv+dsb-isb for the full test and diagram

A.1.2 Test Listing

The test listing is comprised of 4 main sections:

- memory and page-table initialisation code (on the left-hand-side).
- per-thread initial state (in the "Initial state" section on the right-hand-side).
- thread sections (labelled 'Thread 0','Thread 1','Thread 0 EL1 handler', etc).
- final state condition.

A.1.2.1 Pagetable setup The core of the pagetable setup is a small DSL, whose syntax is given by the grammar in Fig. 18.

The setup is a sequence of *constraints*. Initially the table is unconstrained, except for some initial mappings (each code section is identity mapped executable, etc).

Constraints can create new physical, intermediate-physical or virtual addresses to be used in the program, and set initial (and other possible) states of their mappings. The pagetable setup code must describe not only the initial state of all translation tables, but also any intermediate or final state that the translation tables could be in during execution of the test.

This constraint language comes with some built-in functions:

- `raw(N)` is a raw 64-bit number, useful as right-hand-side of `|->` relations.
- `table(addr)` is a mapping for a whole table, useful as the right-hand-side of a `|->` relation.
- `va_to_pa` casts a virtual address to a physical one.
  See also `pa_to_va`, `ipa_to_va`, `ipa_to_pa`, etc.

A.1.2.2 Initial and final state The initial state box is a key-value store, mapping the per-thread registers to initial values. These values are set just after machine reset.

The final state box contains a single expression which asserts the expected values of registers in the relaxed outcome. If the final state is allowed, then the test will have exhibited relaxed behaviours.

Both the final and initial state boxes use a simplified expression language which is common to both, and its full simplified syntax is given by the grammar in Fig. 19.

This expression language comes with some built-in functions, which make it easier to write the tests:

- `extz(v,bits)` zero-extends `v` to be `bits` wide.
- `ttbr(id=id,base=base)` produces a correctly-packed 64-bit number suitable as a `TTBRx_ELy` value, with base and the given asid/vmid.
- `pte3(IA,PTE BASE)` returns the address of the level 3 descriptor used to translate the virtual or intermediate-physical `IA` starting from a table rooted at `PTE BASE`.
  There are also `pte2`, `pte1` and `pte0` variants.
- `desc3(IA, PTE BASE)` which is roughly `*pte3(IA, PTE BASE)`, that is, the actual 64-bit descriptor found at the address given by the `pteN(...)`.
- `raw(N)` is a raw 64-bit number, useful as right-hand-side of `|->` relations.
- `mkdesc3(oa=OA)` constructs a fresh level3 *block* descriptor with default permissions and output address `OA`.
  (See also `mkdesc2`, `mkdesc1`).
- `mkdesc2(table=ADDR)` constructs a fresh level2 *table* descriptor with default permissions and table address `ADDR`.
  (See also `mkdesc1`, `mkdesc0`).
- `page(addr)` is the page the address is found in, defined as `addr` right-shifted 12 bits.
- `asid(id)` is a 64-bit value suitable for use in `TLBI`-by-ASID instructions, defined to be `id` left-shifted 48 bits.

```
PageTableSetup ::= Constraint (";" Constraint)*

Constraint ::=
    "option" name "=" ("true" | "false") # enable/disable option
  | Alignment? "virtual" name+           # va/ipa/pa with optional alignment
  | Alignment? "intermediate" name+
  | Alignment? "physical" name+
  | "identity" Expr WithAttrs? Level?    # identity mapping
  | Expr "|->" Expr WithAttrs? Level?    # mapsto
  | Expr "?->" Expr WithAttrs? Level?    # maybe mapsto
  | "*" Expr "=" Expr                    # deref equality
  | "assert" Expr
  | Stage name Expr?
  | Stage name Expr? "{" PageTableSetup "}"

Expr ::=
    Expr "&&" Expr    # boolean AND
  | Expr "||" Expr    # boolean OR
  | Expr "&" Expr     # bitwise AND
  | Expr "|" Expr     # bitwise OR
  | Expr "^" Expr     # bitwise XOR
  | Expr "==" Expr    # equality
  | Expr "!=" Expr    # inequality
  | "~" Expr          # bitwise negation
  | name "(" (Arg ("," Arg)*)? ")"  # call
  | hex | nat | bin
  | "(" Expr ")"

Arg ::=
    name "=" Expr       # keyword argument
  | Expr

Stage ::= "s1table" | "s2table"
Level ::= "at" "level" nat

Alignment ::= "aligned" u64

AttrField ::= name "=" (hex | bin)
Attrs ::=
    "default"
  | "code"
  | "[" AttrField ("," AttrField)* "]"

WithAttrs ::=
    "with" Attrs              # with S1
  | "with" Attrs "and" Attrs  # with S1 and S2
```

Figure 18: Pagetable Setup DSL — Simplified Grammar

```
Expr ::=
    Loc "=" Expr
  | label ":"       # code label
  | bin | hex | nat
  | name "(" (Arg ("," Arg)*)? ")"
  | "(" Expr ")"
  | "~" Expr
  | "true" | "false"
  | Expr ("&" Expr)+
  | Expr ("|" Expr)+
  | Expr "->" Expr

Loc ::=
    nat ":" regname  # register
  | "*" name         # deref

Arg ::=
    name "=" Expr
  | Expr
```

Figure 19: Litmus Test — Simplified Expression Grammar

A.1.3 Execution witness

The test is run by Isla with a model with no axioms to allow all behaviours. Executions that satisfy the final state then have graphs produced (and if multiple, the 'interesting' execution is hand-picked for display).

The diagrams contain an 'initial state' node which represents all initial writes in the system (which may or may not be writes of zero). Threads are then laid out in a row, with instructions within each Thread box placed in a single column with `po` ('program-order') going top to bottom. Multiple events within the same instruction are then aligned horizontally within the same row, where possible.

Translates are highlighted (in blue) and interesting relations (`iio`, `po`, `co`, `rf`, `trf`, `fr`, `tfr`, `same-va-page`, and `same-ipa-page`) are shown (and where the relation is transitively closed, we display the transitive reduction of that relation to reduce clutter). Labels for `po` are elided to reduce clutter.

A.1.4 Isla output

The test is run in Isla, using the strong model (see App. B).

The generated Isla output that is produced is cut down to just the final line of output, which contains five key pieces of information:

- The test name
- Model outcome (allowed or forbidden)
- The total number of executions, and how many were allowed.
- The total Isla execution time for the test.

A.1.5 Example

Consider CoTW1.inv. It has one thread (Thread 0) and one other code section (Thread 0's EL1 exception vector). The initial state says that the thread starts from EL0 (from the `PSTATEEL` register), with `R1` (aka `X1`) containing the 64-bit virtual address named `x`, `R2` containing the 64-bit level 3 descriptor which the initial pagetable setup uses to translate `y`, `R3` containing the 64-bit virtual address of the location that contains the level 3 descriptor used in translating `x`, and finally that the thread's EL1 vector base address (`VBAR`) is at `0x1000`.

The pagetable setup has two virtual addresses (`x` and `y`), with one physical address (`pa1`). Initially `x` is unmapped, and `y` maps to `pa1` where `pa1` is initially 1. The page containing the vector table (starting at `0x1000`) is identity mapped as executable.

During execution of the test it is expected that at some point `x` may map to `pa1`, and so there is a `x ?-> pa1` constraint. Without this constraint Isla will not generate any executions that involve translating `x` resulting in a translation to `pa1`. (In fact, in this instance Isla will fail on symbolic evaluation of `STR X2,[X3]` as the write is unsatisfiable.)

The exception handler of interest is located at `0x1400`, this is at `VBAR+0x400`. An offset of `0x400` represents a synchronous exception from a lower exception level. The handler overwrites `X0` with 0, to mark that an exception has occured, and then does an exception-return to the next-instruction-address (i.e. `ELR+4`).

The final state asserts that `0:R0=1`, that is that seeing `X0` being 1 would imply a relaxed execution of the above test. See the CoTW1.inv section for an explanation of which relaxed behaviour(s) this outcome corresponds to.

The diagram then shows such an execution.

Finally we see that the test is forbidden by our strong model, that Isla generates 2 candidate executions for this test (and neither are allowed), and that it took 38589 miliseconds for Isla to run the test (just under 40 seconds).

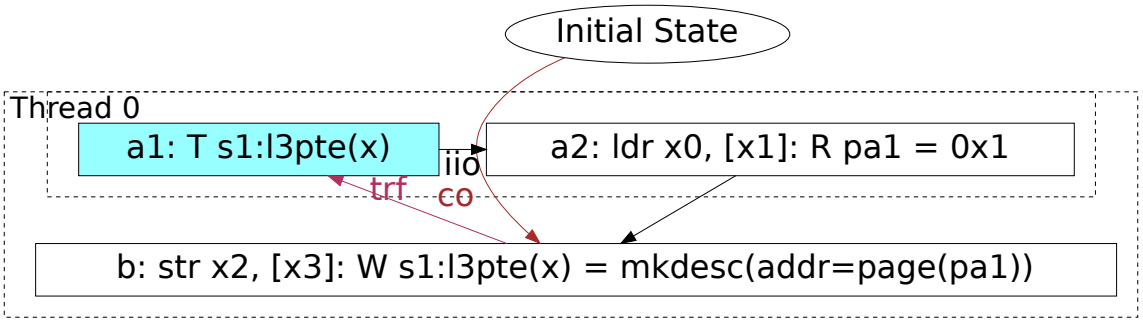
AArch64 CoTW1.inv

Page table setup:
`physical pa1;`
`x ↦ invalid;`
`x ?-> pa1;`
`y ↦ pa1;`
`*pa1 = 1;`
`identity 0x1000 with code;`
	Initial state:
	`PSTATE.EL=0b00`
	`PSTATE.SP=0b0`
	`R1=x`
	`R2=desc3(y,page_table_base)`
	`R3=pte3(x,page_table_base)`
	`VBAR_EL1=0x1000`

	Thread 0
	`LDR X0,[X1]` `STR X2,[X3]`
	thread0 el1 handler

	`0x1400:`
	`MOV X0,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `0:R0=1`



Model	Result
Base	no result for CoTW1.inv
ETS	`CoTW1.inv forbidden (0 of 2) 2714ms`

A.2 Aliasing

A.2.1 Coherence

Arm's notion of *coherence* gives a fixed total order per location of all writes to that location. With virtual memory, that becomes a total order per *physical address* location.

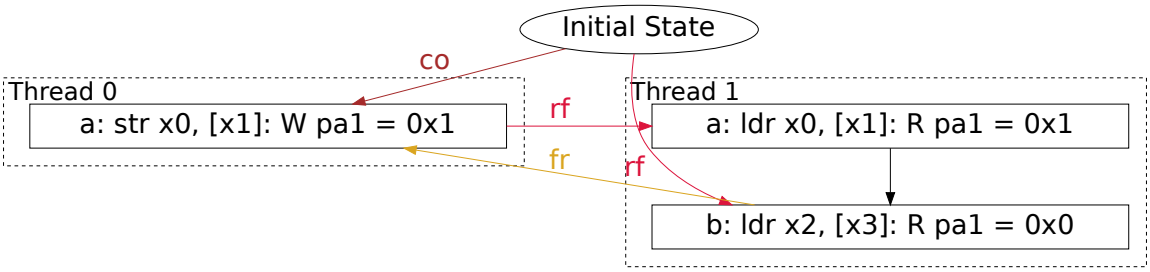
CoRR-shaped tests

A.2.1.1 Test: CoRR0.alias+po forbid

This is the classic coherence shape. Here, we ask whether two reads with different VAs but which map to the same PA are allowed to re-order with respect to each other if they read from different writes. For Arm, they are not, as coherence is with respect to physical addresses.

AArch64 CoRR0.alias+po

Page table setup:	Initial state:
physical pa1; x ↦ pa1; y ↦ pa1; *pa1 = 0;	0:R0=0b1 0:R1=x 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=y
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **LDR** X2,[X3]
	Final state: 1:R0=1 & 1:R2=0



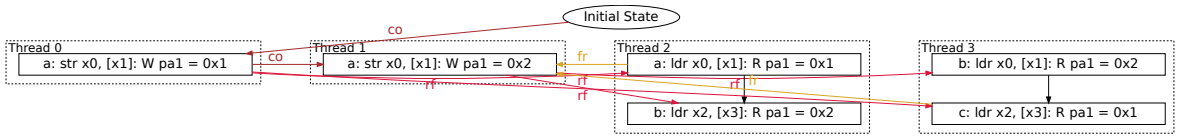
Model	Result
Base	`CoRR0.alias+po forbidden (0 of 1) 2512ms`
ETS	`CoRR0.alias+po forbidden (0 of 1) 1420ms`

A.2.1.2 Test: CoRR2.alias+po forbid

This is another standard variant of CoRR, adapted to physical memory.

AArch64 CoRR2.alias+po

Page table setup: **physical** pa1; u ↦ pa1; v ↦ pa1; w ↦ pa1; x ↦ pa1; y ↦ pa1; z ↦ pa1; *pa1 = 0;	Initial state: 0:R0=0b01 0:R1=u 1:R0=0b10 1:R1=v 2:PSTATE.EL=0b00 2:PSTATE.SP=0b0 2:R1=w 2:R3=x 3:PSTATE.EL=0b00 3:PSTATE.SP=0b0 3:R1=y 3:R3=z
	Thread 0
	STR X0,[X1]
	Thread 1
	STR X0,[X1]
	Thread 2
	LDR X0,[X1] **LDR** X2,[X3]
	Thread 3
	LDR X0,[X1] **LDR** X2,[X3]
	Final state: 2:R0=1 & 2:R2=2 & 3:R0=2 & 3:R2=1



Model	Result
Base	CoRR2.alias+po forbidden (0 **of** 1) 1934ms
ETS	CoRR2.alias+po forbidden (0 **of** 1) 2260ms

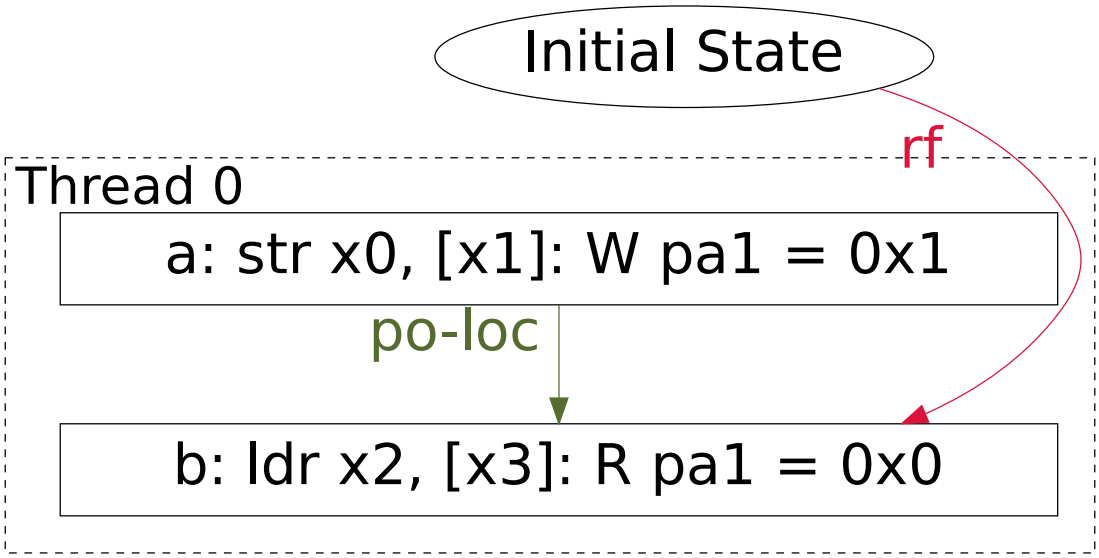
A.2.1.3 Test: CoWR.alias forbid

If one writes to one VA, and reads with another that is mapped to the same PA, must the read read-from the program-order preceding write of the same PA, or something newer, regardless of the second VA? For Armv8-A, yes.

AArch64 CoWR.alias

Page table setup:
`physical pa1;`
`x ↦ pa1;`
`y ↦ pa1;`
`*pa1 = 0;`
	Initial state:
	`R0=0x1`
	`R1=x`
	`R3=y`

	Thread 0
	`STR X0,[X1]` `LDR X2,[X3]`
	Final state: `0:R2=0`



Model	Result
Base	`CoWR.alias forbidden (0 of 1) 977ms`
ETS	`CoWR.alias forbidden (0 of 1) 967ms`

A.2.2 Write-Forwarding

A.2.2.1 Test: PPOCA.alias allow

Can a load from one virtual address have its value forwarded from a store to distinct VA that is mapped to the same PA, on a speculative branch?

Our model says yes.

AArch64 PPOCA.alias

Page table setup:
`physical pa1 pa2 pa3;`
`w ↦ pa1;`
`x ↦ pa1;`
`y ↦ pa2;`
`z ↦ pa3;`
`*pa1 = 0;`
`*pa2 = 0;`
`*pa3 = 0;`
	Initial state:
	`0:R0=0x1`
	`0:R1=z`
	`0:R2=0x1`
	`0:R3=y`
	`1:R1=y`
	`1:R2=0x1`
	`1:R3=x`
	`1:R5=w`
	`1:R7=z`

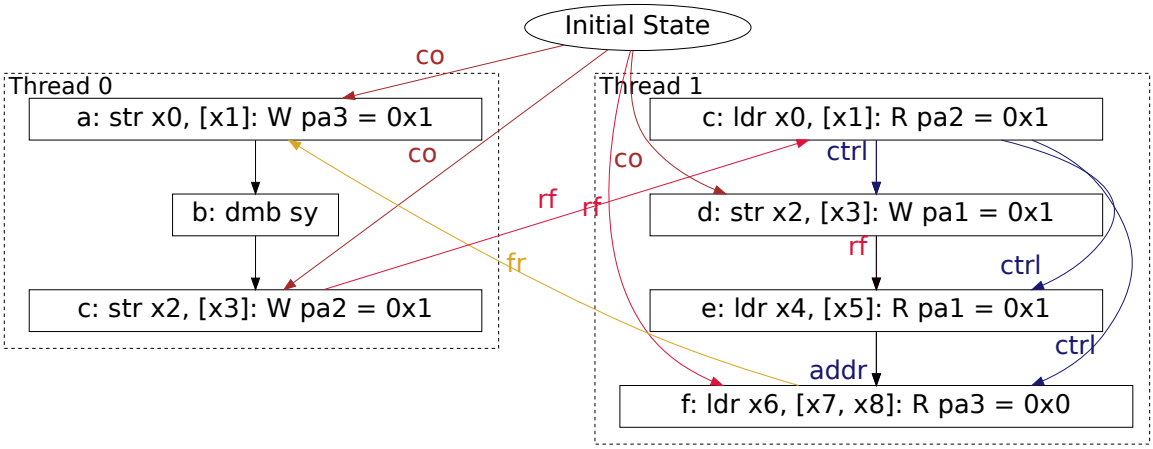
	Thread 0

	`STR X0,[X1]`
	`DMB SY`
	`STR X2,[X3]`

	Thread 1

	`LDR X0,[X1]`
	`CBNZ X0,L0`
	`L0:`
	`STR X2,[X3]`
	`LDR X4,[X5]`
	`EOR X8,X4,X4`
	`LDR X6,[X7,X8]`

	Final state: `1:R0=1 & 1:R4=1 & 1:R6=0`



Model	Result
Base	`PPOCA.alias allowed (1 of 2) 4744ms`
ETS	`PPOCA.alias allowed (1 of 2) 4932ms`

A.2.3 Out-of-order reads

A.2.3.1 Test: RSW.alias allow

If two reads from different VAs which translate to the same PA read from the same write, they can be re-ordered with respect to program-order.

AArch64 RSW.alias

Page table setup:

```
physical pa1 pa2 pa3;
w ↦ pa1;
x ↦ pa1;
y ↦ pa2;
z ↦ pa3;
*pa1 = 0;
*pa2 = 0;
*pa3 = 0;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=0b1
0:R1=z
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R4=w
1:R6=x
1:R9=z
```

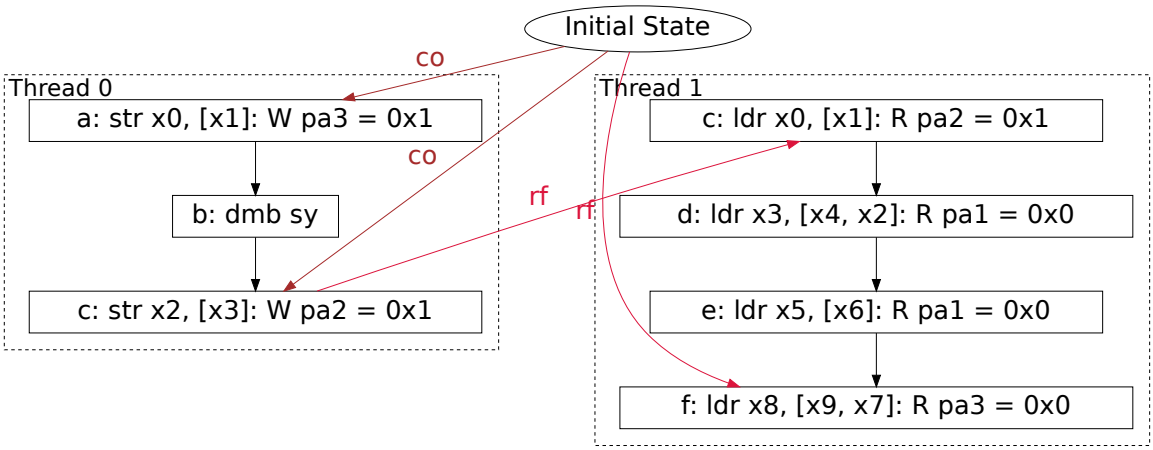
Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 1

```
LDR X0,[X1]
EOR X2,X0,X0
LDR X3,[X4,X2]
LDR X5,[X6]
EOR X7,X5,X5
LDR X8,[X9,X7]
```

Final state: `1:R0=1 & 1:R8=0`



Model	Result
Base	`RSW.alias allowed (1 of 1) 2765ms`
ETS	`RSW.alias allowed (1 of 1) 2609ms`

A.2.3.2 Test: RDW.alias forbid

If two loads of different VAs which translate to the same PA read from different writes, then can they be re-ordered?

Our model says no.

AArch64 RDW.alias

Page table setup:

```
physical pa1 pa2 pa3;
w ↦ pa1;
x ↦ pa1;
y ↦ pa2;
z ↦ pa3;
*pa1 = 0;
*pa2 = 0;
*pa3 = 0;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=0b1
0:R1=z
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R4=w
1:R6=x
1:R9=z
2:R0=0b1
2:R1=x
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

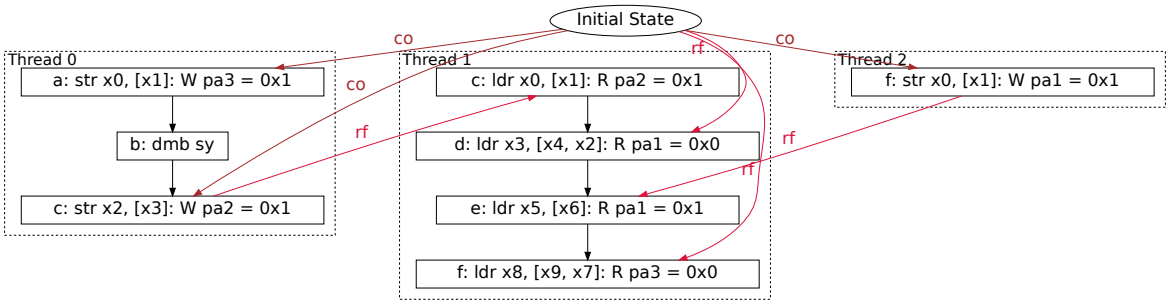
Thread 1

```
LDR X0,[X1]
EOR X2,X0,X0
LDR X3,[X4,X2]
LDR X5,[X6]
EOR X7,X5,X5
LDR X8,[X9,X7]
```

Thread 2

```
STR X0,[X1]
```

Final state: `1:R0=1 & 1:R3=0 & 1:R5=1 & 1:R8=0`



Model	Result
Base	`RDW.alias forbidden (0 of 1) 2972ms`
ETS	`RDW.alias forbidden (0 of 1) 3317ms`

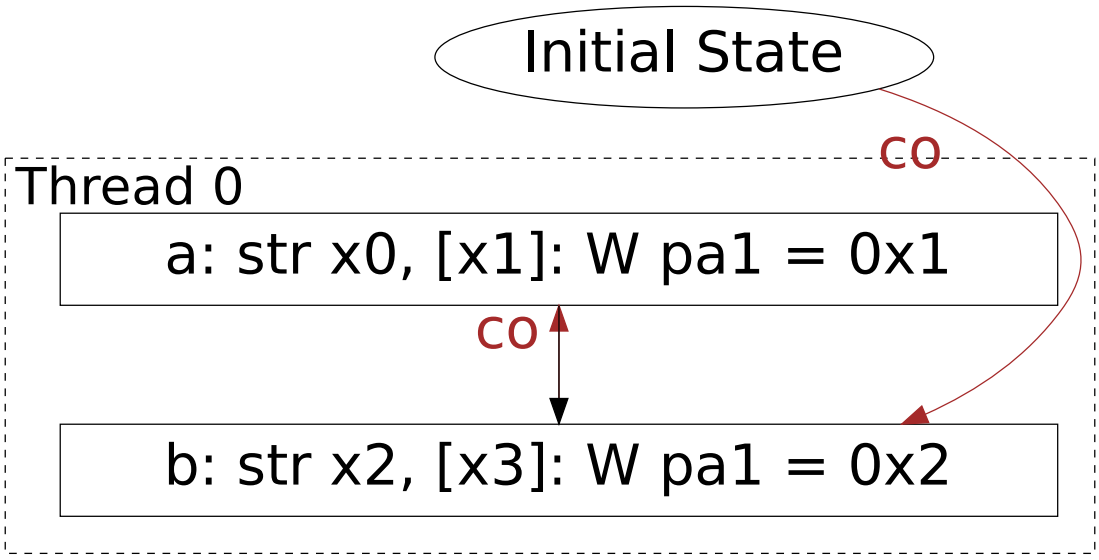
A.2.3.3 Test: CoWW.alias forbid

Should the coherence-order of writes respect program-order in the same thread even if they are to different VAs?

Our model says yes, and forbids the following CoWW.alias test.

AArch64 CoWW.alias

Page table setup:	Initial state:
physical pa1; x ↦ pa1; y ↦ pa1; *pa1 = 0;	PSTATE.EL=0b00 PSTATE.SP=0b0 R0=0b01 R1=x R2=0b10 R3=y
	Thread 0
	STR X0,[X1] **STR** X2,[X3]
	Final state: x=1



Model	Result
Base	CoWW.alias forbidden (0 **of** 1) 822ms
ETS	CoWW.alias forbidden (0 **of** 1) 849ms

A.2.3.4 Test: MP.alias3+rfi-data+dmb allow

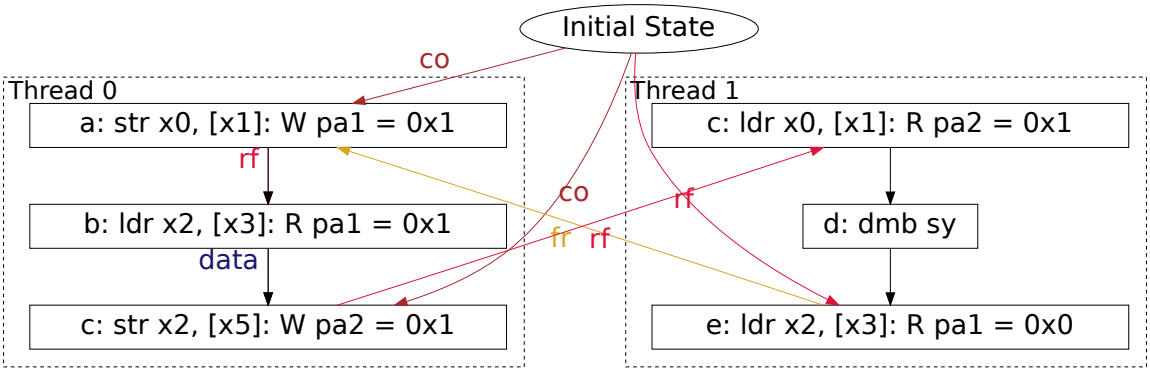
This shows thread-local forwarding of a write to a read with distinct VA but the same PA, in a potentially non-speculative path.

AArch64 MP.alias3+rfi−data+dmb

Page table setup:	Initial state:

physical pa1 pa2;
x ↦ pa1;
y ↦ pa2;
z ↦ pa1;
*pa1 = 0;
*pa2 = 0;
	0:R0=0x1
	0:R1=x
	0:R3=z
	0:R5=y
	1:R1=y
	1:R3=x

	Thread 0
	STR X0,[X1] **LDR** X2,[X3] **STR** X2,[X5]
	Thread 1
	LDR X0,[X1] **DMB** SY **LDR** X2,[X3]
	Final state: 1:R0=1 & 1:R2=0



Model	Result
Base	MP.alias3+rfi-data+dmb allowed (1 **of** 1) 1431ms
ETS	MP.alias3+rfi-data+dmb allowed (1 **of** 1) 1483ms

A.3 Writing new entries

A.3.1 Translation tables as data memory

Writes to the translation tables are treated as completely normal writes to memory as far as normal reads are concerned, like any other location: they can be re-ordered, cached, and take part in coherence as far as their memory attributes allow. We assume here that all reads and writes are to 'normal' cacheable memory.

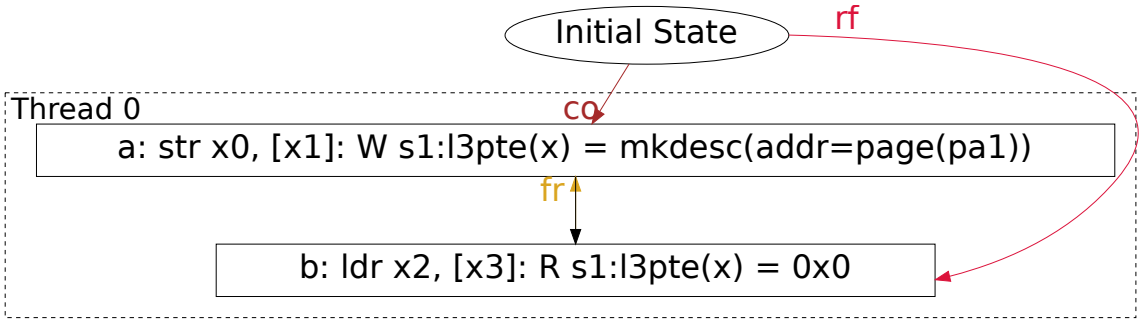
A.3.1.1 Test: CoWR.inv forbid

Writing a new entry to the page-table then loading the location again performs a normal data memory read.

We do not adapt all of the standard "user" data memory tests here with translation tables as memory locations. Instead, we just give one representative co-shaped example.

AArch64 CoWR.inv

Page table setup:	Initial state:
physical pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1;	PSTATE.SP=0b0 R0=**desc3**(y,page_table_base) R1=**pte3**(x,page_table_base) R3=**pte3**(x,page_table_base) VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1] **LDR** X2,[X3]
	Final state: 0:R2=0



Model	Result
Base	CoWR.inv forbidden (0 **of** 1) 825ms
ETS	CoWR.inv forbidden (0 **of** 1) 1104ms

A.3.2 Making a new entry

If a VA is currently unmapped (and that has been fully synchronized with sufficient TLBI and barrier instructions), then, to produce a new virtual-to-physical mapping, all that is needed is to simply write to the physical location that contains the invalid entry for that VA.

To ensure that the new entry is seen by the same processor, the pipeline must be flushed with an `ISB` or other *context-synchronizing* event. Without this, the processor can re-order (or perhaps even speculatively perform) the translation.

CoWTf-shaped tests

A.3.2.1 Test: CoWTf.inv+po allow

If a thread writes to a page table entry initially containing an invalid descriptor, and the translation of the address of the next instruction uses the page table entry, then the translate is allowed to see the old, invalid descriptor.

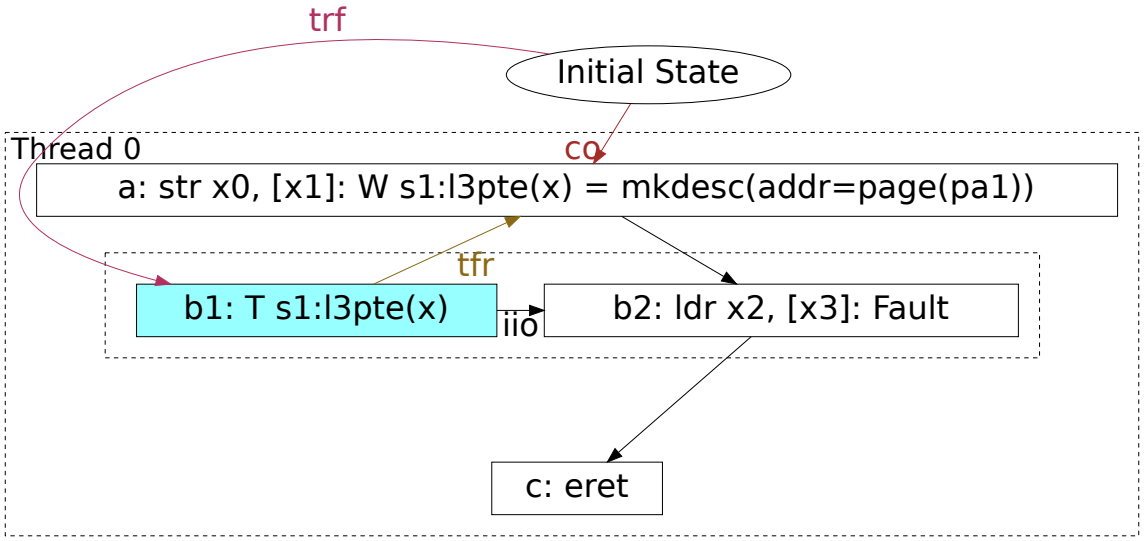
To detect this, we install a handler for synchronous aborts which writes `0` to `X2` before incrementing the `ELR` to the next instruction address and performing an exception-return.

If the final state sees `X2=1`, then we know the load read-from the new physical location, but if it saw `X2=0`, then it must have been caused by a translation-fault.

The role of `y` in this litmus test is to make it possible to succinctly describe the new descriptor for `x`.

AArch64 CoWTf.inv+po

Page table setup:	Initial state:
physical pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**;	PSTATE.SP=0b0 R0=**desc3**(y,page_table_base) R1=**pte3**(x,page_table_base) R3=x VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1] **LDR** X2,[X3]
	thread0 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**
	Final state: 0:R2=0



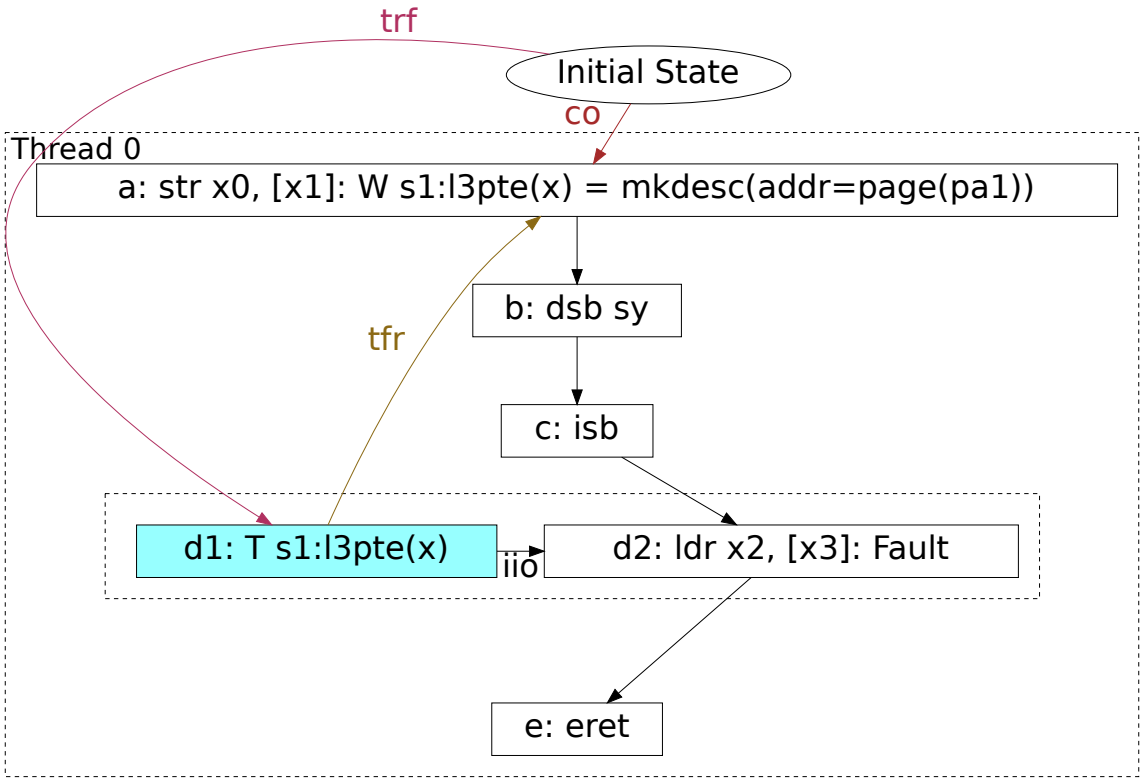
Model	Result
Base	`CoWTf.inv+po allowed (1 of 2) 4112ms`
ETS	`CoWTf.inv+po allowed (1 of 2) 4160ms`

A.3.2.2 Test: CoWTf.inv+dsb-isb forbid

If there is a `DSB; ISB` interposed in between the overwriting of the invalid descriptor with the valid descriptor and the translation, then the translation is required to see a write no older than that of the valid descriptor, as the `DSB; ISB` causes a pipeline flush.

AArch64 CoWTf.inv+dsb−isb

Page table setup:	Initial state:
physical pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**;	PSTATE.SP=0b0 R0=**desc3**(y,page_table_base) R1=**pte3**(x,page_table_base) R3=x VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	thread0 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**
	Final state: 0:R2=0



Model	Result
Base	`CoWTf.inv+dsb-isb forbidden (0 of 2) 3024ms`
ETS	`CoWTf.inv+dsb-isb forbidden (0 of 2) 2475ms`

A.3.3 Creating a new entry for another core

If two CPUs are using the same (or overlapping) translation tables, then, necessarily, writes to the translation table by one CPU can be visible to the other.

A.3.3.1 Test: S.T+dmb+po forbid

In this S-shaped test, Thread 0 writes some data and then gives Thread 1 a new mapping.

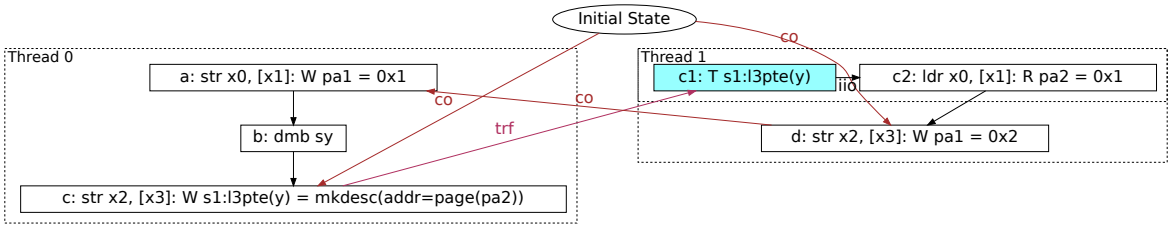
If Thread 1 sees the mapping, then the program-order-later store must wait for the translation to finish before propagating to memory.

AArch64 S.T+dmb+po

Page table setup:	Initial state:

physical pa1 pa2;
x ↦ pa1;
 	
y ↦ invalid;
y ?-> pa2;
 	
*pa1 = 0;
*pa2 = 1;
identity 0x1000 **with code**;
	0:R0=0x1
	0:R1=x
	0:R2=**mkdesc3**(oa=pa2)
	0:R3=**pte3**(y,page_table_base)
	1:PSTATE.EL=0b00
	1:PSTATE.SP=0b0
	1:R1=y
	1:R2=0x2
	1:R3=x
	1:VBAR_EL1=0x1000

	Thread 0
	STR X0,[X1] **DMB** SY **STR** X2,[X3]
	Thread 1
	LDR X0,[X1] **STR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & x=1



Model	Result
Base	S.T+dmb+po forbidden (0 **of** 2) 6152ms
ETS	S.T+dmb+po forbidden (0 **of** 2) 4485ms

MP.RTf.inv-shaped tests

A.3.3.2 Test: MP.RTf.inv+dmb+dsb-isb forbid

Note that Thread 0 only has a `DMB SY`; in fact, any ordered-before relation here would suffice. Thread 1 requires the pipeline flush, as described above.

AArch64 MP.RTf.inv+dmb+dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

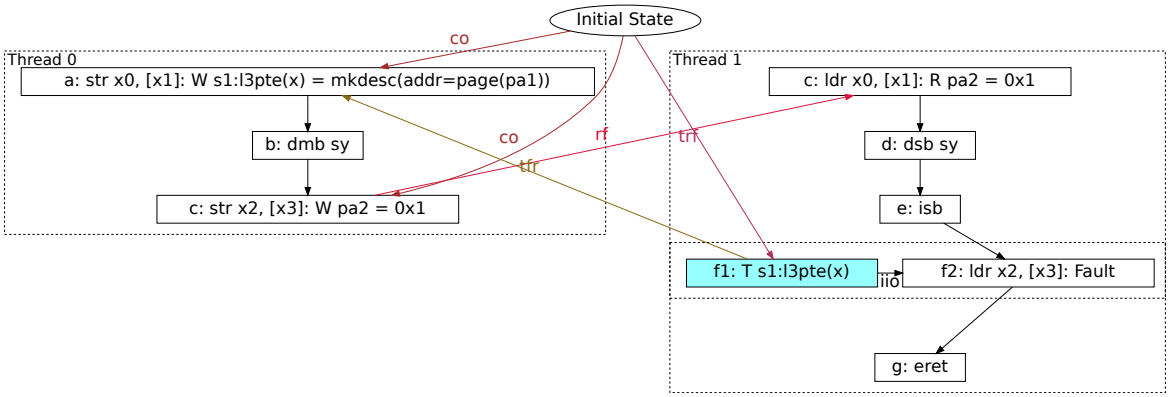
Thread 1

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv+dmb+dsb-isb forbidden (0 of 2) 5345ms`
ETS	`MP.RTf.inv+dmb+dsb-isb forbidden (0 of 2) 3982ms`

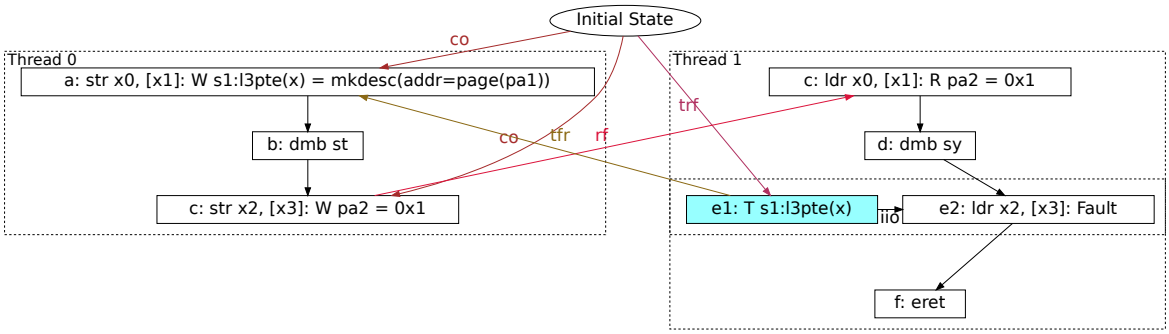
A.3.3.3 Test: MP.RTf.inv+dmbs allow (forbid with ETS)

The `DSB; ISB` is required for the base architecture, as illustrated by the this test.

However, if the implementation has the ETS optional feature ("Enhanced Translation Synchronization"), then this test is forbidden. This is because ETS ensures that a translation-table-walk which results in a translation-fault (that is, one that reads an invalid entry) is ordered-after any memory event which would be ordered-before the read/write of any load/store (as appropriate) in the place of the instruction which generated the translation-fault.

AArch64 MP.RTf.inv+dmbs

Page table setup: **physical** pa1 pa2; x ↦ invalid; x ?-> pa1; z ↦ pa1; *pa1 = 1; y ↦ pa2; **identity** 0x1000 **with code**;	Initial state: 0:R0=**desc3**(z,page_table_base) 0:R1=**pte3**(x,page_table_base) 0:R2=0b1 0:R3=y 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=y 1:R3=x 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1] **DMB** ST **STR** X2,[X3]
	Thread 1
	LDR X0,[X1] **DMB** SY **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



Model	Result
Base	`MP.RTf.inv+dmbs allowed (1 of 2) 5896ms`
ETS	`MP.RTf.inv+dmbs forbidden (0 of 2) 3681ms`

A.3.3.4 Test: MP.RTf.inv+dmb+ctrl-isb forbid?

AArch64 MP.RTf.inv+dmb+ctrl−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

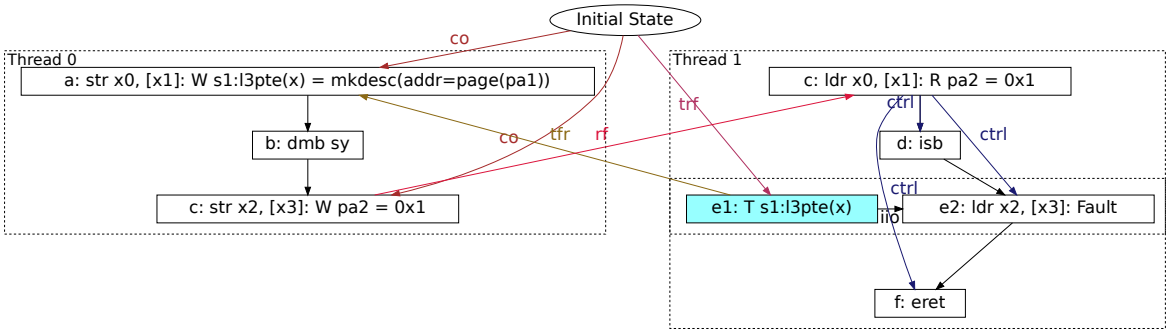
Thread 1

```
LDR X0,[X1]
CBNZ X0,L0
L0:
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv+dmb+ctrl-isb forbidden (0 of 4) 5734ms`
ETS	`MP.RTf.inv+dmb+ctrl-isb forbidden (0 of 4) 5059ms`

A.3.3.5 Test: MP.RTf.inv+dmb+addr forbid?

