Relaxed virtual memory in Armv8-A

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 first.last@arm.com

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.

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 [57,59,58,60,44,38,41] and x86 [34,35,29,62], 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 [25,40,37,44,11,38,43,61] 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 §7).

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 (§5).
– 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 (§6.1).

– 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 (§6.2).

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); on the current Arm prose documentation [13]; on discussion with the pKVM development team; and on 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.

We begin in §2 with an informal introduction to virtual memory in a simple sequential setting, to make this self-contained. This paper is necessarily condensed; an extended version, with our tests, models, proofs, and Isla tooling, is available 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, 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:

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

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. Translation functions are not necessarily injective, and the full translate function has permissions per exception-level, and includes not just access permissions but additional fields for cacheability, shareability, security, contiguity, and others 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. Each non-leaf node of the tree has 512 64-bit entries, indexed by specific bit ranges of the virtual address. Each entry can be either *invalid*, meaning that the translate function is undefined for this part of the domain; a *block* (at levels 1 or 2) or *page descriptor entry* (at level 3), returning an output address and permissions; or a *table* (at levels 0, 1, or 2), with the physical (or intermediate physical) address of a next-level table with which to continue recursively.

This *translation-table walk* function is fully defined in the Arm ASL language.

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.

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

Armv8-A has various *exception levels* (ELs), 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 walk 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 214 hand-written Isla-compatible virtual-memory litmus tests that illustrate the issues, but to keep this concise we just give the main ideas here. In the extended version, we link to tests for each issue. 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.

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, 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.

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.

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.

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.

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 [56] and reads from persistent memory [51].

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.

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. 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 thereof: 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.

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.

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.

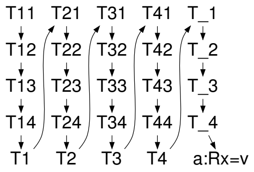
In reality, these six BBM failure cases include some where hardware may give such weakly constrained behaviour and others where, because coherence is over physical addresses and the mapping may be temporarily indeterminate, software might see well-defined but nondeterministic or surprising results. These 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. 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. 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.

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.

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).

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. Because invalid entries cannot be cached, this means that if an entry is initially invalid, 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.

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-satisfying cycles give rise to *thin air* behaviours and the architectural intent is to forbid them.

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.

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.

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`.

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.
2. P0 has DSB; TLBI; DSB, and P1 has any ordered-before relationship.

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

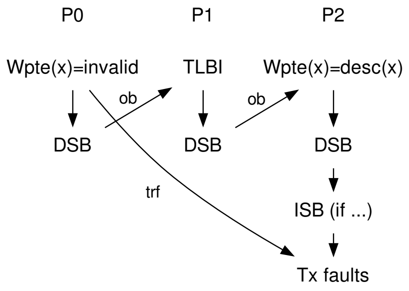
1. *all memory accesses generated by that PE using in-scope old translation information are complete.*

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."*.

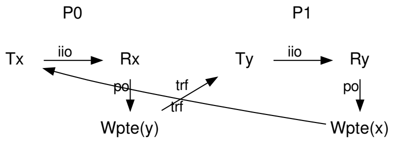
Typically the write of an invalid entry and TLBI would be on the same thread, but more generally, any shape as below should be forbidden, where `Tx` is a translation-walk read for an access of `x` and the `trf` relation shows the page-table write it reads from. In other words, the sequence ensures that 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.



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 DSB waiting for TLBI completion has to be on the same hardware thread as it.

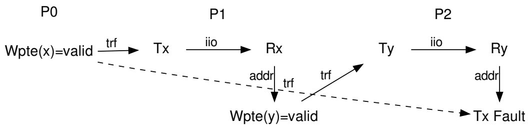
Translation-walk non-TLB reads from other-thread writes, forbidden future: Above we saw that translation-walk non-TLB reads should not read from po-later writes. How should that be generalised to multiple threads? For the simplest example, consider the translation version of the LB test on the right, 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.

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)].

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 others 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 for lack of space we cannot 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 of these, we give one test in detail, which also illustrates the general form of virtual memory litmus tests; the others are described in the extended version.

In the simplest case where pKVM is just switching from one virtual CPU (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, typeset (lightly hand-edited) from the

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; }
*pa2 = 1;
```

Initial state:

```
PSTATE.EL=0b10 // initial exception level is EL2
VBAR_EL2=0x1000 // exception vector base address
ELR_EL2=L0: // exception link register, to return to from EL2
SPSR_EL2=0b00101 // saved program status
TTBR0_EL1=ttbr(asid=0x00,base=vm1_stage1) // EL1 Stage 1
VTTBR_EL2=ttbr(vmid=0x0001,base=vm1_stage2) // Stage 2
TTBR0_EL2=ttbr(base=hyp_map,asid=0x00) // EL2
x0=ttbr(asid=0x00,base=vm2_stage1)
x1=ttbr(base=vm2_stage2,vmid=0x0002)
x3=x
```

Thread 0 (with pKVM source lines)

```
msr ttbr0_el1, x0   // kvm/hyp/sysreg—sr.h:96
msr vttbr_el2, x1   // include/asm/kvm_mmu.h:276
eret                // kvm/hyp/nvhe/host.S
L0:
ldr x2, [x3]        // in guest
```

Thread 0 EL2 handler

