Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware

Runzhou Tao
Columbia University
New York, NY, USA
runzhou.tao@columbia.edu

Jianan Yao
Columbia University
New York, NY, USA
jianan@cs.columbia.edu

Xupeng Li
Columbia University
New York, NY, USA
xupeng.li@columbia.edu

Shih-Wei Li*
Columbia University
New York, NY, USA
shihwei@cs.columbia.edu

Jason Nieh
Columbia University
New York, NY, USA
nieh@cs.columbia.edu

Ronghui Gu
Columbia University
New York, NY, USA
ronghui.gu@columbia.edu

Abstract

Concurrent systems software is widely-used, complex, and error-prone, posing a significant security risk. We introduce VRM, a new framework that makes it possible for the first time to verify concurrent systems software, such as operating systems and hypervisors, on Arm relaxed memory hardware. VRM defines a set of synchronization and memory access conditions such that a program that satisfies these conditions can be mostly verified on a sequentially consistent hardware model and the proofs will automatically hold on relaxed memory hardware. VRM can be used to verify concurrent kernel code that is not data race free, including code responsible for managing shared page tables in the presence of relaxed MMU hardware. Using VRM, we verify the security guarantees of a retrofitted implementation of the Linux KVM hypervisor on Arm. For multiple versions of KVM, we prove KVM's security properties on a sequentially consistent model, then prove that KVM satisfies VRM's required program conditions such that its security proofs hold on Arm relaxed memory hardware. Our experimental results show that the retrofit and VRM conditions do not adversely affect the scalability of verified KVM, as it performs similar to unmodified KVM when concurrently running many multiprocessor virtual machines with real application workloads on Arm multiprocessor server hardware. Our work is the first machine-checked proof for concurrent systems software on Arm relaxed memory hardware.

*Now affiliated with National Taiwan University.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483560

CCS Concepts: • **Software and its engineering** → **Software verification**; *Virtual machines*; *Operating systems*; **Formal software verification**; • **Security and privacy** → *Logic and verification*; **Virtualization and security**.

Keywords: Formal methods; systems verification; relaxed memory; hypervisors; Arm; KVM

ACM Reference Format:
Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483560

1 Introduction

Concurrent systems software such as commodity operating system (OS) kernels and hypervisors form the backbone of safety-critical systems, the latter increasingly used to run virtual machines (VMs) on hosts in the cloud. Despite their importance, these multiprocessor systems are complex and error-prone, posing a significant security risk as large codebases contain many vulnerabilities. It is crucial to ensure the formal correctness of these commodity software systems.

Broadly speaking, a proof of correctness involves three components: (1) a specification, an abstract model of how the program is meant to behave which serves as the standard of correctness; (2) a hardware model, an abstract model of the hardware that executes the program which defines the machine interface with which a program may interact; and (3) an implementation, a concrete program definition representing the software we hope to verify. The proof needs to show that any behavior exhibited by the implementation running on the hardware model is captured by the specification. Its soundness ultimately rests on the accuracy of the hardware model. If the actual hardware exhibits behaviors beyond the specified hardware model, then any proof of correctness constructed upon that model cannot give any meaningful guarantees about the behavior of the real program.

Recent efforts [9, 10, 12, 21, 23, 25, 30, 34, 39, 53] prove the correctness of software systems over simple hardware models. For example, seL4 [30], Komodo [18], and Serval [43] assume a uniprocessor hardware model; it is unknown how their proofs can be extended to multiprocessor hardware. CertiKOS [22–24], AtomFS [56], and Mailboat [8] are multiprocessor systems, but are all verified using a sequentially consistent (SC) hardware model, which assumes that memory accesses behave as if the operations of all the CPUs were executed in some sequential order while preserving program order [33].

Unfortunately, this is not how real multiprocessor systems behave. To alleviate memory access bottlenecks, modern instruction set architectures such as Arm, RISC-V, and x86 support relaxed memory (RM) hardware, allowing CPUs to reorder memory accesses and execute out of program order. CPUs may observe RM effects and disagree about the order they observe each other accessing memory. Consider the following example with two threads:

Example 1 (Out-of-order write)**.**

PRE: `[x] = [y] = r0 = r1 = 0`

CPU 1	CPU 2
`(a) r0 := [x];`	`(c) r1 := [y];`
`(b) [y] := 1`	`(d) [x] := r1`

RM EXECUTION: `(b)` ⟶ `(c)` ⟶ `(d)` ⟶ `(a)`
POST: `r0 = r1 = 1`

On SC hardware, either `(a)` or `(c)` will be executed first, which means that at least one of `r0` and `r1` will be `0`. However, on RM hardware, both `r0` and `r1` can be `1`. Since the accessed address in `(b)` is independent of the one in `(a)`, `(b)` can be executed out-of-order before `(a)`, and the whole execution order can be `(b)` ⟶ `(c)` ⟶ `(d)` ⟶ `(a)`, which cannot occur on SC hardware. Using uniprocessor or SC multiprocessor models may make proofs for software systems tractable, but they fail to provide any guarantees about the behavior of these systems executing on real RM hardware. While more realistic hardware models have been proposed which model RM effects [19, 47, 48], they have not been shown to be practical or feasible to use for verifying real systems software.