AArch64 MP.RTf.inv+dmb+addr

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB ST
STR X2,[X3]
```

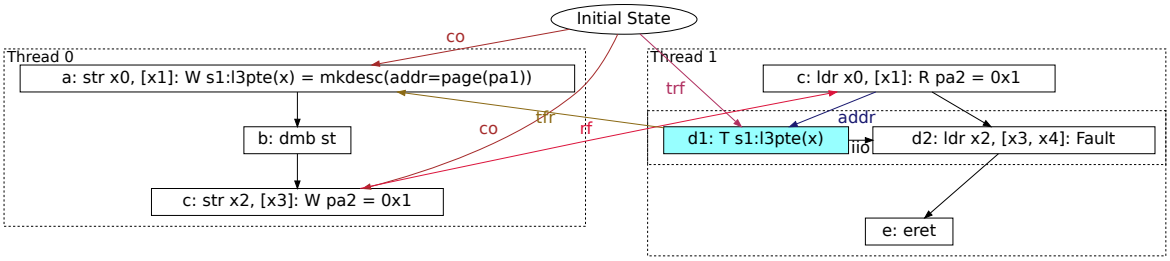
Thread 1

```
LDR X0,[X1]
EOR X4,X0,X0
LDR X2,[X3,X4]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv+dmb+addr forbidden (0 of 2) 3574ms`
ETS	`MP.RTf.inv+dmb+addr forbidden (0 of 2) 4327ms`

A.3.3.6 Test: MP.RTf.inv+dmb+po allow

Even with ETS, program-order alone is not enough to ensure that Thread 1 sees the write of the valid descriptor.

AArch64 MP.RTf.inv+dmb+po

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB ST
STR X2,[X3]
```

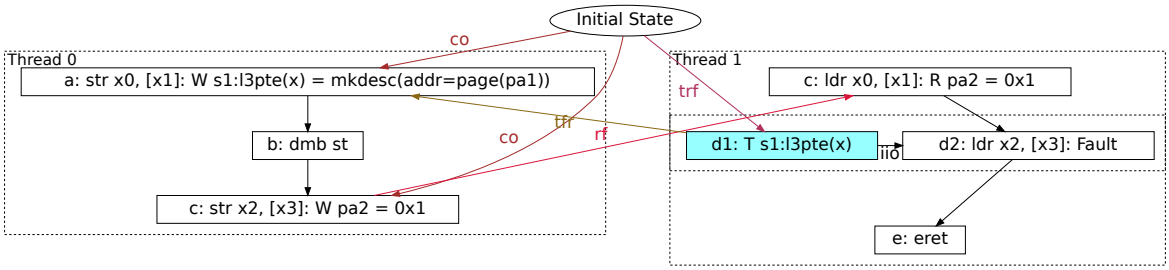
Thread 1

```
LDR X0,[X1]
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv+dmb+po allowed (1 of 2) 5316ms`
ETS	`MP.RTf.inv+dmb+po forbidden (0 of 2) 2598ms`

A.3.3.7 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+po allow

This is a variant of message passing where `x` is initially not mapped, Thread 0 maps `x`, performs a `DSB;TLBI;DSB`, and writes to the flag, and Thread 1 reads the flag and reads `x`. Because there is only program-order in Thread 1, the reads can be completely reordered, and thus the second read can happen entirely before the TLBI.

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+po

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

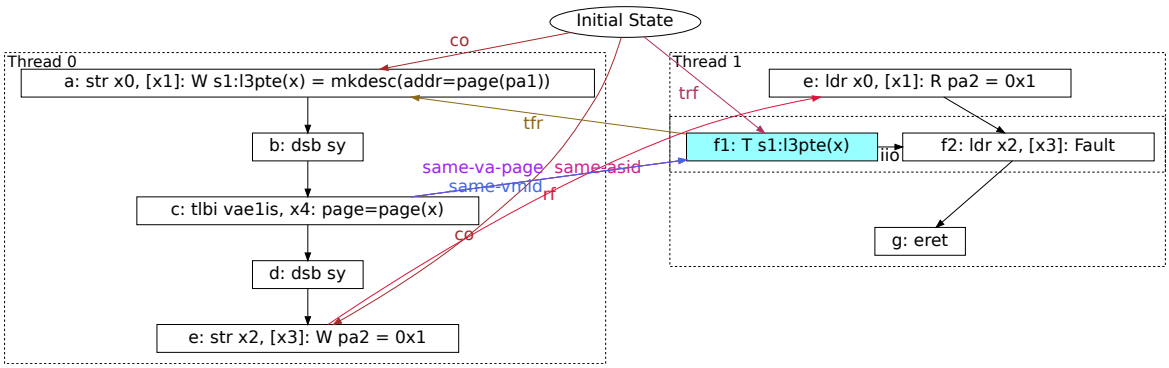
Thread 1

```
LDR X0,[X1]
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+po forbidden (0 of 2) 20910ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+po forbidden (0 of 2) 3710ms`

A.3.3.8 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+dmb forbid

A fault inherits the order that the corresponding memory access would have had if it had not faulted. (With ETS, its translates also inherit the order, making this test forbidden more directly.) Moreover, the pipeline effect of the broadcast TLBI enforces that a memory access and its translate-reads are 'atomically' ordered with respect to the TLBI: they are either both ordered-before it, or both ordered-after it. Therefore, because the counterfactual load of `x` in Thread 1 is ordered after the load of the flag `y` by the `DMB SY`, the translate is also ordered after it. Therefore, if Thread 1 sees that the flag `y` is set to 1, then the translate is guaranteed to translate-read something at least as new as the new, valid mapping that Thread 0 wrote.

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+dmb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

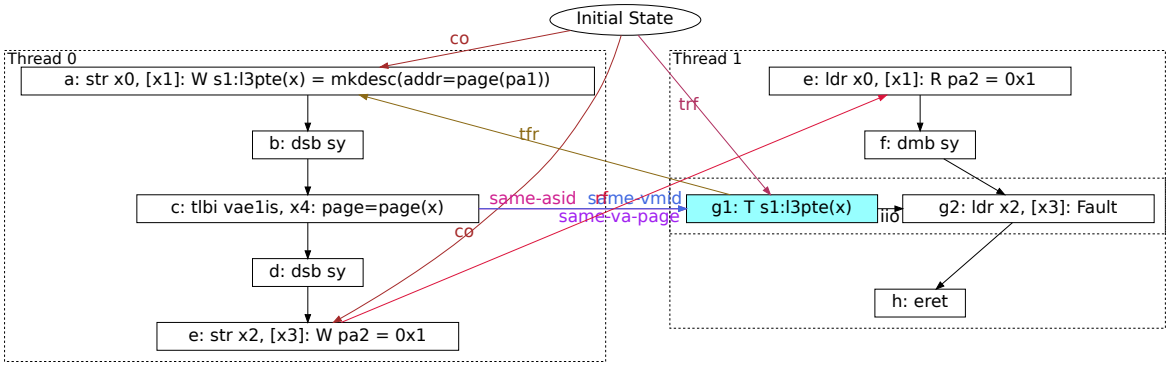
Thread 1

```
LDR X0,[X1]
DMB SY
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+dmb forbidden (0 of 2) 10762ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+dmb forbidden (0 of 2) 5294ms`

A.3.3.9 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+addr forbid

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+addr

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

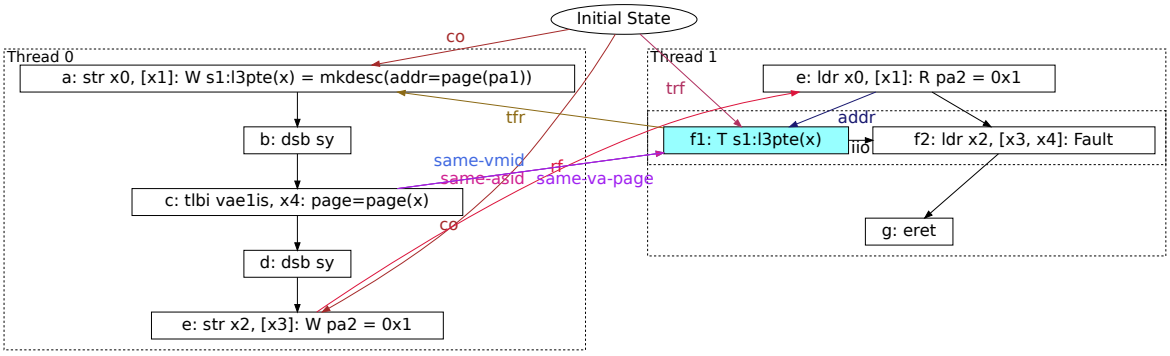
Thread 1

```
LDR X0,[X1]
EOR X4,X0,X0
LDR X2,[X3,X4]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+addr forbidden (0 of 2) 3994ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+addr forbidden (0 of 2) 3882ms`

A.3.3.10 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+data forbid

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+data

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

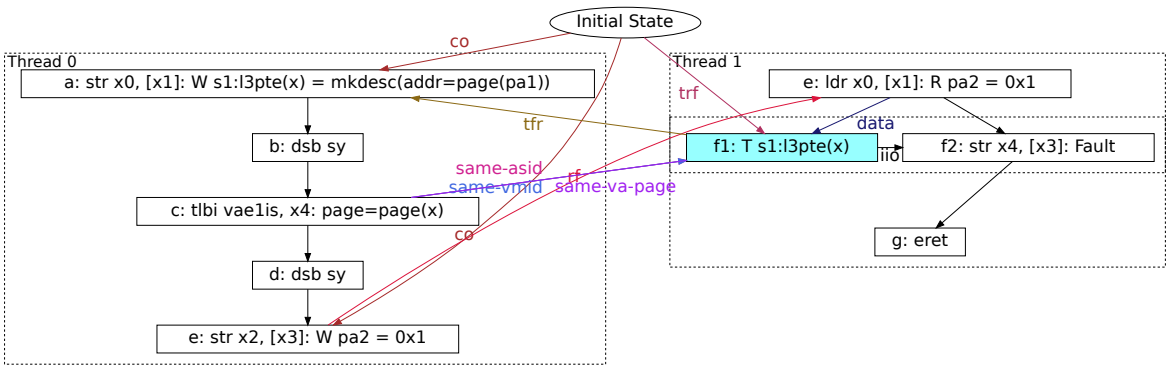
Thread 1

```
LDR X0,[X1]
EOR X4,X0,X0
STR X4,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+data forbidden (0 of 2) 3744ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+data forbidden (0 of 2) 5394ms`

A.3.3.11 Test: MP.RTf.inv+dmb+data allow?

The version with just a `DMB` is not enough.

AArch64 MP.RTf.inv+dmb+data

Page table setup:
`physical pa1 pa2;`
`x ↦ invalid;`
`x ?-> pa1;`
`z ↦ pa1;`
`*pa1 = 1;`
`y ↦ pa2;`
`identity 0x1000 with code;`
	Initial state:
	`0:R0=desc3(z,page_table_base)`
	`0:R1=pte3(x,page_table_base)`
	`0:R2=0b1`
	`0:R3=y`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=y`
	`1:R2=0b1`
	`1:R3=x`
	`1:R5=0b1`
	`1:VBAR_EL1=0x1000`

	Thread 0

	`STR X0,[X1]`
	`DMB ST`
	`STR X2,[X3]`

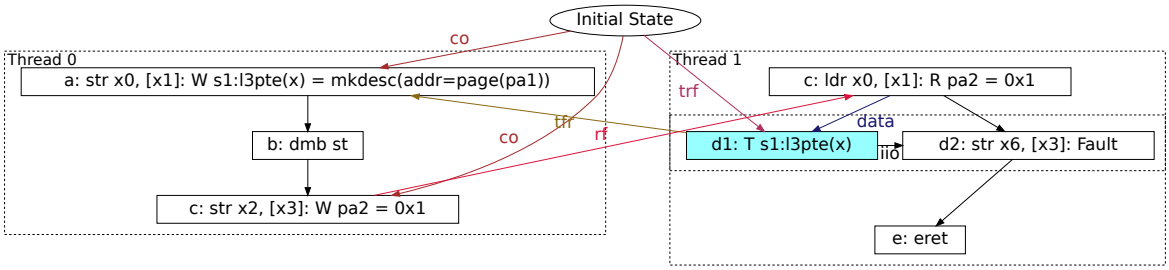
	Thread 1

	`LDR X0,[X1]`
	`EOR X4,X0,X0`
	`ORR X6,X4,X5`
	`STR X6,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv+dmb+data forbidden (0 of 2) 3232ms`
ETS	`MP.RTf.inv+dmb+data forbidden (0 of 2) 3843ms`

A.3.3.12 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl forbid

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+ctrl

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

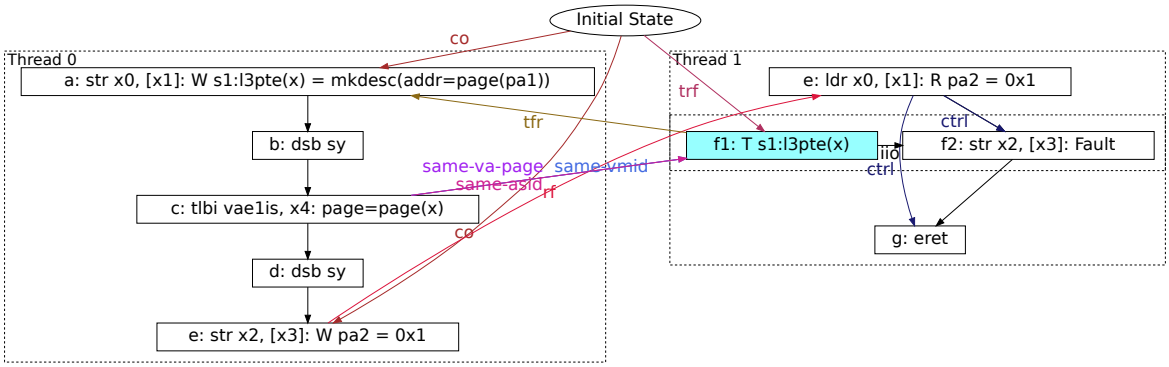
Thread 1

```
LDR X0,[X1]
CBNZ X0,L0
L0:
STR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl forbidden (0 of 4) 15261ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl forbidden (0 of 4) 6025ms`

A.3.3.13 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb forbid

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

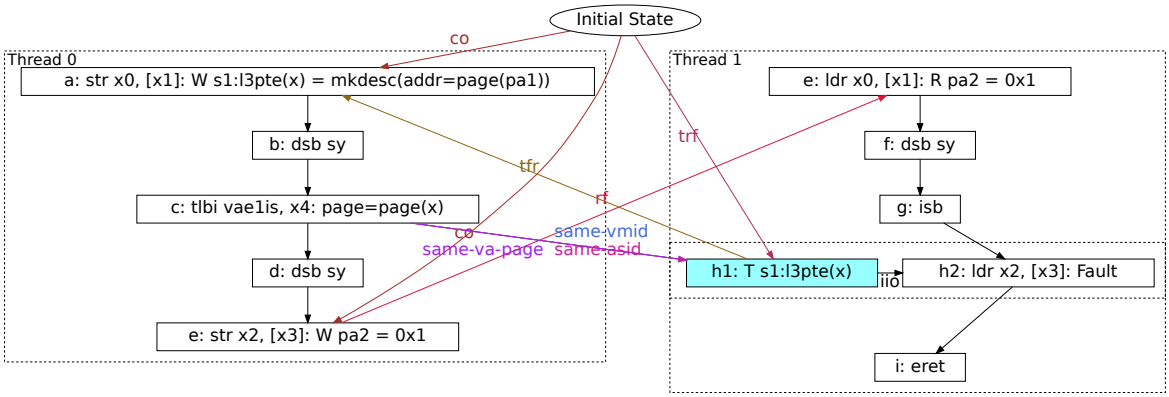
Thread 1

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 2) 52965ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 2) 6204ms`

A.3.3.14 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl-isb forbid

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+ctrl−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

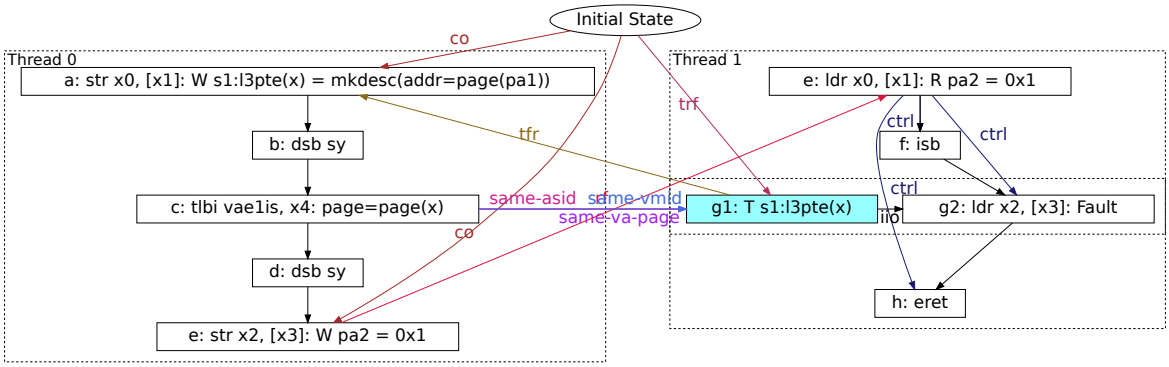
Thread 1

```
LDR X0,[X1]
CBNZ X0,L0
L0:
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl-isb forbidden (0 of 4) 31160ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl-isb forbidden (0 of 4) 8117ms`

A.3.3.15 Test: MP.RTf.inv.EL1+dsb-tlbiis-dsb+poap forbid

AArch64 MP.RTf.inv.EL1+dsb−tlbiis−dsb+poap

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

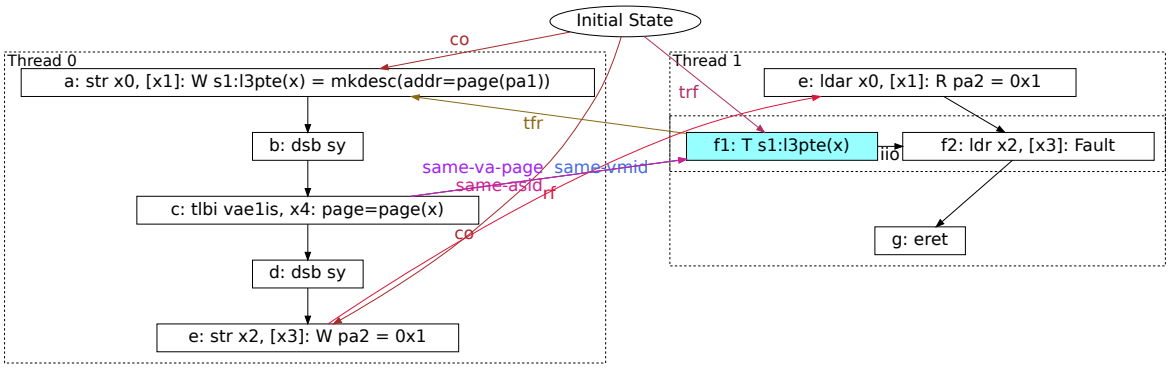
Thread 1

```
LDAR X0,[X1]
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+poap forbidden (0 of 2) 75710ms`
ETS	`MP.RTf.inv.EL1+dsb-tlbiis-dsb+poap forbidden (0 of 2) 3279ms`

A.3.3.16 Test: LB.TT.inv+pos forbid

This is a variant of load buffering where the first thread's store writes the descriptor that the translate for the second thread's load translate-reads from, and symmetrically. This kind of self-satisfying cycle would be very problematic, and this test is forbidden.

AArch64 LB.TT.inv+pos

Page table setup:

```
physical pa1;
x ↦ invalid;
y ↦ invalid;

x ?-> pa1;
y ?-> pa1;

*pa1 = 1;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R1=x
0:R2=mkdesc3(oa=pa1)
0:R3=pte3(y,page_table_base)
0:VBAR_EL1=0x1000
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=mkdesc3(oa=pa1)
1:R3=pte3(x,page_table_base)
1:VBAR_EL1=0x2000
```

Thread 0

```
LDR X0,[X1]
STR X2,[X3]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

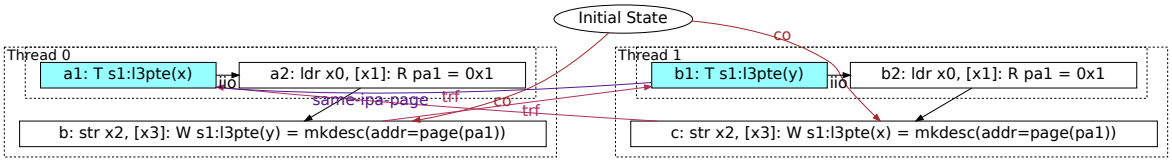
thread0 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R0=1 & 1:R0=1`



Model	Result
Base	`LB.TT.inv+pos forbidden (0 of 4) 15481ms`
ETS	`LB.TT.inv+pos forbidden (0 of 4) 16545ms`

A.3.3.17 Test: S.RTf.inv.EL1+dsb-tlbiis-dsb+data forbid

AArch64 S.RTf.inv.EL1+dsb−tlbiis−dsb+data

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

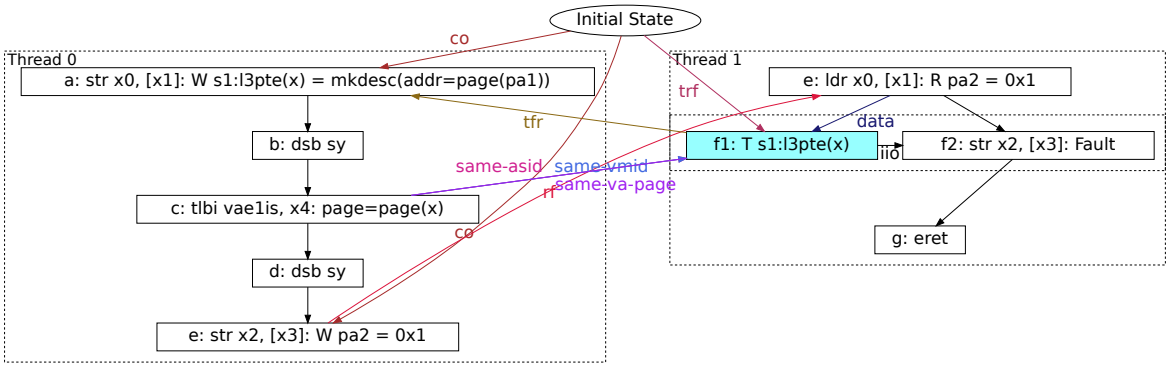
Thread 1

```
LDR X0,[X1]
EOR X4,X0,X0
ORR X2,X2,X4
STR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`S.RTf.inv.EL1+dsb-tlbiis-dsb+data forbidden (0 of 2) 4120ms`
ETS	`S.RTf.inv.EL1+dsb-tlbiis-dsb+data forbidden (0 of 2) 4535ms`

A.3.3.18 Test: S.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl forbid

AArch64 S.RTf.inv.EL1+dsb−tlbiis−dsb+ctrl

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

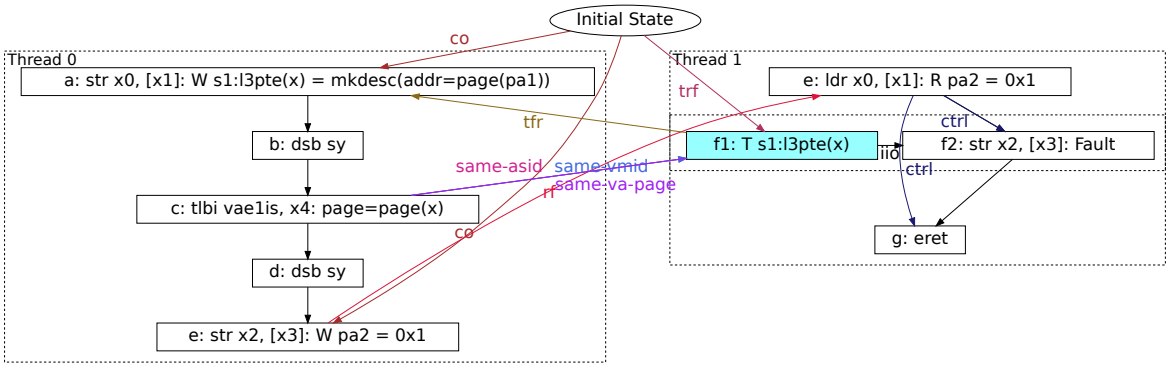
Thread 1

```
LDR X0,[X1]
CBNZ X0,L0
L0:
STR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`S.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl forbidden (0 of 4) 12958ms`
ETS	`S.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl forbidden (0 of 4) 4463ms`

A.3.3.19 Test: S.RTf.inv.EL1+dsb-tlbiis-dsb+dmb forbid

AArch64 S.RTf.inv.EL1+dsb−tlbiis−dsb+dmb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

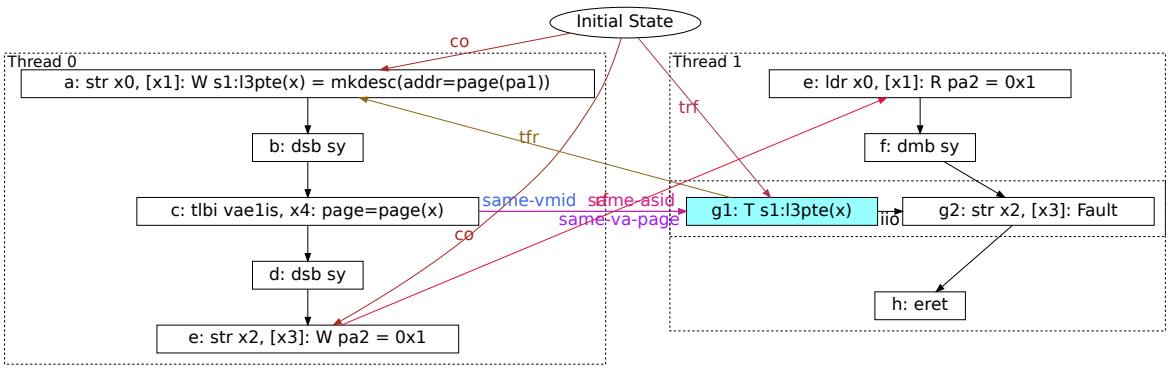
Thread 1

```
LDR X0,[X1]
DMB SY
STR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`S.RTf.inv.EL1+dsb-tlbiis-dsb+dmb forbidden (0 of 2) 11481ms`
ETS	`S.RTf.inv.EL1+dsb-tlbiis-dsb+dmb forbidden (0 of 2) 4349ms`

A.3.3.20 Test: S.RTf.inv.EL1+dsb-tlbiis-dsb+popl forbid

AArch64 S.RTf.inv.EL1+dsb−tlbiis−dsb+popl

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

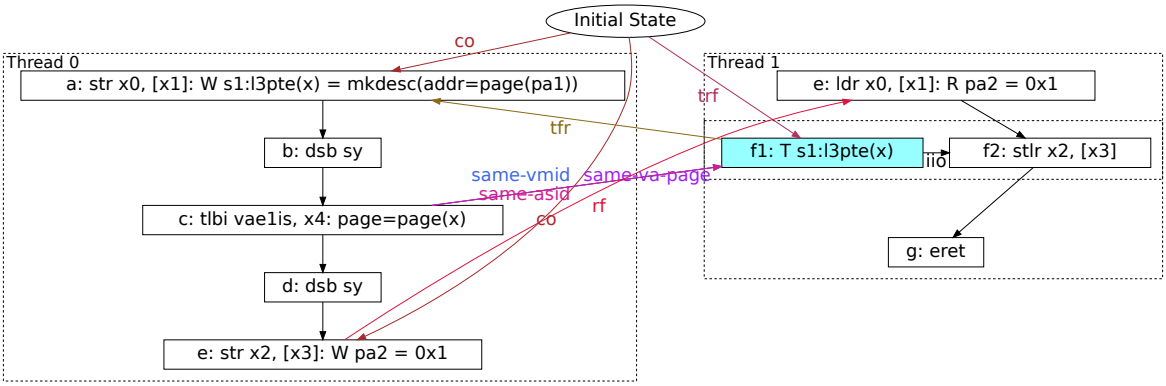
Thread 1

```
LDR X0,[X1]
STLR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`S.RTf.inv.EL1+dsb-tlbiis-dsb+popl forbidden (0 of 2) 28716ms`
ETS	`S.RTf.inv.EL1+dsb-tlbiis-dsb+popl forbidden (0 of 2) 4266ms`

A.3.3.21 Test: S.RTf.inv.EL1+dsb-tlbiis-dsb+poap forbid

AArch64 S.RTf.inv.EL1+dsb−tlbiis−dsb+poap

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=0b1
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