```
0x1400:
mov x2, #0
```

Final state: `0:x2=0`

TOML input format of our Isla tool (§6.1). 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. This starts running at EL2, as one can see from the `PSTATE.EL` register value. 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 `x2=0`, which could occur if the `ldr` translation used the old VM1 mapping instead of VM2's mapping. In this case that should not be allowed.

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`).

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: 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.

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 simply include a translation-read event for each read performed by architected translation-table walks, and define which writes each such translation-read can read from. 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 location coherence order `co`, the reads-from relation `rf` from writes to reads, the address, data, and control-dependency (`addr`, `data`, `ctrl`) subsets of `po`, and others. The base model is essentially the conjunction of 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; 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. We add translation data to events, including virtual, intermediate physical, and physical addresses (as determined by the translation regime). We add events for translation reads (**`T`**), TLB maintenance (**`TLBI`**), taking and returning from an exception (**`TE`** and **`ERET`**), and writing system registers (e.g. **`MSR TTBR`**). We modify the `loc` and `co` relations to relate events with the same *physical* address, and add a translation-reads-from `trf` to relate `W` to the `T` that read from it. To identify events with the same address we add `same-va` and `same-ipa` relations, relating events to the same virtual or intermediate physical address, and `same-{va,ipa}-page` for events in the same page. To identify events with the same address space or virtual machine ID, we use `same-vmid` and `same-asid`. The translate-read events within an instruction are related in the order they appear in the sequential ASL/Sail execution, both to each other and to any memory access or fault event, with the `iio` ("intra-instruction order") relation. We derive the `addr` relation from a new primitive `tdata` relation which relates read events to events that use that read value in the translation or computation of an address. 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 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`).

The model is in Fig. 1, in full except for the `tlb-affects` relation. 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.

```
let tlb-affects =
  (* see extended version *)

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
```

Fig. 1: Strong Model (with baseline Armv8-A model parts in gray)

Most of the changes to the model are in the `external` axiom, where we add several relations to ordered-before (`ob`): `iio` relates the intra-instruction events 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 `TLBI`s 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.

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. Coherence 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.

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. This is done with the `tcache` relations, which decides which TLBIs effect which translations by looking at the addresses each use and the `wco` ordering between the TLBIs and related writes.

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. We elide the full definition here, as it is simply the product of the enumeration of TLBI variants with the set of translations that match the exception level, stage, address, ASID or VMID given in the TLBI instruction. `obtlbi_translate` then uses `tlb-affects` and `wco` to order any translations that read-from 'stale' writes from before the invalidation with the TLBI that invalidated those writes. One notable subtlety here is in Stage 2 translations: since the TLB could store whole VA to PA mappings we must check that the correct Stage 1 invalidations have been performed, in addition to the Stage 2 ones, to be able to order the Stage 2 translation with the TLBI.

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 mentioned 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. 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`. This involves manually adding `[dmb] ; po ; [fault]`, `addr ; po ; [fault & FromW]`, etc. to `ob`. The `obETS` relation then orders translations which result in a translation fault after anything the fault is ordered-after.

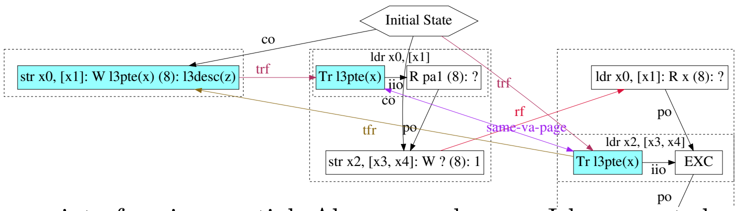
Metatheory: To establish that our models provide a simple and sound abstraction we prove three theorems: that for static injectively-mapped address spaces, any execution which is consistent in the model with translation, erasing translation events gives an execution that is consistent in the original 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.

6 Tooling

6.1 Isla-based model evaluation

Making relaxed-memory semantics exhaustively executable is essential for exploring their behaviour on examples [66,54,53,20,9,36,65,23,63,49,56]. 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].

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 214 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.

We also ran our model on an existing suite of "user" litmus tests, including 1927 additional generated tests, with a constant identity-mapped pagetable and checked the results match RMEM [31] and the official Armv8-A model [26,49,13].

6.2 Experimental testing of hardware

Validation of the models through experimental testing has been a vital part of past relaxed memory semantics [24,54,3,8]. This is equally true here. However experimental testing of the concurrent aspects of virtual memory is a far harder problem: these tests need to be able to access privileged parts of the instruction set; they need to be able to setup and use their own exception handlers, preventing building these tools ontop of standard distributions like Linux; Stage 2 tests and bare-metal Stage 1 tests require direct access to hardware, preventing the use of hypervisors such as KVM around the harness. To achieve this we build a harness that can run bare-metal on Armv8 devices to run Stage 1 (but as yet, not Stage 2) concurrent virtual memory litmus tests, which 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 with standard Arm cores (A53, A72). The data we collected suggests that in practice, aside from known errata, these cores: respect coherence over physical locations; correctly implement TLB maintenance; are multi-copy atomic w.r.t translation-table walks; and generally do not disagree with our model, except in one instance where we observed an anomalous result which is under discussion with Arm.

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.

7 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 [54,5,7,47,55,53,21,52,46,9,36,31,32,49,64].

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. [61] 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. [56] 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 [57,59,58,60], 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. [62], Kolanski [41], and Guanciale et al. [38].

8 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).

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 Ltd. Memory model tool. https://developer.arm.com/architectures/cpu-architecture/a-profile/memory-model-tool, January 2022. Accessed 2022-01-18.
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, 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.
57. 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.
58. Hira Taqdees Syeda. *Low-level program verification under cached address translation.* PhD thesis, University of New South Wales, Sydney, Australia, 2019.
59. 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.
60. Hira Taqdees Syeda and Gerwin Klein. Formal reasoning under cached address translation. *J. Autom. Reason.*, 64(5):911–945, 2020.
61. 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.
62. 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.
63. 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.
64. 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.
65. 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.

66. 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.