There are some circumstances in which a program's behavior on SC hardware is equivalent to its behavior on RM hardware. If a program is data race free (DRF), i.e., no two threads can access one memory location at the same time, its behavior on SC and most RM hardware is the same [1]. However, this theorem is of limited utility in practice because it requires the entire program be DRF, including the lock implementations used to guarantee that the program is DRF, and locks are inherently not DRF as they are designed to allow concurrent reads of a lock value which may be updated.

A more useful notion is the local DRF condition, which allows a program to have some parts that are DRF and other parts that are not. If a program satisfies the local DRF condition, such as a program in which all shared objects are correctly protected by locks even though the locks themselves may not be DRF, its behavior on SC hardware is equivalent to its behavior on RM hardware that obeys certain architectural constraints [17, 27]. However, the local DRF condition must hold on RM hardware, not just an SC model. Using an SC model to attempt to prove that a program is local DRF so that an SC model can be used would be circular and insufficient to make any claims about whether the program is actually local DRF on RM hardware. Furthermore, although x86-TSO hardware satisfies the architectural constraints required for applying the local DRF condition, other architectures such as Power and Arm do not, the latter being particularly important given its dominance in mobile and embedded systems and its increasing use in personal computers and cloud computing. For those architectures, whether or not a program is local DRF provides no guarantee about whether its behavior is the same on SC and RM hardware.

To address this problem, we introduce VRM (Verification on Relaxed Memory), a new framework for verifying concurrent systems software on RM hardware. We focus on kernel code, code that runs at a higher privilege level, as used in OS kernels and hypervisors, given their central importance in protecting our computing infrastructure. VRM defines a set of six synchronization and memory access conditions, which we refer to as weak data race free (wDRF) conditions. VRM proves that kernel code that satisfies these conditions can be verified mostly on an SC hardware model and have the proofs hold when running on Arm RM hardware.

wDRF conditions can be thought of as a weaker notion of DRF which requires that the kernel code is DRF except for (1) data races in lock implementations bounded by memory barriers and (2) limited read/write races in page table implementations. The second relaxation of DRF is important in practice because kernel code is often responsible for managing page tables, which may be shared across multiple CPUs on multiprocessor hardware and concurrently read via MMU hardware while they are being updated by kernel code. To account for page table implementations in the presence of MMU hardware that may exhibit RM behaviors, the wDRF conditions include additional requirements regarding the kernel's own page table, shared page table writes, and TLB invalidations. The wDRF conditions further limit the propagation of RM behaviors from user programs to the kernel by requiring a degree of memory isolation between user and kernel memory. The wDRF conditions required by VRM must themselves hold on RM hardware.

VRM provides a multi-layer hardware model so that program properties that must be proven in the presence of RM behavior can be verified on a RM hardware model, which can then be lifted to an SC hardware model to simplify the rest of the program verification. VRM's bottom-layer RM operational model is Promising Arm [48], which has been previously shown to be equivalent to the Armv8 axiomatic model [47] in Coq. For kernel code which satisfies wDRF conditions, we prove that properties of the kernel code verified on

an SC model hold on the Promising Arm model, and therefore on Arm RM hardware.

We demonstrate the effectiveness of VRM on real software by using it to verify a retrofitted version of the Linux KVM hypervisor [29]. We build on our previous work on SeKVM [35–37], a verified KVM implementation for Arm, and show how VRM extends its proofs to hold in the presence of Arm RM behavior. Using VRM, we prove for the first time that a KVM implementation can guarantee the confidentiality and integrity of VMs on Arm RM hardware. We prove this for multiple implementations of KVM across multiple Linux kernel versions for multiple hardware configurations, showing that the wDRF conditions hold for all KVM versions. No changes to the verified KVM implementation or its proofs were required to use VRM. We measure the performance of the verified KVM versions and show that it performs comparably to unmodified KVM when running up to 32 multiprocessor VMs concurrently with real application workloads on Arm server hardware. These results show that even though verified KVM requires some modifications to KVM to prove its security guarantees, and satisfies the wDRF conditions so that those guarantees extend to RM hardware, neither of these requirements adversely affect its performance scalability on Arm multiprocessor hardware.