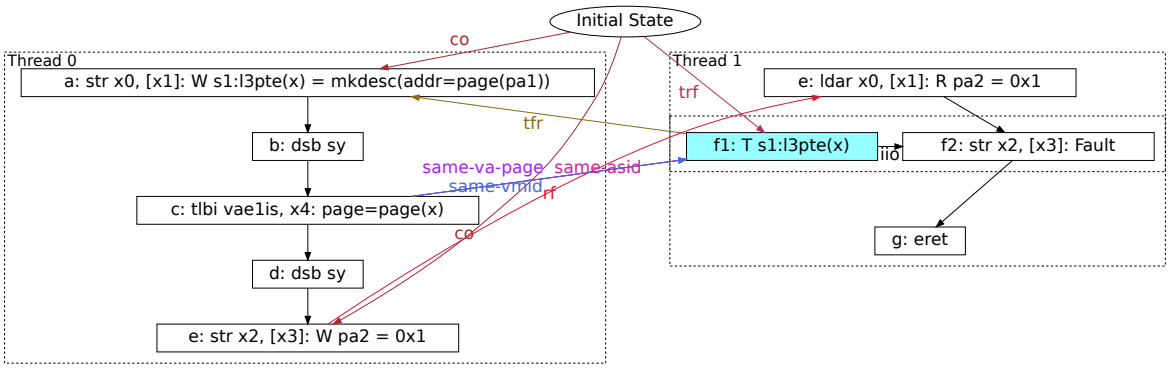
Thread 1

```
LDAR X0,[X1]
STR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`S.RTf.inv.EL1+dsb-tlbiis-dsb+poap forbidden (0 of 2) 45733ms`
ETS	`S.RTf.inv.EL1+dsb-tlbiis-dsb+poap forbidden (0 of 2) 3117ms`

A.3.4 Coherence

Similarly to our previous questions about Instruction↔Data coherence, we can ask questions about Translation↔Data coherence:

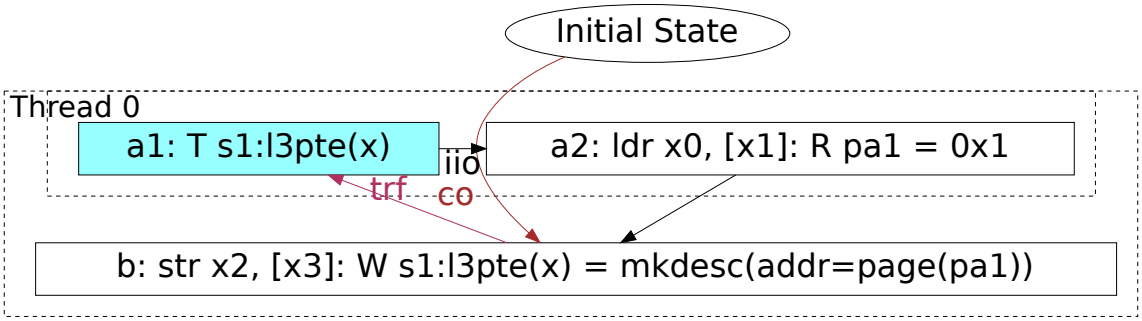
1. If a translation-table-walk reads-from a write, must a later translation-table-walk that reads the same location read-from the same write or something coherence-newer? (Translation→Translation Coherence).
2. If a translation-table-walk reads-from a write, must a later load/store that reads/writes that location read-from something at least that new, or write something coherence-after it? (Data→Translation Coherence).
3. If a load reads-from a write, must a later translation-table-walk which reads that location read-from that write or something newer? (Translation→Data Coherence).

A.3.4.1 Test: CoTW1.inv forbid

Translations cannot read-from writes which appear program-order after the instruction that does the translation.

AArch64 CoTW1.inv

Page table setup: **physical** pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**;	Initial state: PSTATE.EL=0b00 PSTATE.SP=0b0 R1=x R2=**desc3**(y,page_table_base) R3=**pte3**(x,page_table_base) VBAR_EL1=0x1000
	Thread 0
	LDR X0,[X1] **STR** X2,[X3]
	thread0 el1 handler
	0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 0:R0=1



Model	Result
Base	no result for CoTW1.inv
ETS	CoTW1.inv forbidden (0 **of** 2) 2714ms

CoTTf-shaped tests

A.3.4.2 Test: CoTTf.inv+dsb-isb forbid

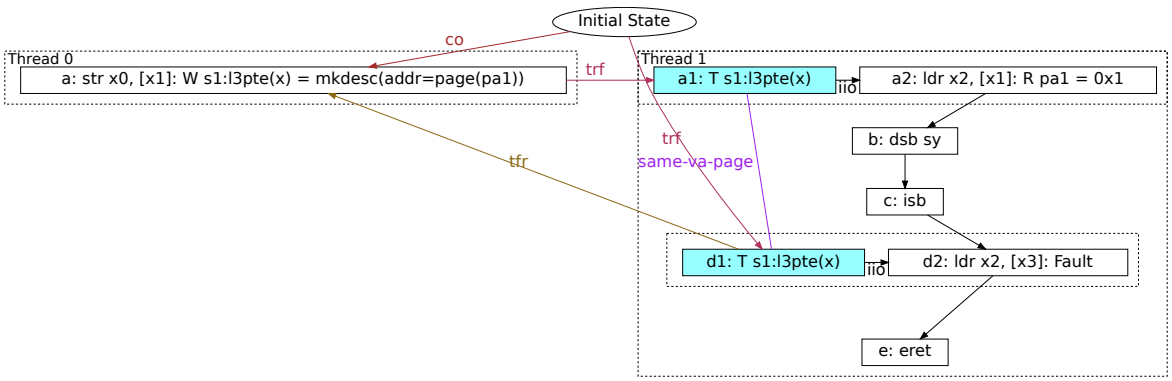
Here, Thread 0 makes a new mapping, and Thread 1 observes that new mapping by performing a translation using it, and then later tries to load that same location. If the first read is translated using the new entry, then the second one is not allowed to fault.

Note this test's handler writes to `X2`, so the test saves it into `X0` after the first load.

This suggests a kind of translation→translation coherence. In general, you do observe such coherence when TLB-misses (and therefore walks in memory) occur. However, the CoTfT+dsb-isb test (later in this document) shows that this is not guaranteed for all translations.

AArch64 CoTTf.inv+dsb−isb

Page table setup:	Initial state:
physical pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**;	0:R0=**desc3**(y,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=x 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X2,[X1] **MOV** X0,X2 **DSB** SY **ISB** **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



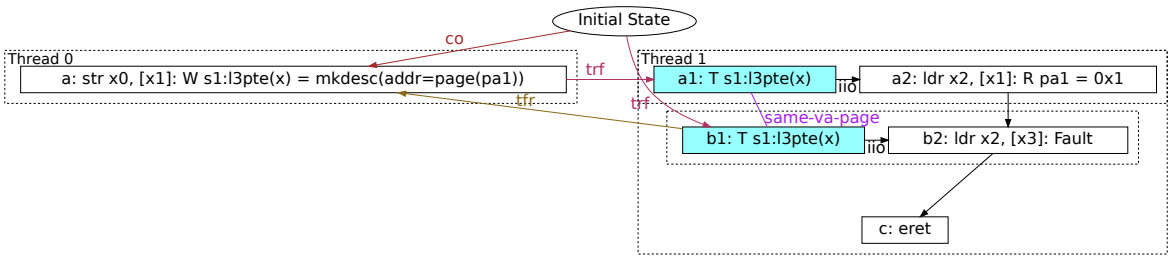
Model	Result
Base	`CoTTf.inv+dsb-isb forbidden (0 of 4) 21525ms`
ETS	`CoTTf.inv+dsb-isb forbidden (0 of 4) 26198ms`

A.3.4.3 Test: CoTTf.inv+po allow

Same as above, but with no explicit order between the two loads.
This is allowed.

AArch64 CoTTf.inv+po

Page table setup: **physical** pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**;	Initial state: 0:R0=**desc3**(y,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=x 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X2,[X1] **MOV** X0,X2 **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



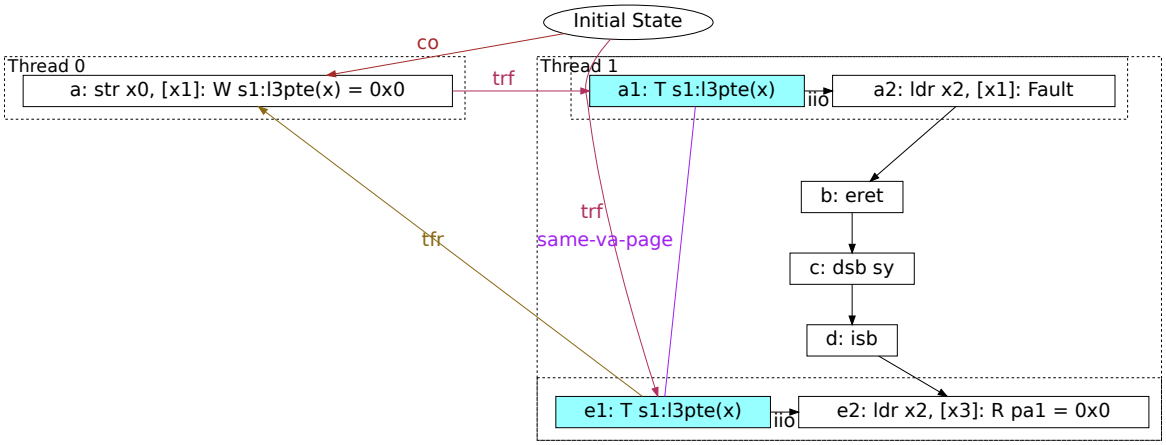
Model	Result
Base	CoTTf.inv+po allowed (1 **of** 4) 51495ms
ETS	CoTTf.inv+po forbidden (0 **of** 4) 29031ms

A.3.4.4 Test: CoTfT+dsb-isb allow

This test is in contrast to the previous test, where the VA used by the loads in Thread 1 are instead valid from the start, and Thread 0 attempts to 'break' the entry by writing an invalid descriptor (e.g. `0`) to the entry. In contrast to the previous test, this one does not obey the translation↔translation coherence principle.

AArch64 CoTfT+dsb−isb

Page table setup:	Initial state:
physical pa1; x ↦ pa1; x ?-> invalid; y ↦ pa1; *pa1 = 0; **identity** 0x1000 **with code**;	0:R0=0b0 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=x 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X2,[X1] **MOV** X0,X2 **DSB** SY **ISB** **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



Model	Result
Base	`CoTfT+dsb-isb allowed (1 of 4) 78905ms`
ETS	`CoTfT+dsb-isb allowed (1 of 4) 95644ms`

CoRpteTf.inv-shaped tests

A.3.4.5 Test: CoRpteTf.inv+dsb-isb forbid

In this test, Thread 0 writes a new mapping, which Thread 1 reads-from with a load before trying to access the location mapped by that entry.

So long as the later translation is ordered after the read (with a context-synchronizing event, or if with ETS then any ordered-before), then the translation must see the new mapping too.

This implies a kind of Translation→Data coherence. Note that this only applies going Invalid→Valid; removing a mapping does not guarantee this coherence and TLB maintenance is required (c.f. CoRT+dsb-isb)

AArch64 CoRpteTf.inv+dsb−isb

Page table setup:
`option default_tables = true;`
`physical pa1;`
`intermediate ipa1;`
 	
`x ↦ invalid;`
`x ?-> pa1;`
`y ↦ pa1;`
`identity 0x1000 with code;`
 	
`*pa1 = 1;`
	Initial state:
	`0:R0=desc3(y,page_table_base)`
	`0:R1=pte3(x,page_table_base)`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=pte3(x,page_table_base)`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`

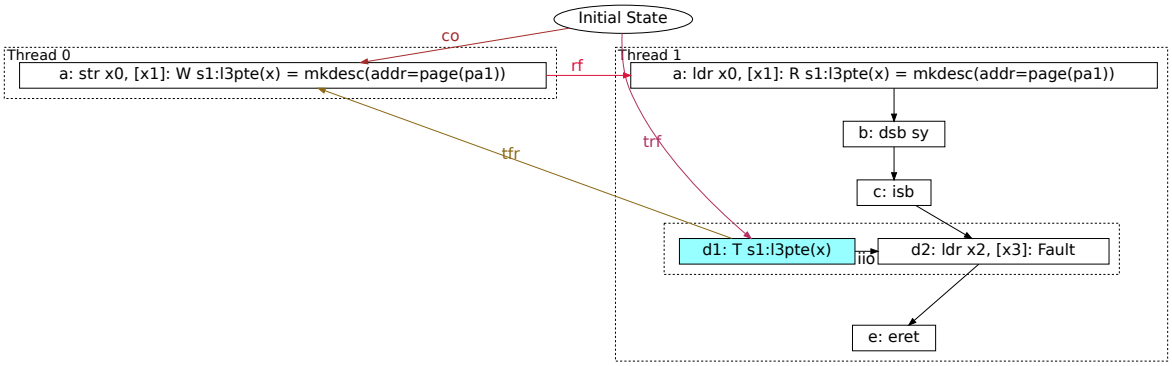
	Thread 0
	`STR X0,[X1]`
	Thread 1

	`LDR X0,[X1]`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=desc3(y,page_table_base) & 1:R2=0`



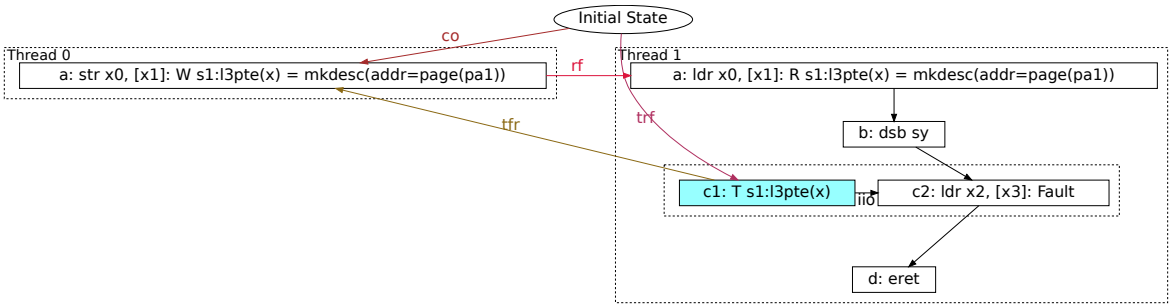
Model	Result
Base	`CoRpteTf.inv+dsb-isb forbidden (0 of 2) 3347ms`
ETS	`CoRpteTf.inv+dsb-isb forbidden (0 of 2) 3472ms`

A.3.4.6 Test: CoRpteTf.inv+dsb allow (unless ETS)

Same as previous, but without the `ISB`. Allowed (unless ETS, then forbidden).

AArch64 CoRpteTf.inv+dsb

Page table setup:	Initial state:
physical pa1; **intermediate** ipa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; **identity** 0x1000 **with code**; *pa1 = 1;	0:R0=**desc3**(y,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=**pte3**(x,page_table_base) 1:R3=x 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **DSB** SY **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=**desc3**(y,page_table_base) & 1:R2=0



Model	Result
Base	`CoRpteTf.inv+dsb allowed (1 of 2) 5413ms`
ETS	`CoRpteTf.inv+dsb forbidden (0 of 2) 2582ms`

A.3.4.7 Test: CoRpteT+dsb-isb allow

Here, Thread 0 *unmaps* a location, and Thread 1 loads the translation table entry containing the invalid descriptor. Given Thread 1 read the invalid descriptor, is a 'later' translation of the possibly-unmapped location required to fault?

On Arm, no. The TLB can cache the old mapping and then the later translation can read that cached value.

AArch64 CoRpteT+dsb−isb

Page table setup:	Initial state:

`physical pa1;`
`x ↦ pa1;`
`x ?-> invalid;`
`identity 0x1000 with code;`
`*pa1 = 0;`
	`0:R0=0b0`
	`0:R1=pte3(x,page_table_base)`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=pte3(x,page_table_base)`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`

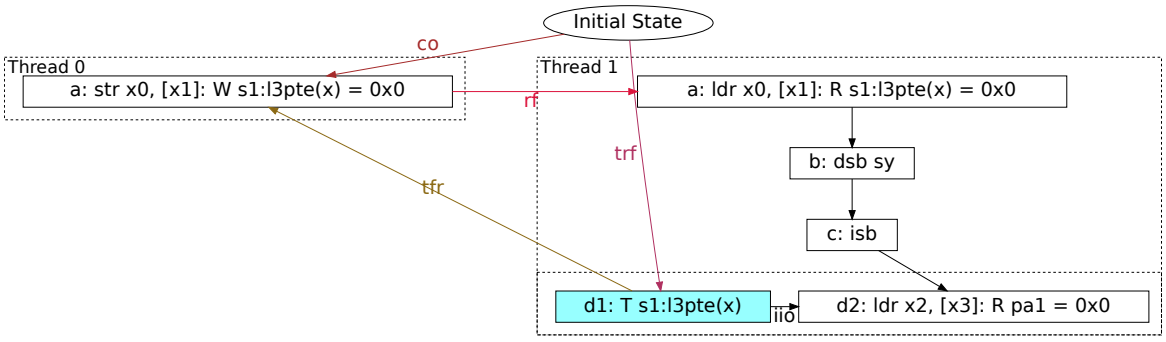
	Thread 0
	`STR X0,[X1]`
	Thread 1

	`LDR X0,[X1]`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=0 & 1:R2=0`



Model	Result
Base	`CoRpteT+dsb-isb allowed (1 of 2) 7347ms`
ETS	`CoRpteT+dsb-isb allowed (1 of 2) 7604ms`

A.3.4.8 Test: CoRpteT.EL1+dsb-tlbi-dsb-isb forbid

In order to forbid the case in the previous test, an extra `TLBI` (with correct synchronization) must be inserted to remove those cached entries before trying to load the possibly-unmapped location.

AArch64 CoRpteT.EL1+dsb−tlbi−dsb−isb

Page table setup:
`physical pa1;`
`intermediate ipa1;`
`x ↦ pa1;`
`x ?-> invalid;`
`identity 0x1000 with code;`
`*pa1 = 0;`
	Initial state:
	`0:R0=0b0`
	`0:R1=pte3(x,page_table_base)`
	`1:PSTATE.EL=0b01`
	`1:R1=pte3(x,page_table_base)`
	`1:R3=x`
	`1:R4=page(x)`
	`1:VBAR_EL1=0x1000`

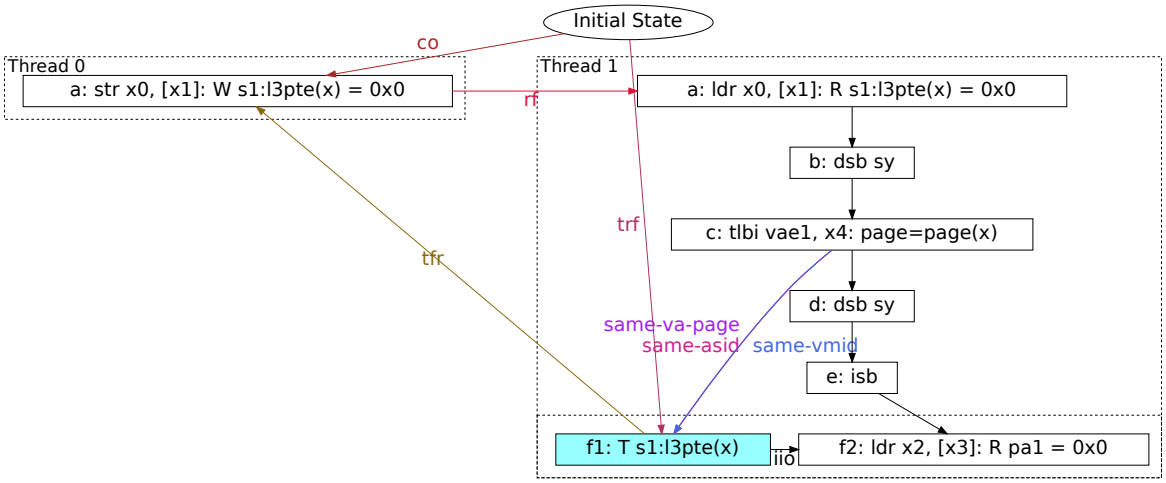
	Thread 0
	`STR X0,[X1]`
	Thread 1

	`LDR X0,[X1]`
	`DSB SY`
	`TLBI VAE1,X4`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=0 & 1:R2=0`



Model	Result
Base	`CoRpteT.EL1+dsb-tlbi-dsb-isb forbidden (0 of 2) 53579ms`
ETS	`CoRpteT.EL1+dsb-tlbi-dsb-isb forbidden (0 of 2) 28345ms`

A.3.4.9 Test: CoRpteT.EL1+dsb-tlbi-dsb allow

Same as previous test, but without the `ISB`.

AArch64 CoRpteT.EL1+dsb−tlbi−dsb

Page table setup:
`physical pa1;`
`x ↦ pa1;`
`x ?-> invalid;`
`identity 0x1000 with code;`
`*pa1 = 0;`
	Initial state:
	`0:R0=0b0`
	`0:R1=pte3(x,page_table_base)`
	`1:PSTATE.EL=0b01`
	`1:R1=pte3(x,page_table_base)`
	`1:R3=x`
	`1:R4=page(x)`
	`1:VBAR_EL1=0x1000`

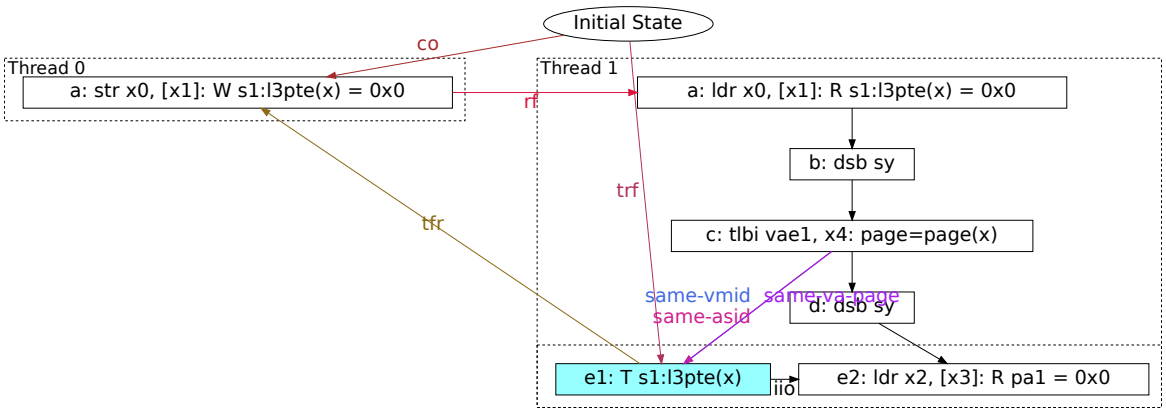
	Thread 0
	`STR X0,[X1]`
	Thread 1

	`LDR X0,[X1]`
	`DSB SY`
	`TLBI VAE1,X4`
	`DSB SY`
	`LDR X2,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=0 & 1:R2=0`



Model	Result
Base	`CoRpteT.EL1+dsb-tlbi-dsb allowed (1 of 2) 10293ms`
ETS	`CoRpteT.EL1+dsb-tlbi-dsb allowed (1 of 2) 11815ms`

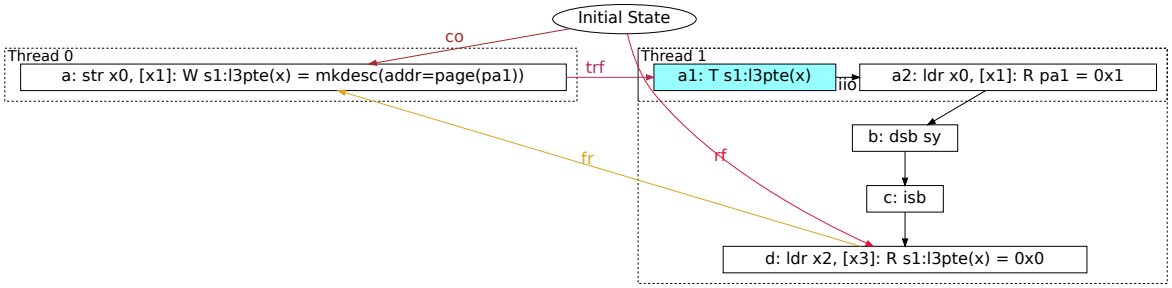
A.3.4.10 Test: CoTRpte.inv+dsb-isb forbid

Here, Thread 0 makes a new mapping, and Thread 1 reads-from that entry during a translation-table-walk before trying to load the entry itself.

If Thread 1 sees the new mapping, then 'later' loads of the translation table must see that entry or something newer. This is Data→Translation coherence.

AArch64 CoTRpte.inv+dsb−isb

Page table setup:	Initial state:
physical pa1; x ↦ invalid; x ?-> pa1; y ↦ pa1; *pa1 = 1; **identity** 0x1000 **with code**;	0:R0=**desc3**(y,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=**pte3**(x,page_table_base) 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



Model	Result
Base	CoTRpte.inv+dsb-isb forbidden (0 **of** 2) 4436ms
ETS	CoTRpte.inv+dsb-isb forbidden (0 **of** 2) 5367ms

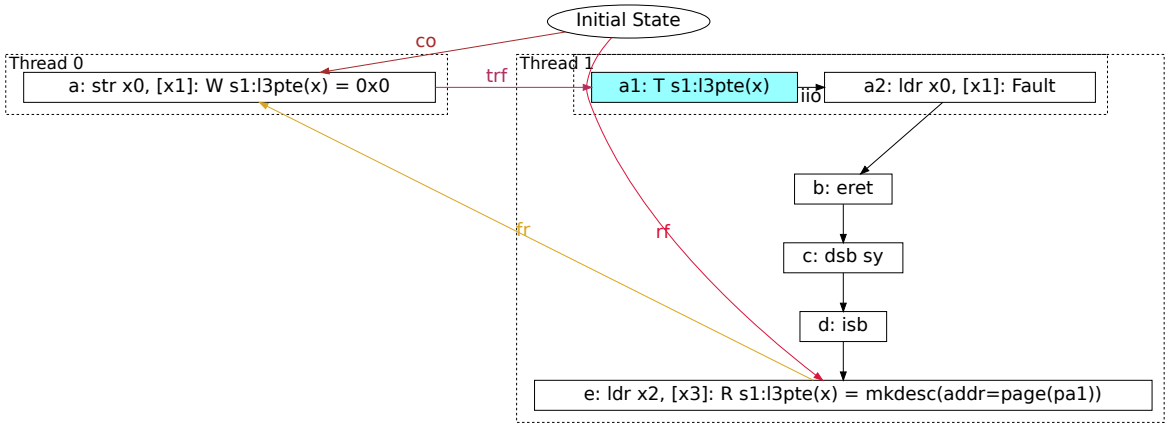
CoTfRpte-shaped tests

A.3.4.11 Test: CoTfRpte+dsb-isb forbid

Here, Thread 0 unmaps a location and Thread 1 translates that location getting a translation fault. Is a later load of the translation-table entry in Thread 1 required to see the write of the invalid descriptor or something newer? On Armv8-A, yes, since the fault is from a TLB-miss it reads from memory and therefore respects the total coherence-order for that entry.

AArch64 CoTfRpte+dsb−isb

Page table setup: **physical** pa1; x ↦ pa1; x ?-> invalid; y ↦ pa1; **identity** 0x1000 **with code**;	Initial state: 0:R0=0b0 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=**pte3**(x,page_table_base) 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X0,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=**desc3**(y,page_table_base)



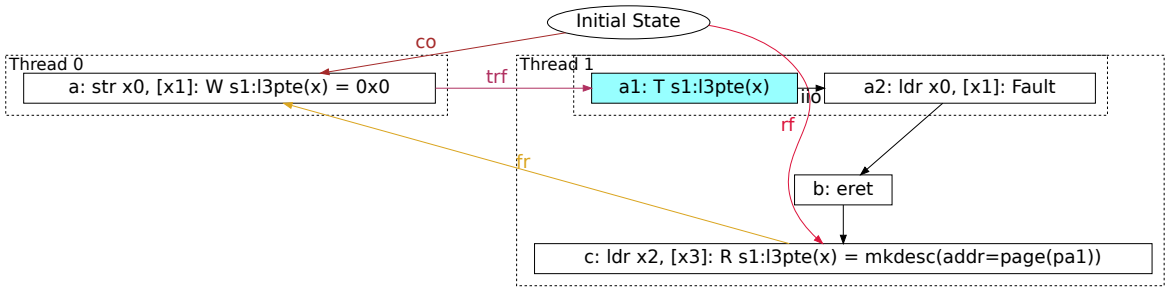
Model	Result
Base	CoTfRpte+dsb-isb forbidden (0 **of** 2) 5543ms
ETS	CoTfRpte+dsb-isb forbidden (0 **of** 2) 5852ms

A.3.4.12 Test: CoTfRpte+po This test cannot exist, because a fault causes an exception, which the test has to take, and from which the test has to return, both of which cause synchronisation.

A.3.4.13 Test: CoTfRpte+eret forbid

AArch64 CoTfRpte+eret

Page table setup: **physical** pa1; x ↦ pa1; x ?-> invalid; y ↦ pa1; **identity** 0x1000 **with code**;	Initial state: 0:R0=0b0 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R3=**pte3**(x,page_table_base) 1:VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X0,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=**desc3**(y,page_table_base)



Model	Result
Base	CoTfRpte+eret forbidden (0 **of** 2) 2995ms
ETS	CoTfRpte+eret forbidden (0 **of** 2) 4168ms

CoTfW.inv-shaped tests

A.3.4.14 Test: CoTfW.inv+dsb-isb forbid

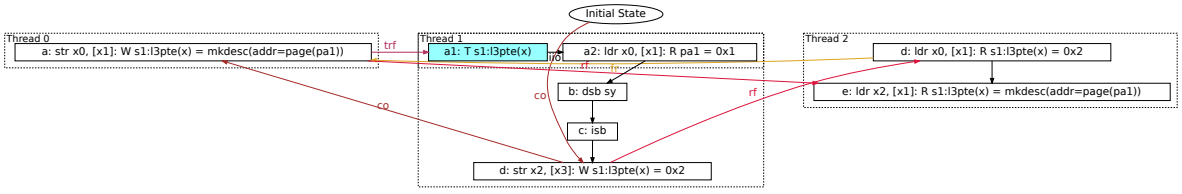
Here, Thread 0 writes a new mapping, and Thread 1 then sees this mapping with a translation-table walk, before overwriting it with another entry. Can this later write be coherence-before the original write?

Our model forbids this, requiring that writes pass the point of coherence before being visible to translations.

AArch64 CoTfW.inv+dsb−isb

Page table setup:
physical pa1;
x ↦ invalid;
x ?-> pa1;
x ?-> raw(2);
*pa1 = 1;
identity 0x1000 **with code**;
	Initial state:
	0:PSTATE.EL=0b00
	0:PSTATE.SP=0b0
	0:R0=**mkdesc3**(oa=pa1)
	0:R1=**pte3**(x,page_table_base)
	1:PSTATE.EL=0b00
	1:PSTATE.SP=0b0
	1:R1=x
	1:R2=0b10
	1:R3=**pte3**(x,page_table_base)
	1:VBAR_EL1=0x1000
	2:R1=**pte3**(x,page_table_base)

	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **DSB** SY **ISB** **STR** X2,[X3]
	Thread 2
	LDR X0,[X1] **LDR** X2,[X1]
	thread1 el1 handler
	0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 2:R0=2 & 2:R2=**mkdesc3**(oa=pa1)



Model	Result
Base	`CoTfW.inv+dsb-isb forbidden (0 of 2) 5572ms`
ETS	`CoTfW.inv+dsb-isb forbidden (0 of 2) 8392ms`

A.3.4.15 Test: CoTfW.inv+po forbid

This is like the previous test, except that there is no barrier between the instruction which faults and the program-order-later write.

However, this write is still ordered after the translation-table-walk. This is due to the write being unable to propagate until after the translation-table-walk has finished and the fault is known, as stores cannot propagate while speculative.

AArch64 CoTfW.inv+po

Page table setup:

```
physical pa1;
x ↦ invalid;
x ?-> pa1;
x ?-> raw(2);
*pa1 = 1;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b10
1:R3=pte3(x,page_table_base)
1:VBAR_EL1=0x1000
2:R1=pte3(x,page_table_base)
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

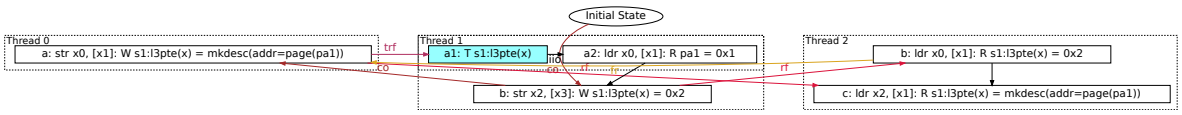
Thread 2

```
LDR X0,[X1]
LDR X2,[X1]
```

thread0 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=2 & 2:R2=mkdesc3(oa=pa1)`



Model	Result
Base	`CoTfW.inv+po forbidden (0 of 2) 6038ms`
ETS	`CoTfW.inv+po forbidden (0 of 2) 7738ms`

A.3.4.16 Test: PPODA.RT.inv allow?

Can writes be forwarded to translation-table-walks in general?

AArch64 PPODA.RT.inv

Page table setup:

```
physical pa1 pa2 pa3;
w ↦ pa1;
x ↦ invalid;
x ?-> pa1;
y ↦ pa2;
z ↦ pa3;
*pa1 = 1;
*pa2 = 0;
*pa3 = 0;

identity 0x1000 with code;
```

Initial state:

```
0:R0=0x1
0:R1=z
0:R2=0x1
0:R3=y
1:R1=y
1:R3=desc3(w,page_table_base)
1:R4=pte3(x,page_table_base)
1:R5=x
1:R7=z
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

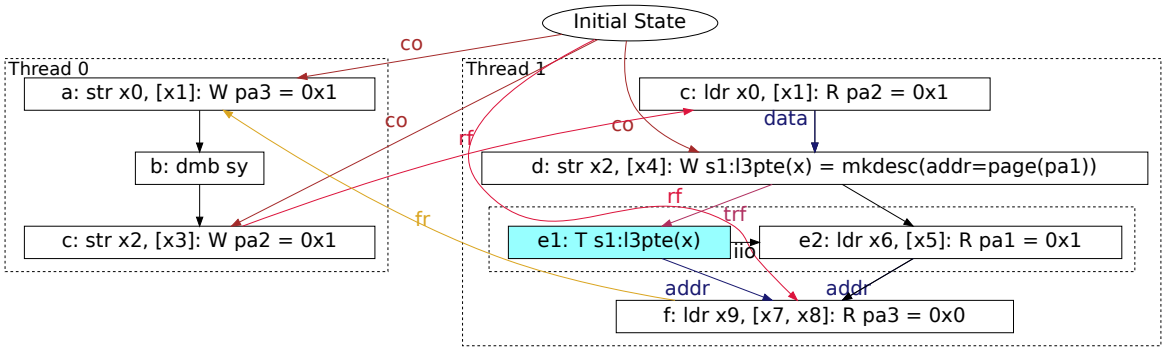
Thread 1

```
LDR X0,[X1]
EOR X2,X0,X0
ORR X2,X2,X3
STR X2,[X4]
LDR X6,[X5]
EOR X8,X6,X6
LDR X9,[X7,X8]
```

thread1 el1 handler

```
0x1400:
MOV X6,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R6=1 & 1:R9=0`



Model	Result
Base	`PPODA.RT.inv forbidden (0 of 2) 5779ms`
ETS	`PPODA.RT.inv forbidden (0 of 2) 6596ms`

A.3.5 Write-forwarding

A.3.5.1 Test: MP.RT.inv+dmb+ctrl-trfi forbid

It seems that write forwarding should not allowed down speculative paths. Allowing it could be problematic, as forwarding in a non-taken path from a write to the translate for a read would make it possible for the read to read from device memory, which the device could observe.

AArch64 MP.RT.inv+dmb+ctrl−trfi

Page table setup:

```
physical pa1 pa2;
w ↦ invalid;
w ?-> pa1;
x ↦ pa1;
*pa1 = 0;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:R0=0b1
0:R1=x
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R2=mkdesc3(oa=pa1)
1:R3=pte3(w,page_table_base)
1:R5=w
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

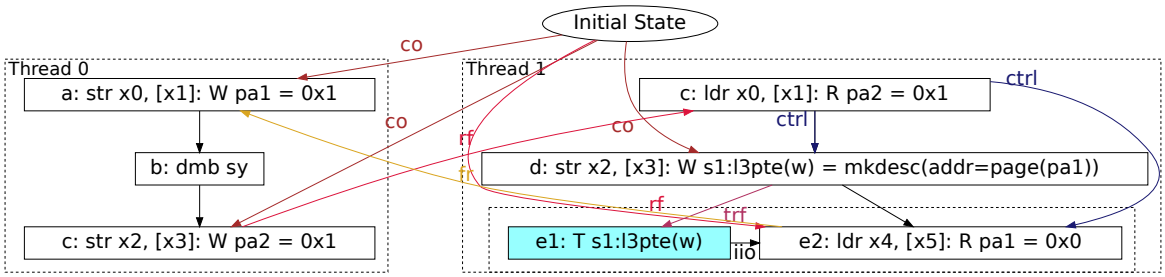
Thread 1

```
LDR X0,[X1]
CBZ X0,LC00
LC00:
STR X2,[X3]
LDR X4,[X5]
```

thread1 el1 handler

```
0x1400:
MOV X4,#2
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R4=0`



Model	Result
Base	`MP.RT.inv+dmb+ctrl-trfi forbidden (0 of 4) 6011ms`
ETS	`MP.RT.inv+dmb+ctrl-trfi forbidden (0 of 4) 6144ms`

A.3.5.2 Test: MP.RT.inv+dmb+addr-trfi forbid

Model	Result
Base	no result for MP.RT.inv+dmb+addr-trfi
ETS	`MP.RT.inv+dmb+addr-trfi forbidden (0 of 2) 7235ms`

A.3.6 Address dependencies

A.3.6.1 **MP.RTf.inv+dmb+addr** forbid

(See MP.RTf.inv+dmb+addr from earlier)

A.3.7 Data dependencies

A.3.7.1 **MP.RTf.inv+dmb+data** allow

The data dependency (unlike an address dependency) is to the memory access itself, not its translates, so a translate can happen much earlier.

(See MP.RTf.inv+dmb+data for diagrams/results)

A.4 Unmapping memory and TLB invalidation

In the previous section, we explored the sequences required to successfully create, or *map* a given virtual address in the current address space.

Removing a mapping (or *unmapping*) is more subtle: simply overwriting the entry back to zero is not enough, due to caching of those old entries in the thread's TLB.

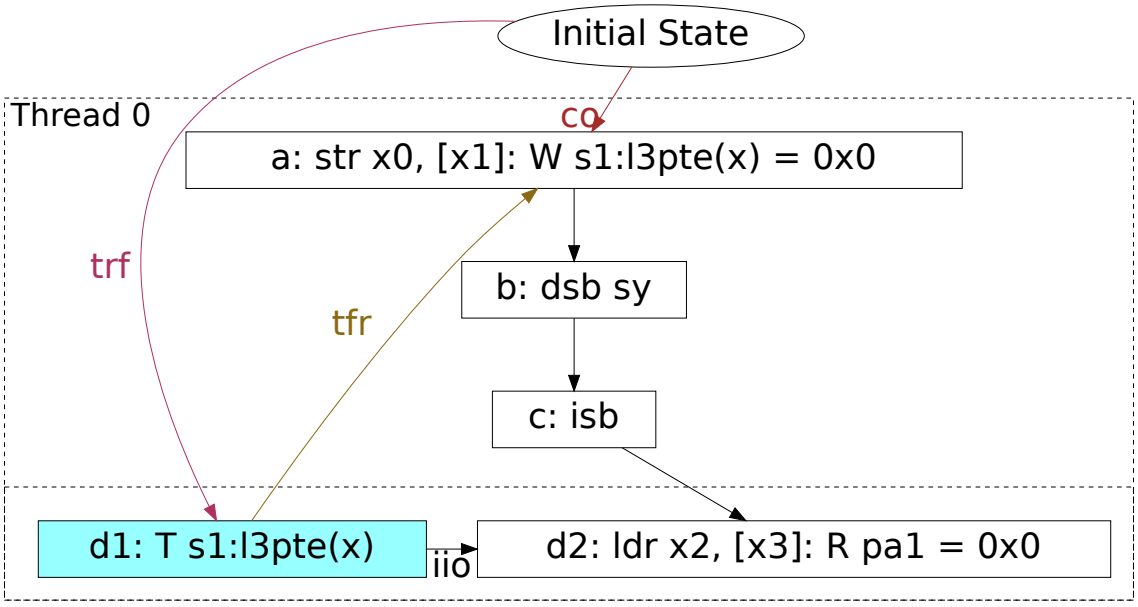
A.4.1 Same-thread unmap

CoWinvT-shaped tests

A.4.1.1 Test: CoWinvT+dsb-isb allow

AArch64 CoWinvT+dsb−isb

Page table setup:	Initial state:
physical pa1 pa2; x ↦ pa1; x ?-> invalid; **identity** 0x1000 **with code**;	PSTATE.SP=0b0 R0=0b0 R1=**pte3**(x,page_table_base) R3=x VBAR_EL1=0x1000
	Thread 0
	STR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	thread0 el1 handler
	0x1400: **MOV** X2,#1 **MRS** X20,ELR_EL1 **ADD** X20,X20,#4 **MSR** ELR_EL1,X20 **ERET**
	Final state: 0:R2=0



Model	Result
Base	CoWinvT+dsb-isb allowed (1 **of** 2) 5292ms
ETS	CoWinvT+dsb-isb allowed (1 **of** 2) 5212ms

A.4.1.2 Test: CoWinvT.EL1+dsb-tlbi-dsb allow

This is the 'break' side of break-before-make, but without an `ISB` at the end on the same thread, so it is not guaranteed that the po-later translations for this core are restarted.

AArch64 CoWinvT.EL1+dsb−tlbi−dsb

Page table setup:
`physical pa1 pa2;`
`x ↦ pa1;`
`x ?-> invalid;`
`identity 0x1000 with code;`
	Initial state:
	`PSTATE.EL=0b01`
	`R0=0b0`
	`R1=pte3(x,page_table_base)`
	`R3=x`
	`R5=page(x)`
	`VBAR_EL1=0x1000`

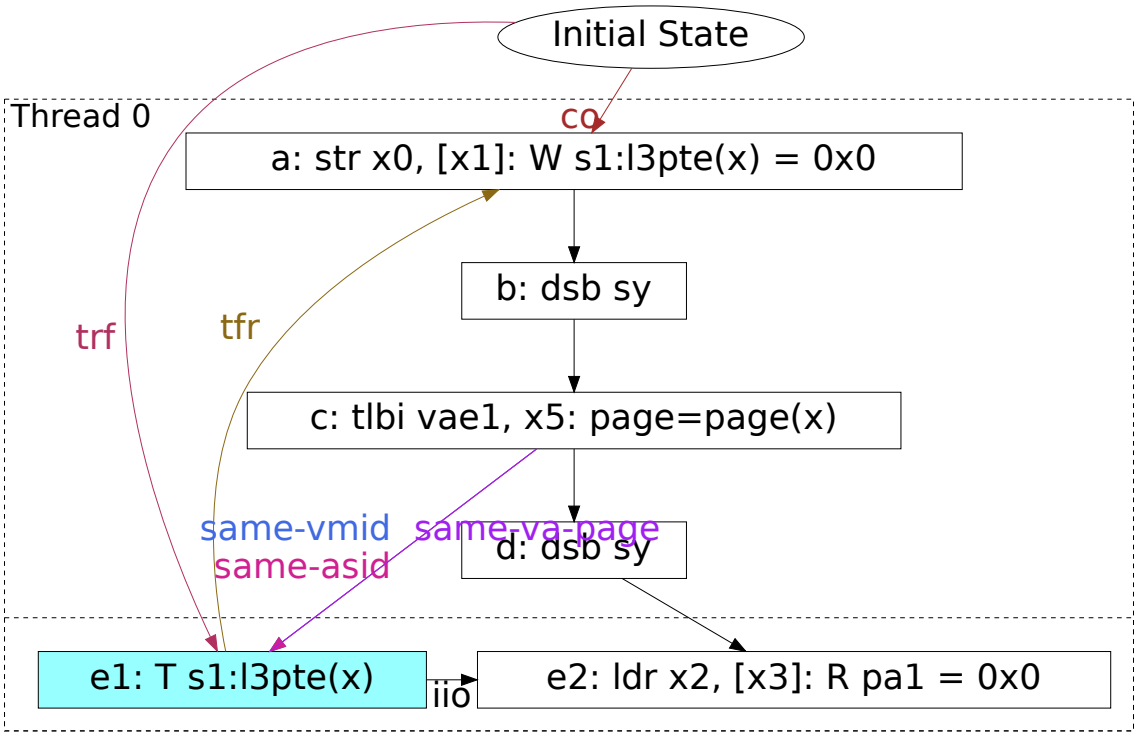
	Thread 0

	`STR X0,[X1]`
	`DSB SY`
	`TLBI VAE1,X5`
	`DSB SY`
	`MOV X2,#0`
	`LDR X2,[X3]`

	thread0 el1 handler

	`0x1000:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `0:R2=0`



Model	Result
Base	`CoWinvT.EL1+dsb-tlbi-dsb allowed (1 of 2) 10117ms`
ETS	`CoWinvT.EL1+dsb-tlbi-dsb allowed (1 of 2) 12882ms`

A.4.1.3 Test: CoWinvT.EL1+dsb-tlbiis-dsb allow

This is the same as the previous test, but with a broadcast TLBI.

AArch64 CoWinvT.EL1+dsb−tlbiis−dsb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;
identity 0x1000 with code;
```

Initial state:

```
PSTATE.EL=0b01
R0=0b0
R1=pte3(x,page_table_base)
R3=x
R5=page(x)
VBAR_EL1=0x1000
```

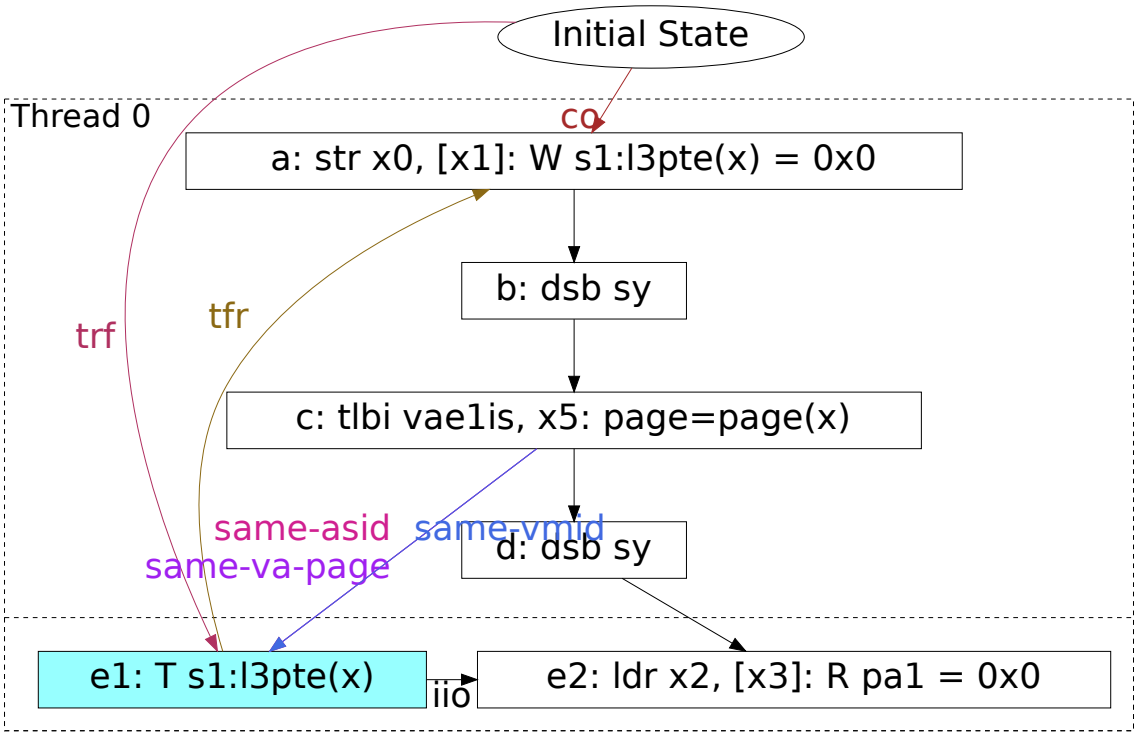
Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X5
DSB SY
MOV X2,#0
LDR X2,[X3]
```

thread0 el1 handler

```
0x1000:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R2=0`



Model	Result
Base	`CoWinvT.EL1+dsb-tlbiis-dsb allowed (1 of 2) 11490ms`
ETS	`CoWinvT.EL1+dsb-tlbiis-dsb allowed (1 of 2) 12759ms`

A.4.1.4 Test: CoWinvT.EL1+dsb-tlbiis-dsb-isb forbid

This (similarly to the previous test) is the 'break' side of break-before-make, but now including the `ISB` at the end on the same thread, so this time it is guaranteed that the po-later translations for this core are restarted.

AArch64 CoWinvT.EL1+dsb−tlbiis−dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;
identity 0x1000 with code;
```

Initial state:

```
PSTATE.EL=0b01
R0=0b0
R1=pte3(x,page_table_base)
R3=x
R5=page(x)
VBAR_EL1=0x1000
```

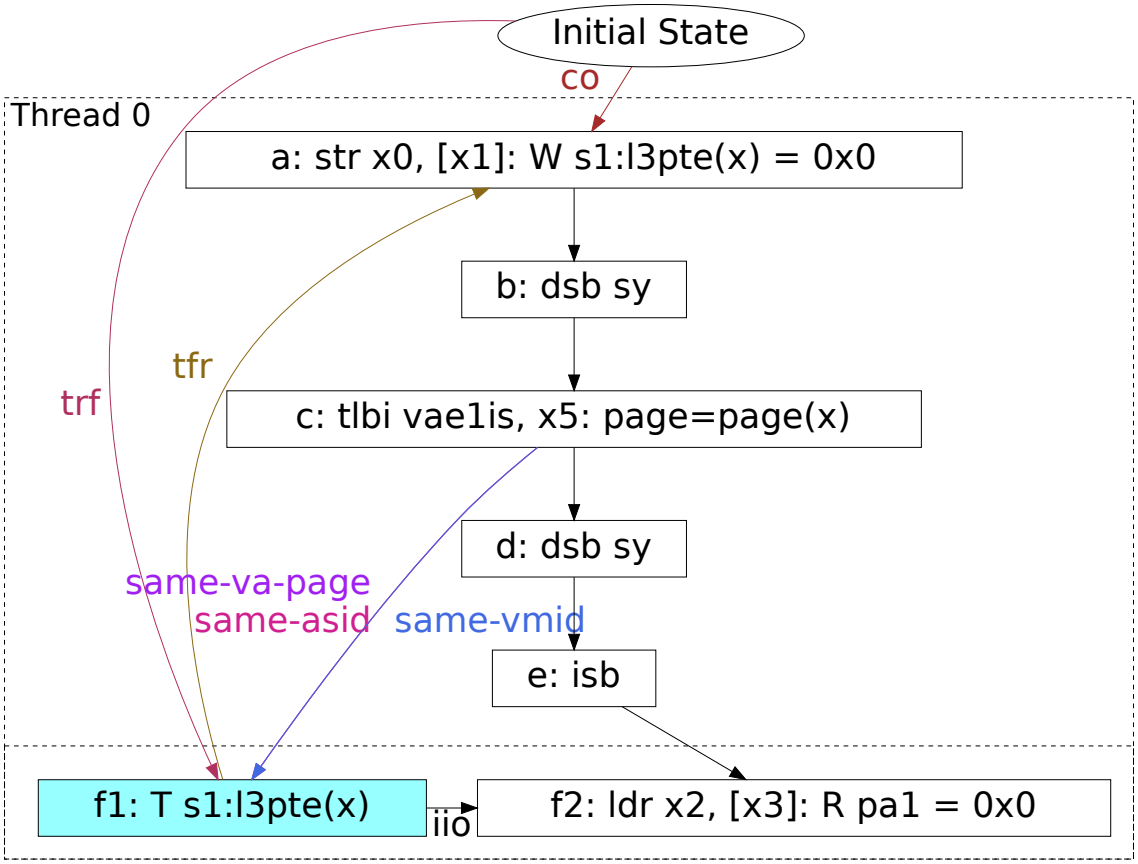
Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X5
DSB SY
ISB
MOV X2,#0
LDR X2,[X3]
```

thread0 el1 handler

```
0x1000:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R2=0`



Model	Result
Base	`CoWinvT.EL1+dsb-tlbiis-dsb-isb forbidden (0 of 2) 7548ms`
ETS	`CoWinvT.EL1+dsb-tlbiis-dsb-isb forbidden (0 of 2) 12986ms`

A.4.1.5 Test: MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb forbid

This is the 'break' side of break-before-make, now with a message passing to another core.

AArch64 MP.RT.EL1+dsb−tlbiis−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

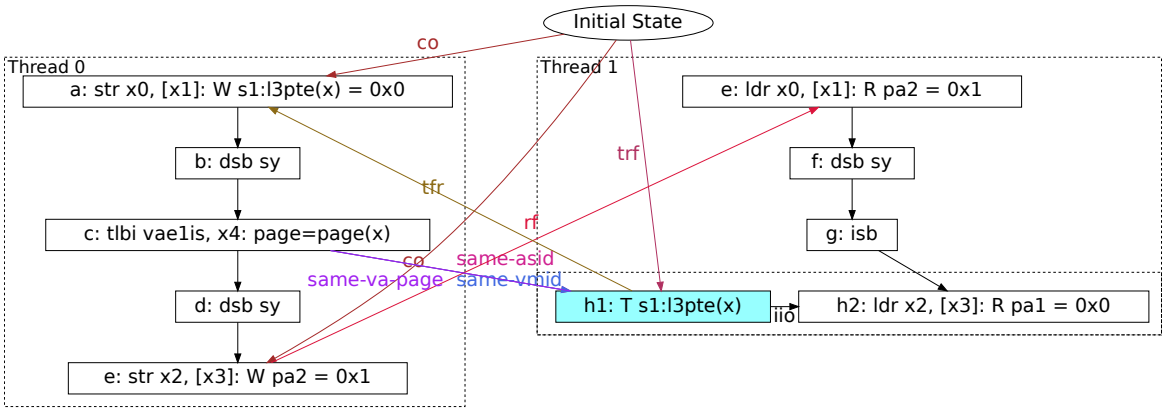
Thread 1

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 2) 348942ms`
ETS	`MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 2) 16656ms`

A.4.1.6 Test: RBS+dsb-tlbiis-dsb forbid

This 'read-broken-secret' (RBS) test is a fundamental test for the security guarantees 'break' gives you. Thread 0 unmaps a VA before writing to the original PA (for example, here, through an alias). Thread 1 attempts to read that VA.

It is allowed for Thread 1 to see the translation-fault, or to translate the VA using the old mapping and see the old write, but it is forbidden to translate the VA to the PA and see the new write.

This ensures that, once a mapping to a location is 'broken', later writes to that location are 'secret' for any cores that were using that VA.

AArch64 RBS+dsb−tlbiis−dsb

Page table setup:

```
physical pa1;
x ↦ pa1;
x ?-> invalid;
y ↦ pa1;
*pa1 = 0;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=0x2
0:R3=y
0:R5=page(x)
1:R1=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X5
DSB SY
STR X2,[X3]
```

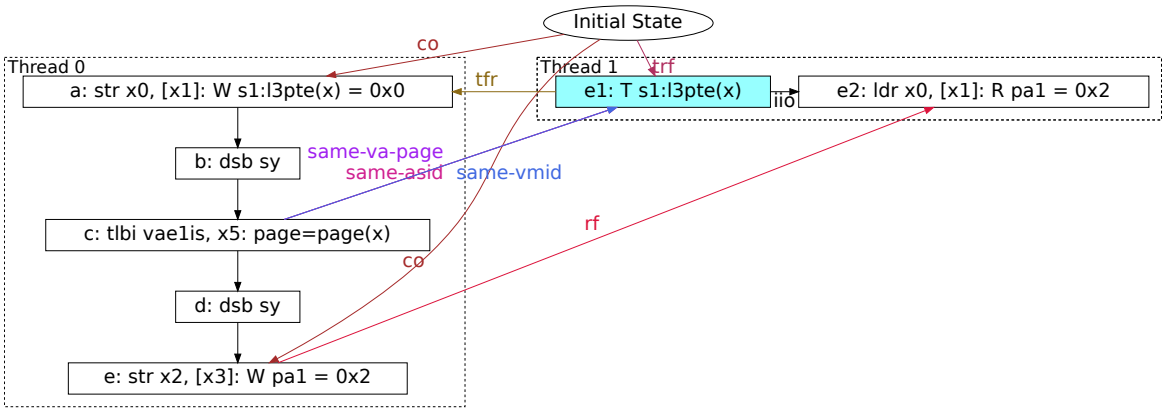
Thread 1

```
LDR X0,[X1]
```

thread1 el1 handler

```
0x1400:
MOV X0,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=2`



Model	Result
Base	`RBS+dsb-tlbiis-dsb forbidden (0 of 2) 4158ms`
ETS	`RBS+dsb-tlbiis-dsb forbidden (0 of 2) 12912ms`

A.5 More TLB invalidation

A.5.1 TLBI-pipeline interactions

Broadcast TLBI variants, also called TLB *shootdowns*, not only clean the cached entries in the TLB, but also go into the pipeline of other cores to invalidate any unfinished instructions that had already started using any of the old translations.

MP.RT.EL1-shaped tests

A.5.1.1 Test: MP.RT.EL1+dsb-tlbiis-dsb+dmb forbid

The message-passing ensures that the read of `x` is ordered after the broadcast `TLBI`, and the translate of `x` has to be on the same side as the access it translates for, so it has to be after the `TLBI` too, and therefore has to fault.

AArch64 MP.RT.EL1+dsb−tlbiis−dsb+dmb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

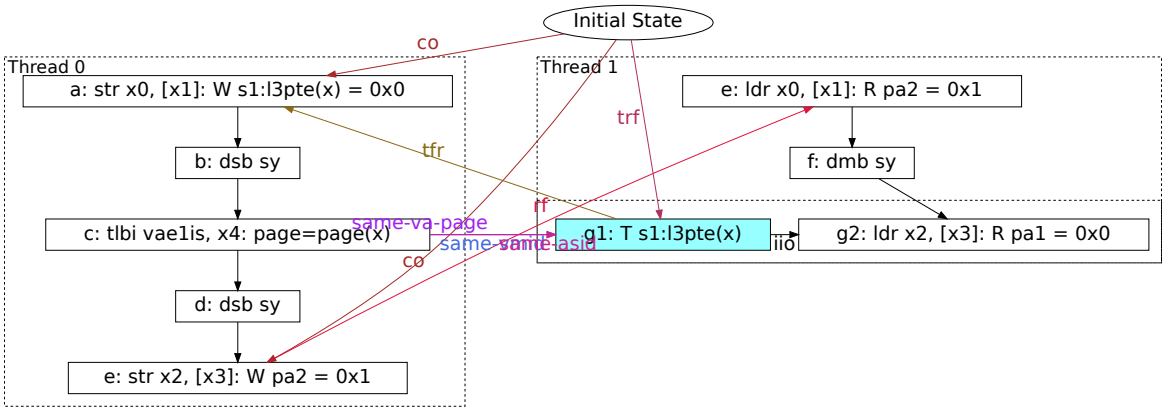
Thread 1

```
LDR X0,[X1]
DMB SY
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RT.EL1+dsb-tlbiis-dsb+dmb forbidden (0 of 2) 57372ms`
ETS	`MP.RT.EL1+dsb-tlbiis-dsb+dmb forbidden (0 of 2) 5360ms`

A.5.2 Thread-local TLBIs

In the previous section, we described broadcast TLB maintenance (aka 'TLB shootdowns'). But Arm also have thread-local TLB-maintenance instructions. These thread-local TLBIs can be used to clear thread-local context information (such as local TLB-cached entries), and can be combined together to emulate a broadcast TLB-maintenance by interrupting other cores and performing a thread-local TLBI.

A.5.2.1 Test: CoWinvT.EL1+dsb-tlbi-dsb-isb forbid

The effect of a thread-local TLBI is enough for the thread executing it.

AArch64 CoWinvT.EL1+dsb−tlbi−dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

identity 0x1000 with code;
```

Initial state:

```
PSTATE.EL=0b01
R0=0b0
R1=pte3(x,page_table_base)
R3=x
R5=page(x)
VBAR_EL1=0x1000
```

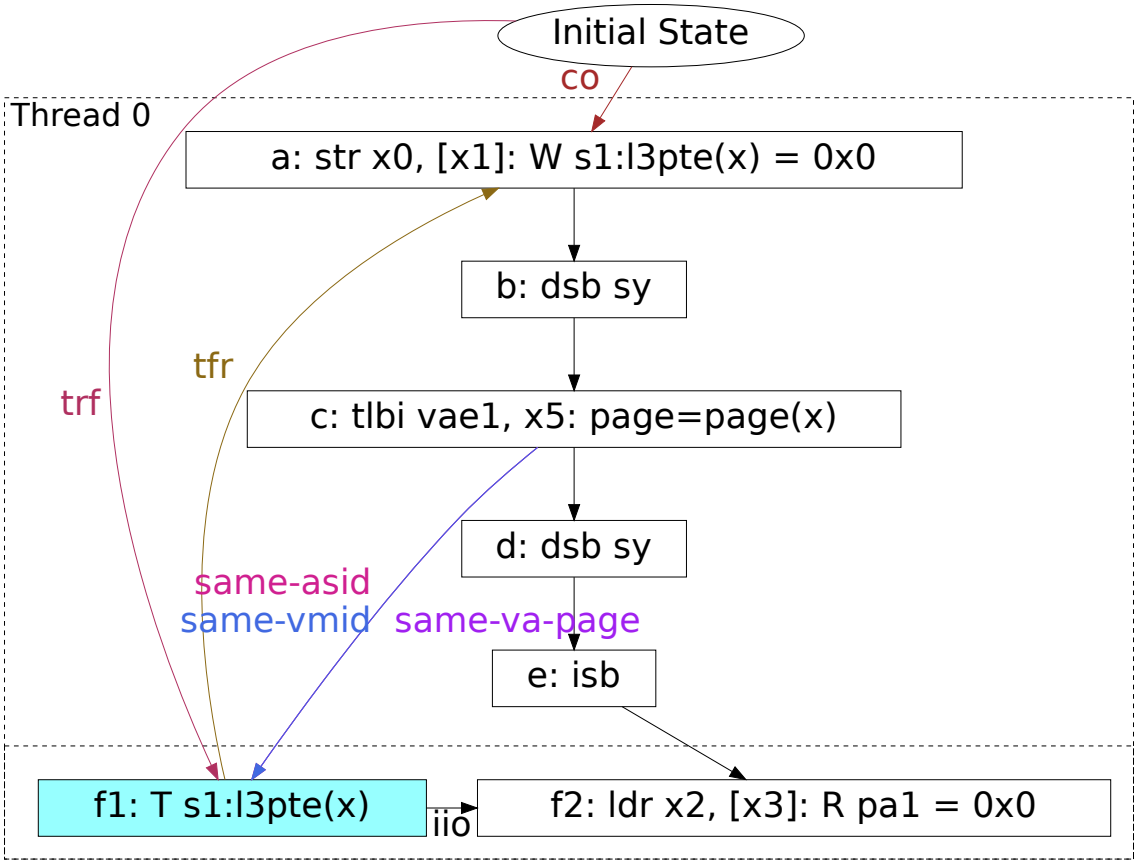
Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1,X5
DSB SY
ISB
LDR X2,[X3]
```

thread0 el1 handler

```
0x1000:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R2=0`



Model	Result
Base	`CoWinvT.EL1+dsb-tlbi-dsb-isb forbidden (0 of 2) 18380ms`
ETS	`CoWinvT.EL1+dsb-tlbi-dsb-isb forbidden (0 of 2) 25267ms`

A.5.2.2 Test: MP.RT.EL1+dsb-tlbi-dsb+dsb-isb allow

The effect of a thread-local TLBI is indeed thread-local, and does not get carried over to another thread by message-passing.

AArch64 MP.RT.EL1+dsb−tlbi−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

y ↦ pa2;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1,X4
DSB SY
STR X2,[X3]
```

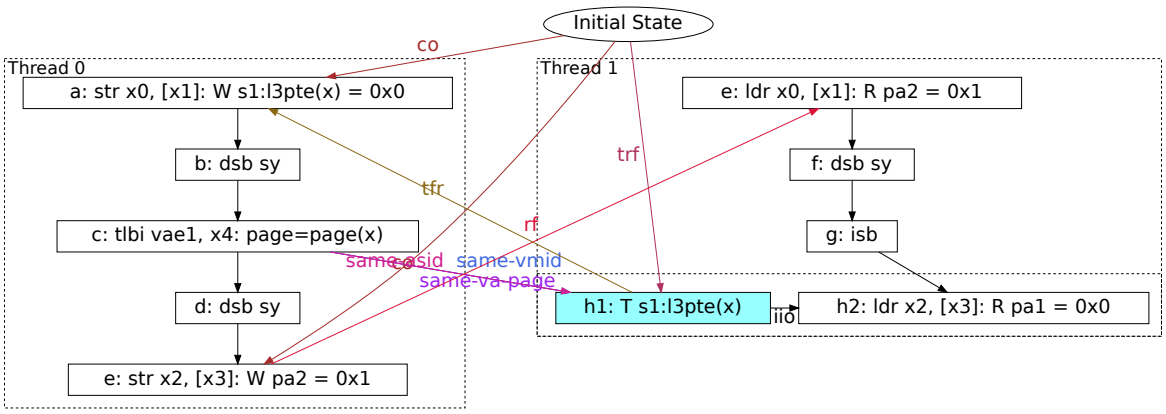
Thread 1

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RT.EL1+dsb-tlbi-dsb+dsb-isb allowed (1 of 2) 10678ms`
ETS	`MP.RT.EL1+dsb-tlbi-dsb+dsb-isb allowed (1 of 2) 87237ms`

A.5.2.3 Test: MP.RT.EL1+dsb-shootdown-dsb+dsb-isb forbid

A broadcast TLBI can be emulated by performing a thread-local TLBI on each core, with sufficient synchronization between them.

In the following test, we take MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb and split the broadcast TLBI over many threads. Thread 0 'breaks' the location and then sends messages to each core requesting it perform the TLB maintenance locally.

Note that to correctly emulate the behaviour of the broadcast TLBI, each core must perform an `ISB` (or other context-synchronizing event) to get the pipeline effects of the TLBI. This

requirement is slightly stronger than the TLBI semantics, also flushing unrelated accesses.

AArch64 MP.RT.EL1+dsb−shootdown−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2 pa3;
x ↦ pa1;
x ?-> invalid;

y ↦ pa2;
*pa2 = 0;

f ↦ pa3;
*pa3 = 0;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
0:R4=0b1
0:R5=f
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:R5=f
1:R6=page(x)
1:R7=0b10
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
STR X4,[X5]
LDR X6,[X5]
DSB SY
STR X2,[X3]
```

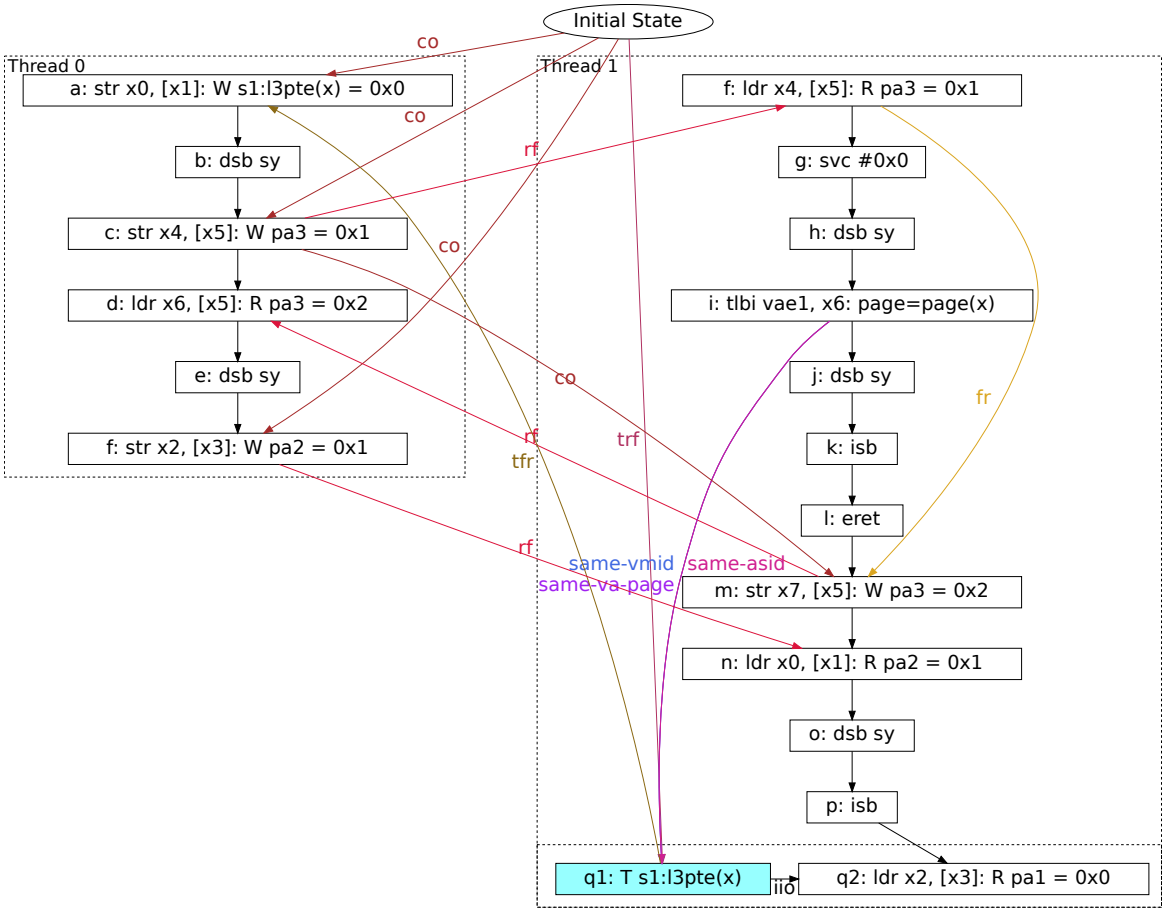
Thread 1

```
LDR X4,[X5]
SVC #0
STR X7,[X5]
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MRS X9,ESR_EL1
LSR X9,X9,#26
SUB X9,X9,#0b010101
CBNZ X9,1f
0:
DSB SY
TLBI VAE1,X6
DSB SY
ISB
ERET
1:
MOV X2,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R4=1 & 0:R6=2 & 1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.RT.EL1+dsb-shootdown-dsb+dsb-isb forbidden (0 of 2) 765865ms`
ETS	`MP.RT.EL1+dsb-shootdown-dsb+dsb-isb forbidden (0 of 2) 49466ms`

A.5.3 Multiple locations

A.5.3.1 Test: MP.RTT.EL1+dsb-tlbiis-tlbiis-dsb+dsb-isb forbid

Here, we invalidate *two* different VAs, and perform their TLBIs together in the same thread, effectively allowing concurrent execution of the two TLBIs on the same core.

AArch64 MP.RTT.EL1+dsb−tlbiis−tlbiis−dsb+dsb−isb

Page table setup:
`physical pa1 pa2 pa3;`
`x ↦ pa1;`
`x ?-> invalid;`
 	
`y ↦ pa2;`
 	
`z ↦ pa3;`
`z ?-> invalid;`
 	
`identity 0x1000 with code;`
	Initial state:
	`0:PSTATE.EL=0b01`
	`0:R0=0b0`
	`0:R1=pte3(x,page_table_base)`
	`0:R2=0b0`
	`0:R3=pte3(z,page_table_base)`
	`0:R4=page(x)`
	`0:R5=page(z)`
	`0:R6=0x1`
	`0:R7=y`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=y`
	`1:R3=x`
	`1:R5=z`
	`1:VBAR_EL1=0x1000`

	Thread 0

	`STR X0,[X1]`
	`STR X2,[X3]`
	`DSB SY`
	`TLBI VAE1IS,X4`
	`TLBI VAE1IS,X5`
	`DSB SY`
	`STR X6,[X7]`

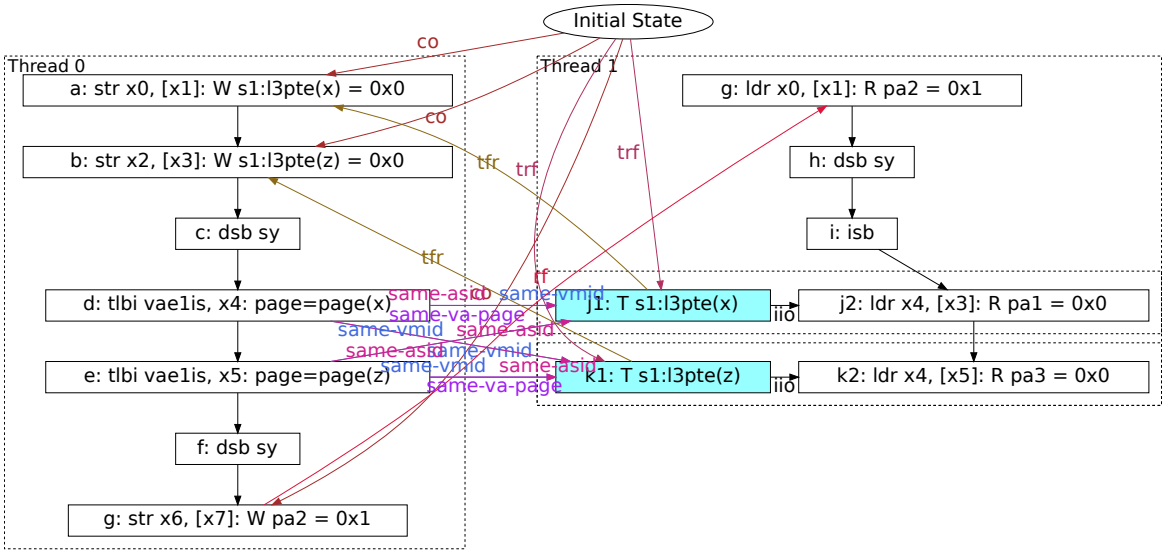
	Thread 1

	`LDR X0,[X1]`
	`DSB SY`
	`ISB`
	`LDR X4,[X3]`
	`MOV X2,X4`
	`LDR X4,[X5]`

	thread1 el1 handler

	`0x1400:`
	`MOV X4,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=1 & 1:R2=0 & 1:R4=0`



Model	Result
Base	`MP.RTT.EL1+dsb-tlbiis-tlbiis-dsb+dsb-isb forbidden (0 of 4) 520042ms`
ETS	`MP.RTT.EL1+dsb-tlbiis-tlbiis-dsb+dsb-isb forbidden (0 of 4) 305092ms`

A.6 Stage 1 Re-mapping and break-before-make

A.6.1 Break-before-make

BBM-shaped tests

A.6.1.1 Test: BBM+dsb-tlbiis-dsb allow

This is the smallest example of a safe change of output address mapping, using the 'break-before-make' (BBM) pattern.

Thread 1 loads a fixed VA, and Thread 0 tries to re-map that VA from the initial PA to a new one.

To do this safely, Arm prescribe a "break-before-make" sequence to ensure that the other threads will not ever see both the new and old mappings at the same time. Instead, the mapping must be 'broken' (unmapped) and cleaned between the two states.

This test is the minimum required to correctly change OA ("output address") for a given mapping.

AArch64 BBM+dsb−tlbiis−dsb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;
x ?-> pa2;

identity 0x1000 with code;

*pa2 = 2;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R4=0b1
0:R6=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X6
DSB SY
STR X2,[X1]
```

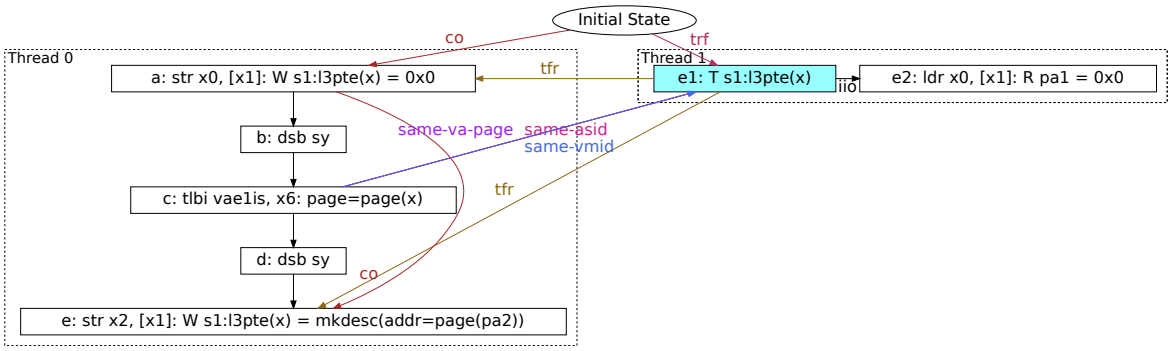
Thread 1

```
LDR X0,[X1]
```

thread1 el1 handler

```
0x1400:
MOV X0,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=0`



Model	Result
Base	`BBM+dsb-tlbiis-dsb allowed (1 of 3) 12702ms`
ETS	`BBM+dsb-tlbiis-dsb allowed (1 of 3) 19328ms`

A.6.1.2 Test: BBM.Tf+dsb-tlbiis-dsb allow

This illustrates the other allowed outcome of the previous test: the correct use of break-before-make ensures that the change of output address is safe, but does not guarantee that the new page table entry is seen. Thread 1 sees a translation-fault from the transient invalid entry during the break-before-make sequence.

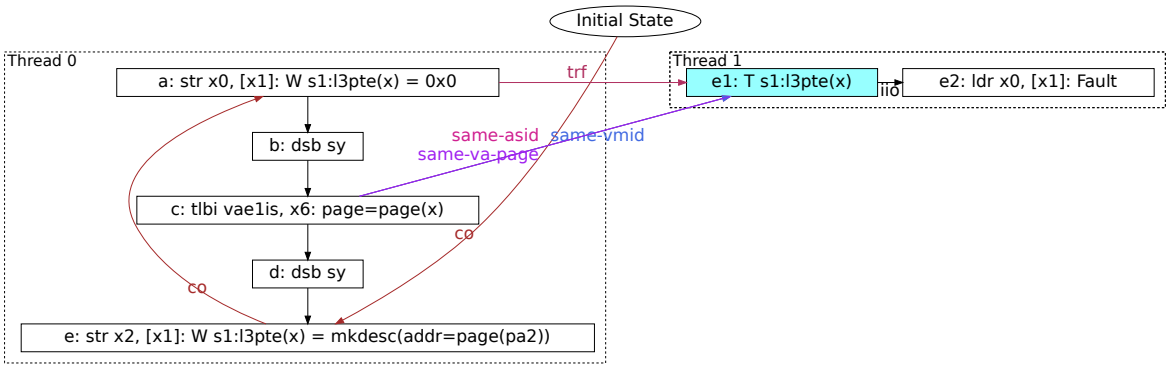
AArch64 BBM.Tf+dsb−tlbiis−dsb

Page table setup:
`physical pa1 pa2;`
`x ↦ pa1;`
`x ?-> invalid;`
`x ?-> pa2;`
 	
`identity 0x1000 with code;`
 	
`*pa2 = 2;`
	Initial state:
	`0:PSTATE.EL=0b01`
	`0:R0=0b0`
	`0:R1=pte3(x,page_table_base)`
	`0:R2=mkdesc3(oa=pa2)`
	`0:R4=0b1`
	`0:R6=page(x)`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=x`
	`1:VBAR_EL1=0x1000`

	Thread 0

	`STR X0,[X1]`
	`DSB SY`
	`TLBI VAE1IS,X6`
	`DSB SY`
	`STR X2,[X1]`

	Thread 1
	`LDR X0,[X1]`
	thread1 el1 handler
	`0x1400:` `MOV X0,#1`
	Final state: `1:R0=1`



Model	Result
Base	`BBM.Tf+dsb-tlbiis-dsb allowed (1 of 3) 9196ms`
ETS	`BBM.Tf+dsb-tlbiis-dsb allowed (1 of 3) 13659ms`

MP.BBM-shaped tests

A.6.1.3 Test: MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb forbid

In this test, Thread 0 break-before-makes a new mapping, and then synchronises with Thread 1 with a message pass.

While this is a slightly unusual setup, as one would mostly expect break-before-make to happen concurrently with the other thread, rather than be synchronised-before it, this is still an interesting test to explore the architecture.

Arm forbid both the translation-fault and the translation with the old entry.

AArch64 MP.BBM1+dsb−tlbiis−dsb−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2 pa3;
x ↦ pa1;
x ?-> invalid;
x ?-> pa2;

y ↦ pa3;
identity 0x1000 with code;

*pa2 = 1;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R4=0b1
0:R5=y
0:R6=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X6
DSB SY
STR X2,[X1]
DSB SY
STR X4,[X5]
```

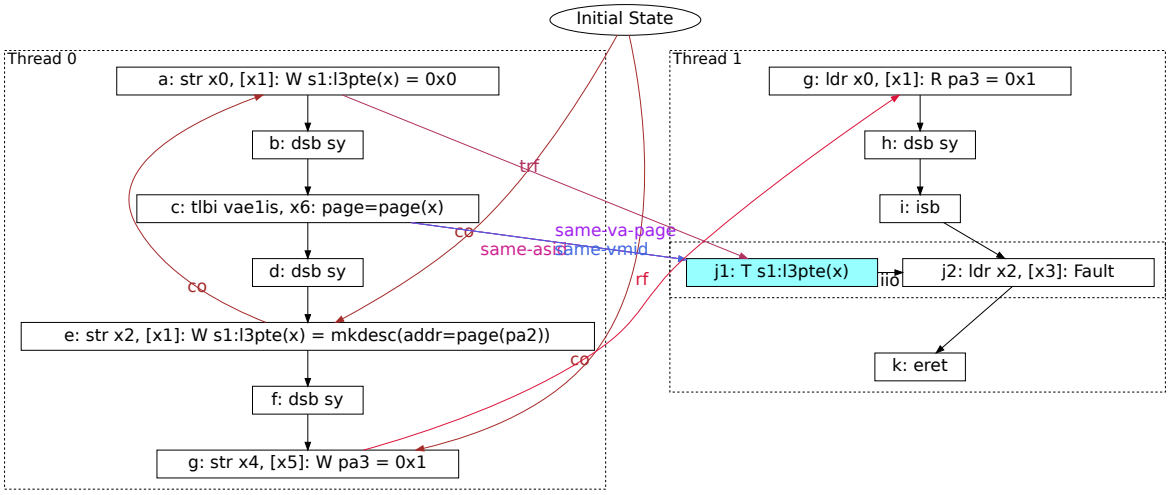
Thread 1

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb forbidden (0 of 3) 116116ms`
ETS	`MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb error (0 of 3) 182456ms`

A.6.1.4 Test: MP.BBM1+dsb-tlbiis-dsb-dsb+ctrl-isb forbid

AArch64 MP.BBM1+dsb−tlbiis−dsb−dsb+ctrl−isb

Page table setup:

```
physical pa1 pa2 pa3;
x ↦ pa1;
x ?-> invalid;
x ?-> pa2;

y ↦ pa3;
identity 0x1000 with code;

*pa2 = 1;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R4=0b1
0:R5=y
0:R6=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X6
DSB SY
STR X2,[X1]
DSB SY
STR X4,[X5]
```

Thread 1

```
LDR X0,[X1]
CBNZ X0,L0
L0:
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`

Model	Result
Base	`MP.BBM1+dsb-tlbiis-dsb-dsb+ctrl-isb forbidden (0 of 6) 1068905ms`
ETS	`MP.BBM1+dsb-tlbiis-dsb-dsb+ctrl-isb forbidden (0 of 6) 571232ms`

A.7 Translation-table-walk ordering

A.7.1 Inter-instruction ordering

Typically, translations of separate instructions are not ordered with respect to each other: simply having program-order between them introduces no strength. Earlier, we saw that even same-VA did not give strength (CoTTf.inv+po)

MP.TT.inv-shaped tests

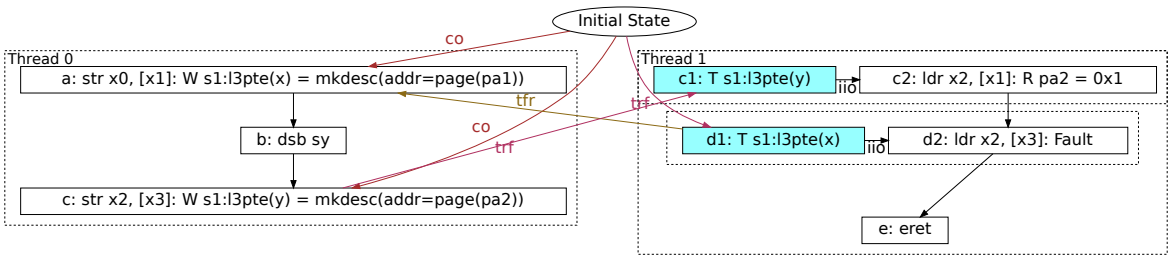
A.7.1.1 Test: MP.TTf.inv+dsb+po allow

The translates on the reader side are not ordered.

AArch64 MP.TTf.inv+dsb+po

Page table setup:
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;
 	
y ↦ invalid;
y ?-> pa2;
 	
*pa1 = 1;
*pa2 = 1;
identity 0x1000 **with code**;
	Initial state:
	0:R0=**mkdesc3**(oa=pa1)
	0:R1=**pte3**(x,page_table_base)
	0:R2=**mkdesc3**(oa=pa2)
	0:R3=**pte3**(y,page_table_base)
	1:PSTATE.EL=0b00
	1:PSTATE.SP=0b0
	1:R1=y
	1:R3=x
	1:VBAR_EL1=0x1000

	Thread 0
	STR X0,[X1] **DSB** SY **STR** X2,[X3]
	Thread 1
	LDR X2,[X1] **MOV** X0,X2 **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



Model	Result
Base	MP.TTf.inv+dsb+po allowed (1 **of** 4) 42061ms
ETS	MP.TTf.inv+dsb+po forbidden (0 **of** 4) 23415ms

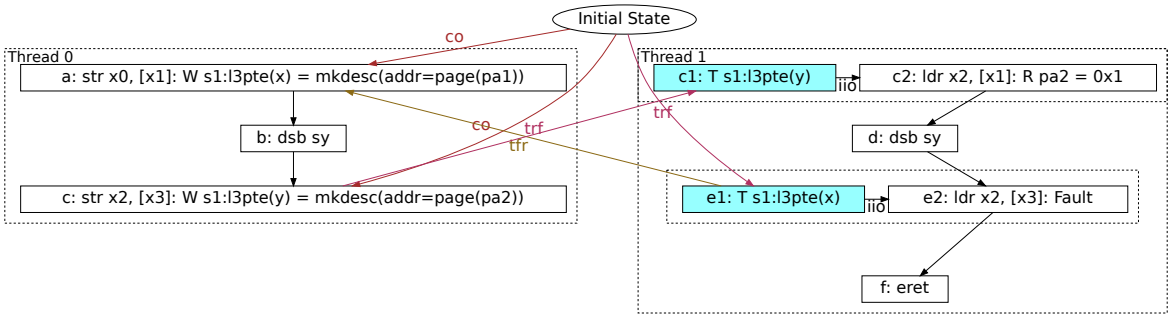
A.7.1.2 Test: MP.TTf.inv+dsbs allow (unless ETS, then forbid)

The translates on the reader side are not ordered, as the `DSB SY` does not order them by itself (it needs an `ISB`), except with ETS, which would provide order with the faults.

AArch64 MP.TTf.inv+dsbs

Page table setup:
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;
 	
y ↦ invalid;
y ?-> pa2;
 	
*pa1 = 1;
*pa2 = 1;
identity 0x1000 **with code**;
	Initial state:
	0:R0=**mkdesc3**(oa=pa1)
	0:R1=**pte3**(x,page_table_base)
	0:R2=**mkdesc3**(oa=pa2)
	0:R3=**pte3**(y,page_table_base)
	1:PSTATE.EL=0b00
	1:PSTATE.SP=0b0
	1:R1=y
	1:R3=x
	1:VBAR_EL1=0x1000

	Thread 0
	STR X0,[X1] **DSB** SY **STR** X2,[X3]
	Thread 1
	LDR X2,[X1] **MOV** X0,X2 **DSB** SY **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 1:R2=0



Model	Result
Base	`MP.TTf.inv+dsbs allowed (1 of 4) 64484ms`
ETS	`MP.TTf.inv+dsbs forbidden (0 of 4) 28800ms`

A.7.1.3 Test: MP.TTf.inv+dsb+dsb-isb forbid

The translates on the reader side are ordered by the `DSB SY;ISB` combination.

AArch64 MP.TTf.inv+dsb+dsb−isb

Page table setup:
`physical pa1 pa2;`
`x ↦ invalid;`
`x ?-> pa1;`
 	
`y ↦ invalid;`
`y ?-> pa2;`
 	
`*pa1 = 1;`
`*pa2 = 1;`
`identity 0x1000 with code;`
	Initial state:
	`0:R0=mkdesc3(oa=pa1)`
	`0:R1=pte3(x,page_table_base)`
	`0:R2=mkdesc3(oa=pa2)`
	`0:R3=pte3(y,page_table_base)`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=y`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`

	Thread 0

	`STR X0,[X1]`
	`DSB SY`
	`STR X2,[X3]`

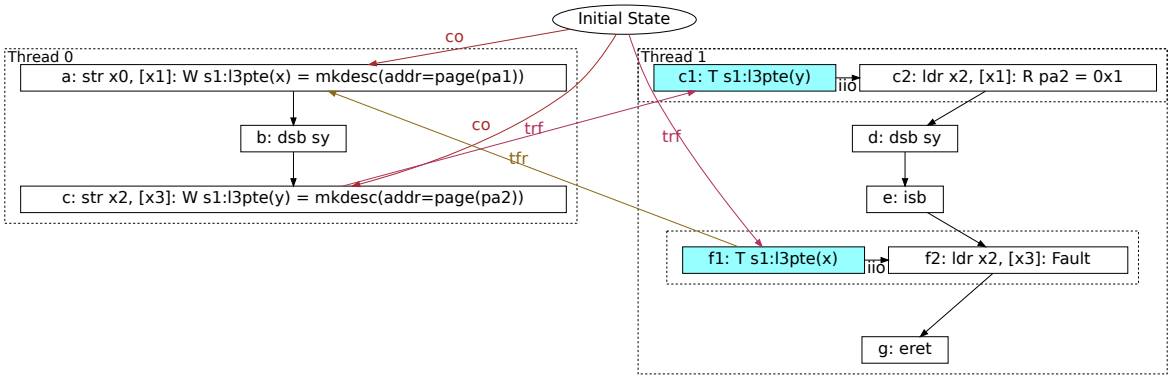
	Thread 1

	`LDR X2,[X1]`
	`MOV X0,X2`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.TTf.inv+dsb+dsb-isb forbidden (0 of 4) 29708ms`
ETS	`MP.TTf.inv+dsb+dsb-isb forbidden (0 of 4) 22505ms`

A.7.1.4 Test: MP.TTf.inv+dsb+ctrl-isb forbid

AArch64 MP.TTf.inv+dsb+ctrl−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 1;
*pa2 = 1;
identity 0x1000 with code;
```

Initial state:

```
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
STR X2,[X3]
```

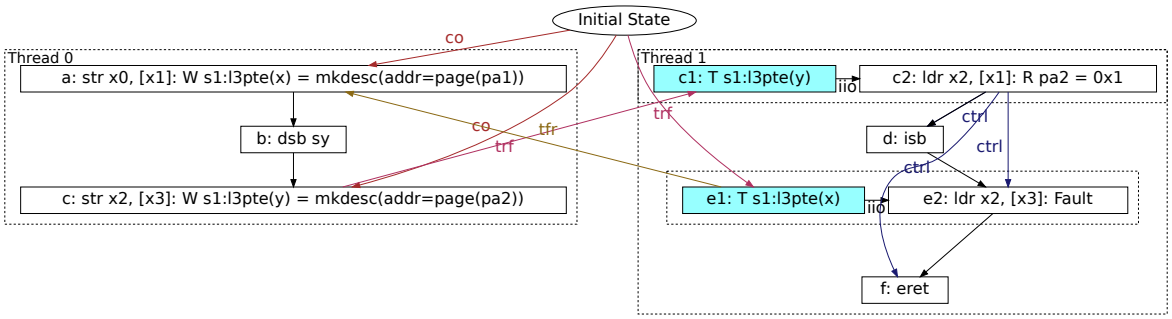
Thread 1

```
LDR X2,[X1]
MOV X0,X2
CBNZ X0,L0
L0:
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.TTf.inv+dsb+ctrl-isb forbidden (0 of 6) 30317ms`
ETS	`MP.TTf.inv+dsb+ctrl-isb forbidden (0 of 6) 37812ms`

A.7.1.5 Test: MP.TTf.inv+dmb+dsb-isb forbid

The `DMB SY` on the writer thread is enough (a `DSB SY` is not needed), as it is only here to order the writes as normal writes.

AArch64 MP.TTf.inv+dmb+dsb−isb

Page table setup:	Initial state:

`physical pa1 pa2;`
`x ↦ invalid;`
`x ?-> pa1;`
 	
`y ↦ invalid;`
`y ?-> pa2;`
 	
`*pa1 = 1;`
`*pa2 = 1;`
`identity 0x1000 with code;`
	`0:R0=mkdesc3(oa=pa1)`
	`0:R1=pte3(x,page_table_base)`
	`0:R2=mkdesc3(oa=pa2)`
	`0:R3=pte3(y,page_table_base)`
	`1:PSTATE.EL=0b00`
	`1:PSTATE.SP=0b0`
	`1:R1=y`
	`1:R3=x`
	`1:VBAR_EL1=0x1000`

	Thread 0

	`STR X0,[X1]`
	`DMB SY`
	`STR X2,[X3]`

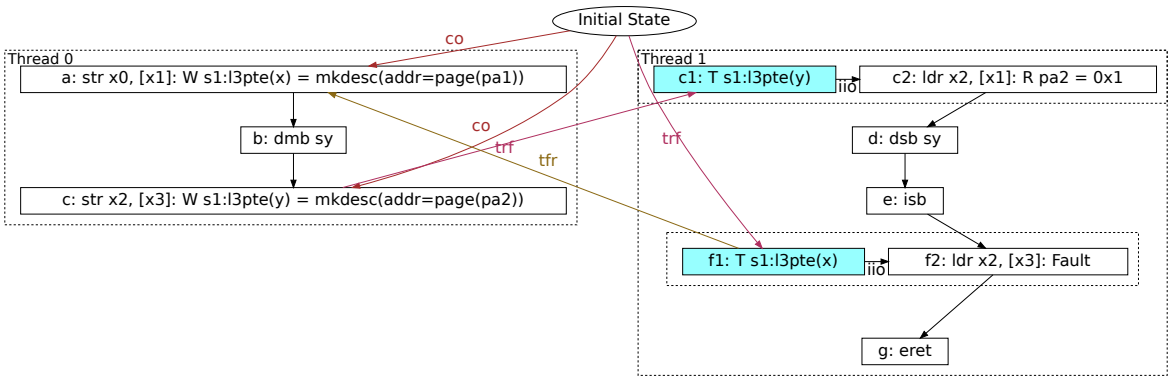
	Thread 1

	`LDR X2,[X1]`
	`MOV X0,X2`
	`DSB SY`
	`ISB`
	`LDR X2,[X3]`

	thread1 el1 handler

	`0x1400:`
	`MOV X2,#0`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.TTf.inv+dmb+dsb-isb forbidden (0 of 4) 27192ms`
ETS	`MP.TTf.inv+dmb+dsb-isb forbidden (0 of 4) 47972ms`

A.7.1.6 Test: MP.TTf.inv+dmb+po allow

Note that although the final translation-fault in the second thread must be ordered-before the write of the valid entry in Thread 1 (as the fault must come from a non-TLB read of memory), the translation-fault is not necessarily ordered-after the initial translate or even the first load's read.

AArch64 MP.TTf.inv+dmb+po

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 1;
*pa2 = 1;
identity 0x1000 with code;
```

Initial state:

```
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

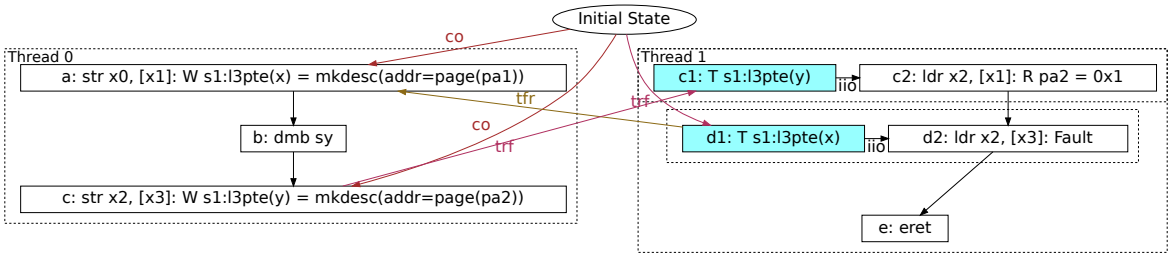
Thread 1

```
LDR X2,[X1]
MOV X0,X2
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.TTf.inv+dmb+po allowed (1 of 4) 52201ms`
ETS	`MP.TTf.inv+dmb+po forbidden (0 of 4) 9113ms`

A.7.1.7 Test: MP.TTf.inv.EL1+dsb-tlbiis-dsb+po allow

Program-order between the loads does not induce order between the translates of the loads.

AArch64 MP.TTf.inv.EL1+dsb−tlbiis−dsb+po

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 1;
*pa2 = 1;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

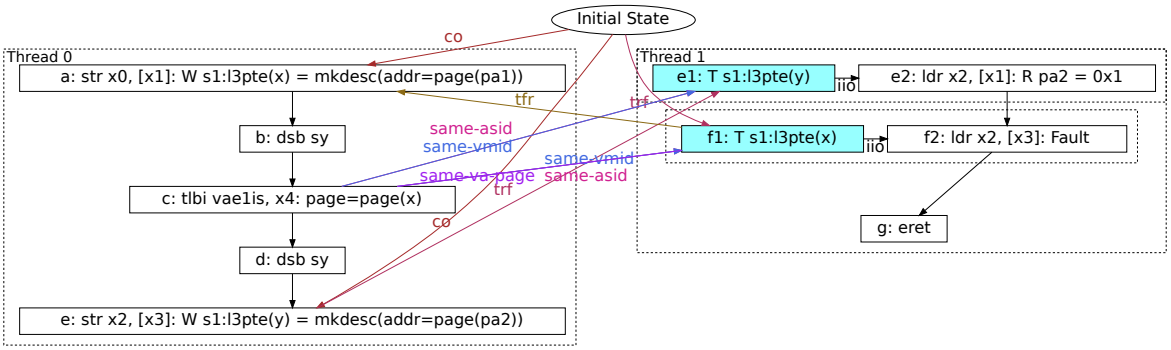
Thread 1

```
LDR X2,[X1]
MOV X0,X2
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.TTf.inv.EL1+dsb-tlbiis-dsb+po forbidden (0 of 4) 162552ms`
ETS	`MP.TTf.inv.EL1+dsb-tlbiis-dsb+po forbidden (0 of 4) 200958ms`

A.7.1.8 Test: MP.TTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb forbid

A `DSB; ISB` between the loads does not induce order between the translates of the loads.

AArch64 MP.TTf.inv.EL1+dsb−tlbiis−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 1;
*pa2 = 1;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
0:R4=page(x)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X4
DSB SY
STR X2,[X3]
```

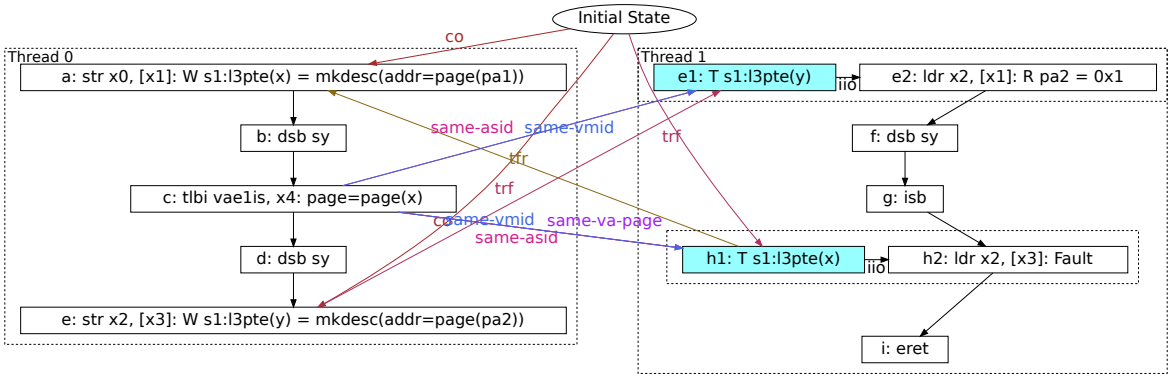
Thread 1

```
LDR X2,[X1]
MOV X0,X2
DSB SY
ISB
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 1:R2=0`



Model	Result
Base	`MP.TTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 4) 16436ms`
ETS	`MP.TTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 4) 29491ms`

A.7.2 Multi-level translations

ROT.inv-shaped tests

A.7.2.1 Test: ROT.inv+dsb forbid

In this ROT test ("reorder translation"), Thread 0 writes to the leaf entry of a fresh (unused) translation-table, and then replaces an (initially invalid) leaf higher in the table with a new table entry which points to the freshly created table.

Thread 1 then tries to load an address that would use this new freshly made entry. If the individual accesses during the translation-table-walk are allowed to re-order, then it would be possible for Thread 1 to see the updated table but still see the old leaf entry.

This must be forbidden, requiring that the translation-table-walk happens 'in-order' and the ordering on Thread 0 ensures the two writes are visible to the walker in that order.

The exception handler code records which translation level caused the exception.

AArch64 ROT.inv+dsb

Page table setup:

```
physical pa1;
intermediate ipa1;

assert pa1 == ipa1;
ipa1↦pa1;

s1table new_table 0x280000 {
  x↦invalid;
  x ?-> ipa1;
};

identity 0x283000 with default;

x↦invalid at level 2;
x ?-> table(0x283000) at level 2;

identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=mkdesc3(oa=ipa1)
0:R1=pte3(x,new_table)
0:R2=mkdesc2(table=0x283000)
0:R3=pte2(x,page_table_base)
1:R1=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
STR X2,[X3]
```

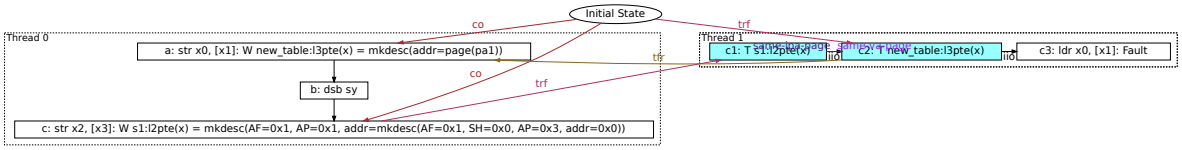
Thread 1

```
LDR X0,[X1]
```

thread 1 el1 handler

```
0x1400:
MRS X14,ESR_EL1
AND X14,X14,#0b111
CMP X14,#0b111
MOV X17,#1
MOV X18,#2
// if ESR_EL1.ISS.DFSC == Translation Level 3 then x2 = 1 else x2 = 2
CSEL X0,X17,X18,eq
```

Final state: `1:R0=1`



Model	Result
Base	`ROT.inv+dsb forbidden (0 of 3) 2805ms`
ETS	`ROT.inv+dsb forbidden (0 of 3) 3202ms`

A.7.2.2 Test: ROT.inv+dmbst forbid

This is like the previous test, but with a much weaker barrier between the two writes. This is also forbidden: any respected ordering between the writes would suffice.

AArch64 ROT.inv+dmbst

Page table setup:

```
physical pa1;
intermediate ipa1;

assert pa1 == ipa1;
ipa1 ↦ pa1;

s1table new_table 0x280000 {
  x ↦ invalid;
  x ?-> ipa1;
};

identity 0x283000 with default;

x ↦ invalid at level 2;
x ?-> table(0x283000) at level 2;

identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=mkdesc3(oa=ipa1)
0:R1=pte3(x,new_table)
0:R2=mkdesc2(table=0x283000)
0:R3=pte2(x,page_table_base)
1:R1=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DMB ST
STR X2,[X3]
```

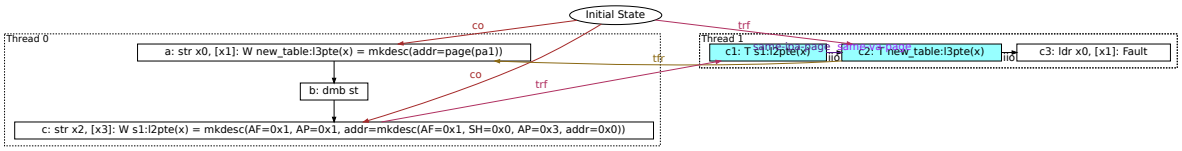
Thread 1

```
LDR X0,[X1]
```

thread 1 el1 handler

```
0x1400:
MRS x14,esr_el1
AND X14,X14,#0b111
CMP x14,#0b111
MOV X17,#1
MOV X18,#2
// if ESR_EL1.ISS.DFSC == Translation Level 3 then x2 = 1 else x2 = 2
CSEL x0,x17,x18,eq
```

Final state: `1:R0=1`



Model	Result
Base	`ROT.inv+dmbst forbidden (0 of 3) 3998ms`
ETS	`ROT.inv+dmbst forbidden (0 of 3) 3919ms`

A.7.2.3 Test: LB+data-trfis forbid

This is a variant of LB+datas+WW.