2 Relaxed Memory Behavior Bugs

To illustrate the challenges with verifying kernel code on RM hardware, we discuss six kinds of RM behavior bugs that can be verified to be correct using an SC model, showing that systems verified using an SC model cannot provide any meaningful guarantees about program behavior on RM hardware. Unlike SC hardware which preserves the program order, RM hardware can reorder instructions, for example, when the two instructions do not have data dependencies (i.e., value written to a register by one instruction is used as data by another), address dependencies (i.e., value written to a register by one instruction is used in an address calculation by another), coherence constraints (i.e., two instructions access the same memory location), or any explicit barrier between them.

Shared memory access synchronized via locks. Locks are commonly used for mutual exclusion on shared memory accesses. However, a lock implementation that is proven to be correct using an SC model may misbehave on RM hardware.

Example 2 (VM booting)**.** Figure 1 shows a `gen_vmid()` function used in a hypervisor to get the next unused VM identifier (VMID) when booting a new VM, which is synchronized using a standard ticket lock implementation. One can easily prove that this function is correct and well synchronized on an SC model—only one CPU at a time can execute the critical section (lines 11-14) and each new VM will have a unique `vmid`, assuming the number of VMs is less than `MAX_VM`.

Consider the following execution with two CPUs getting `vmid` simultaneously:

```
1  // ticket, now, and next_vmid are shared vars
2  void acquire_lock() {
3      u32 my_ticket = fetch_and_incr(ticket);
4      while (my_ticket != now) {};
5  }
6  void release_lock() {
7      now++;
8  }
9  u32 gen_vmid() {
10     acquire_lock();
11     u32 vmid = next_vmid;
12     if (vmid < MAX_VM)
13         next_vmid++;
14     else panic();
15     release_lock();
16     return vmid;
17 }
```

Figure 1. A ticket lock implementation that is correct on SC hardware, and its application to synchronize VM booting.

PRE: $my\_ticket_1 = 4$, $my\_ticket_2 = 5$, $now = 4$

CPU 1	CPU 2
`gen_vmid():`	`gen_vmid():`
`  acquire_lock():`	`  acquire_lock():`
`    …`	`    …`
`(a)  while(my_ticket!=now);`	`(d)  while(my_ticket!=now);`
`(b) vmid = next_vmid;`	`(e) vmid = next_vmid;`
`  …`	`  …`
`(c) release_lock();`	`  release_lock();`
`  return vmid;`	`  return vmid;`

RM EXECUTION: (a) → (b) → (e) → ⋯ → (c) → (d)
POST: $vmid_1 = vmid_2$

Suppose CPU 1's ticket is 4, CPU 2's ticket is 5, and the current ticket `now` is 4. One would expect that CPU 2 is trapped in the while loop until CPU 1 releases the lock, such that the returned `vmid` must be different for CPU 1 and CPU 2. However, the Arm architecture allows CPU 2 to predict that the loop condition `my_ticket != now` evaluates to `false`, and then execute `(e)` speculatively. It is possible that the speculative execution of CPU 2's `(e)` happens right after CPU 1 sets its `vmid` in `(b)`, before CPU 1 updates `next_vmid`, such that both CPU 1 and CPU 2 get the same `vmid`. CPU 2 may then execute the loop condition `(d)` after CPU 1 releases the lock in `(c)`. In this case, `now` becomes 5 and the previous prediction of the speculative execution of `(e)` can be validated and it will not reverted. In other words, with the implementation in Example 2, two VMs may incorrectly receive the same `vmid`, causing further problems in other hypervisor code that expects each VM to have a unique `vmid`.

Shared memory access synchronized via auxiliary variables. Shared memory access can also be synchronized using other methods such as auxiliary variables. However, they are harder to implement correctly on RM hardware.