AArch64 LB+data−trfis

Page table setup:

```
physical pa1 pa2 pa3 pa4;
x ↦ pa1;
y ↦ pa2;
u ↦ invalid;
u ?-> pa2;
v ↦ invalid;
v ?-> pa1;
identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R1=x
0:R3=desc3(y,page_table_base)
0:R4=pte3(u,page_table_base)
0:R5=0b1
0:R6=u
0:R7=0b1
0:VBAR_EL1=0x1000
1:R1=y
1:R3=desc3(x,page_table_base)
1:R4=pte3(v,page_table_base)
1:R5=0b1
1:R6=v
1:R7=0b1
1:VBAR_EL1=0x2000
```

Thread 0

```
LDR X0,[X1]
EOR X2,X0,X0
ORR X3,X3,X2
STR X3,[X4]
STR X5,[X6]
```

Thread 1

```
LDR X0,[X1]
EOR X2,X0,X0
ORR X3,X3,X2
STR X3,[X4]
STR X5,[X6]
```

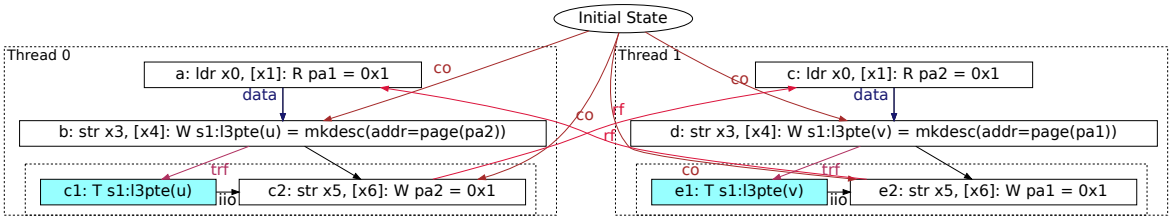
thread1 el0 handler

```
0x1400:
MOV X7,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el1 handler

```
0x2400:
MOV X7,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R0=1 & 0:R7=1 & 1:R0=1 & 1:R7=1`



Model	Result
Base	`LB+data-trfis forbidden (0 of 4) 26579ms`
ETS	`LB+data-trfis forbidden (0 of 4) 18056ms`

A.7.2.4 Test: LB+addr-trfis forbid

This is a variant of LB+datas+WW.

AArch64 LB+addr−trfis

Page table setup:

```
physical pa1 pa2 pa3 pa4;
x ↦ pa1;
y ↦ pa2;
u ↦ invalid;
u ?-> pa2;
v ↦ invalid;
v ?-> pa1;
identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R1=x
0:R3=desc3(y,page_table_base)
0:R4=pte3(u,page_table_base)
0:R5=0b1
0:R6=u
0:R7=0b1
0:VBAR_EL1=0x1000
0:__isla_monomorphize_writes=true
1:R1=y
1:R3=desc3(x,page_table_base)
1:R4=pte3(v,page_table_base)
1:R5=0b1
1:R6=v
1:R7=0b1
1:VBAR_EL1=0x2000
1:__isla_monomorphize_writes=true
```

Thread 0

```
LDR X0,[X1]
EOR X2,X0,X0
STR X3,[X4,X2]
STR X5,[X6]
```

Thread 1

```
LDR X0,[X1]
EOR X2,X0,X0
STR X3,[X4,X2]
STR X5,[X6]
```

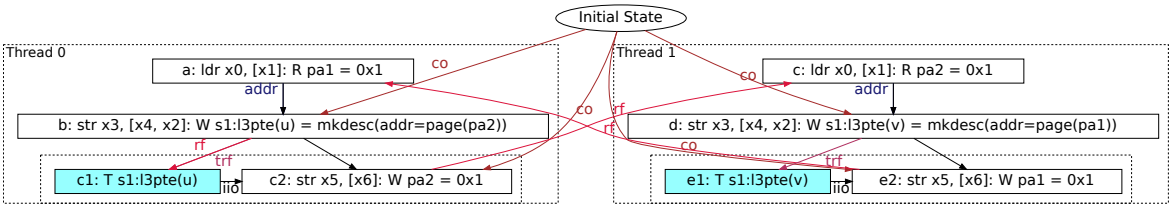
thread1 el0 handler

```
0x1400:
MOV X7,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el1 handler

```
0x2400:
MOV X7,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R0=1 & 0:R7=1 & 1:R0=1 & 1:R7=1`



Model	Result
Base	`LB+addr-trfis forbidden (0 of 4) 7042ms`
ETS	`LB+addr-trfis forbidden (0 of 4) 7977ms`

A.7.2.5 Test: WRC.TfRT+po+dsb-isb allow

AArch64 WRC.TfRT+po+dsb−isb

Page table setup:

```
physical pa1 pa2;

x ↦ pa1;
x ?-> invalid;
*pa1 = 1;

y ↦ pa2;
*pa2 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=0b0
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

Thread 2

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

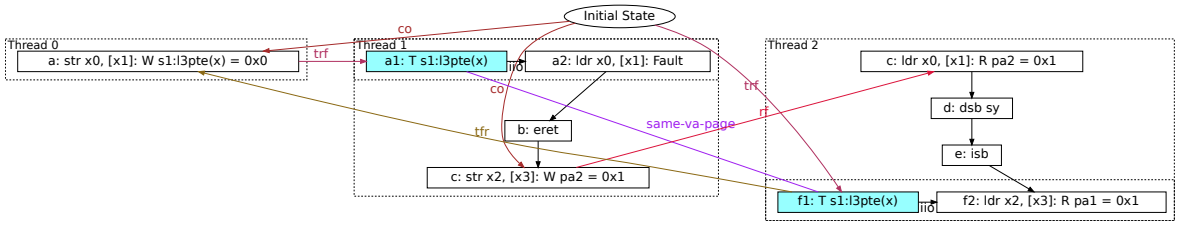
thread1 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=0 & 2:R0=1 & 2:R2=1`



Model	Result
Base	`WRC.TfRT+po+dsb-isb allowed (1 of 4) 39075ms`
ETS	`WRC.TfRT+po+dsb-isb allowed (1 of 4) 57628ms`

A.7.2.6 Test: WRC.TfRT+dsb-tlbiis-dsb+dsb-isb allow

AArch64 WRC.TfRT+dsb−tlbiis−dsb+dsb−isb

Page table setup:

```
physical pa1 pa2;

x ↦ pa1;
x ?-> invalid;
*pa1 = 1;

y ↦ pa2;
*pa2 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=0b0
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:R4=page(x)
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
// the TLBI is in the handler
STR X2,[X3]
```

Thread 2

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

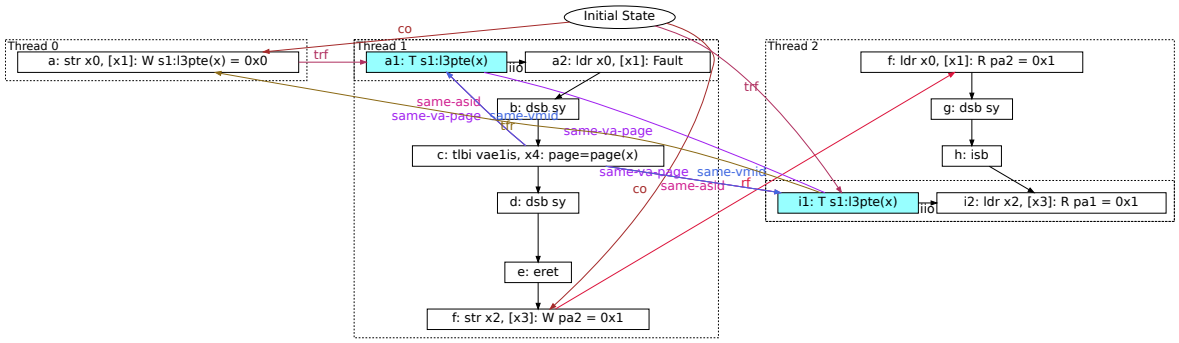
thread1 el1 handler

```
0x1400:
MOV X0,#0
DSB SY
TLBI VAE1IS,X4
DSB SY
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=0 & 2:R0=1 & 2:R2=1`



Model	Result
Base	`WRC.TfRT+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 4) 549315ms`
ETS	`WRC.TfRT+dsb-tlbiis-dsb+dsb-isb forbidden (0 of 4) 1582501ms`

A.8 Multi-copy atomicity

A.8.1 MCA translation-table-walk

A fundamental guarantee given by Armv8 over data memory is that of *multi-copy atomicity*, that is, once a write is seen by one other core, then all cores must see it or something newer if they try read that location.

Translation-table-walks are a kind of read, and we can ask whether those reads come with the same guarantee. There are multiple ways in which multi-copy atomicity *could* be violated here:

1. If a translation-table-walk reads from a write, must another core's translation-table-walk that reads the same entry read-from that same write or something newer?
2. If a load reads a translation table entry directly, must another core's translation-table-walk that reads that location read-from that write or something newer?
3. If a translation-table-walk reads from a write, must another core which loads that entry read-from that write or something newer?

We tackle each in turn.

Note that questions about multi-copy atomicity are only interesting under assumptions about coherence, and due to the lack of data→translation coherence in the non-TLB-miss case (c.f. CoRT+dsb-isb), all the tests below start from an invalid state to avoid those uninteresting cases.

A.8.1.1 Test: CoWTf.inv+po-ctrl-isb+po forbid?

Can another core see a write propagate to its translation-table-walk 'before' the writer thread's own translation-table-walker does?

In this test Thread 0 writes a new valid descriptor which Thread 1 uses in a translation-table-walk before sending a message back to Thread 0; if Thread 0 sees that message can a later translation-table-walk still see the old invalid entry?

AArch64 CoWTf.inv+po−ctrl−isb+po

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;
u ↦ pa1;
*pa1 = 1;

y ↦ pa2;
*pa2 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(u,page_table_base)
0:R1=pte3(x,page_table_base)
0:R3=y
0:R5=x
0:VBAR_EL1=0x1000
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
LDR X2,[X3]
CBNZ X2,L0
L0:
ISB
LDR X4,[X5]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

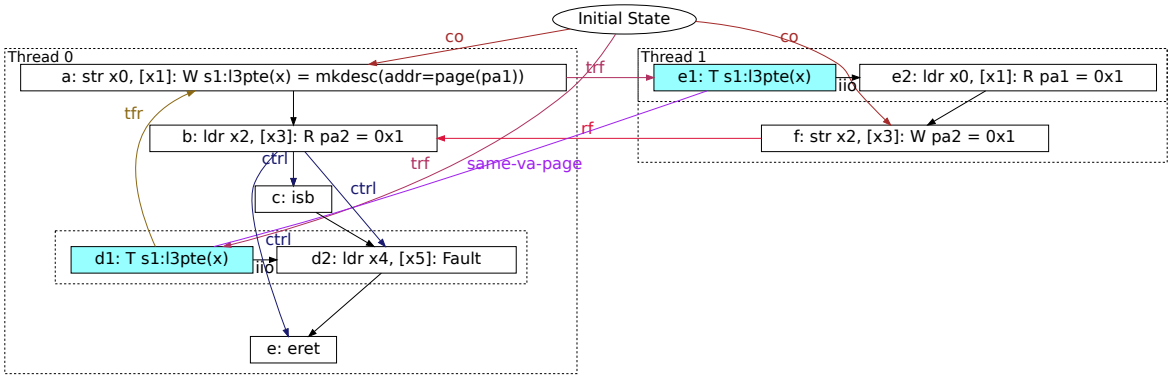
thread0 el1 handler

```
0x1400:
MOV X4,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el1 handler

```
0x2400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `0:R2=1 & 0:R4=0 & 1:R0=1`



Model	Result
Base	`CoWTf.inv+po-ctrl-isb+po allowed (1 of 8) 64271ms`
ETS	`CoWTf.inv+po-ctrl-isb+po allowed (1 of 8) 70941ms`

WRC.TRTf.inv-shaped tests

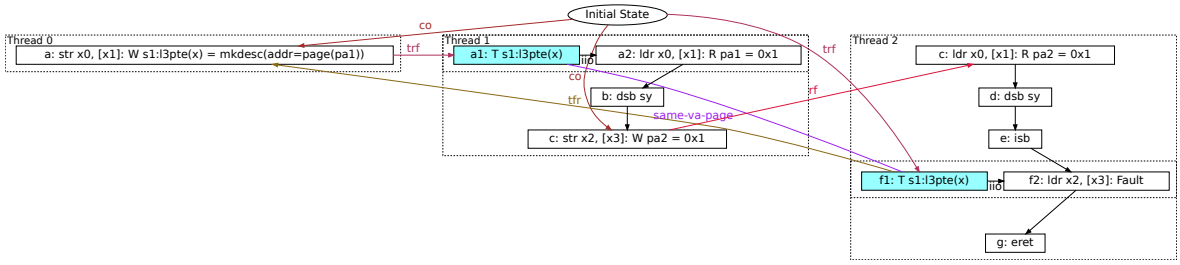
A.8.1.2 Test: WRC.TRTf.inv+dsb+dsb-isb forbid

In this WRC-shaped test, Thread 0 writes a new mapping before Thread 1 translates using that entry. Thread 1 then messages Thread 2, which then tries to translate the same location that Thread 1 did. If Thread 2 were allowed to see a translation fault, then this would be a kind of *non*-multi-copy atomic behaviour.

Multi-copy atomicity would forbid this, requiring that translation-table walks are multi-copy atomic reads of flat memory.

AArch64 WRC.TRTf.inv+dsb+dsb−isb

Page table setup: **physical** pa1 pa2; x ↦ invalid; x ?-> pa1; z ↦ pa1; *pa1 = 1; y ↦ pa2; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;	Initial state: 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0 0:R0=**desc3**(z,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R2=0b1 1:R3=y 1:VBAR_EL1=0x1000 2:PSTATE.EL=0b00 2:PSTATE.SP=0b0 2:R1=y 2:R3=x 2:VBAR_EL1=0x2000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **DSB** SY **STR** X2,[X3]
	Thread 2
	LDR X0,[X1] **DSB** SY **ISB** **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	thread2 el1 handler
	0x2400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 2:R0=1 & 2:R2=0



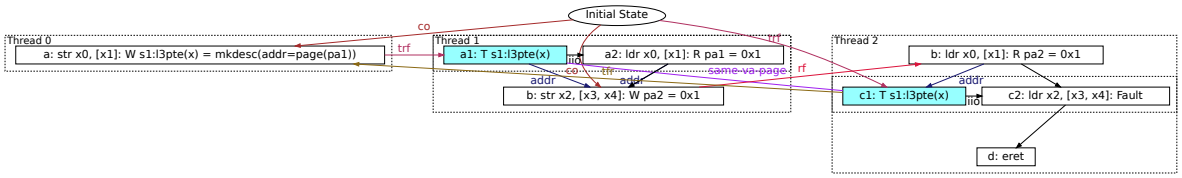
Model	Result
Base	`WRC.TRTf.inv+dsb+dsb-isb forbidden (0 of 4) 9146ms`
ETS	`WRC.TRTf.inv+dsb+dsb-isb forbidden (0 of 4) 16960ms`

A.8.1.3 Test: WRC.TRTf.inv+addrs forbid

The address-dependency into the instruction which yields a translation fault ensures that the translation-table-walk happens after the address is determined, and so the fault is ordered-after the read which its address depends on.

AArch64 WRC.TRTf.inv+addrs

Page table setup: **physical** pa1 pa2; x ↦ invalid; x ?-> pa1; z ↦ pa1; *pa1 = 1; y ↦ pa2; **identity** 0x1000 **with code**; **identity** 0x2000 **with code**;	Initial state: 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0 0:R0=**desc3**(z,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R2=0b1 1:R3=y 1:VBAR_EL1=0x1000 2:PSTATE.EL=0b00 2:PSTATE.SP=0b0 2:R1=y 2:R3=x 2:VBAR_EL1=0x2000
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **EOR** X4,X0,X0 **STR** X2,[X3,X4]
	Thread 2
	LDR X0,[X1] **EOR** X4,X0,X0 **LDR** X2,[X3,X4]
	thread1 el1 handler
	0x1400: **MOV** X0,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	thread2 el1 handler
	0x2400: **MOV** X2,#0 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 2:R0=1 & 2:R2=0



Model	Result
Base	`WRC.TRTf.inv+addrs forbidden (0 of 4) 10005ms`
ETS	`WRC.TRTf.inv+addrs forbidden (0 of 4) 11073ms`

A.8.1.4 Test: WRC.TRTf.inv+dsbs allow (unless ETS, then forbid)

AArch64 WRC.TRTf.inv+dsbs

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DSB SY
STR X2,[X3]
```

Thread 2

```
LDR X0,[X1]
DSB SY
LDR X2,[X3]
```

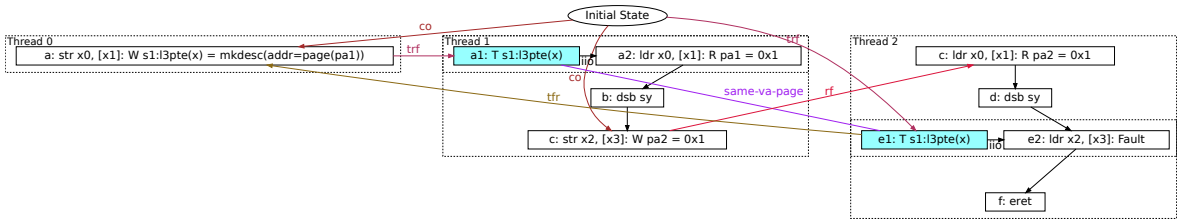
thread1 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.TRTf.inv+dsbs allowed (1 of 4) 48023ms`
ETS	`WRC.TRTf.inv+dsbs forbidden (0 of 4) 8487ms`

A.8.1.5 Test: WRC.TRTf.inv+dmbs allow (unless ETS, then forbid)

AArch64 WRC.TRTf.inv+dmbs

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 2

```
LDR X0,[X1]
DMB SY
LDR X2,[X3]
```

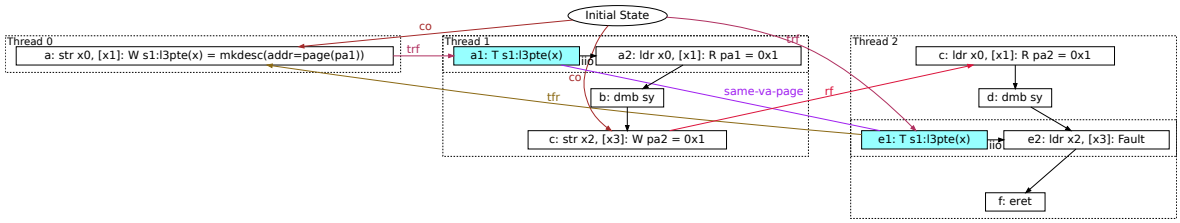
thread1 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.TRTf.inv+dmbs allowed (1 of 4) 35492ms`
ETS	`WRC.TRTf.inv+dmbs forbidden (0 of 4) 8559ms`

A.8.1.6 Test: WRC.TRTf.inv+pos allow

AArch64 WRC.TRTf.inv+pos

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

Thread 2

```
LDR X0,[X1]
LDR X2,[X3]
```

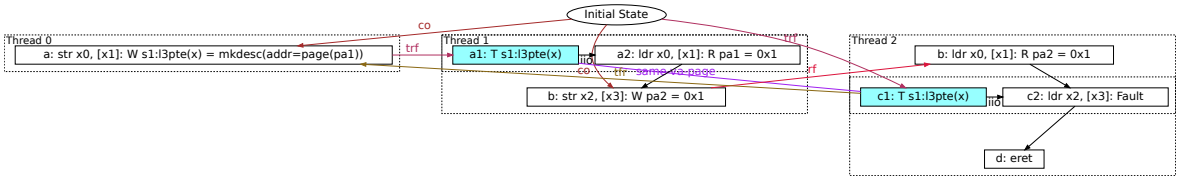
thread1 el1 handler

```
0x1400:
MOV X0,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.TRTf.inv+pos allowed (1 of 4) 26642ms`
ETS	`WRC.TRTf.inv+pos forbidden (0 of 4) 9414ms`

A.8.1.7 Test: WRC.TTTf.inv+addrs forbid

AArch64 WRC.TTTf.inv+addrs

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 1;
*pa2 = 1;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=mkdesc3(oa=pa2)
1:R3=pte3(y,page_table_base)
1:VBAR_EL1=0x1000
1:__isla_monomorphize_writes=true
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
MOV X0,#0
LDR X0,[X1]
EOR X4,X0,X0
STR X2,[X3,X4]
```

Thread 2

```
MOV X0,#0
LDR X0,[X1]
EOR X4,X0,X0
MOV X2,#0
LDR X2,[X3,X4]
```

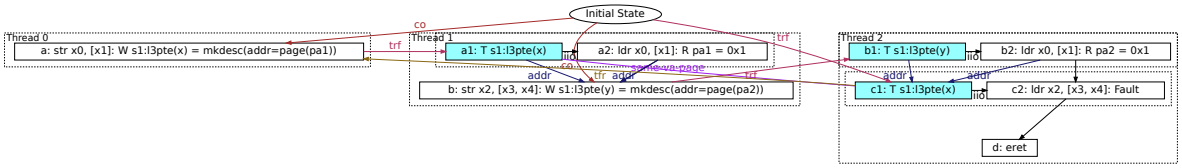
thread1 el1 handler

```
0x1400:
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread2 el1 handler

```
0x2400:
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.TTTf.inv+addrs forbidden (0 of 8) 31851ms`
ETS	`WRC.TTTf.inv+addrs forbidden (0 of 8) 23959ms`

A.8.1.8 Test: WRC.TTTf.inv+data+addr forbid

AArch64 WRC.TTTf.inv+data+addr

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;
x ?-> pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 1;
*pa2 = 1;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=mkdesc3(oa=pa2)
1:R3=pte3(y,page_table_base)
1:VBAR_EL1=0x1000
1:__isla_monomorphize_writes=true
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
MOV X0,#0
LDR X0,[X1]
EOR X4,X0,X0
ORR X2,X2,X4
STR X2,[X3]
```

Thread 2

```
MOV X0,#0
LDR X0,[X1]
EOR X4,X0,X0
MOV X2,#0
LDR X2,[X3,X4]
```

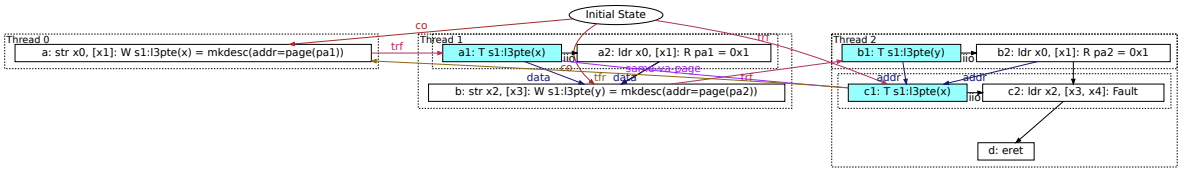
thread1 el1 handler

```
0x1400:
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread2 el1 handler

```
0x2400:
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.TTTf.inv+data+addr forbidden (0 of 8) 49883ms`
ETS	`WRC.TTTf.inv+data+addr forbidden (0 of 8) 40834ms`

WRC.RRTf.inv-shaped tests

A.8.1.9 Test: WRC.RRTf.inv+dsb+dsb-isb forbid

This test is like the previous one, except, instead of loading the unmapped location in Thread 1 (therefore reading from the entry during the translation-table walk), it loads the entry itself directly.

Arm also forbid this test, as the load in Thread 1 will ensure that the write is visible to the translation-table-walk that would be performed if Thread 2 had a translation-fault.

AArch64 WRC.RRTf.inv+dsb+dsb−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=pte3(x,page_table_base)
1:R2=0b1
1:R3=y
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DSB SY
STR X2,[X3]
```

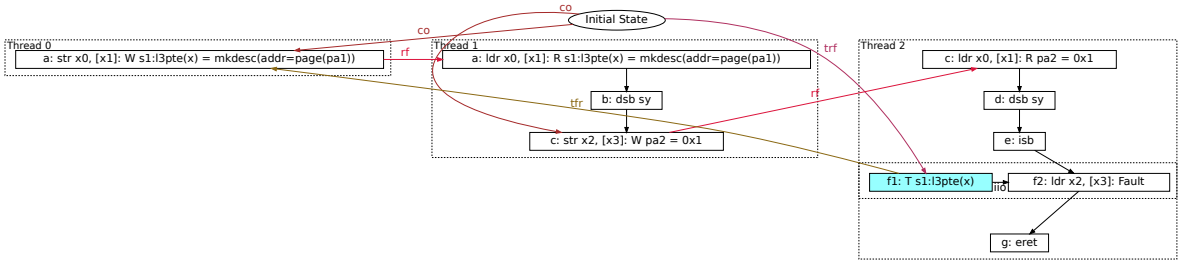
Thread 2

```
LDR X0,[X1]
DSB SY
ISB
LDR X2,[X3]
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=desc3(z,page_table_base) & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.RRTf.inv+dsb+dsb-isb forbidden (0 of 2) 5393ms`
ETS	`WRC.RRTf.inv+dsb+dsb-isb forbidden (0 of 2) 5591ms`

A.8.1.10 Test: WRC.RRTf.inv+dsb+ctrl-isb forbid

AArch64 WRC.RRTf.inv+dsb+ctrl−isb

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=pte3(x,page_table_base)
1:R2=0b1
1:R3=y
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DSB SY
STR X2,[X3]
```

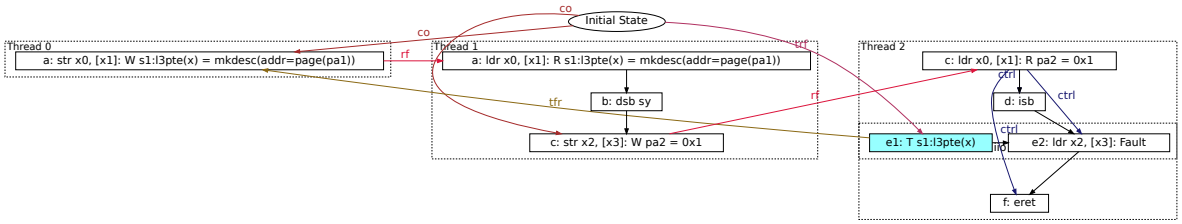
Thread 2

```
LDR X0,[X1]
CBNZ X0,L0
L0:
ISB
LDR X2,[X3]
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=desc3(z,page_table_base) & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.RRTf.inv+dsb+ctrl-isb forbidden (0 of 4) 5218ms`
ETS	`WRC.RRTf.inv+dsb+ctrl-isb forbidden (0 of 4) 5459ms`

A.8.1.11 Test: WRC.RRTf.inv+dsbs allow (unless ETS, then forbid)

AArch64 WRC.RRTf.inv+dsbs

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=pte3(x,page_table_base)
1:R2=0b1
1:R3=y
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DSB SY
STR X2,[X3]
```

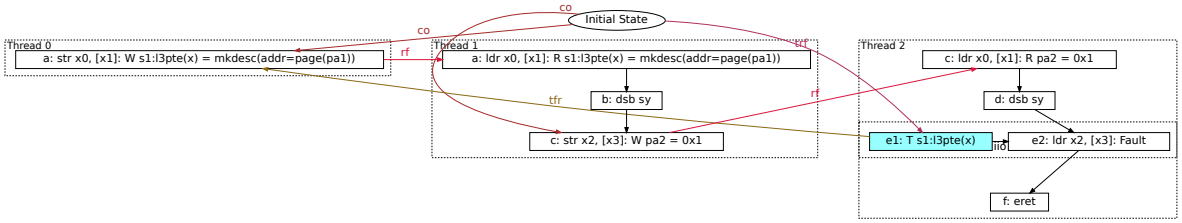
Thread 2

```
LDR X0,[X1]
DSB SY
LDR X2,[X3]
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=desc3(z,page_table_base) & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.RRTf.inv+dsbs allowed (1 of 2) 8964ms`
ETS	`WRC.RRTf.inv+dsbs forbidden (0 of 2) 4568ms`

A.8.1.12 Test: WRC.RRTf.inv+dmbs allow (unless ETS, then forbid)

AArch64 WRC.RRTf.inv+dmbs

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=pte3(x,page_table_base)
1:R2=0b1
1:R3=y
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DMB SY
STR X2,[X3]
```

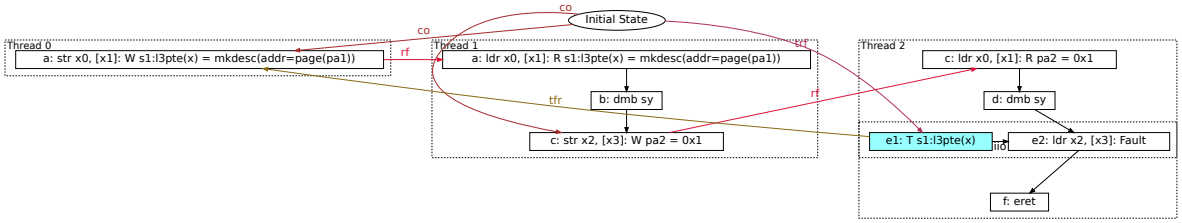
Thread 2

```
LDR X0,[X1]
DMB SY
LDR X2,[X3]
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=desc3(z,page_table_base) & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.RRTf.inv+dmbs allowed (1 of 2) 6441ms`
ETS	`WRC.RRTf.inv+dmbs forbidden (0 of 2) 3589ms`

A.8.1.13 Test: WRC.RRTf.inv+pos allow

AArch64 WRC.RRTf.inv+pos

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=pte3(x,page_table_base)
1:R2=0b1
1:R3=y
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

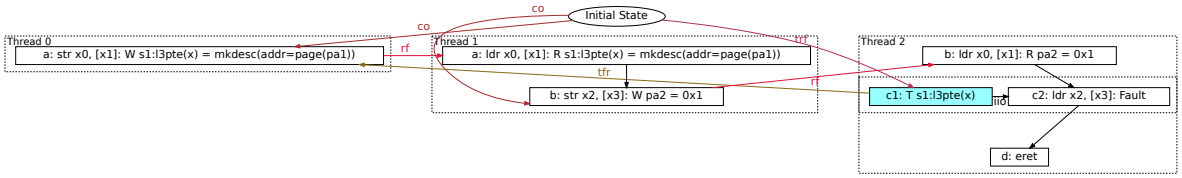
Thread 2

```
LDR X0,[X1]
LDR X2,[X3]
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=desc3(z,page_table_base) & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.RRTf.inv+pos allowed (1 of 2) 5334ms`
ETS	`WRC.RRTf.inv+pos allowed (1 of 2) 5239ms`

A.8.1.14 Test: WRC.RRTf.inv+addrs forbid

AArch64 WRC.RRTf.inv+addrs

Page table setup:

```
physical pa1 pa2;
x ↦ invalid;

x ?-> pa1;
z ↦ pa1;
*pa1 = 1;

y ↦ pa2;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=pte3(x,page_table_base)
1:R2=0b1
1:R3=y
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=x
2:VBAR_EL1=0x2000
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
EOR X4,X0,X0
STR X2,[X3,X4]
```

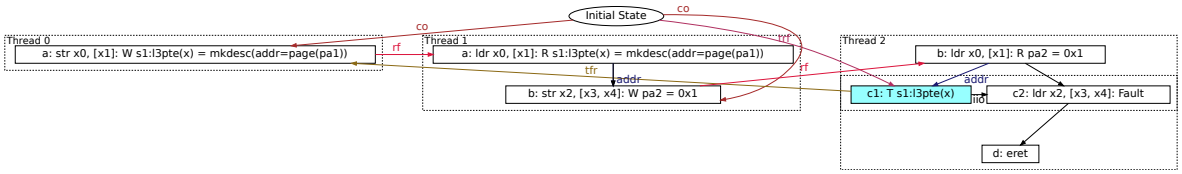
Thread 2

```
LDR X0,[X1]
EOR X4,X0,X0
LDR X2,[X3,X4]
```

thread2 el1 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=desc3(z,page_table_base) & 2:R0=1 & 2:R2=0`



Model	Result
Base	`WRC.RRTf.inv+addrs forbidden (0 of 2) 4100ms`
ETS	`WRC.RRTf.inv+addrs forbidden (0 of 2) 4411ms`

WRC.TfRR-shaped tests

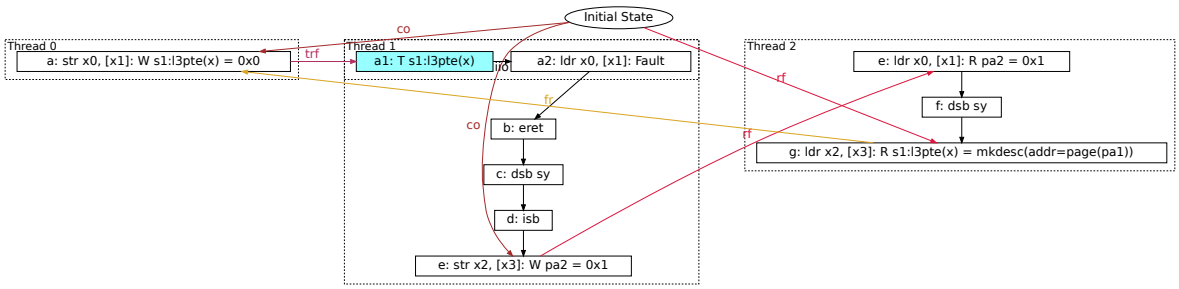
A.8.1.15 Test: WRC.TfRR+dsb-isb+dsb forbid

This is like the previous tests, except that, here, Thread 1 loads the unmapped address and suffers a translation-fault. Can Thread 2 load the entry and read-from a write before the break?

As before, Arm forbid this, enforcing a kind of multi-copy atomicity for translation-table-walks.

AArch64 WRC.TfRR+dsb−isb+dsb

Page table setup: **physical** pa1 pa2; x ↦ pa1; x ?-> invalid; *pa1 = 0; y ↦ pa2; **identity** 0x1000 **with code**;	Initial state: 0:PSTATE.EL=0b00 0:PSTATE.SP=0b0 0:R0=**desc3**(z,page_table_base) 0:R1=**pte3**(x,page_table_base) 1:PSTATE.EL=0b00 1:PSTATE.SP=0b0 1:R1=x 1:R2=0b1 1:R3=y 1:VBAR_EL1=0x1000 2:PSTATE.EL=0b00 2:PSTATE.SP=0b0 2:R1=y 2:R3=**pte3**(x,page_table_base)
	Thread 0
	STR X0,[X1]
	Thread 1
	LDR X0,[X1] **DSB** SY **ISB** **STR** X2,[X3]
	Thread 2
	LDR X0,[X1] **DSB** SY **LDR** X2,[X3]
	thread1 el1 handler
	0x1400: **MOV** X0,#1 **MRS** X13,ELR_EL1 **ADD** X13,X13,#4 **MSR** ELR_EL1,X13 **ERET**
	Final state: 1:R0=1 & 2:R0=1 & ~2:R2=0



Model	Result
Base	`WRC.TfRR+dsb-isb+dsb forbidden (0 of 2) 5353ms`
ETS	`WRC.TfRR+dsb-isb+dsb forbidden (0 of 2) 5963ms`

A.8.1.16 Test: WRC.TfRR+ctrl-isb+dsb forbid

AArch64 WRC.TfRR+ctrl−isb+dsb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

*pa1 = 0;

y ↦ pa2;

identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=pte3(x,page_table_base)
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
CBNZ X0,L0
L0:
ISB
STR X2,[X3]
```

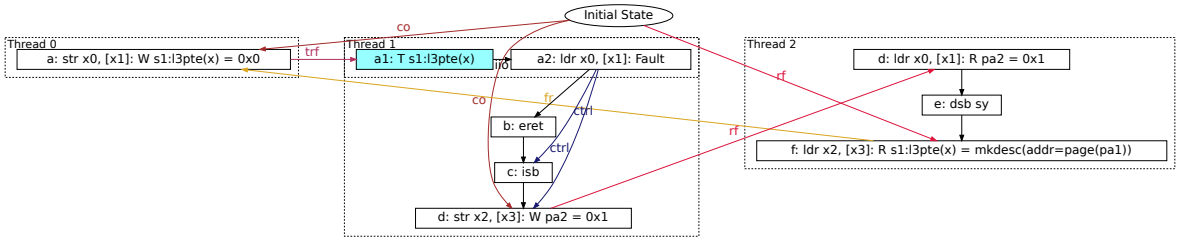
Thread 2

```
LDR X0,[X1]
DSB SY
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X0,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & ~2:R2=0`



Model	Result
Base	`WRC.TfRR+ctrl-isb+dsb forbidden (0 of 3) 5699ms`
ETS	`WRC.TfRR+ctrl-isb+dsb forbidden (0 of 3) 6787ms`

A.8.1.17 Test: WRC.TfRR+dsbs forbid

AArch64 WRC.TfRR+dsbs

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

*pa1 = 0;

y ↦ pa2;

identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=pte3(x,page_table_base)
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
DSB SY
STR X2,[X3]
```

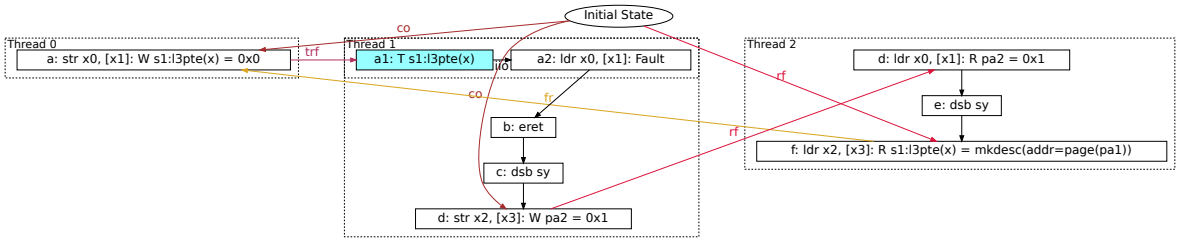
Thread 2

```
LDR X0,[X1]
DSB SY
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X0,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & ~2:R2=0`



Model	Result
Base	`WRC.TfRR+dsbs forbidden (0 of 2) 3850ms`
ETS	`WRC.TfRR+dsbs forbidden (0 of 2) 5423ms`

A.8.1.18 Test: WRC.TfRR+po+dsb forbid

Note that the translation-fault to store ordering is preserved (See CoTfW.inv+po).

AArch64 WRC.TfRR+po+dsb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

*pa1 = 0;

y ↦ pa2;

identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=pte3(x,page_table_base)
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

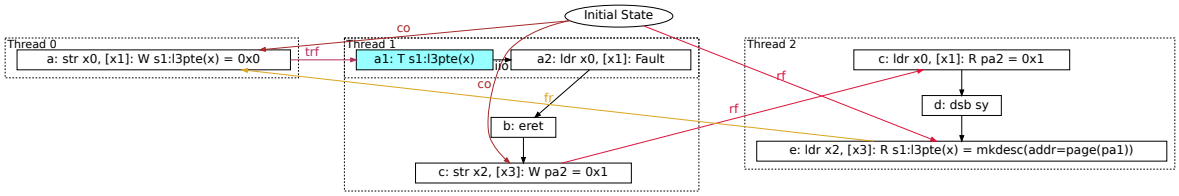
Thread 2

```
LDR X0,[X1]
DSB SY
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X0,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & ~2:R2=0`



Model	Result
Base	`WRC.TfRR+po+dsb forbidden (0 of 2) 3508ms`
ETS	`WRC.TfRR+po+dsb forbidden (0 of 2) 3727ms`

A.8.1.19 Test: WRC.TfRR+pos allow

AArch64 WRC.TfRR+pos

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;
x ?-> invalid;

*pa1 = 0;

y ↦ pa2;

identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b00
0:PSTATE.SP=0b0
0:R0=desc3(z,page_table_base)
0:R1=pte3(x,page_table_base)
1:PSTATE.EL=0b00
1:PSTATE.SP=0b0
1:R1=x
1:R2=0b1
1:R3=y
1:VBAR_EL1=0x1000
2:PSTATE.EL=0b00
2:PSTATE.SP=0b0
2:R1=y
2:R3=pte3(x,page_table_base)
```

Thread 0

```
STR X0,[X1]
```

Thread 1

```
LDR X0,[X1]
STR X2,[X3]
```

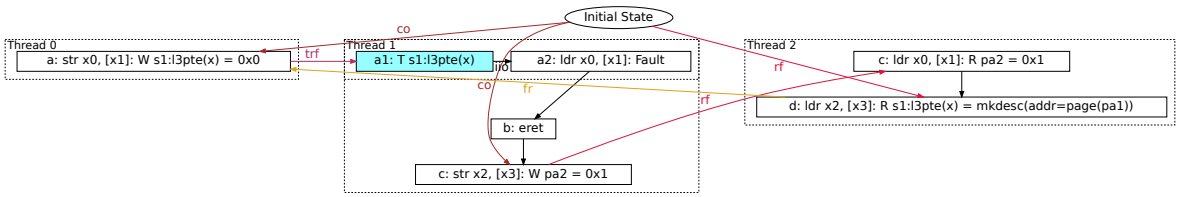
Thread 2

```
LDR X0,[X1]
LDR X2,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X0,#1
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `1:R0=1 & 2:R0=1 & ~2:R2=0`



Model	Result
Base	`WRC.TfRR+pos allowed (1 of 2) 5616ms`
ETS	`WRC.TfRR+pos allowed (1 of 2) 7637ms`

A.9 Multi-address-space support with ASIDs

To support systems software with multiple address spaces, such as operating systems with many concurrently executing processes, Arm provide two features that allow the hardware and software to manage the translation tables for these processes more effectively:

§A.9.1 A `TTBR` ("Translation Table Base Register"), which can be changed to point to a new translation table.

§A.9.2 ASIDs ("Address Space Identifiers"), which are used to tag TLB entries with their process/address space, to reduce TLB maintenance burden.

A.9.1 TTBRs

The translation tables are stored in normal memory in a hierarchical tree structure. In order for the processor to know where the root of this tree is, it reads a register called the `TTBR` (or "Translation-Table Base Register"). Each *translation regime* has its own base register:

- `TTBR0_EL1`: for Stage 1 translations in the 'low' (positive) portion of the address map, from EL1&0.
- `TTBR1_EL1`: for Stage 1 translations in the 'high' (negative) portion of the address map, from EL1&0.
- `TTBR0_EL2`: for Stage 1 translations from EL2.
- `VTTBR_EL2`: for Stage 2 translations from accesses from EL1&0.

A.9.2 ASIDs

A.9.2.1 Test: CoWinvTa1.1+dsb-tlbiasidis-dsb-eret forbid

In this test, a virtual address is unmapped, and only ASID #1 is cleaned; since the thread uses that ASID, the TLB invalidation affects all translations in that thread, and so the final outcome is forbidden.

AArch64 CoWinvTa1.1+dsb−tlbiasidis−dsb−eret

Page table setup:
`physical pa1;`
`x ↦ pa1;`
`x ?-> invalid;`
 	
`identity 0x1000 with code;`
	Initial state:
	`ELR_EL1=L0:`
	`PSTATE.EL=0b01`
	`R0=0b0`
	`R1=pte3(x,page_table_base)`
	`R3=x`
	`R4=asid(0x1)`
	`SPSR_EL1=0b00000`
	`TTBR0_EL1=ttbr(asid=0x0001,base=page_table_base)`
	`VBAR_EL1=0x1000`

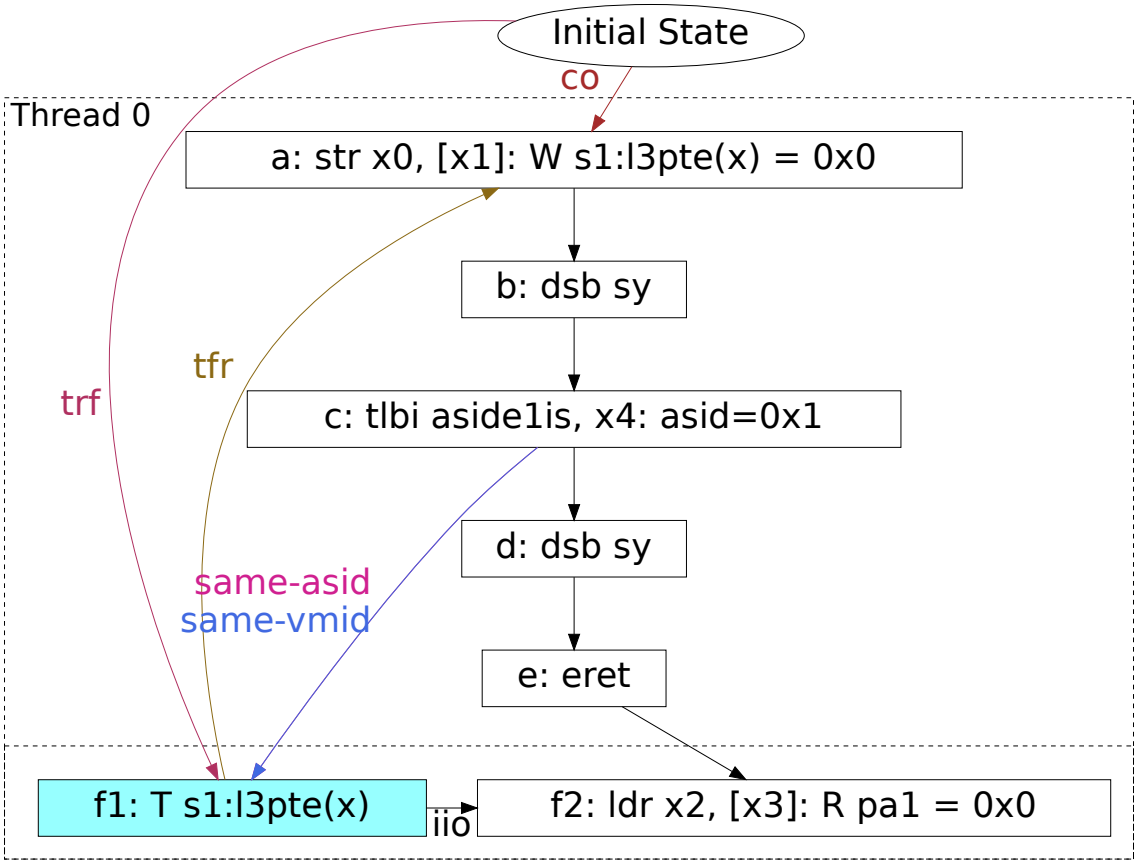
	Thread 0

	`STR X0,[X1]`
	`DSB SY`
	`TLBI ASIDE1IS,X4`
	`DSB SY`
	`ERET`
	`L0:`
	`LDR X2,[X3]`

	thread0 el1 handler

	`0x1400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `0:R2=0`



Model	Result
Base	`CoWinvTa1.1+dsb-tlbiasidis-dsb-eret forbidden (0 of 2) 7638ms`
ETS	`CoWinvTa1.1+dsb-tlbiasidis-dsb-eret forbidden (0 of 2) 99884ms`

A.9.2.2 Test: CoWinvTa2.1+dsb-tlbiasidis-dsb-eret allow

Same as previous, but invalidating the 'wrong' ASID.

AArch64 CoWinvTa2.1+dsb−tlbiasidis−dsb−eret

Page table setup:
`physical pa1;`
`x ↦ pa1;`
`x ?-> invalid;`
 	
`identity 0x1000 with code;`
	Initial state:
	`ELR_EL1=L0:`
	`PSTATE.EL=0b01`
	`R0=0b0`
	`R1=pte3(x,page_table_base)`
	`R3=x`
	`R4=asid(0x2)`
	`SPSR_EL1=0b00000`
	`TTBR0_EL1=ttbr(base=page_table_base,asid=0x0001)`
	`VBAR_EL1=0x1000`

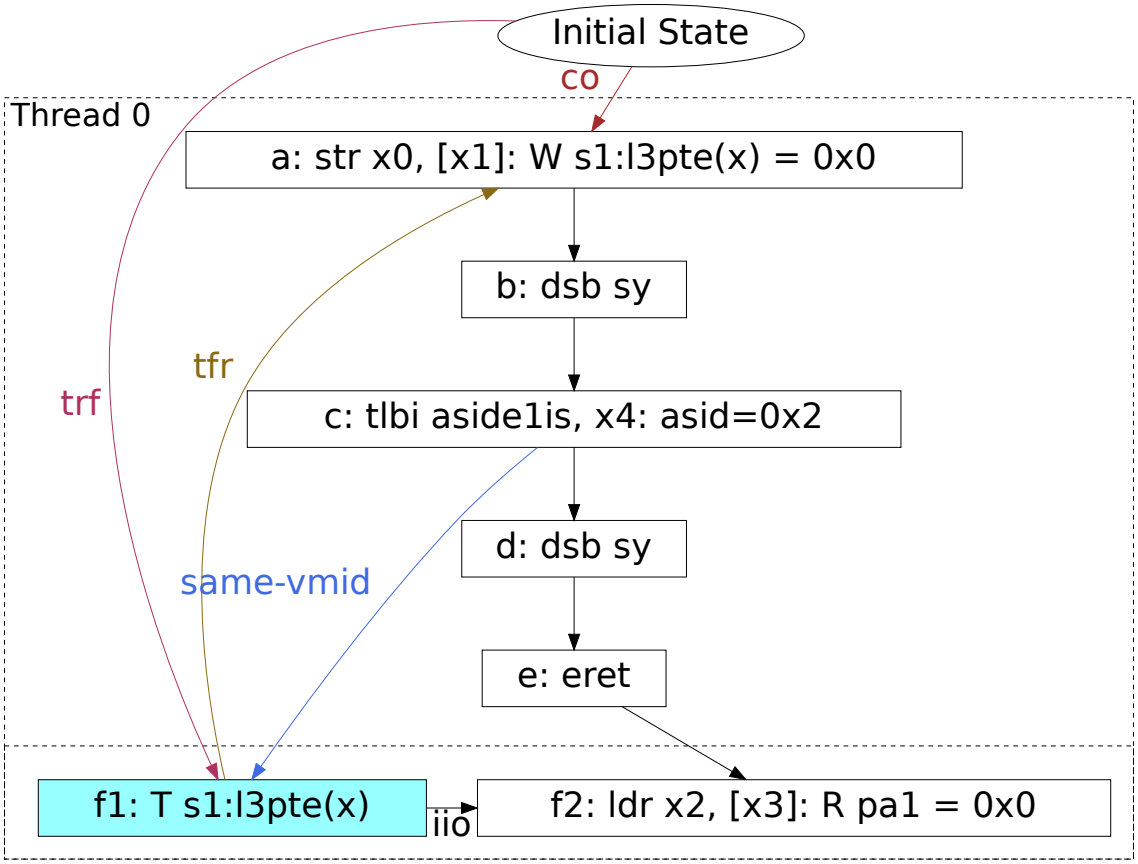
	Thread 0

	`STR X0,[X1]`
	`DSB SY`
	`TLBI ASIDE1IS,X4`
	`DSB SY`
	`ERET`
	`L0:`
	`LDR X2,[X3]`

	thread0 el1 handler

	`0x1400:`
	`MOV X2,#1`
	`MRS X13,ELR_EL1`
	`ADD X13,X13,#4`
	`MSR ELR_EL1,X13`
	`ERET`

	Final state: `0:R2=0`



Model	Result
Base	`CoWinvTa2.1+dsb-tlbiasidis-dsb-eret allowed (1 of 2) 55920ms`
ETS	`CoWinvTa2.1+dsb-tlbiasidis-dsb-eret allowed (1 of 2) 9894ms`

A.10 Additional tests, as-yet unsorted

A.10.0.1 Test: MP.RT.inv+dmb+addr-po-msr-isb forbid

This exercises the ctxob edges via
`speculative ; [MSR]` +
`[ContextChange] ; po ; [CSE]` +
`[CSE] ; instruction-order`

AArch64 MP.RT.inv+dmb+addr−po−msr−isb

Page table setup:

```
physical pa1 pa2 pa3;

x ↦ invalid;
x ?-> pa1;

y ↦ pa2;

z ↦ pa3;

*pa1 = 1;
*pa2 = 0;
*pa3 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b01
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:R5=0x1000
1:R8=z
1:VBAR_EL1=0x1000
1:VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 1

```
LDR X2,[X1]
EOR X6,X2,X2
LDR X7,[X8,X6]
MSR ELR_EL1,X5
ISB
LDR X4,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X4,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X4,#0
MRS X13,ELR_EL2
ADD X13,X13,#4
MSR ELR_EL2,X13
ERET
```

Final state: `1:R2=1 & 1:R4=0`

Model	Result
Base	`MP.RT.inv+dmb+addr-po-msr-isb forbidden (0 of 2) 4687ms`
ETS	`MP.RT.inv+dmb+addr-po-msr-isb forbidden (0 of 2) 3356ms`

A.10.0.2 Test: MP.RT.inv+dmb+addr-po-isb allow

AArch64 MP.RT.inv+dmb+addr−po−isb

Page table setup:

```
physical pa1 pa2 pa3;

x ↦ invalid;
x ?-> pa1;

y ↦ pa2;

z ↦ pa3;

*pa1 = 1;
*pa2 = 0;
*pa3 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R0=mkdesc3(oa=pa1)
0:R1=pte3(x,page_table_base)
0:R2=0b1
0:R3=y
1:PSTATE.EL=0b01
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:R8=z
1:VBAR_EL1=0x1000
1:VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 1

```
LDR X2,[X1]
EOR X6,X2,X2
LDR X7,[X8,X6]
ISB
LDR X4,[X3]
```

thread1 el1 handler

```
0x1400:
MOV X4,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X4,#0
MRS X13,ELR_EL2
ADD X13,X13,#4
MSR ELR_EL2,X13
ERET
```

Final state: `1:R2=1 & 1:R4=0`

Model	Result
Base	`MP.RT.inv+dmb+addr-po-isb forbidden (0 of 2) 3366ms`
ETS	`MP.RT.inv+dmb+addr-po-isb forbidden (0 of 2) 3223ms`

A.10.0.3 Test: MP.TR.inv+dmb+msr allow

AArch64 MP.TR.inv+dmb+msr

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 0;
*pa2 = 1;
identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R0=0b1
0:R1=x
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
1:PSTATE.EL=0b01
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:R5=0x1000
1:VBAR_EL1=0x1000
1:VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 1

```
LDR X2,[X1]
MSR ELR_EL1,X5
LDR X4,[X3]
```

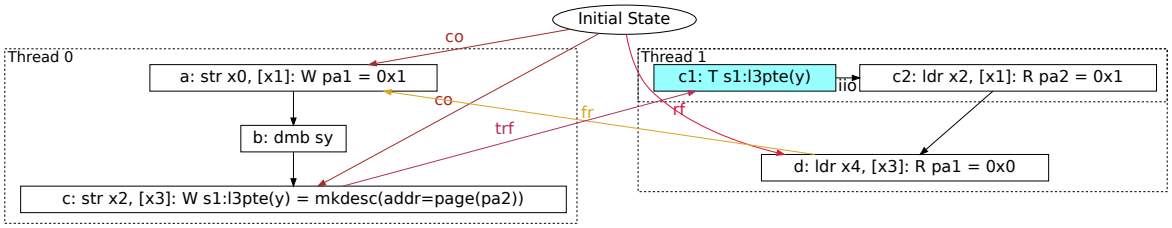
thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL2
ADD X13,X13,#4
MSR ELR_EL2,X13
ERET
```

Final state: `1:R2=1 & 1:R4=0`



Model	Result
Base	`MP.TR.inv+dmb+msr allowed (1 of 2) 6174ms`
ETS	`MP.TR.inv+dmb+msr allowed (1 of 2) 6987ms`

A.10.0.4 Test: MP.TR.inv+dmb+isb allow

AArch64 MP.TR.inv+dmb+isb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 0;
*pa2 = 1;
identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R0=0b1
0:R1=x
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
1:PSTATE.EL=0b01
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:VBAR_EL1=0x1000
1:VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 1

```
LDR X2,[X1]
ISB
LDR X4,[X3]
```

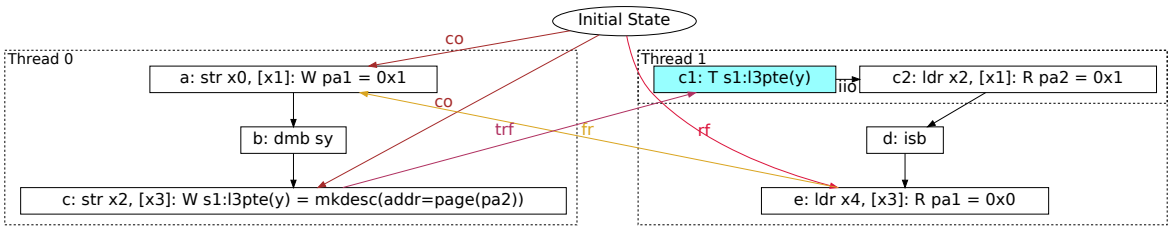
thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL2
ADD X13,X13,#4
MSR ELR_EL2,X13
ERET
```

Final state: `1:R2=1 & 1:R4=0`



Model	Result
Base	`MP.TR.inv+dmb+isb forbidden (0 of 2) 2406ms`
ETS	`MP.TR.inv+dmb+isb forbidden (0 of 2) 3016ms`

A.10.0.5 Test: MP.TR.inv+dmb+msr-isb forbid

AArch64 MP.TR.inv+dmb+msr−isb

Page table setup:

```
physical pa1 pa2;
x ↦ pa1;

y ↦ invalid;
y ?-> pa2;

*pa1 = 0;
*pa2 = 1;
identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
0:R0=0b1
0:R1=x
0:R2=mkdesc3(oa=pa2)
0:R3=pte3(y,page_table_base)
1:PSTATE.EL=0b01
1:PSTATE.SP=0b0
1:R1=y
1:R3=x
1:R5=0x1000
1:VBAR_EL1=0x1000
1:VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DMB SY
STR X2,[X3]
```

Thread 1

```
LDR X2,[X1]
MSR ELR_EL1,X5
ISB
LDR X4,[X3]
```

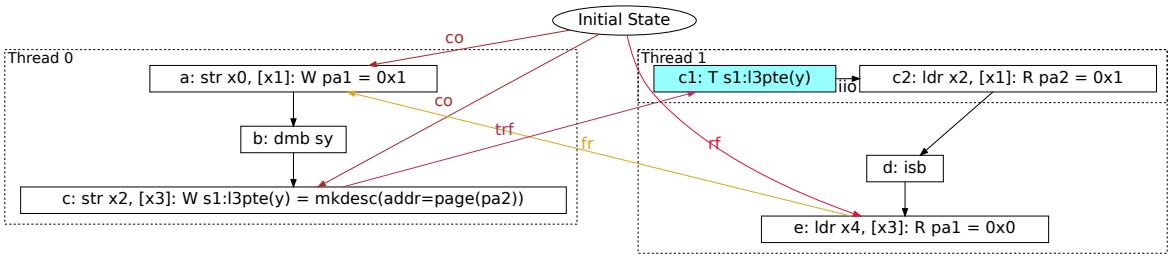
thread1 el1 handler

```
0x1400:
MOV X2,#0
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

thread1 el2 handler

```
0x2400:
MOV X2,#0
MRS X13,ELR_EL2
ADD X13,X13,#4
MSR ELR_EL2,X13
ERET
```

Final state: `1:R2=1 & 1:R4=0`



Model	Result
Base	`MP.TR.inv+dmb+msr-isb forbidden (0 of 2) 4042ms`
ETS	`MP.TR.inv+dmb+msr-isb forbidden (0 of 2) 4240ms`

A.10.0.6 Test: SwitchTable.different-asid+eret forbid

If the page the page table root is changed, together with an unused ASID, then a new translation has to read-from a page table entry from the new page table.

Model	Result
Base	no result for SwitchTable.different-asid+eret
ETS	no result for SwitchTable.different-asid+eret

A.10.0.7 Test: SwitchTable.same-asid+eret allow

If the page the page table root is changed, together with an already-used ASID, then a new translation can read-from a page table entry from the old page table.

Model	Result
Base	no result for SwitchTable.same-asid+eret
ETS	no result for SwitchTable.same-asid+eret

A.10.0.8 Test: WDS+po-dsb-tlbiipa-dsb-tlbiis-dsb-eret forbid

Write to Different Stages.

If two stages of translation are updated, then both stages need to be invalidated, in the right order, to be guaranteed to see the new mapping.

AArch64 WDS+po−dsb−tlbiipa−dsb−tlbiis−dsb−eret

Page table setup:

```
physical pa1;
intermediate ipa1;

x ↦ ipa1;
x ?-> invalid;

ipa1 ↦ pa1;
ipa1 ?-> invalid;

*pa1 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
ELR_EL2=L0:
PSTATE.EL=0b10
R0=0b0
R1=pte3(ipa1,s2_page_table_base)
R2=0b0
R3=pte3(x,page_table_base)
R5=x
R6=page(x)
SPSR_EL2=0b00100
VBAR_EL1=0x1000
VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
STR X2,[X3]
DSB SY
TLBI IPAS2E1,X6
DSB SY
TLBI VMALLE1IS
DSB SY
ERET
L0:
LDR X4,[X5]
```

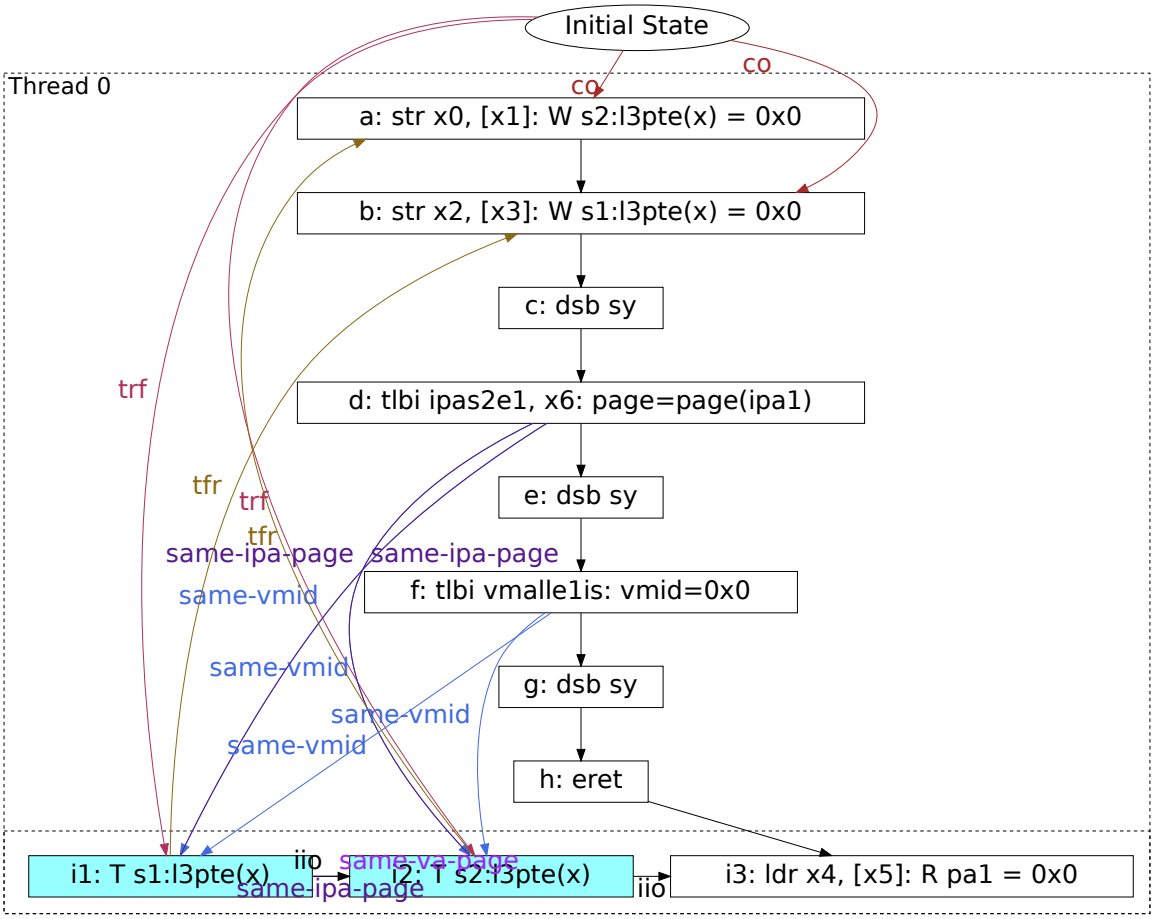
thread0 el1 handler

```
0x1200:
MOV X4,#1
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
```

thread0 el2 handler lower exc

```
0x2400:
MOV X4,#2
MRS X20,ELR_EL2
ADD X20,X20,#4
MSR ELR_EL2,X20
ERET
```

Final state: `0:R4=0`



Model	Result
Base	`WDS+po-dsb-tlbiipa-dsb-tlbiis-dsb-eret forbidden (0 of 6) 186945ms`
ETS	`WDS+po-dsb-tlbiipa-dsb-tlbiis-dsb-eret forbidden (0 of 6) 576750ms`

A.10.0.9 Test: WDS+po-dsb-tlbiipa-dsb-eret allow

AArch64 WDS+po−dsb−tlbiipa−dsb−eret

Page table setup:

```
physical pa1;
intermediate ipa1;

x ↦ ipa1;
x ?-> invalid;

ipa1 ↦ pa1;
ipa1 ?-> invalid;

*pa1 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
ELR_EL2=L0:
PSTATE.EL=0b10
R0=0b0
R1=pte3(ipa1,s2_page_table_base)
R2=0b0
R3=pte3(x,page_table_base)
R5=x
R6=page(x)
SPSR_EL2=0b00100
VBAR_EL1=0x1000
VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
STR X2,[X3]
DSB SY
TLBI IPAS2E1,X6
DSB SY
ERET
L0:
LDR X4,[X5]
```

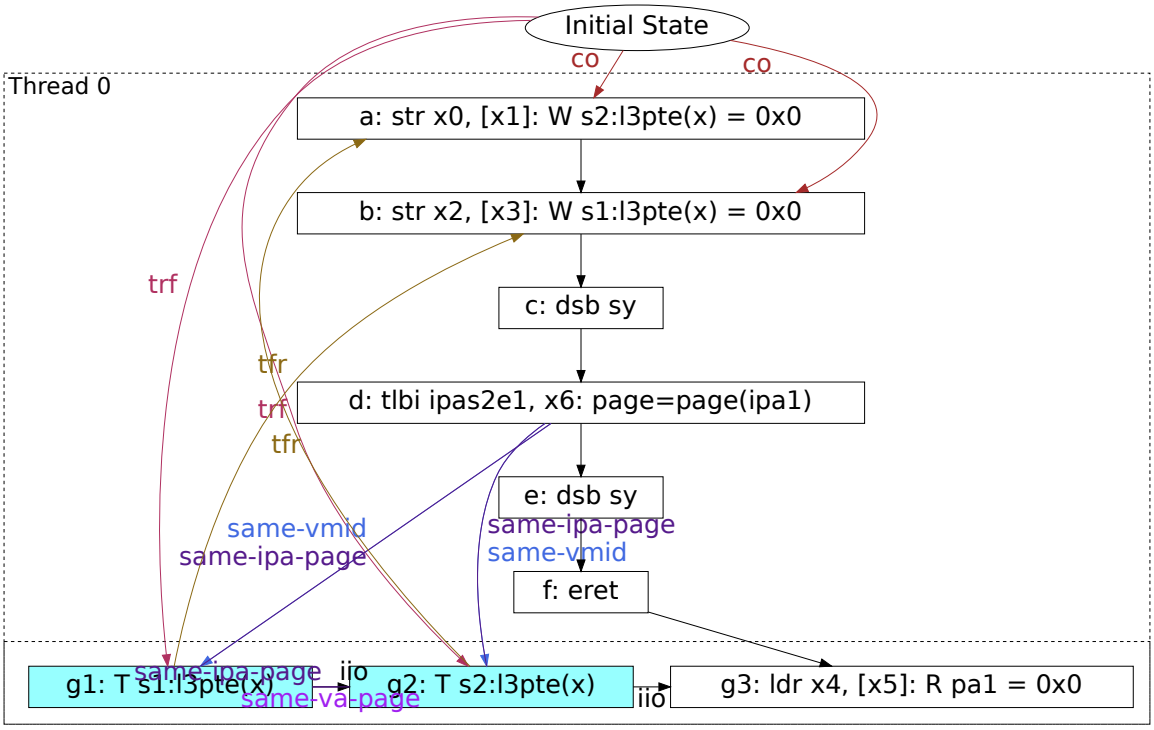
thread0 el1 handler

```
0x1200:
MOV X4,#1
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
```

thread0 el2 handler lower exc

```
0x2400:
MOV X4,#2
MRS X20,ELR_EL2
ADD X20,X20,#4
MSR ELR_EL2,X20
ERET
```

Final state: `0:R4=0`



Model	Result
Base	`WDS+po-dsb-tlbiipa-dsb-eret allowed (1 of 6) 267478ms`
ETS	`WDS+po-dsb-tlbiipa-dsb-eret allowed (1 of 6) 25136ms`

A.10.0.10 Test: WDS+dsb-tlbiipa-dsb-eret-po

AArch64 WDS+dsb−tlbiipa−dsb−eret−po

Page table setup:

```
physical pa1;
intermediate ipa1;

x ↦ ipa1;
x ?-> invalid;

ipa1 ↦ pa1;
ipa1 ?-> invalid;

*pa1 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
ELR_EL2=L0:
PSTATE.EL=0b10
R0=0b0
R1=pte3(ipa1,s2_page_table_base)
R2=0b0
R3=pte3(x,page_table_base)
R5=x
R6=page(x)
SPSR_EL2=0b00100
VBAR_EL1=0x1000
VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI IPAS2E1,X6
DSB SY
ERET
L0:
STR X2,[X3]
LDR X4,[X5]
```

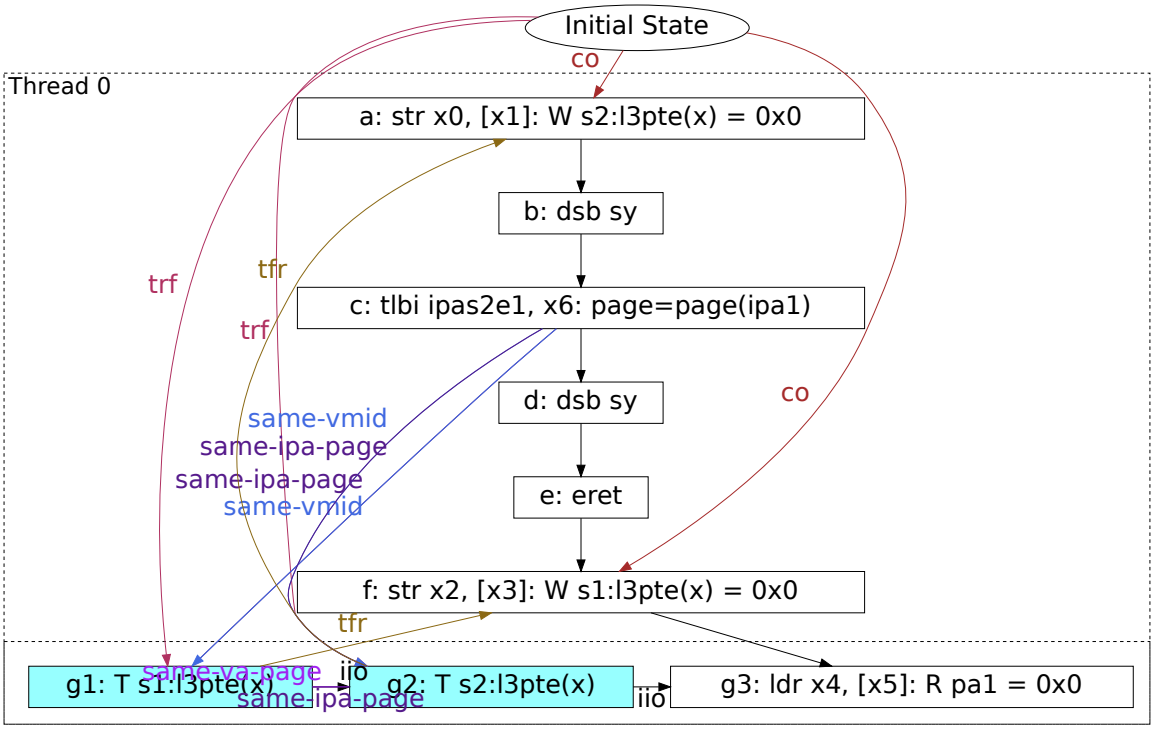
thread0 el1 handler

```
0x1200:
MOV X4,#1
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
```

thread0 el2 handler lower exc

```
0x2400:
MOV X4,#2
MRS X20,ELR_EL2
ADD X20,X20,#4
MSR ELR_EL2,X20
ERET
```

Final state: `0:R4=0`



Model	Result
Base	`WDS+dsb-tlbiipa-dsb-eret-po allowed (1 of 6) 59690ms`
ETS	`WDS+dsb-tlbiipa-dsb-eret-po allowed (1 of 6) 216640ms`

A.10.0.11 Test: WDS+dsb-tlbiipa-dsb-po-eret

AArch64 WDS+dsb−tlbiipa−dsb−po−eret

Page table setup:

```
physical pa1;
intermediate ipa1;

x ↦ invalid;
x ?-> ipa1;

ipa1 ↦ invalid;
ipa1 ?-> pa1;

*pa1 = 0;

identity 0x1000 with code;
identity 0x2000 with code;
```

Initial state:

```
ELR_EL2=L0:
PSTATE.EL=0b10
R0=mkdesc3(oa=pa1)
R1=pte3(ipa1,s2_page_table_base)
R2=mkdesc3(oa=ipa1)
R3=pte3(x,page_table_base)
R5=x
R6=page(x)
SPSR_EL2=0b00100
VBAR_EL1=0x1000
VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI IPAS2E1,X6
DSB SY
STR X2,[X3]
ERET
L0:
LDR X4,[X5]
```

thread0 el1 handler

```
0x1200:
MOV X4,#1
MRS X20,ELR_EL1
ADD X20,X20,#4
MSR ELR_EL1,X20
ERET
```

thread0 el2 handler lower exc

```
0x2400:
MOV X4,#2
MRS X20,ELR_EL2
ADD X20,X20,#4
MSR ELR_EL2,X20
ERET
```

Final state: `0:R4=2`

Model	Result
Base	`WDS+dsb-tlbiipa-dsb-po-eret forbidden (0 of 6) 61832ms`
ETS	`WDS+dsb-tlbiipa-dsb-po-eret forbidden (0 of 6) 9009ms`

A.10.0.12 Test: WBM+dsb-tlbiis-dsb forbid

AArch64 WBM+dsb−tlbiis−dsb

Page table setup:

```
physical pa1;
x ↦ pa1;
y ↦ pa1;
*pa1 = 0;
x ?-> invalid;
identity 0x1000 with code;
```

Initial state:

```
0:PSTATE.EL=0b01
0:R0=0b0
0:R1=pte3(x,page_table_base)
0:R2=0x2
0:R3=y
0:R5=page(x)
1:R0=0x1
1:R1=x
1:VBAR_EL1=0x1000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI VAE1IS,X5
DSB SY
STR X2,[X3]
```

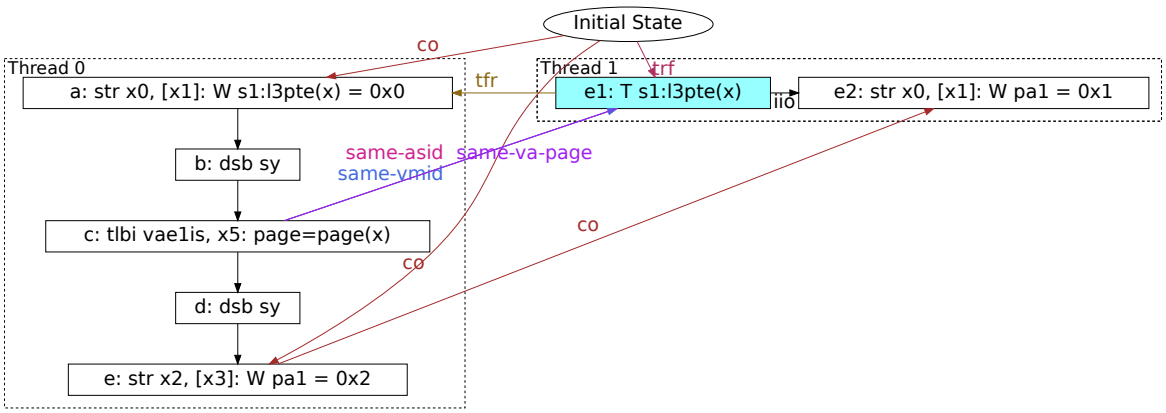
Thread 1

```
STR X0,[X1]
```

thread1 el1 handler

```
0x1400:
MRS X13,ELR_EL1
ADD X13,X13,#4
MSR ELR_EL1,X13
ERET
```

Final state: `x=1`



Model	Result
Base	`WBM+dsb-tlbiis-dsb forbidden (0 of 2) 5104ms`
ETS	`WBM+dsb-tlbiis-dsb forbidden (0 of 2) 4658ms`

A.10.0.13 Test: WBM+dsb-tlbiis-dsb-[dmb]-dmb forbid

Model	Result
Base	no result for WBM+dsb-tlbiis-dsb-[dmb]-dmb
ETS	no result for WBM+dsb-tlbiis-dsb-[dmb]-dmb

Write to Broken Mapping.

This is a variant of RBS, but with a write to the invalidated location, instead of the read from it.

A.10.0.14 Test: CoWTf.inv.EL2+dsb-tlbiipa-dsb-tlbiis-dsb-eret forbid

AArch64 CoWTf.inv.EL2+dsb−tlbiipa−dsb−tlbiis−dsb−eret

Page table setup:

```
physical pa1;
intermediate ipa1;

x ↦ ipa1;

ipa1 ↦ invalid;
ipa1 ?-> pa1;

*pa1 = 1;

identity 0x2000 with code;
```

Initial state:

```
ELR_EL2=L0:
PSTATE.EL=0b10
R0=mkdesc3(oa=pa1)
R1=pte3(ipa1,s2_page_table_base)
R3=x
R4=page(x)
SPSR_EL2=0b00100
VBAR_EL2=0x2000
```

Thread 0

```
STR X0,[X1]
DSB SY
TLBI IPAS2E1,X4
DSB SY
TLBI VMALLE1IS
DSB SY
ERET
L0:
LDR X2,[X3]
```

thread0 el2 handler lower exc

```
0x2400:
MOV X2,#0
MRS X20,ELR_EL2
ADD X20,X20,#4
MSR ELR_EL2,X20
ERET
```

Final state: `0:R2=0`

Model	Result
Base	`CoWTf.inv.EL2+dsb-tlbiipa-dsb-tlbiis-dsb-eret forbidden (0 of 5) 221283ms`
ETS	`CoWTf.inv.EL2+dsb-tlbiipa-dsb-tlbiis-dsb-eret forbidden (0 of 5) 5688ms`

B Full models

Here we include the entire strong and weak model (Note that the main relations are the same as those in §5 but may be presented differently).

B.1 Common

The models both include a common core, which defines the shared set of derived relations and events.

B.1.1 Barriers

First we define a hierarchy of barriers, so that an ordering `[e1] ; dmb ; [e2]` implies `[e1] ; dsb ; [e2]`.

```
(* define a hierarchy of barriers *)
(*  e.g. if  [e1] ; dmbst ; [e2]  is forbidden
 *  then
 *           [e1] ; dmbsy ; [e2]
 *       and [e1] ; dsbsy ; [e2]
 *  are forbidden too
 *)
(* we do not model NSH so pretend it's SY *)
let dsbsy = DSB.ISH | DSB.SY | DSB.NSH
let dsbst = dsbsy | DSB.ST | DSB.ISHST | DSB.NSHST
let dsbld = dsbsy | DSB.LD | DSB.ISHLD | DSB.NSHLD
let dsbnsh = DSB.NSH
let dmbsy = dsbsy | DMB.SY
let dmbst = dmbsy | dsbst | DMB.ST | DSB.ST | DSB.ISHST | DSB.NSHST
let dmbld = dmbsy | dsbld | DMB.LD | DSB.ISHLD | DSB.NSHLD
let dmb = dmbsy | dmbst | dmbld
let dsb = dsbsy | dsbst | dsbld
```

B.1.2 Common Core

Here we define all the relations common to both models, as they are given to isla-axiomatic:

```
include "barriers.cat"

(* For each instruction, for each read performed by the translation
   table walk ASL code, we generate one translate-read (T) event. If
   the translation finds an invalid entry, the translate-read event
   will additionally belong to T_f. *)
set T
set T_f

(* T events which are part of a Stage 1 or 2 walk *)
set Stage1
set Stage2

set read_VMID
set read_ASID
relation same-translation

(* A write of an invalid descriptor (an even value) is in W_invalid *)
set W_invalid

(* A write of a valid descriptor (an odd value) is in W_valid *)
set W_valid
```

```
(* initial writes *)
set is_IW

(* trf is the translate analogue of rf, such that writes are
   trf-related to translates that read them. trf1 is trf
   restricted to stage 1 reads, and trf2 for stage 2 reads *)
relation trf
relation trf1
relation trf2
relation wco

relation iio
relation instruction-order

(* e1 speculative e2
 * iff e2 was conditionally executed based on the value of e1
 *)
let speculative =
    ctrl
  | addr; po
  | [T] ; instruction-order

(* po-pa relates all events from instructions in instruction-order to the same PA *)
let po-pa = instruction-order & loc

let trfi = trf & int
let trfe = trf \ trfi

(* likewise, tfr is the translate analogue of fr *)
(* we use overlap-loc not loc here to handle the case where
 * multiple translations get merged into a single event *)
relation overlap-loc
let tfr1 = (((trf1^-1); co) \ id) & overlap-loc
let tfr2 = (((trf2^-1); co) \ id) & overlap-loc
let tfr = tfr1 | tfr2
let tfri = tfr & int
let tfre = tfr \ tfri

(* translate and TLBI events with VAs within the same 4K region
 * are related by same-va-page, similarly for IPAs and same-ipa-page *)
relation same-va-page
relation same-ipa-page
relation same-asid-internal
relation same-vmid-internal
relation tlbi-to-asid-read
relation tlbi-to-vmid-read

(* for convenience, derive some handy embeddings of program-order into
 * some subset of the events *)
(*
 * THESE ARE GENERATED BY ISLA
let instruction-order = iio^-1 ; fpo ; iio
let po = [M|F|C] ; instruction-order ; [M|F|C]
let tpo = [T] ; instruction-order ; [T]
*)

(* addr is now derived from the data dependency into the translate-reads
 * if a translation exists *)
(*
 * THIS IS GENERATED BY ISLA
let _addr =
    tdata ; [M]
  | tdata ; iio+ ; [R|W]
```

```
  | tdata ; [T_f]
*)

(* CSEs are context-synchronization-events
 * that is an ISB, and taking/returning from an exception *)
(*
let CSE = ISB | TE | ERET
*)

(* Context changing operations
 * are those that write to system registers
 *)
let ContextChange = MSR | TE | ERET

(* fault events come from reads or writes *)
let Fault = TE  (* TakeException, this is overly general *)
let IsTranslationFault = Fault
let IsPermissionFault = Fault
set IsFromR  (* events originating from an Arm LDR instruction *)
set IsFromW  (* events originating from an Arm STR instruction *)
set IsFromReleaseW  (* events originating from an Arm STLR instruction *)

(* Currently we only use same-vmid/same-asid between TLBIs and
 * translates, so this relation defines them in a minimal way.
 *)
let same-vmid = tlbi-to-vmid-read; [read_VMID]; same-translation
let same-asid = tlbi-to-asid-read; [read_ASID]; same-translation

(* A TLBI barriers some writes, making them unobservable to "future" reads from a
    translation table walk.
 *
 * tseq1 relates writes with TLBIs that ensure their visibility
 * e.g.  'a: Wpte(x) ; b: Wpte(x) ; c: Wpte(x) ; d: TLBI x'
 *  then 'c ; tseq1 ; d'
 *  as a, b are no longer visible to translation table walks
 *)
let tlb_might_affect =
    [ TLBI-S1 & ~TLBI-S2 &  TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-va-page & same-
    asid & same-vmid) ; [T & Stage1]
  | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  &  TLBI-ASID & TLBI-VMID] ; (same-asid & same-vmid
    ) ; [T & Stage1]
  | [ TLBI-S1 & ~TLBI-S2 & ~TLBI-VA  & ~TLBI-ASID & TLBI-VMID] ; same-vmid ; [T &
    Stage1]
  | [~TLBI-S1 &  TLBI-S2 &  TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; (same-ipa-page & same-
    vmid) ; [T & Stage2]
  | [~TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid ; [T &
    Stage2]
  | [ TLBI-S1 &  TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & TLBI-VMID] ; same-vmid ; [T]
  | ( TLBI-S1 &            ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T & Stage1)
  | (            TLBI-S2 & ~TLBI-IPA & ~TLBI-ASID & ~TLBI-VMID) * (T & Stage2)
(*  | (TLBI-ALL * T) *)

let tlb-affects =
    [TLBI-IS] ; tlb_might_affect
  | ([~TLBI-IS] ; tlb_might_affect) & int

(* [T] -> [TLBI] where the T reads-from a write before the TLBI and the TLBI is to the
    same addr
 * this doesn't mean the T happened before the TLBI, but it does mean there could have
    been a cached version
 * which the TLBI threw away
 *)
let maybe_TLB_cached =
  ([T] ; trf^-1 ; wco ; [TLBI-S1]) & tlb-affects^-1
```

```
(* translation-ordered-before *)
let tob =
  (* a faulting translation must read from flat memory or newer *)
    [T_f] ; tfre
  (* cannot forward past a DSB *)
  | ([T_f] ; tfri ; [W]) & (po ; [dsbst] ; instruction-order)^-1
  (* no forwarding from speculative writes *)
  | speculative ; trfi

let tlb_barriered =
  ([T] ; tfr ; wco ; [TLBI]) & tlb-affects^-1

let obtlbi_translate =
  (* A S1 translation must read from TLB/memory before the TLBI which
   * invalidates that entry happens *)
  [T & Stage1] ; tlb_barriered ; [TLBI-S1]
  (* if the S2 translation is ordered before some S2 write
   * then the S1 translation has to be ordered before the subsequent
   * S1 invalidate which would force the S2 write to be visible
   *
   * this applies to S2 translations during a S1 walk as well
   * here the Stage2 translation is only complete once the TLBI VA which
   * invalidates previous translation-table-walks have been complete *)
  (* if the S1 translation is from after the TLBI VA
   * then the S2 translation is only ordered after the TLBI IPA
   *)
  | ([T & Stage2] ; tlb_barriered ; [TLBI-S2])
     & (same-translation ; [T & Stage1] ; trf^-1 ; wco^-1)
  (* if the S1 translation is from before the TLBI VA,
   * then the S2 translation is ordered after the TLBI VA
   *)
  | (([T & Stage2] ; tlb_barriered ; [TLBI-S2]) ; wco? ; [TLBI-S1])
    & (same-translation ; [T & Stage1] ; maybe_TLB_cached)

(* ordered-before-TLBI *)
let obtlbi =
    obtlbi_translate
  (*
   * a TLBI ensures all instructions that use the old translation
   * and their respective memory events
   * are ordered before the TLBI.
   *)
  | [R|W|Fault] ; iio^-1 ; (obtlbi_translate & ext) ; [TLBI]

(* context-change ordered-before *)
(* note that this is under-approximate and future work is needed
 * on exceptions and context-changing operations in general *)
let ctxob =
 (* no speculating past context-changing operations *)
    speculative ; [MSR]
 (* context-synchronization orders everything po-after with the synchronization point
    *)
  | [CSE] ; instruction-order
 (* context-synchronization acts as a barrier for context-changing operations *)
  | [ContextChange] ; po ; [CSE]
 (* context-synchronization-events cannot happen speculatively *)
  | speculative ; [CSE]

(* ordered-before a translation fault *)
let obfault =
    data ; [Fault & IsFromW]
  | speculative ; [Fault & IsFromW]
  | [dmbst] ; po ; [Fault & IsFromW]
```

```
  | [dmbld] ; po ; [Fault & (IsFromW | IsFromR)]
  | [A|Q] ; po ; [Fault & (IsFromW | IsFromR)]
  | [R|W] ; po ; [Fault & IsFromW & IsFromReleaseW]

(* ETS-ordered-before *)
(* if FEAT_ETS then if E1 is ordered-before some Fault
 * then E1 is ordered-before the translation-table-walk read which generated that fault
 * (but not *every* read from the walk, only the one that directly led to the
    translation fault)
 *
 * Additionally, if ETS then TLBIs are guaranteed completed after DSBs
 * hence po-later translations must be ordered after the TLBI (D5.10.2)
 *)
let obETS =
    (obfault ; [Fault]) ; iio^-1 ; [T_f]
  | ([TLBI] ; po ; [dsb] ; instruction-order ; [T]) & tlb-affects

include "shows.cat"
```

B.2 Strong Model

```
"VMSA strong"
include "cos.cat"
include "barriers.cat"
include "aarch64_mmu_common.cat"

(* observed by *)
let obs = rfe | fr | wco
  (* observing a write through a fetch or translate
   * means the write is now visible to the rest of the system
   * aka {instruction, translation}->data coherence *)
  | trfe

(* dependency-ordered-before *)
let dob =
    addr | data
  | speculative ; [W]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | data); trfi

(* atomic-ordered-before *)
let aob = rmw
  | [range(rmw)]; rfi; [A | Q]

(* barrier-ordered-before *)
let bob = [R] ; po ; [dmbld]
  | [W] ; po ; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb] ; po

(* Ordered-before *)
let _ob = obs | dob | aob | bob | iio | tob | obtlbi | ctxob | obfault
let ob = _ob^+

(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal

(* External visibility requirement *)
irreflexive ob as external

(* Atomic: Basic LDXR/STXR constraint to forbid intervening writes. *)
empty rmw & (fre; coe) as atomic

(* Writes cannot forward to po-future translations *)
acyclic (po-pa | trfi) as translation-internal

(* No translations interposing well-bracketed take/return exceptions *)
(* empty take-to-return & ((ob & int) ; [T] ; (ob & higher-EL)) *)
```

Figure 20: Strong Model

B.2.1 Translation Faults

To correctly implement ETS and TLBI-completion ordering for translation-faults we produce `fault` events which exist `iio`-after the `T` event which causes them.

To get the correct ETS ordering, we add `FromR` and `FromW` sets for faults that originate from load or store instructions.

Then we duplicate edges from `ob` which end in a `[R]` or `[W]` to also end in `[Fault & FromR]` or `[Fault & FromW]`, and those get included in the `obfault` relation which is included in `ob`. To model ETS, we can then simply add `[R|W] ; obfault ; [fault] ; iio^-1 ; [T_f]` to `ob`.

See the relevant part of the Arm ARM (D.5.10.2 — **Ordering and completion of TLB maintenance instructions**)

```
A TLB maintenance operation without the nXS qualifier generated by a TLB maintenance
```

```
    instruction is
finished for a PE when:
- All memory accesses generated by that PE using in-scope old translation information
    are complete.
- All memory accesses RWx generated by that PE are complete.

RWx is the set of all memory accesses generated by instructions for that PE that
    appear in program order
before an instruction I1 executed by that PE where all of the following apply:
- I1 uses the in-scope old translation information.
- The use of the in-scope old translation information generates a synchronous Data
    Abort.
- If I1 did not generate an abort from use of the in-scope old translation
    information, I1 would generate
    a memory access that RWx would be locally-ordered-before.
```

B.2.2 Edges justification

We justify existence of edges in `ob` with the following tests:

B.2.2.1 obs

- `[W] ; trfe ; [T]` (CoTRpte.inv+dsb-isb)
- `[W] ; trfe ; [T_f]` (CoTfRpte+dsb-isb)

B.2.2.2 tob

- `[T_f] ; tfr ; [W]` (CoRpteTf.inv+dsb-isb)
- `[T] ; iio ; [R|W] ; po ; [W]` (see also `speculative ; [W]`, S.T+dmb+po)
- `speculative ; trfi` (MP.RT.inv+dmb+ctrl-trfi)

B.2.2.3 obtlbi_translate

- `tcache1` (MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb)
- `tcache2 & (same-trans ; [T & Stage1] ; trf^1 ; wco^1)` (WDS+dsb-tlbiipa-dsb-po-eret)
- `(tcache2 ; wco?  ; [TLBI S2]) & (same-trans ; [T & Stage1] ; maybe_TLB_cached)` (WDS+po-dsb-tlbiipa-dsb-tlbiis-dsb-eret)

B.2.2.4 obtlbi

- `obtlbi_translate` (see previous)
- `[R] ; iio^1 ; (obtlbi_translate & ext)` (RBS+dsb-tlbiis-dsb)
- `[W] ; iio^1 ; (obtlbi_translate & ext)` (WBM+dsb-tlbiis-dsb)
- `[Fault] ; iio^1 ; (obtlbi_translate & ext)` (see `obfault` for relevant tests)

B.2.2.5 ctxob These edges are over-approximate compared to the assumed real semantics of context-synchronizing events.

- `speculative ; [MSR]` (MP.RT.inv+dmb+addr-po-msr)
- `[CSE] ; instruction-order` (MP+dmb+ctrl-isb)
- `[ContextChange] ; po ; [CSE]` (SwitchTable.different-asid+eret)
- `speculative ; [ISB]` (MP+dmb+ctrl-isb, MP+dmb+addr-po-isb, MP.TR.inv+dmb+isb)

B.2.2.6 obfault

- `[R] ; data ; [Fault_W]` (S.RTf.inv.EL1+dsb-tlbiis-dsb+data)
- `speculative ; [Fault_W]` (S.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl)
- `[dmb] ; po ; [Fault_R]` (MP.RTf.inv.EL1+dsb-tlbiis-dsb+dmb)
- `[dmb] ; po ; [Fault_W]` (S.RTf.inv.EL1+dsb-tlbiis-dsb+dmb)
- `[A|Q] ; po ; [Fault]` (MP.RTf.inv.EL1+dsb-tlbiis-dsb+poap, MP.RTf.inv.EL1+dsb-tlbiis-dsb+poqp, S.RTf.inv.EL1+dsb-tlbiis-dsb+poap, S.RTf.inv.EL1+dsb-tlbiis-dsb+poqp
- `[R|W] ; po ; [Fault_L]` (S.RTf.inv.EL1+dsb-tlbiis-dsb+popl, R.Tf.inv.EL1+dsb-tlbiis-dsb+popl)

B.2.2.7 obETS

- `obfault ; [Fault] ; iio^1 ; [T_f]` (MP.RTf.inv+dmbs, MP.RTf.inv+dmb+addr)

B.2.2.8 dob These edges ensure that self-satisfying cycles cannot be constructed, which could otherwise lead to new translation table entries out of thin air.

- `addr ; trfi` (LB+addr-trfis)
- `data ; trfi` (LB+data-trfis)

Note the lack of `ctrl ; trfi` here does not imply weakness, as `tob` already covers this.

B.2.2.9 axioms

- `acyclic (po-pa | trfi)` (CoTW1.inv)

B.3 Weak Model

```
"VMSA weak"
include "cos.cat"
include "aarch64_mmu_common.cat"
include "barriers.cat"

let obs = rfe | fr | wco

let dob = addr | data
  | ctrl; [W]
  | (ctrl | (addr; po)); [ISB]
  | addr; po; [W]
  | (addr | data); rfi
  | (addr | ctrl | data); trfi

let aob = rmw
  | [range(rmw)]; rfi; [A | Q]

(* barrier-ordered-before *)
let bob = [R] ; po ; [dmbld]
  | [W] ; po ; [dmbst]
  | [dmbst]; po; [W]
  | [dmbld]; po; [R|W]
  | [L]; po; [A]
  | [A | Q]; po; [R | W]
  | [R | W]; po; [L]
  | [F | C]; po; [dsbsy]
  | [dsb] ; po
  | [CSE] ; instruction-order

(* Ordered-before *)
let ob = (obs | dob | aob | bob | ctxob)^+

(* Internal visibility requirement *)
acyclic po-loc | fr | co | rf as internal

(* External visibility requirement *)
irreflexive ob as external

(* Atomic: Basic LDXR/STXR constraint to forbid intervening writes. *)
empty rmw & (fre; coe) as atomic

(* Writes cannot forward to po-future translations *)
acyclic (po-pa | trfi) as translation-internal

(* break-before-make S1 *)
empty
  ([is_IW | W_invalid] ; co ; [W_valid] ; ob ; [CSE] ; instruction-order ; [T & Stage1
    ])
  & (ob ; [dsbsy] ; po ; ([TLBI-S1] ; po ; [dsbsy] ; ob ; [CSE] ; instruction-order ; [
    T]) & tlb-affects)
  & trf & loc
  as bbm

(* break S1 *)
empty ([is_IW | W] ; co ; [W_invalid] ; ob ; [dsbsy] ; po
      ; ([TLBI-S1] ; po ; [dsbsy] ; ob ; [M]; iio^-1; [T]) & tlb-affects & ext
      ) & trf & loc
      as brk1

empty ([is_IW | W] ; co ; [W_invalid] ; ob ; [dsbsy] ; po
      ; ([TLBI-S1] ; po ; [dsbsy] ; ob ; [CSE] ; instruction-order ; [T]) & tlb-affects
      ) & trf & loc
```

```
      as brk2

(* break-before-make S2 *)
empty
  ([is_IW | W_invalid] ; co ; [W_valid] ; ob ; [CSE] ; instruction-order ; [T & Stage2
    ])
  & (ob ; [dsbsy] ; po
    ; ([TLBI-S2] ; po ; [dsbsy] ; po ;
      ; ([TLBI-S1] ; po ; [dsbsy] ; ob ; [CSE]; instruction-order; [T]) & tlb-affects
      ; iio ; [T]) & tlb-affects)
  & trf & loc
    as bbms2

(* break S2 *)
empty ([is_IW | W] ; co ; [W_invalid] ; ob ; [dsbsy] ; po
      ; ([TLBI-S2] ; po ; [dsbsy] ; po ; ([TLBI-S1] ; po ; [dsbsy] ; ob ; [M]; iio^-1;
    [T]) & tlb-affects & ext
      ; iio ; [T]) & tlb-affects & ext
      ) & trf & loc
      as brk1s2

empty ([is_IW | W] ; co ; [W_invalid] ; ob ; [dsbsy] ; po
      ; ([TLBI-S2] ; po ; [dsbsy] ; po ; ([TLBI-S1] ; po ; [dsbsy] ; ob ; [CSE];
    instruction-order; [T]) & tlb-affects
      ; iio ; [T]) & tlb-affects
      ) & trf & loc
      as brk2s2
```

B.4 Break-before-make detection predicate

```
; Check for break-before-make violations
; This is set of constraints is satisfiable iff there is a BBM violation
(declare-const BBM_Wl0 Event)
(declare-const BBM_Wl1 Event)
(declare-const BBM_Wl2 Event)
(declare-const BBM_Wl3 Event)

(declare-const BBM_Wl0_pa (_ BitVec 64))
(declare-const BBM_Wl1_pa (_ BitVec 64))
(declare-const BBM_Wl2_pa (_ BitVec 64))
(declare-const BBM_Wl3_pa (_ BitVec 64))

(assert (not (= BBM_Wl0_pa BBM_Wl1_pa)))
(assert (not (= BBM_Wl1_pa BBM_Wl2_pa)))
(assert (not (= BBM_Wl2_pa BBM_Wl3_pa)))

(declare-const BBM_Wl0_data (_ BitVec 64))
(declare-const BBM_Wl1_data (_ BitVec 64))
(declare-const BBM_Wl2_data (_ BitVec 64))
(declare-const BBM_Wl3_data (_ BitVec 64))

(declare-const BBM_ia (_ BitVec 36))

(define-fun ia_offset3 ((ia (_ BitVec 36))) (_ BitVec 12)
  (concat ((_ extract 8 0) ia) #b000))

(define-fun ia_offset2 ((ia (_ BitVec 36))) (_ BitVec 12)
  (concat ((_ extract 17 9) ia) #b000))

(define-fun ia_offset1 ((ia (_ BitVec 36))) (_ BitVec 12)
  (concat ((_ extract 26 18) ia) #b000))

(define-fun ia_offset0 ((ia (_ BitVec 36))) (_ BitVec 12)
  (concat ((_ extract 35 27) ia) #b000))

(define-fun page_offset ((pa (_ BitVec 64))) (_ BitVec 12)
  ((_ extract 11 0) pa))

(define-fun table_address ((desc (_ BitVec 64))) (_ BitVec 64)
  (concat #x0000 ((_ extract 47 12) desc) #x000))

(assert (= (page_offset BBM_Wl0_pa) (ia_offset0 BBM_ia)))
(assert (= (page_offset BBM_Wl1_pa) (ia_offset1 BBM_ia)))
(assert (= (page_offset BBM_Wl2_pa) (ia_offset2 BBM_ia)))
(assert (= (page_offset BBM_Wl3_pa) (ia_offset2 BBM_ia)))

(assert (tt_write BBM_Wl0 BBM_Wl0_pa BBM_Wl0_data))

(define-fun valid_desc ((desc (_ BitVec 64))) Bool
   (= (bvand desc #x0000000000000001) #x0000000000000001))

(define-fun valid_table_desc ((desc (_ BitVec 64))) Bool
   (= (bvand desc #x0000000000000011) #x0000000000000011))

; For each level, if it is valid, then its parent must be a valid table entry
(assert
  (and
    (implies (valid_desc BBM_Wl3_data) (valid_table_desc BBM_Wl2_data))
    (implies (valid_desc BBM_Wl2_data) (valid_table_desc BBM_Wl1_data))
    (implies (valid_desc BBM_Wl1_data) (valid_table_desc BBM_Wl0_data))))

; If an entry is pointed to by its parent, then it must be actually
```

```
; represented by a valid page table write at the correct location.
; The alternative is if the parent is invalid, in which case anything
; goes
(assert
  (implies (valid_table_desc BBM_Wl0_data)
    (and (tt_write BBM_Wl1 BBM_Wl1_pa BBM_Wl1_data)
         (= (table_address BBM_Wl0_data) (table_address BBM_Wl1_pa)))))

(assert
  (implies (valid_table_desc BBM_Wl1_data)
    (and (tt_write BBM_Wl2 BBM_Wl2_pa BBM_Wl2_data)
         (= (table_address BBM_Wl1_data) (table_address BBM_Wl2_pa)))))

(assert
  (implies (valid_table_desc BBM_Wl2_data)
    (and (tt_write BBM_Wl3 BBM_Wl3_pa BBM_Wl3_data)
         (= (table_address BBM_Wl2_data) (table_address BBM_Wl3_pa)))))

(declare-const BBM_W1 Event)
(declare-const BBM_W1_pa (_ BitVec 64))
(declare-const BBM_W1_data (_ BitVec 64))

(declare-const BBM_W2 Event)

; BBM_W1 and BBM_W2 conflict
(assert (and (tt_write BBM_W1 BBM_W1_pa BBM_W1_data) (valid_desc BBM_W1_data)))
(assert (W_valid BBM_W2))
(assert (not (= ((_ extract 47 12) BBM_W1_data) ((_ extract 47 12) (val_of_64 BBM_W2)))
    ))
(assert (= BBM_W1_pa (addr_of BBM_W2)))

(assert (or
  (and (= BBM_W1 BBM_Wl3) (= BBM_W1_pa BBM_Wl3_pa) (= BBM_W1_data BBM_Wl3_data))
  (and (= BBM_W1 BBM_Wl2) (= BBM_W1_pa BBM_Wl2_pa) (= BBM_W1_data BBM_Wl2_data))
  (and (= BBM_W1 BBM_Wl1) (= BBM_W1_pa BBM_Wl1_pa) (= BBM_W1_data BBM_Wl1_data))
  (and (= BBM_W1 BBM_Wl0) (= BBM_W1_pa BBM_Wl0_pa) (= BBM_W1_data BBM_Wl0_data))))

(assert (co BBM_W1 BBM_W2))

(define-fun BBM_sequence1 ((S_Wp Event) (S_tlbi Event)) Bool
  (and
    (wco BBM_W1 S_Wp)
    (W_invalid S_Wp)
    (implies (= BBM_W1 BBM_Wl3) (or (= S_Wp BBM_Wl3) (= S_Wp BBM_Wl2) (= S_Wp BBM_Wl1)
    (= S_Wp BBM_Wl0)))
    (implies (= BBM_W1 BBM_Wl2) (or (= S_Wp BBM_Wl2) (= S_Wp BBM_Wl1) (= S_Wp BBM_Wl0))
    )
    (implies (= BBM_W1 BBM_Wl1) (or (= S_Wp BBM_Wl1) (= S_Wp BBM_Wl0)))
    (implies (= BBM_W1 BBM_Wl0) (= S_Wp BBM_Wl0))
    (wco S_Wp S_tlbi)
    (TLBI-VA S_tlbi)
    (= (tlbi_va (val_of_cache_op S_tlbi)) (concat #x0000 BBM_ia #x000))
    (wco S_tlbi BBM_W2)))

; If there are no valid BBM sequence between BBM_W1 and BBM_W2, we have a BBM violation
(assert (forall ((BBM_Wp Event) (BBM_tlbi Event))
  (not (BBM_sequence1 BBM_Wp BBM_tlbi))))
```

C Relationships between models

In this appendix, we illustrate how our models support a relatively simple abstraction to higher-level code. We prove three theorems: that for static injectively-mapped address spaces, any execution which is consistent in the model with translation, erasing translation events gives an execution that is consistent in the original Armv8-A model without translation (Theorem 2); that for any consistent execution in the original Armv8-A model, there is a corresponding consistent execution in our extended model with translations (Theorem 3); and that our weak model is a sound over-approximation of our full translation model, i.e., that for any consistent execution in our full translation model, that same execution is consistent in the weak translation model (Theorem 1).

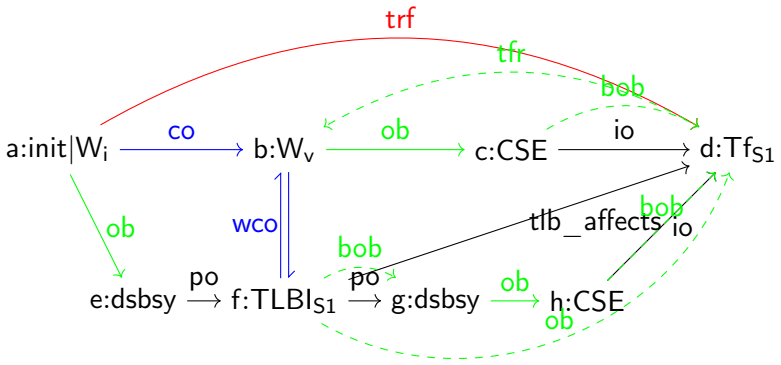
C.1 Soundness of the weak model

Theorem 1 *The weak model is a sound over-approximation of the strong model.*

Proof: The definition of `ob` in the strong model contains all the clauses of the weak model (and more).
The extra axioms of the weak model are subsumed by those of the strong model:
We label events with identifiers, write given edges solid, and derived edges dashed.

- bbm

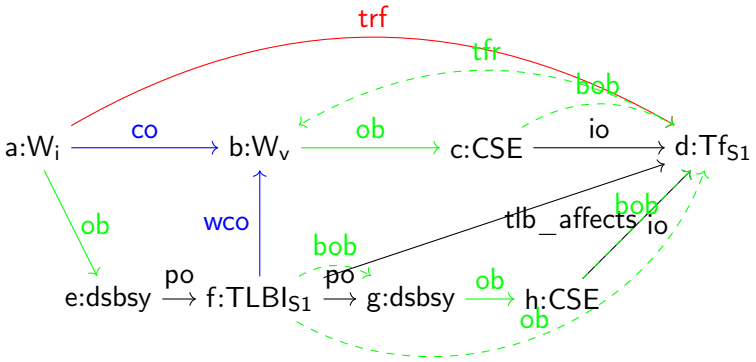
```
([a:is_IW | W_invalid] ; co ; [b:W_valid]
 ; ob ; [c:CSE] ; instruction-order ; [d:Tf & Stage1])
&
(ob ; [e:dsb] ; po
 ; (([f:TLBI-S1] ; po ; [g:dsb] ; ob ; [h:CSE] ; instruction-order)
    & tlb_affects))
& trf & loc
```



From `[f] ; po ; [g]`, we have `[f] ; bob ; [g]`, and from `[h] ; io ; [d]`, we have `[h] ; bob ; [d]` Therefore, together with `[g] ; ob ; [h]`, we have `[f] ; ob ; [d]`.
From `[c] ; po ; [d]`, we `[c] ; bob ; [d]`.

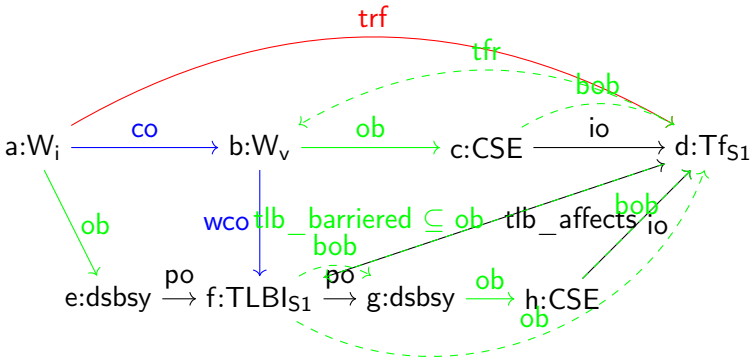
`wco` relates `b` and `f`.

  - If `[f] ; wco ; [b]`, then this is not a BBM violation.



From `[d] ; tfr ; [b]`, we have `[d] ; ob ; [b]`.
Therefore, we have a cycle in `ob`.

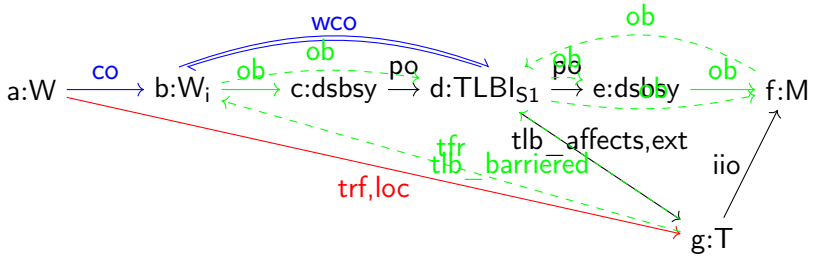
– If `[b] ; wco ; [f]`, then:



∗ If there is another TLBI preventing a BBM violation involving `a` and `b`, then there is another subgraph of the execution that corresponds to the previous case.
∗ If not, then there is also a BBM violation in the strong model, because they have the same execution candidate, and use the same BBM check.

- brk1

```
([a:is_IW | W] ; co ; [b:W_invalid] ; ob ; [c:dsbsy] ; po
 ; ([d:TLBI-S1] ; po ; [e:dsbsy] ; ob ; [f:M]; iio^-1; [g:T])
   & tlb_affects & ext)
& trf & loc
```

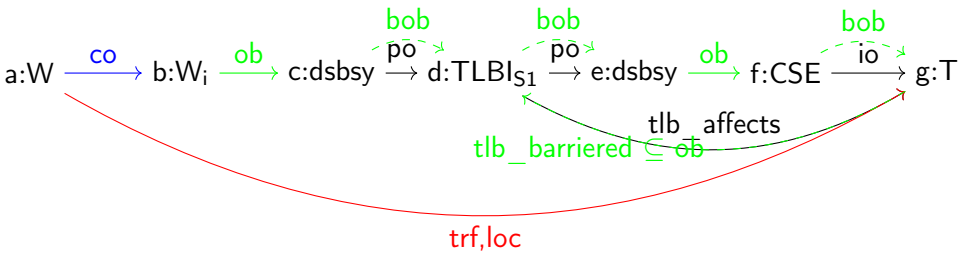


From `[a] ; trf ; [g]` and `[a] ; co ; [b]`, we have `[g] ; tfr ; [b]`.
`wco` relates `b` and `d`.

  - If `[d] ; wco ; [b]`.
    then we have `[d] ; ob ; [b]`
    Moreover, from `[c] ; po ; [d]`, in the weak model, we have `[c] ; ob ; [d]`.
    From `[b] ; ob ; [c]` and `[c] ; ob ; [d]`, we have `[b] ; ob ; [d]`.
    Therefore, we have a cycle in `ob`.
  - If `[b] ; wco ; [d]`.
    From `[g] ; tfr ; [b]`, `[b] ; wco ; [d]`, and `[d] ; tlb-affects^-1 ; [g]`, we have `[g] ; tlb_barriered ; [d]`,
    and therefore `[g] ; obtlbi_translate ; [d]`.
    From `[g] ; obtlbi_translate ; [d]`, `[g] ; ext ; [d]`, and `[f] ; iio^-1 ; [g]`, we have `[f] ; obtlbi_translate ; [d]`,
    and therefore `[f] ; ob ; [d]`.
    Moreover, from `[d] ; po ; [e]`, we have `[d] ; bob ; [e]`, and therefore `[d] ; ob ; [e]`.
    From `[d] ; ob ; [e]` and `[e] ; ob ; [f]`, we have `[d] ; ob ; [f]`.
    Therefore, we have a cycle in `ob`.

- brk2

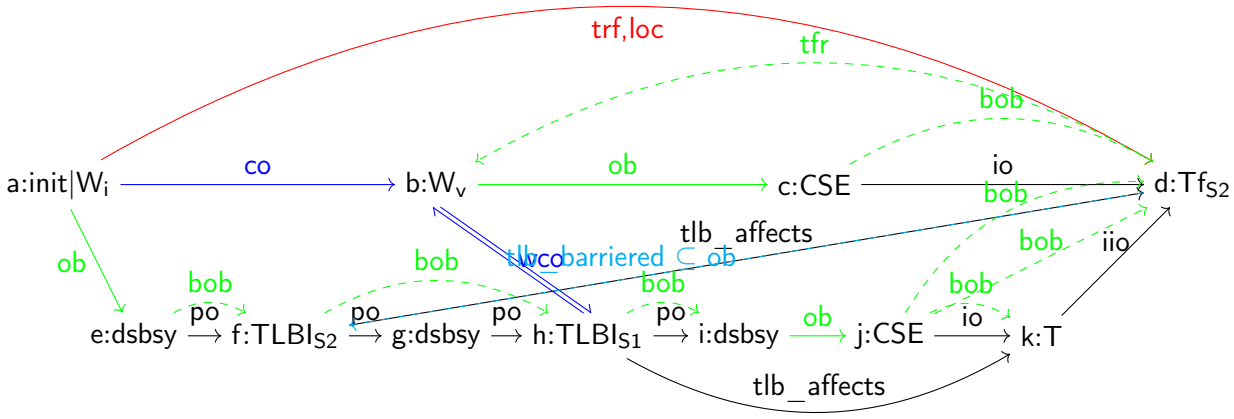
```
([a:is_IW | W] ; co ; [b:W_invalid] ; ob ; [c:dsbsy] ; po
 ; ([d:TLBI-S1] ; po ; [e:dsbsy] ; ob ; [f:CSE] ; instruction-order ; [g:T])
    & tlb-affects)
& trf & loc
```



From `[c] ; po ; [d]`, we have `[c] ; bob ; [d]`, and therefore `[c] ; ob ; [d]`.
Therefore, as before, by examination of `wco`, we have `[g] ; ob ; [d]`.
From `[d] ; po ; [e]`, we have `[d] ; bob ; [e]`, and therefore `[d] ; ob ; [e]`.
Moreover, from `[f] ; instruction-order ; [g]`, we have `[f] ; bob ; [g]`.
Therefore, we have a cycle in `ob`.

- bbms2

```
([a:is_IW | W_invalid] ; co ; [b:W_valid] ; ob
 ; [c:CSE] ; instruction-order ; [d:Tf & Stage2])
& (ob ; [e:dsbsy] ; po
   ; ([f:TLBI-S2] ; po ; [g:dsbsy] ; po ;
      ; ([h:TLBI-S1] ; po ; [i:dsbsy] ; ob ; [j:CSE]; instruction-order; [k:T])
         & tlb-affects
      ; iio) & tlb-affects)
& trf & loc
```

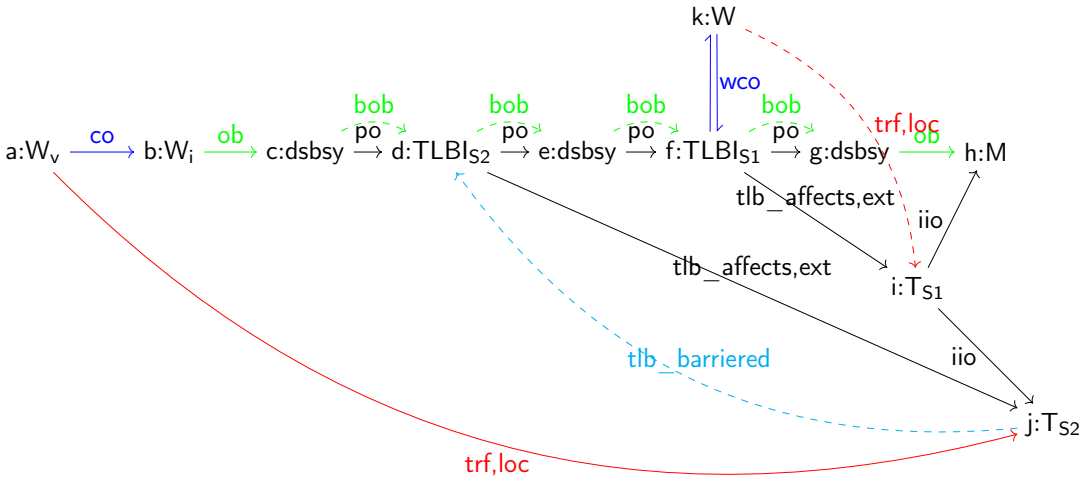


From `[c] ; instruction-order ; [d]`, we have `[c] ; bob ; [d]`, and therefore `[c] ; ob ; [d]`.
From `[j] ; instruction-order ; [k]`, we have `[j] ; bob ; [k]`, and therefore `[j] ; ob ; [k]`.
From `[e] ; po ; [f]`, we have `[e] ; bob ; [f]`.
From `[f] ; po [g:dsbsy] ; po ; [h]`, we have `[f] ; bob ; [h]`.
From `[h] ; po ; [i]`, we have `[h] ; bob ; [i]`.
From `[a] ; trf ; [d]` and `[a] ; co ; [b]`, we have `[d] ; tfr ; [b]`.
From `[j] ; instruction-order ; [k] ; iio ; [d]`, we have `[j] ; bob ; [d]`.
`wco` relates `b` and `h`.

- Assume `[h] ; wco ; [b]`. Then this is not a BBM violation.
  From `[d] ; tfr ; [b]`, we have `[d] ; ob ; [b]`.
- Assume `[b] ; wco ; [h]`.
  - If there is another TLBI preventing a BBM violation involving `a` and `b`, then there is another subgraph of the execution that corresponds to the previous case.
  - If not, then there is also a BBM violation in the strong model, because they have the same execution candidate, and use the same BBM check.

- brk1s2

```
([a:is_IW | W] ; co ; [b:W_invalid] ; ob ; [c:dsbsy] ; po
 ; ([d:TLBI-S2] ; po ; [e:dsbsy] ; po ;
    ([f:TLBI-S1] ; po ; [g:dsbsy] ; ob ; [h:M]; iio^-1; [i:T & Stage1])
     & tlb-affects & ext
; iio ; [j:T & Stage2]) & tlb-affects & ext
) & trf & loc
```



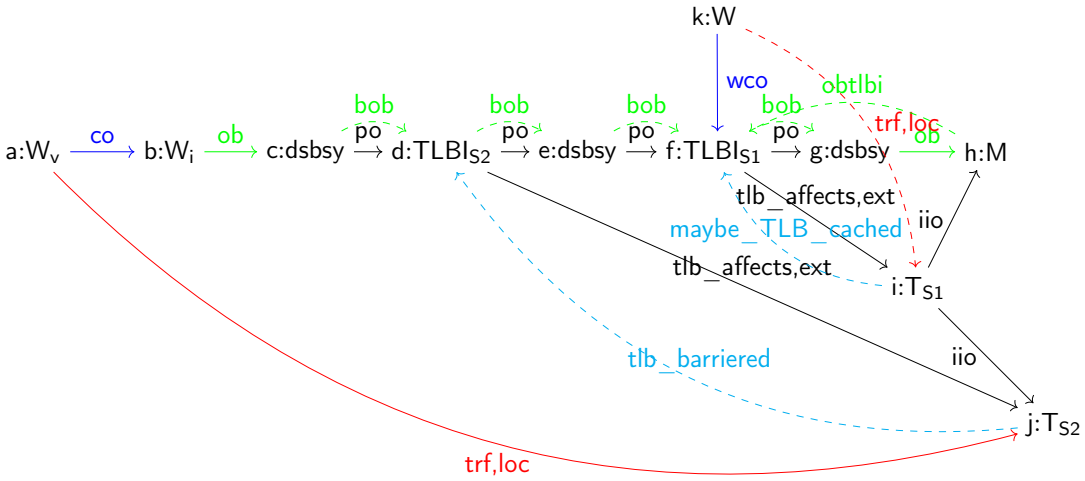
Consider `wco`:
From `[c] ; po ; [d]`, we have `[c] ; bob ; [d]`, and therefore `[c] ; ob ; [d]`.
From `[d] ; po ; [e]` and `[e] ; po ; [f]` we have `[d] ; ob ; [f]`.

Therefore `wco` must give us `[a] ; wco ; [b] ; wco ; [d] ; wco ; [f]` otherwise there is a cycle in `ob` and the execution is trivially forbidden.

For `[d] ; ob ; [e]`, `[e] ; ob ; [f]`, and `[f] ; ob ; [g]`.

`wco` relates `b` and `d`.

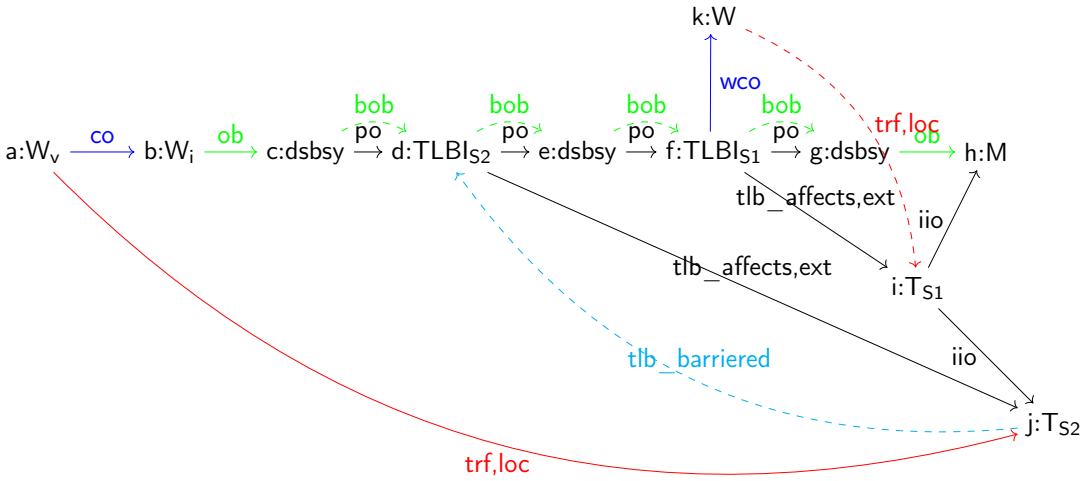
- If `[d] ; wco ; [b]`, we have a cycle in `ob`.
- If `[b] ; wco ; [d]`.
  From `[j] ; tfr ; [b]`, `[b] ; wco ; [d]`, and `[j] ; tlb-affects^-1 ; [d]`, we have `[j] ; tlb_barriered ; [d]`.
  Moreover, there must exist `k` such that `[k] ; trf ; [i]`, and `k` must be related by `wco` to `f`.
  * If `[k] ; wco ; [f]`:



Then we have `[j] ; tlb_barriered ; [d]` and `[i] ; maybe_TLB_cached ; [f]`, then from third clause of `obltbi_translate` we get

`[j] ; obtlbi_translate ; [f]`. From the second clause of `obtlbi` we have `[h] ; obtlbi ; [f]`, and so `[h] ; ob ; [f]`, and therefore we have a cycle in `ob`.

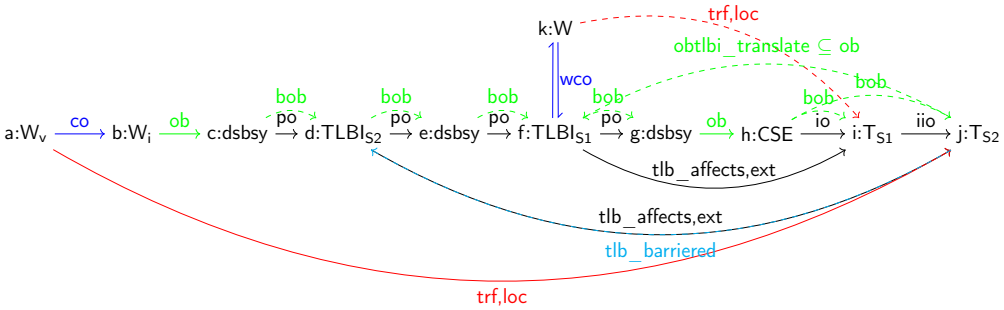
* If `[f] ; wco ; [k]`:



Then `[d] ; wco ; [k]`, and `[j] ; tlb_barriered ; [d]`. Then from the second clause of `obtlbi_translate` we have `[j] ; obtlbi_translate ; [d]` From the second clause of `obtlbi` we have `[h] ; obtlbi ; [d]`, Which implies `[h] ; ob ; [d]`, but `[d] ; ob ; [h]` by `bob` so we have a cycle in `ob`.

- brk2s2

```
([a:is_IW | W] ; co ; [b:W_invalid] ; ob ; [c:dsbsy] ; po
 ; ([d:TLBI-S2] ; po ; [e:dsbsy] ; po ;
    ([f:TLBI-S1] ; po ; [g:dsbsy] ; ob ; [h:CSE] ; instruction-order; [i:T & Stage1])
     & tlb-affects
    ; iio ; [j:T & Stage2]) & tlb-affects)
& trf & loc
```



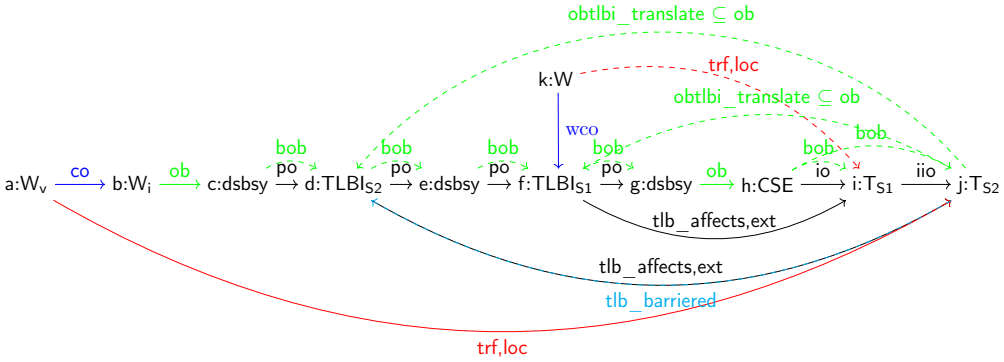
Similar to the previous case, we have `[c] ; ob ; [d]`, `[d] ; ob ; [e]`, `[e] ; ob ; [f]`, `[f] ; ob ; [g]`, `[h] ; ob ; [i]`, and `[h] ; ob ; [j]`.

`wco` relates `b` and `d`.

  - If `[d] ; wco ; [b]`, we have a cycle in `ob`.
  - If `[b] ; wco ; [d]`.
    From `[j] ; tfr ; [b]`, `[b] ; wco ; [d]`, and `[j] ; tlb-affects^-1 ; [d]`, we have

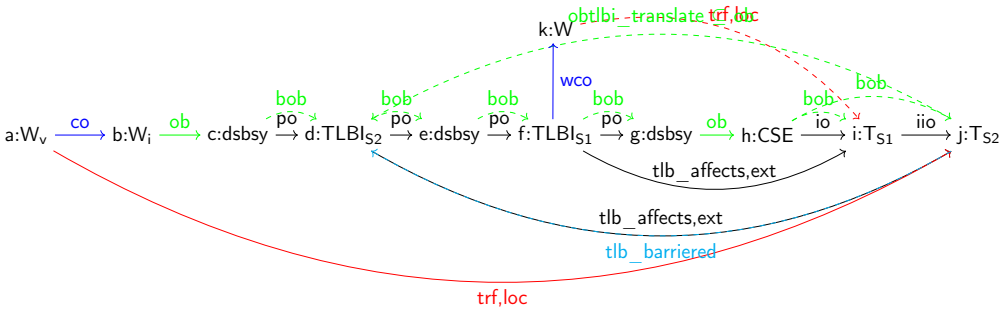
`[j] ; tlb_barriered ; [d]`.
Moreover, there must exist `k` such that `[k] ; trf ; [i]`, and `k` must be related by `wco` to `f`.

* If `[k] ; wco ; [f]`:



then we have `[i] ; maybe_TLB_cached ; [f]`, and therefore `[j] ; obtlbi_translate ; [f]`, and therefore `[j] ; ob ; [f]` so there is a cycle in `ob`.

* If `[f] ; wco ; [k]`:



then we have `[j] ; obtlbi_translate ; [d]`, and therefore `[j] ; ob ; [d]`, so there is a cycle in `ob`.

□

C.2 Virtual address abstraction and anti-abstraction

We consider a simple case when the virtual address abstraction ought to hold: under some conditions, the model with translation and the original model without translations coincide. Here, we only consider the consistency of the pre-executions, but not how these pre-executions arise.

C.2.1 Abstraction

Definition 1 (VA abstraction subcondition) `G` *satisfies the* VA abstraction subcondition *when it has no page-table-affecting instructions: no TLBI, no context-changing operations (for example via writing to registers, for example via MSR TTBR), etc.*

Definition 2 (VA abstraction condition) `Gtr` *satisfies the* VA abstraction condition *when it satisfies the VA abstraction subcondition, and has a static injective page table.*

Theorem 2 (VA abstraction) *For all (*`Gtr : concrete execution`*)*
if `Gtr` *is consistent wrt. the model with translation*
and respects the VA abstraction condition, then
let `Gabs` = *erase* `Gtr` *in*
`Gabs` *is consistent wrt. the model without translation.*

Proof: First, the builtin `addr` of the abstract model is assumed to coincide with the derived `addr` of the concrete model by the erasure. Showing that the two definitions of pre-executions do relate in this way is outside of our scope. Given that the definitions `addr` coincide, the definitions of all the other derived relations of the abstract model, including `ob` in the translation model, are syntactically supersets of their definition in the concrete model, so a cycle in `ob` in the abstract model is also a cycle in `ob` in the concrete model. □

C.2.2 Anti-abstraction

For this direction, we need to be able to put the translation table somewhere.

C.2.2.1 Step 1: Building the candidate execution in the translation model

Definition 3 (translation extension condition) *The* translation extension condition *is the data of*
(`Gabs : execution`*)*
such that `Gabs` *is consistent wrt. the model without translation*
and has no TLBI, and no MSR TTBR
and
(`va_space : va_address -> bool`*)*
such that all the memory accesses of `Gabs` *are in* `va_space`
and
(`pt_pa_space : pa_address -> bool`*)*
(`pt_initial_state : pa_address -> option (list byte)`*),*
such that the domains of `pt_pa_space` *and* `pt_initial_state` *coincide*
and
(`tr_ctxt : translation_context`*),*
such that `id_map_lifted va_space` *and* `pt_pa_space` *are disjoint address spaces*
and
(`translate : translation_function`*),*
such that translating `abstract_va_space` *translate-reads from within* `pt_pa_space` *and gives the injective map.*

Definition 4 (translation extension) *Given the translation extension condition, the* translation extension `Gtr` *of* `Gabs` *is constructed by:*

- *adding all the initial writes for the page tables,*
- *adding all the translate reads obtained by running the* `translate` *function with the* `tr_ctxt`*,*
- *adding the translate reads in* `iio` *between the fetch and the explicit event,*
- *adding* `tdata` *to match* `addr`*,*
- *adding* `trf` *from the corresponding initial writes to the translates.*

Definition 5 (VA anti abstraction condition) `Gtr` *satisfies the* VA anti-abstraction condition *when it is derived from a consistent execution which satisfies the VA abstraction subcondition by the translation extension.*

Lemma 1 (VA abstraction condition for translation extension) *If* `Gtr` *satisfies the* VA anti-abstraction condition*, then* `Gtr` *satisfies the VA abstraction condition.*

Proof: The translation extension does not add any extra instructions, and sets up static injective page tables. □

Lemma 2 (obtlbi-empty) *If* `Gtr` *satisfies the VA anti-abstraction condition, then* `obtlbi` *is empty.*

Proof: `obtlbi` has

- `obtlbi_translate` which has
  - `tcache1`
    which is `[T & Stage1] ; tfr ; tseq1`
    the latter is
    `[W] ; (maybe_TLB_barriered_by_va & ob) ; [TLBI VA]`
    which requires a TLBI, so it is empty
  - `tcache2 & ...`
    which requires a TLBI, so it is empty
  - `(tcache2 ; ...) & ...`
    which requires a TLBI, so it is empty
- `[M] ; iio^-1 ; obtlbi_translate`
  to which the same reasoning applies

□

C.2.2.2 Step 2: Consistency

Lemma 3 *If* `Gtr` *satisfies the VA anti-abstraction condition, then translation-internal is acylic.*

Proof: `po-pa; [W]; trf` is empty
because by the VA anti-abstraction condition there are no non-initial writes to page tables. □

So we only need to show `external` is acyclic.

Lemma 4 (ob-to-T) *If* `G` *satisfies the VA anti-abstraction condition, then, for all* $n \geq 1$*,*

```
imm(ob)^n ; [T] ==
  iio
  | imm(ob)^(n-1) ; trfe
  | imm(ob)^(n-1) ; [T] ; iio ; [T]
  | imm(ob)^(n-1) ; [CSE] ; instruction-order
  | imm(ob)^(n-1) ; po ; [ERET] ; instruction-order ; [T]
```

Proof:

- The `addr` clause
  `| tdata ; [T_f]`
  is empty because there are no translation failures.
- `tob` does not contribute: there are no faults, and no non-initial writes to page table entries.
- The first clause of `ctxob` is empty because there are no `MSR TTBR`. The third and fourth are also empty, because they do not end in a `[T]`.

- Given a static injective mapping, the new `| (addr | data | ctrl) ; trfi` clause of `dob` is empty.

□

Lemma 5 (no-cycle-ob-to-init) *If* `Gtr` *is well-formed and consistent (in either model), then there is cycle in* `ob` *via the initial writes.*

Proof: By well-formedness, `wco ; [INIT] = [INIT] ; wco ; [INIT]`, and `wco` is acyclic.
By examination of the other edges. □

Lemma 6 (ob-from-T) *If* `Gtr` *satisfies the VA anti-abstraction condition, then*

```
[T] ; imm(ob) ==
  iio
  | [T] ; iio ; [M] ; po ; [W]
```

Proof: By examination of the edges. □

Lemma 7 (instruction-order-compress)
`instruction-order ; [T] ; iio ; [M] ; po` $\subseteq$ `instruction-order`

Proof: If we unfold the definitions of `instruction-order` and `po`, we have
`iio^-1 ; fpo ; iio ; [T] ; iio ; [M] ; [M|F|C] ; iio^-1 ; fpo ; iio ; [M|F|C]`
which we can simplify into
`iio^-1 ; fpo ; fpo ; iio ; [M|F|C]`
which means we have
`instruction-order`. □

Lemma 8 (instruction-order-compress-iio) `instruction-order ; iio ; po` $\subseteq$ `instruction-order`

Proof: `iio` is transitive, and is the RHS of `instruction-order`. □

Lemma 9 (ob-acyclic-preserved) *If* `G` *satisfies the VA anti-abstraction condition, if there is a cycle in translate-*`ob`*, then there is a cycle in plain-*`ob`*.*

Proof:

Consider a minimal cycle in translate-`imm(ob)` (that is, the transitive closure of the `ob` of the model with translation). Let n be its length.
We show that there is a cycle in plain-`ob`.
Assume, for contradiction, that the cycle contains an edge that is not in plain-`ob` (that is, the `ob` of the model without translation):

- iio
  by case split:
  - `[T] ; iio; [M]`: by Lemma ob-to-T, the `ob` edge to the left has to be either
    - `iio` in which case, by transitivity of `iio`, there is a shorter cycle, so we have a contradiction.
      Let us call this Case IIOtrans.
    - `trfe`, which is from an initial write by the VA abstraction condition,
      but by Lemma no-cycle-ob-to-init, the cycle cannot exist.

    * `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]`
      then we have `imm(ob)^(n-2); [T]; iio; [M]`, which involves one fewer translate, so we have a contradiction.
    * `imm(ob)^(n-2) ; [CSE] ; instruction-order`
      This is similar to IIOtrans.
    * `imm(ob)^(n-2) ; po ; [ERET] ; instruction-order ; [T]`
      This is similar to IIOtrans.
  - `[T] ; iio ; [T]`:
    So the whole cycle looks like `imm(ob)^(n-1) ; [T] ; iio ; [T]`
    By Lemma ob-to-T, we have either
    * `imm(ob)^(n-2) ; iio ; [T] ; iio ; [T]`
      See Case IIOtrans.
    * `imm(ob)^(n-2) ; trfe`
      the `trfe` is from an initial write by the VA abstraction condition, and by Lemma no-cycle-ob-to-init, the cycle cannot exist.
    * `imm(ob)^(n-2); [T]; iio; [T]`
      but we already have `iio` to the second T, so we have a cycle involving one fewer translate, so we have a contradiction.
    * `imm(ob)^(n-2) ; [CSE] ; instruction-order`
      This is similar to IIOtrans.
    * `imm(ob)^(n-2) ; po ; [ERET] ; instruction-order ; [T]`
      This is similar to IIOtrans.

- `tob` has
  - `[T_f] ; tfr`
    which has a fault, so we have a contradiction.
  - `([T_f] ; tfri) & (po ; [dsb.sy] ; instruction-order)^-1`
    which has a fault, so we have a contradiction.
  - `speculative ; trfi` which is empty, because of the static page table.
- `obtlbi`, which is empty by Lemma obtlbi-empty.
- `ctxob` has
  - `speculative ; [MSR TTBR]`
    by the VA abstraction condition, there is no MSR TTBR
  - `[CSE] ; instruction-order`
    So the whole cycle looks like
    `[CSE] ; instruction-order ; imm(ob)^(n-1)`
    Because `instruction-order` is acyclic, $n \geq 1$, so we have
    `[CSE] ; instruction-order ; imm(ob) ; imm(ob)^(n-2)`
    By Lemma ob-from-T, we have either:
    * `[CSE] ; instruction-order ; iio ; imm(ob)^(n-2)`
      which means that by Lemma instruction-order-compress, we have
      `[CSE] ; instruction-order ; imm(ob)^(n-2)`
      so we have a cycle involving one edge fewer, so we have a contradiction.
    * `[CSE] ; instruction-order ; [T] ; iio ; [M] ; po ; [W] ; imm(ob)^(n-2)`
      which means that by Lemma instruction-order-compress, we have
      `[CSE] ; instruction-order ; imm(ob)^(n-2)`
      so we have a cycle involving one edge fewer, so we have a contradiction.

- `[ContextChange] ; po ; [CSE]`
  by the VA abstraction condition, there is no `ContextChange`.
- `speculative ; [CSE]`
  The `CSE` has to be an `ISB`, because there are no exceptions, and the `speculative` is either in `dob` in the plain model, so we have a contradiction, or in `[T]; instruction-order`.
  So the whole cycle looks like `imm(ob)^(n-1) ; [T] ; iio ; [M] ; po ; [ISB]`
  Because `po | iio` is acyclic, $n-1$ has to be ≥ 1, so by Lemma ob-to-T, we have either
  - `imm(ob)^(n-2); iio; [T]; iio; [M]; po; [ISB]`
    See Case IIOtrans.
  - `trfe`, which is from an initial write by the VA abstraction condition,
    but by Lemma no-cycle-ob-to-init, the cycle cannot exist
  - `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]; po; [ISB]`
    but we already have `iio` to the second T,
    so we have a cycle involving one fewer translate,
    so we have a contradiction.
  - `imm(ob)^(n-2); [CSE] ; instruction-order ; [T] ; iio ; [M] ; po ; [ISB]`
    which means that by Lemma instruction-order-compress, we have
    `imm(ob)^(n-2); [CSE] ; instruction-order`
    so we have a cycle involving one edge fewer,
    so we have a contradiction.
  - `imm(ob)^(n-2) ; po ; [ERET] ; instruction-order ; [T] ; iio ; [M] ; po ; [ISB]`
    is similar
- `po ; [ERET] ; instruction-order ; [T]`
  So the whole cycle looks like
  `po ; [ERET] ; instruction-order ; [T] ; imm(ob)^(n-1)`
  Because instruction-order is acyclic, $n \geq 1$, so we have
  `po ; [ERET] ; instruction-order ; [T] ; imm(ob) ; imm(ob)^(n-2)`
  By Lemma ob-from-T, we have either:
  - `po ; [ERET] ; instruction-order ; [T] ; iio ; imm(ob)^(n-2)`
    which means that by Lemma instruction-order-compress-iio, we have
    `po ; [ERET] ; instruction-order ; imm(ob)^(n-2)`
    so we have a cycle involving one edge fewer, so we have a contradiction.
  - `po ; [ERET] ; instruction-order ; [T] ; ([T] ; iio ; [M]; po ; [W]) ; imm(ob)^(n-2)`
    which means that by Lemma instruction-order-compress, we have
    `po ; [ERET] ; instruction-order ; imm(ob)^(n-2)`
    so we have a cycle involving one edge fewer, so we have a contradiction.

- extended `dob`:
  - involving `trfi` from non-initial writes, which contradicts our assumption about static translation.
  - or `[T] ; instruction-order ; [W]`,
    so `[T] ; iio ; [M] ; po ; [W]`
    So the whole cycle looks like `imm(ob)^(n-1) ; [T] ; iio ; [M] ; po ; [W]`
    Because `po | iio` is acyclic, $n-1$ has to be ≥ 1, so by Lemma ob-to-T, we have either

    * `imm(ob)^(n-2); iio; [T]; iio; [M]; po; [W]`
      See Case IIOtrans.
    * `trfe`, which is from an initial write by the VA abstraction condition,
      but by Lemma no-cycle-ob-to-init, the cycle cannot exist
    * `imm(ob)^(n-2); [T]; iio; [T]; iio; [M]; po; [W]`
      but we already have `iio` to the second T,
      so we have a cycle involving one fewer translate,
      so we have a contradiction.
    * `imm(ob)^(n-2); [CSE] ; instruction-order ; [T] ; iio ; [M] ; po ; [W]`
      which means that by Lemma instruction-order-compress, we have
      `imm(ob)^(n-2); [CSE] ; instruction-order`
      so we have a cycle involving one edge fewer,
      so we have a contradiction.
    * `imm(ob)^(n-2) ; po ; [ERET] ; instruction-order ; [T] ; iio ; [M] ; po ; [W]`
      is similar

- extended `bob`, but only involving TLBI, which contradicts our assumption of no TLBI.
- extended `obs`, but only involving `trfe`, by the VA abstraction condition, the only writes to page tables are from initial writes, and by Lemma no-cycle-ob-to-init, there are no `ob` cycles via initial writes, so there is no cycle.
- `obfault`, which involves a fault, which contradicts our assumptions.
- `obets`, which involves a fault or a TLBI, which contradicts our assumptions.

All the other edges are in plain-`ob` by definition. □

Theorem 3 (VA anti-abstraction) *If the translation extension condition holds, then there exists a* `Gtr` *that satisfies the VA anti-abstraction condition such that* `Gtr` *is a stitching of* `Gabs` *with the* `pt_initial_state` *according to* `translate` *in* `tr_ctxt` *and* `Gtr` *is consistent wrt. the model with translation.*

Proof: `Gtr` exists by the translation extension construction,
and it is consistent by Lemma ob-acyclic-preserved. □

D Test results

D.1 Isla model results

Here ✓ and × indicate whether or not the model allows an execution with satisfying the final-state constraint given in the test. All these are as intended.

	Strong model		Ets model	
Test Name	allow?	time	allow?	time
BBM+dsb-tlbiis-dsb	✓	12702ms	✓	19328ms
BBM.Tf+dsb-tlbiis-dsb	✓	9196ms	✓	13659ms
Break2.news1	×	43306ms	×	22677ms
CoRR0.alias+po	×	2512ms	×	1420ms
CoRR2.alias+po	×	1934ms	×	2260ms
CoRT.inv+addr-trfi	!	1098ms	!	4778ms
CoRpteT+dsb	✓	5978ms	✓	6586ms
CoRpteT+dsb-isb	✓	7347ms	✓	7604ms
CoRpteT.EL1+dsb-tlbi-dsb	✓	10293ms	✓	11815ms
CoRpteT.EL1+dsb-tlbi-dsb-isb	×	53579ms	×	28345ms
CoRpteTf.inv+dsb	✓	5413ms	×	2582ms
CoRpteTf.inv+dsb-isb	×	3347ms	×	3472ms
CoTRpte.inv+dsb	×	5781ms	×	5372ms
CoTRpte.inv+dsb-isb	×	4436ms	×	5367ms
CoTRpte.inv+po	✓	7044ms	✓	6839ms
CoTT.ro+dmb	✓	36785ms	✓	52018ms
CoTT.ro+dsb-isb	✓	88419ms	✓	103258ms
CoTT.ro+po	✓	38533ms	✓	38279ms
CoTTf.inv+dsb-isb	×	21525ms	×	26198ms
CoTTf.inv+po	✓	51495ms	×	29031ms
CoTW1.inv	×	2597ms	×	2714ms
CoTWinv	×	2846ms	×	3277ms
CoTfRpte+dsb	×	4134ms	×	4433ms
CoTfRpte+dsb-isb	×	5543ms	×	5852ms
CoTfRpte+eret	×	2995ms	×	4168ms
CoTfRpte+po	×	3872ms	×	3740ms
CoTfT+dsb-isb	✓	78905ms	✓	95644ms
CoTfT+po	✓	71605ms	✓	72598ms
CoTfW.inv+dsb-isb	×	5572ms	×	8392ms
CoTfW.inv+po	×	6038ms	×	7738ms
CoTfW1.inv+si	!	220ms	!	227ms
CoWR.alias	×	977ms	×	967ms
CoWR.inv	×	825ms	×	1104ms
CoWTa1.1.inv+dsb-tlbiasidis-dsb-eret	×	4653ms	×	3693ms
CoWTa2.1.inv+dsb-tlbiasidis-dsb-eret	×	2833ms	×	3180ms
CoWTf.inv+[dmb]-dmb-addr	×	3845ms	×	5057ms
CoWTf.inv+dsb-isb	×	3024ms	×	2475ms
CoWTf.inv+po	✓	4112ms	✓	4160ms
CoWTf.inv+po-ctrl+po	✓	25871ms	✓	24818ms
CoWTf.inv+po-ctrl-isb+po	✓	64271ms	✓	70941ms
CoWTf.inv+poloc-ctrl-isb	✓	7161ms	✓	8420ms
CoWTf.inv+rfi-addr	✓	6419ms	✓	6075ms
CoWTf.inv+rfi-ctrl-isb	✓	8242ms	✓	8255ms

CoWTf.inv+svc	✓	6843ms	✓	7037ms
CoWTf.inv.EL1+dsb-eret	×	2425ms	×	2073ms
CoWTf.inv.EL1+dsb-svc	×	4906ms	×	4442ms
CoWTf.inv.EL1+eret	✓	4684ms	✓	5627ms
CoWTf.inv.EL2+dsb-tlbiipa-dsb-tlbiis-dsb-eret	×	221283ms	×	5688ms
CoWTf.inv.EL2+po	✓	7031ms	✓	8213ms
CoWTv2.2.inv+dsb-tlbivmidis-dsb-eret	×	5223ms	×	4592ms
CoWW.alias	×	822ms	×	849ms
CoWinvRpte+po	×	1024ms	×	938ms
CoWinvT+dsb-isb	✓	5292ms	✓	5212ms
CoWinvT+po	✓	5406ms	✓	5832ms
CoWinvT.EL1+dsb-tlbi-dsb	✓	10117ms	✓	12882ms
CoWinvT.EL1+dsb-tlbi-dsb-isb	×	18380ms	×	25267ms
CoWinvT.EL1+dsb-tlbiis-dsb	✓	11490ms	✓	12759ms
CoWinvT.EL1+dsb-tlbiis-dsb-isb	×	7548ms	×	12986ms
CoWinvT2+dsb-tlbiipa-dsb-eret	×	12608ms	×	4645ms
CoWinvT2+dsb-tlbiipa-dsb-tlbivmall-dsb-eret	×	114842ms	×	196759ms
CoWinvTa1.1+dsb-tlbiasidis-dsb-eret	×	7638ms	×	99884ms
CoWinvTa2.1+dsb-tlbiasidis-dsb-eret	✓	55920ms	✓	9894ms
CoWinvTv1.2+dsb-tlbivmidis-dsb-eret	×	13931ms	×	8229ms
CoWinvTv2.2+dsb-tlbivmidis-msrvttbr-dsb-eret	✓	10489ms	✓	20604ms
ETS	×	9590ms	×	9297ms
IRIW.TTf.TTf.inv+addrs	×	132223ms	×	147911ms
ISA2.RRTf.inv+dsb+addr+addr	×	4464ms	×	4866ms
ISA2.RRTf.inv+dsb+data+addr	×	5082ms	×	5085ms
LB+addr-trfis	×	7042ms	×	7977ms
LB+data-trfis	×	26579ms	×	18056ms
LB.TT.inv+pos	×	15481ms	×	16545ms
Load.inv	—	—	×	3345ms
MP.BBM1+dsb-tlbiis-dsb-dsb+ctrl-isb	×	1068905ms	×	571232ms
MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb	×	116116ms	!	182456ms
MP.RT.EL1+dsb+dsb-tlbi-dsb+dsb+dsb-isb	✓	146008ms	✓	208363ms
MP.RT.EL1+dsb+dsb-tlbiis-dsb+dsb+dsb-isb	×	307568ms	×	760981ms
MP.RT.EL1+dsb-shootdown-dsb+dsb-isb	×	765865ms	×	49466ms
MP.RT.EL1+dsb-tlbi-dsb+dsb-isb	✓	10678ms	✓	87237ms
MP.RT.EL1+dsb-tlbiis-dsb+dmb	×	57372ms	×	5360ms
MP.RT.EL1+dsb-tlbiis-dsb+dsb-isb	×	348942ms	×	16656ms
MP.RT.EL2+dsb-tlbiipa-dsb-tlbiis-dsb+dsb-isb	✓	108556ms	✓	87273ms
MP.RT.EL2+dsb-tlbiipais-dsb+dsb-isb	✓	112919ms	✓	120764ms
MP.RT.EL2+dsb-tlbiipais-dsb-tlbiis-dsb+dsb-isb	✓	145150ms	×	435166ms
MP.RT.EL2+dsb-tlbiis-dsb-tlbiipais-dsb+dsb-isb	✓	146011ms	✓	97074ms
MP.RT.inv+dmb+addr-po-isb	×	3366ms	×	3223ms
MP.RT.inv+dmb+addr-po-msr	×	3230ms	×	3146ms
MP.RT.inv+dmb+addr-po-msr-isb	×	4687ms	×	3356ms
MP.RT.inv+dmb+addr-trfi	—	—	×	7235ms
MP.RT.inv+dmb+ctrl-trfi	×	6011ms	×	6144ms
MP.RT.inv+trfi-data+addr.toml	✓	40551ms	✓	32442ms
MP.RTT.EL1+dsb-tlbiis-tlbiis-dsb+dsb-isb	×	520042ms	×	305092ms
MP.RTf.inv+dmb+addr	×	3574ms	×	4327ms
MP.RTf.inv+dmb+ctrl-isb	×	5734ms	×	5059ms
MP.RTf.inv+dmb+data	×	3232ms	×	3843ms

MP.RTf.inv+dmb+dsb-isb	×	5345ms	×	3982ms
MP.RTf.inv+dmb+po	✓	5316ms	×	2598ms
MP.RTf.inv+dmbs	✓	5896ms	×	3681ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+addr	×	3994ms	×	3882ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl	×	15261ms	×	6025ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl-isb	×	31160ms	×	8117ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+data	×	3744ms	×	5394ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+dmb	×	10762ms	×	5294ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb	×	52965ms	×	6204ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+po	×	20910ms	×	3710ms
MP.RTf.inv.EL1+dsb-tlbiis-dsb+poap	×	75710ms	×	3279ms
MP.TR.inv+dmb+isb	×	2406ms	×	3016ms
MP.TR.inv+dmb+msr	✓	6174ms	✓	6987ms
MP.TR.inv+dmb+msr-isb	×	4042ms	×	4240ms
MP.TTf.inv+dmb+addr	×	12067ms	×	16288ms
MP.TTf.inv+dmb+dsb-isb	×	27192ms	×	47972ms
MP.TTf.inv+dmb+po	✓	52201ms	×	9113ms
MP.TTf.inv+dsb+ctrl-isb	×	30317ms	×	37812ms
MP.TTf.inv+dsb+dsb-isb	×	29708ms	×	22505ms
MP.TTf.inv+dsb+po	✓	42061ms	×	23415ms
MP.TTf.inv+dsbs	✓	64484ms	×	28800ms
MP.TTf.inv.EL1+dsb-tlbiis-dsb+dmb	×	235651ms	×	81641ms
MP.TTf.inv.EL1+dsb-tlbiis-dsb+dsb-isb	×	16436ms	×	29491ms
MP.TTf.inv.EL1+dsb-tlbiis-dsb+po	×	162552ms	×	200958ms
MP.TfR+dmb+eret	×	3508ms	×	4473ms
MP.alias3+rfi-data+dmb	✓	1431ms	✓	1483ms
PPOAA.alias	×	2645ms	×	3113ms
PPOCA.alias	✓	4744ms	✓	4932ms
PPODA.RT.inv	×	5779ms	×	6596ms
R.RTf.inv.EL1+dsb-tlbiis-dsb+popl	×	13739ms	×	4164ms
R.TR.inv+dmb+trfi	✓	8652ms	✓	9603ms
R.inv	×	1976ms	×	2050ms
RBS+dsb-tlbiis-dsb	×	4158ms	×	12912ms
RDW.alias	×	2972ms	×	3317ms
ROT.inv+dmbst	×	3998ms	×	3919ms
ROT.inv+dsb	×	2805ms	×	3202ms
ROT.inv+po	✓	4749ms	✓	4899ms
RSW.alias	✓	2765ms	✓	2609ms
RWC.RTfR.inv+addr+dmb	×	3022ms	×	3609ms
S+tlbiall+po	✓	3838ms	✓	3787ms
S.RT.ro+dsb-tlbiis-dsb+dsb-isb	×	36657ms	×	80116ms
S.RTW+tlbiall+addr-po	—	—	✓	4452ms
S.RTf.inv.EL1+dsb-tlbiis-dsb+ctrl	×	12958ms	×	4463ms
S.RTf.inv.EL1+dsb-tlbiis-dsb+data	×	4120ms	×	4535ms
S.RTf.inv.EL1+dsb-tlbiis-dsb+dmb	×	11481ms	×	4349ms
S.RTf.inv.EL1+dsb-tlbiis-dsb+poap	×	45733ms	×	3117ms
S.RTf.inv.EL1+dsb-tlbiis-dsb+popl	×	28716ms	×	4266ms
S.T+dmb+po	×	6152ms	×	4485ms
S.T.alias+tlbiall+po	—	—	×	17937ms
SB.TfTf.inv+dmb-ctrl-isbs	×	19569ms	×	16904ms
SB.TfTf.inv+dsb-isbs	×	9504ms	×	8368ms

SB.TfTf.invv+rfi-ctrl-isbs	✓	34795ms	×	15059ms
SwitchTable.same-asid+eret.toml	×	1245ms	×	1294ms
W	✓	663ms	✓	694ms
W+T	✓	3748ms	✓	10144ms
WBM+dsb-tlbiis-dsb	×	5104ms	×	4658ms
WDS+dsb-tlbiipa-dsb-eret-po	✓	59690ms	✓	216640ms
WDS+dsb-tlbiipa-dsb-po-eret	×	61832ms	×	9009ms
WDS+po-dsb-tlbiipa-dsb-eret	✓	267478ms	✓	25136ms
WDS+po-dsb-tlbiipa-dsb-po-eret	✓	146279ms	✓	30736ms
WDS+po-dsb-tlbiipa-dsb-tlbiis-dsb-eret	×	186945ms	×	576750ms
WRC.RRTf.inv+addrs	×	4100ms	×	4411ms
WRC.RRTf.inv+dmbs	✓	6441ms	×	3589ms
WRC.RRTf.inv+dsb+ctrl-isb	×	5218ms	×	5459ms
WRC.RRTf.inv+dsb+dsb-isb	×	5393ms	×	5591ms
WRC.RRTf.inv+dsbs	✓	8964ms	×	4568ms
WRC.RRTf.inv+pos	✓	5334ms	✓	5239ms
WRC.TRTf.inv+addrs	×	10005ms	×	11073ms
WRC.TRTf.inv+dmbs	✓	35492ms	×	8559ms
WRC.TRTf.inv+dsb+dsb-isb	×	9146ms	×	16960ms
WRC.TRTf.inv+dsbs	✓	48023ms	×	8487ms
WRC.TRTf.inv+pos	✓	26642ms	×	9414ms
WRC.TTTf.inv+addrs	×	31851ms	×	23959ms
WRC.TTTf.inv+data+addr	×	49883ms	×	40834ms
WRC.TfRR+ctrl-isb+dsb	×	5699ms	×	6787ms
WRC.TfRR+dmbs	×	4533ms	×	4800ms
WRC.TfRR+dsb-isb+dsb	×	5353ms	×	5963ms
WRC.TfRR+dsbs	×	3850ms	×	5423ms
WRC.TfRR+po+dsb	×	3508ms	×	3727ms
WRC.TfRR+pos	✓	5616ms	✓	7637ms
WRC.TfRT+dsb-tlbiis-dsb+dsb-isb	×	549315ms	×	1582501ms
WRC.TfRT+po+dsb-isb	✓	39075ms	✓	57628ms

Below are the pKVM tests. The two tests without results timed out after 4 hours.

Test Name	Strong model allow?	Strong model time
pKVM.create_hyp_mappings.inv.l2	×	2493ms
pKVM.create_hyp_mappings.inv.l3	×	807ms
pKVM.host_handle_trap.free_table.toml	×	15718265ms
pKVM.host_handle_trap .stage2_idmap.change_block_size	×	7010271ms
pKVM.host_handle_trap .stage2_idmap.change_block_size.change_permissions	×	2370057ms
pKVM.host_handle_trap.stage2_idmap.l3	×	166475ms
pKVM.host_handle_trap .stage2_idmap.l3.already_exists.concurrent	—	—
pKVM.host_handle_trap .stage2_idmap.l3.already_exists	—	—
pKVM.host_handle_trap_twice.stage2_idmap.l3	×	149172ms
pKVM.switch_to_new_table	×	1405ms
pKVM.vcpu_run	×	956ms
pKVM.vcpu_run.same_vm	×	3107ms
pKVM.vcpu_run.update_vmid	×	1817ms
pKVM.vcpu_run.update_vmid.concurrent	×	8604ms

D.2 Hardware results

Below is a table of our results from running our hand-written hardware tests on the various machines we have available: a Raspberry Pi 4; a Raspberry Pi 3B+; and an AWS `m6g-metal` instance (claiming to be an A76). Our hardware test harness uses a different form of test to our Isla tooling; tests with the same name have manually-checked correspondence.

Type	Name	rpi4b			rpi3bp			graviton2		
		Total	Distribution		Total	Distribution		Total	Distribution	
pgtable	CoRT	964.72K/8M	60.30K/500K	± 31.30K/500K	520.06K/3M	86.68K/500K	± 13.08K/500K	2.29M/108M	10.61K/500K	± 14.22K/500K
pgtable	CoRT+dsb-isb	802.86K/8M	50.18K/500K	± 15.93K/500K	327.02K/3M	54.50K/500K	± 5.67K/500K	3.41M/108M	15.77K/500K	± 30.96K/500K
pgtable	CoTR	2.51M/8M	156.63K/500K	± 31.44K/500K	0/3M			21.70M/107.50M	100.92K/500K	± 21.38K/500K
pgtable	CoTR+addr	0/8M			1/3M	0.17/500K	± 0.37/500K	0/107.50M		
pgtable	CoTR+dmb	1/8M	0.06/500K	± 0.24/500K	0/3M			4/107.50M	0.02/500K	± 0.14/500K
pgtable	CoTR+dsb	2/8M	0.12/500K	± 0.33/500K	0/2.50M			5/107M	0.02/500K	± 0.15/500K
pgtable	CoTR+dsb-isb	1/8M	0.06/500K	± 0.24/500K	0/2.50M			1/107M	0.00/500K	± 0.07/500K
pgtable	CoTR.inv	3.63M/6.50M	279.43K/500K	± 62.37K/500K	0/2.50M			32.28M/43M	375.33K/500K	± 91.32K/500K
pgtable	CoTR.inv+dsb-isb	0/6.50M			0/2.50M			0/43M		
pgtable	CoTR1+dsb-dc-dsb-tlbi-dsb-isb	2/6.50M	0.15/500K	± 0.36/500K	0/2.50M			4/43M	0.05/500K	± 0.21/500K
pgtable	CoTR1+dsb-tlbi-dsb-isb	2/6.50M	0.15/500K	± 0.36/500K	0/2.50M			3/43M	0.03/500K	± 0.18/500K
pgtable	CoTR1.tlbi+dsb-isb	6/6.50M	0.46/500K	± 0.93/500K	1/2.50M	0.20/500K	± 0.40/500K	29/43M	0.34/500K	± 0.58/500K
pgtable	CoTT	0/6.50M			0/2M			0/43M		
pgtable	CoTW	0/1.50M			0/1.50M			0/10.50M		
pgtable	CoWT	3.77M/6.50M	289.86K/500K	± 19.23K/500K	1.85M/2M	462.33K/500K	± 53.82K/500K	22.64M/43M	263.28K/500K	± 22.56K/500K
pgtable	CoWT+dsb	3.76M/6.50M	289.17K/500K	± 19.74K/500K	995.06K/2M	248.76K/500K	± 159.86/500K	21.50M/43M	250.05K/500K	± 561.54/500K
pgtable	CoWT+dsb-isb	3.78M/6.50M	290.59K/500K	± 19.12K/500K	995.77K/2M	248.94K/500K	± 228.37/500K	21.50M/43M	250.04K/500K	± 521.08/500K
pgtable	CoWT+dsb-svc-tlbi-dsb	0/6.50M			0/2M			0/42.50M		
pgtable	CoWT.inv	10/6.50M	0.77/500K	± 1.42/500K	1.73M/2M	432.85K/500K	± 73.79K/500K	169/42.50M	1.99/500K	± 3.39/500K
pgtable	CoWT.inv+dmb	8/6.50M	0.62/500K	± 0.92/500K	69.38K/2M	17.35K/500K	± 16.32K/500K	42/42.50M	0.49/500K	± 0.85/500K
pgtable	CoWT.inv+dsb	1/6.50M	0.08/500K	± 0.27/500K	0/2M			57/42M	0.68/500K	± 1.00/500K
pgtable	CoWT.inv+dsb-isb	0/6.50M			0/2M			0/42M		
pgtable	CoWT1+dsb-tlbi-dsb	0/6.50M			0/2M			0/42.50M		
pgtable	CoWT1+dsb-tlbi-dsb-isb	0/6.50M			0/2M			0/42.50M		
pgtable	CoWinvT	4.17M/6.50M	320.63K/500K	± 98.49K/500K	1.79M/2M	447.54K/500K	± 88.94K/500K	26.81M/42M	319.17K/500K	± 105.14K/500K
pgtable	CoWinvT+dsb-isb	4.19M/6.50M	321.95K/500K	± 97.84K/500K	1.83M/2M	458.52K/500K	± 70.27K/500K	26.80M/42M	319.06K/500K	± 105.27K/500K
pgtable	CoWinvT1+dsb-tlbi-dsb	0/6.50M			0/2M			0/42M		
pgtable	CoWinvWT1+dsb-tlbi-dsb-dsb-isb	0/6.50M			0/2M			0/42M		
pgtable	ISA2.TRR+dmb+po+dmb	0/6.50M			0/2M			0/42M		
pgtable	MP.BBM1+[dmb.ld]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M			0/1.50M			0/437.50M		
pgtable	MP.BBM1+[dmb.ld]-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/198.50M			0/1.06G			0/129.50M		
pgtable	MP.BBM1+[po]-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/108.50M			0/1.50M			0/145.50M		
pgtable	MP.BBM1+dsb-isb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/6.50M			0/2M			52/135.50M	0.19/500K	± 0.43/500K
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb+dsb	1/6.50M	0.08/500K	± 0.27/500K	0/2M			7/42.50M	0.08/500K	± 0.31/500K
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb+dsb-isb	0/6.50M			0/2M			2/42.50M	0.02/500K	± 0.15/500K
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb	1/6M	0.08/500K	± 0.28/500K	0/2M			0/42.50M		
pgtable	MP.BBM1+dsb-tlbiis-dsb-dsb-isb+dsb-isb	2/6M	0.17/500K	± 0.37/500K	0/2M			3/42.50M	0.04/500K	± 0.24/500K
pgtable	MP.BBM1+po-dsb-tlbiis-dsb-isb-dsb-isb+dsb-isb	0/1M			0/1.50M			9/191.50M	0.02/500K	± 0.15/500K
pgtable	MP.BBM1.id+dsb-tlbiis-dsb-dsb+dsb-isb	10/6M	0.83/500K	± 0.37/500K	2/2M	0.50/500K	± 0.50/500K	87/42.50M	1.02/500K	± 0.15/500K
pgtable	MP.RT+svc-dsb-tlbi-dsb+dsb-isb	1/6M	0.08/500K	± 0.28/500K	0/2M			3/42M	0.04/500K	± 0.19/500K
pgtable	MP.RT+svc-dsb-tlbiis-dsb+dsb-isb	1/6M	0.08/500K	± 0.28/500K	0/2M			3/42M	0.04/500K	± 0.19/500K
pgtable	MP.RT.inv+dmb+addr	0/6M			0/2M			0/42M		
pgtable	MP.RT.inv+dmb+po	0/6M			6/1.50M	2.00/500K	± 2.16/500K	0/42M		
pgtable	MP.RT1+[dmb.ld]-dmb+dsb-isb	7.15K/6M	595.58/500K	± 290.84/500K	986/1.50M	328.67/500K	± 464.80/500K	1.26K/42M	15.01/500K	± 32.31/500K
pgtable	MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dmb	0/1M			0/1M			0/23M		
pgtable	MP.RT1+[dmb.ld]-dsb-isb-tlbiis-dsb-isb+dsb-isb	0/1M			0/1M			0/23M		
pgtable	MP.RT1+[dmb.ld]-dsb-tlbiis-dsb-isb+dmb	0/6M			0/1.50M			0/42M		
pgtable	MP.RT1+dc-dsb-tlbiall-dsb+dsb-isb	4/6M	0.33/500K	± 0.47/500K	1/1.50M	0.33/500K	± 0.47/500K	5/41.50M	0.06/500K	± 0.24/500K
pgtable	MP.RT1+dc-dsb-tlbiall-dsb-isb+dsb-isb	3/6M	0.25/500K	± 0.43/500K	0/1.50M			2/41.50M	0.02/500K	± 0.15/500K
pgtable	MP.RT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/6M			0/1.50M			4/41M	0.05/500K	± 0.22/500K
pgtable	MP.RT1+dsb-tlbi-dsb+dsb-isb	0/6M			0/1.50M			2/41M	0.02/500K	± 0.15/500K
pgtable	MP.RT1+dsb-tlbiall-dsb+dsb-isb	5/6M	0.42/500K	± 0.49/500K	0/1.50M			6/41M	0.07/500K	± 0.26/500K
pgtable	MP.RT1+dsb-tlbiallis-dsb+dsb-isb	3/6M	0.25/500K	± 0.60/500K	0/1.50M			2/41M	0.02/500K	± 0.15/500K
pgtable	MP.RT1+dsb-tlbiis-dsb+dsb-isb	1/6M	0.08/500K	± 0.28/500K	0/1.50M			1/41M	0.01/500K	± 0.11/500K
pgtable	MP.RT1+dsb-tlbiis-dsb-isb+dmb	0/6M			0/1.50M			1/41M	0.01/500K	± 0.11/500K
pgtable	MP.RT1+dsb-tlbiis-dsb-isb+dsb-isb	0/6M			0/1.50M			1/41M	0.01/500K	± 0.11/500K
pgtable	MP.RT1+dsb-tlbiis-dsb-tlbiis-dsb+dsb-isb	0/6M			0/1.50M			3/41M	0.04/500K	± 0.19/500K
pgtable	MP.TT+Winv-dmb-Winv+tpo	254.83K/6M	21.24K/500K	± 7.81K/500K	114.48K/1.50M	38.16K/500K	± 13.77K/500K	170.96K/41M	2.08K/500K	± 2.06K/500K
pgtable	MP.TT+dmb+dsb-isb	688.65K/5.50M	62.60K/500K	± 14.12K/500K	174.78K/1.50M	58.26K/500K	± 2.25K/500K	492.98K/41M	6.01K/500K	± 6.55K/500K
pgtable	MP.TT+dmb+tpo	843.79K/5.50M	76.71K/500K	± 12.64K/500K	157.80K/1.50M	52.60K/500K	± 5.28K/500K	480.31K/41M	5.86K/500K	± 5.24K/500K
pgtable	MP.TT.inv+dmb+dsb-isb	0/5.50M			0/1.50M			0/41M		
pgtable	MP.TT.inv+dmb+tpo	0/5.50M			0/1.50M			0/41M		
pgtable	MP.invRT+dsb+dsb-isb	871.53K/5M	87.15K/500K	± 33.34K/500K	101.75K/1.50M	33.91K/500K	± 10.14K/500K	1.78M/40.50M	21.99K/500K	± 19.45K/500K
pgtable	MP.invRT1+dsb-isb-tlbiis-dsb-isb+dsb-isb	0/5.50M			0/1.50M			1/41M	0.01/500K	± 0.11/500K
pgtable	MP.invRT1+dsb-tlbiis-dsb+dsb	0/5M			0/1.50M			2/41M	0.02/500K	± 0.15/500K
pgtable	MP.invRT1+dsb-tlbiis-dsb+dsb-isb	1/4.50M	0.11/500K	± 0.31/500K	0/1.50M			1/41M	0.01/500K	± 0.11/500K
pgtable	WRC.AT+ctrl+dsb	128.64K/4.50M	14.29K/500K	± 3.29K/500K	77.36K/1.50M	25.79K/500K	± 4.61K/500K	214.45K/40M	2.68K/500K	± 2.81K/500K
pgtable	WRC.TRR+addr+dmb	0/4.50M			0/1.50M			0/40M		
pgtable	WRC.TRR.inv+addrs	0/4.50M			0/1.50M			0/40M		
pgtable	WRC.TRT+addr+dmb	35.28K/4.50M	3.92K/500K	± 989.41/500K	32.50K/1.50M	10.83K/500K	± 5.05K/500K	103.16K/40M	1.29K/500K	± 802.83/500K
pgtable	WRC.TRT+dmbs	53.60K/4.50M	5.96K/500K	± 4.36K/500K	36.76K/1.50M	12.25K/500K	± 7.35K/500K	171.51K/40M	2.14K/500K	± 1.83K/500K
pgtable	WRC.TRT+dsb-isbs	18.80K/4.50M	2.09K/500K	± 878.54/500K	30.44K/1.50M	10.15K/500K	± 1.19K/500K	104.62K/39.50M	1.32K/500K	± 501.19/500K
pgtable	WRC.TRT.inv+addrs	0/4M			0/1.50M			0/38.50M		
pgtable	WRC.TRT.inv+dsb-isbs	0/4M			0/1M			0/38M		
pgtable	WRC.TRT.inv+po+addr	0/4M			0/1M			0/37.50M		
pgtable	WRC.TRT.inv+po+dmb	0/4M			0/1M			0/37M		
pgtable	WRC.TRT1+dsb-tlbiis-dsb+dmb	0/4.50M			0/1M			0/38M		
pgtable	WRC.TRT1+dsb-tlbiis-dsb+dsb-isb	0/4.50M			0/1M			0/38M		
aliasing	CoWR.alias	0/6M			0/1.50M			0/36M		
aliasing	MP+dmb-data+dmb	0/5M			0/1.50M			0/36M		
aliasing	MP.alias+dmbs	0/5M			0/1.50M			0/36M		
aliasing	MP.alias2+dmb-data+dmb	0/5M			0/1.50M			0/36M		
aliasing	MP.alias2+dmbs	0/3M			0/1.50M			0/19.50M		
aliasing	MP.alias2+po-data+dmb	2.23K/5M	222.60/500K	± 44.19/500K	3.17K/1.50M	1.06K/500K	± 107.89/500K	407.36K/36M	5.66K/500K	± 1.83K/500K
aliasing	MP.alias3+rfi-data+dmb	51/3M	8.50/500K	± 2.36/500K	16/1.50M	5.33/500K	± 2.05/500K	36.35K/19.50M	932.18/500K	± 337.68/500K
aliasing	SB.alias+dmbs	0/5M			0/1M			0/35.50M		
aliasing	WRC.alias2+addrs	0/4M			0/43M			0/19M		
aliasing	WRC.alias2+dmbs	0/4M			0/43M			0/18.50M		
cacheability	MP.NC+dsb-dc-dsb-dmb+dmb	138.80K/8M	8.67K/500K	± 1.69K/500K	364.97K/26M	7.02K/500K	± 555.37/500K	54.95K/25.50M	1.08K/500K	± 1.54K/500K
cacheability	MP.NC+po-dmb+dmb	345.33K/7.50M	23.02K/500K	± 6.66K/500K	642.90K/25.50M	12.61K/500K	± 1.14K/500K	333.55K/25.50M	6.54K/500K	± 1.09K/500K
cacheability	MP.NC1+dsb-tlbiis-dsb-dc-dsb-dmb+dmb	0/7.50M			0/25.50M			0/25.50M		
cacheability	MP.NC1+dsb-tlbiis-dsb-dmb+dmb	556/7.50M	37.07/500K	± 16.14/500K	482/25.50M	9.45/500K	± 3.23/500K	6/25.50M	0.12/500K	± 0.43/500K
cacheability	WR.NC+dsb	0/0			0/0			0/0		
cacheability	WR.NC+po	0/0			0/0			0/0		
cacheability	WR.WARA-NC+dsb	0/0			0/0			0/0		
cacheability	WR.WARA-NC+po	0/0			0/0			0/0		
cacheability	WWR.NC+po-po	0/0			0/0			0/0		
pmds	CoWT.L23+dsb-isb	11.45M/13M	440.53K/500K	± 17.12K/500K	6.73M/13.50M	249.08K/500K	± 384.26/500K	48.94M/84.50M	289.56K/500K	± 15.50K/500K
pmds	CoWT.L23+po	12.88M/13M	495.27K/500K	± 2.43K/500K	13.39M/13.50M	495.87K/500K	± 909.80/500K	80.61M/84.50M	476.97K/500K	± 42.40K/500K
pmds	CoWT1.L23+dsb-tlbi-dsb-isb	0/13M			0/13.50M			0/84.50M		
pmds	ROT+dsb-dsb	0/13M			0/13.50M			0/84.50M		
pmds	ROT+po-po	0/13M			0/13.50M			0/84M		
pmds	ROT1+dsb-dsb-tlbi-dsb	0/13M			0/13.50M			0/84M		
pmds	ROT1+dsb-dsb-tlbivaa-dsb	0/13M			0/13.50M			0/84M		
same_page	CoTT+dsb-popage	0/35.50M			0/31M			0/1.12G		
same_page	CoTT+po-popage	1/47M	0.01/500K	± 0.10/500K	0/43.50M			0/1.20G		
sysreg	WR.MAIR1+dsb-isb-dc-dsb	0/0			0/0			0/0		
sysreg	WR.MAIR1+dsb-isb-po	0/0			0/0			0/0		
sysreg	WR.MAIR1+dsb-tlbi-dsb-isb-dc-dsb	0/0			0/0			0/0		
sysreg	WR.MAIR1+dsb-tlbi-dsb-isb-po	0/0			0/0			0/0		
sysreg	WR.MAIR1+po-po	0/0			0/0			0/0		

References

[1] *Power ISA™ Version 2.07*. IBM, 2013.

[2] pKVM source. https://android-kvm.googlesource.com/linux/+/refs/heads/pkvm/arch/arm64/kvm/hyp/nvhe/, 2021. Accessed 2021-07-06.

[3] Allon Adir, Hagit Attiya, and Gil Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.

[4] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, New York, NY, USA, 1990. ACM.

[5] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[6] Jade Alglave and Luc Maranget. The `herd7` tool. http://diy.inria.fr/doc/herd.html/, 2019. Accessed 2019-07-08.

[7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM TOPLAS*, 36(2):7:1–7:74, July 2014.

[10] Eyad Alkassar, Ernie Cohen, Mark A. Hillebrand, Mikhail Kovalev, and Wolfgang J. Paul. Verifying shadow page table algorithms. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 267–270. IEEE, 2010.

[11] Eyad Alkassar, Ernie Cohen, Mikhail Kovalev, and Wolfgang J. Paul. Verification of TLB virtualization implemented in C. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, volume 7152 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2012.

[12] ARM Limited. ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest/, March 2017. B.a Armv8.1 EAC, v8.2 Beta. ARM DDI 0487B.a (ID0331117). 6354pp.

[13] Arm Limited. Arm architecture reference manual. Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest/, January 2021. G.a Armv8.7 EAC. ARM DDI 0487G.a (ID011921). 8538pp.

[14] 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. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[15] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *In Proc. 33rd International Conference on Computer-Aided Verification*, July 2021. Extended version available at https://www.cl.cam.ac.uk/~pes20/isla/isla-cav2021-extended.pdf.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.

[21] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of POPL 2012: The 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia)*, pages 509–520, 2012.

[22] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[23] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 467–481. ACM, 2017.

[24] William W. Collier. *Reasoning about parallel architectures*. Prentice Hall, 1992.

[25] Data61/CSIRO. Frequently asked questions on seL4: The proof. http://sel4.systems/Info/FAQ/proof.pml, accessed 2019-07-01, 2019.

[26] Will Deacon. The ARMv8 application level memory model. https://github.com/herd/herdtools7/blob/master/herd/libdir/aarch64.cat (accessed 2019-07-01), 2016.

[27] Will Deacon. Virtualization for the masses: Exposing KVM on Android. https://www.youtube.com/watch?v=wY-u6n75iXc, November 2020. KVM Forum Talk.

[28] Ulan Degenbaev. *Formal specification of the x86 instruction set architecture*. PhD thesis, Saarland University, 2012.

[29] 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.

[30] Jake Edge. KVM for Android. https://lwn.net/Articles/836693/, November 2020.

[31] 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 POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[32] 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 *The 44st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, pages 429–442, January 2017.

[33] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *J. Parallel Distributed Comput.*, 15(4):399–407, 1992.

[34] Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, University of Texas at Austin, 2016. https://repositories.lib.utexas.edu/handle/2152/46437.

[35] Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In *Provably Correct Systems*, pages 173–209. 2017.

[36] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2015.

[37] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016.

[38] 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.

[39] 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.

[40] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[41] Rafal Kolanski. *Verification of programs in virtual memory using separation logic*. PhD thesis, University of New South Wales, Sydney, Australia, 2011.

[42] Mikhail Kovalev. *TLB virtualization in the context of hypervisor verification*. PhD thesis, Saarland University, 2013.

[43] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3953–3970. USENIX Association, 2021.

[44] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified Linux KVM hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 839–856, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[45] Arm Limited. https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool, 2022. Accessed 2022-02-23.

[46] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Draft available from http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf, 2012.

[47] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009.

[48] Christopher Pulte. *The Semantics of Multicopy Atomic ARMv8 and RISC-V*. PhD thesis, University of Cambridge, 2019. https://doi.org/10.17863/CAM.39379.

[49] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2018.

[50] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1–15. ACM, 2019.

[51] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[52] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*, pages 311–322, 2012.

[53] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 175–186, 2011.

[54] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 379–391, January 2009.

[55] 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. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights).

[56] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in Armv8-A. In *Proceedings of ESOP 2022: 31st European Symposium on Programming, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany*, April 2022.

[57] 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 (extended version). In *Proceedings of the 29th European Symposium on Programming*, April 2020.

[58] Hira Syeda and Gerwin Klein. Reasoning about translation lookaside buffers. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pages 490–508, 2017.

[59] Hira Taqdees Syeda. *Low-level program verification under cached address translation.* PhD thesis, University of New South Wales, Sydney, Australia, 2019.

[60] Hira Taqdees Syeda and Gerwin Klein. Program verification in the presence of cached address translation. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 542–559, 2018.

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

[62] 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 *SOSP 2021: Proceedings of the 28th ACM Symposium on Operating Systems Principles*, October 2021.

[63] 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.

[64] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Tricheck: Memory model verification at the trisection of software, hardware, and ISA. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 119–133. ACM, 2017.

[65] Andrew Waterman and Krste Asanović, editors. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. December 2018. Document Version 20181221-Public-Review-draft. Contributors: Arvind, Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Roger Espasa, Shaked Flur, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David Horner, Bruce Hoult, Alexandre Joannou, Olof Johansson, Ben Keller, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Joseph Myers, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Christopher Pulte, Jose Renau, Colin Schmidt, Peter Sewell, Susmit Sarkar, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Andrew Waterman, Robert Watson, Derek Williams, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang.

[66] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. 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 190–204. ACM, 2017.

[67] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, 2004.