Example 3 (VM context switch)**.** Figure 2 shows the VM context switch code in a hypervisor. An auxiliary variable `vcpu_state` maintains whether or not a virtual CPU (vCPU) of a VM is actively running on a physical CPU and can have value `ACTIVE` or `INACTIVE`. To stop running a vCPU, the hypervisor saves the context of a vCPU (lines 2-3) then sets the `vcpu_state`

```
1  void save_vm(void) {
2      // save the context of a vCPU
3      ...
4      u32 vmid = get_vmid();
5      u32 vcpuid = get_vcpuid();
6      vcpu_state[vmid][vcpuid] = INACTIVE;
7  }
8  void restore_vm(void) {
9      u32 vmid = get_vmid();
10     u32 vcpuid = get_vcpuid();
11     acquire_lock_vm(vmid);
12     if (vcpu_state[vmid][vcpuid] == INACTIVE)
13         vcpu_state[vmid][vcpuid] == ACTIVE
14     else panic();
15     release_lock_vm(vmid);
16     // restore the context of a vCPU
17     ...
18 }
```

Figure 2. An implementation of context switch in a hypervisor.

of the vCPU to `INACTIVE`. To start running a vCPU, the hypervisor first confirms that the `vcpu_state` is `INACTIVE`, in which case it sets the `vcpu_state` to `ACTIVE` and proceeds to restoring the context of the vCPU, Otherwise, the hypervisor panics.

On an SC model, this example can be verified to be correct, because the hypervisor only restores a vCPU's context and starts running the vCPU if the corresponding ownership variable is `INACTIVE`, and such an ownership variable can only be set to `INACTIVE` after the context is saved. Thus, the context is correctly restored to its previously saved state. On RM hardware, however, an older version of the vCPU context may be wrongly restored, as depicted in the following execution:

PRE: a vCPU running on CPU 1

CPU 1	CPU 2
in `save_vm()`	
`(a) //save vCPU context;`	`(c) restore_vm();`
`...`	
`(b) vcpu_state[vmid]` `[vcpuid] = INACTIVE;`	

RM EXECUTION: `(b)` → `(c)` → `(a)`
POST: Incorrect vCPU context restored by CPU 2

When CPU 1 executes `save_vm()`, the step to save the vCPU's context `(a)` can be reordered after the step `(b)` setting the ownership variable, since there is no data or address dependency between `(a)` and `(b)`. When CPU 2 invokes `restore_vm()`, the ownership variable may have already been set to `INACTIVE`, but the vCPU's context has not been saved by CPU 1 yet. The vCPU context restored by CPU 2 may then be an incorrect older version.

Shared page table access. Shared page tables can be accessed concurrently on multiple CPUs. They are used by OS kernels to support threads sharing the same address space, or hypervisors to support multiprocessor VMs. Even if all page table updates are well synchronized, shared page tables may still have data races due to MMU hardware. When a shared page table is updated by an OS kernel within a critical section on one CPU, threads using the same page table on other CPUs can simultaneously read the page table via MMU hardware during address translation. In other words, read/write data races are inevitable for shared page tables that may lead to unexpected bugs on RM hardware.

Example 4 (Out-of-order page table reads)**.** Consider two threads and a page table such that `0x10` and `0x11` are all-0 physical pages, `0x20` and `0x21` are all-1 physical pages, and variables `x` and `y` are in virtual pages `0x80` and `0x81`, respectively:

PRE: page mapping `0x80` ↦ `0x10`, `0x81` ↦ `0x11`

CPU 1	CPU 2
`(a) pte[0x80] := 0x20;`	`(c) r0 := [y];`
`(b) pte[0x81] := 0x21`	`(d) r1 := [x]`

RM EXECUTION: `(d)` → `(a)` → `(b)` → `(c)`
POST: `r0 = 1, r1 = 0`

On an SC model, the result "`r0 = 1, r1 = 0`" can never be produced because when `[y]` returns 1 in the instruction `(c)`, both remaps of the page table entries (PTEs) on CPU 1 must have completed, so `[x]` in `(d)` must also return value 1. On RM hardware, such a result is allowed by the execution order `(d)` → `(a)` → `(b)` → `(c)` because two reads in `(c)` and `(d)` are neither data nor address dependent and can be executed in an out-of-order manner.

Example 5 (Out-of-order page table writes)**.** Consider two threads and a two-level page table:

PRE: `y` is a leaf of `x`, `z` is an address in `y`

CPU 1	CPU 2
`(a) pgd[x] := EMPTY;`	`(c) r1 := [z]`
`(b) pte[y] := p`	

RM EXECUTION: `(b)` → `(c)` → `(a)`
POST: `(c)` reads values in physical page `p`

CPU 1 modifies the shared page table in the critical section by first unmapping the page directory (PGD) `x` and then setting the PTE `y`, a leaf entry of `x`, to some page `p`. Simultaneously, CPU 2 accesses the virtual address `z` mapped using `x` and `y`.

On an SC model, CPU 2 cannot access the physical page `p` because when `p` is mapped to `y` in the instruction `(b)`, the PGD `x` has already been unmapped. The page table walk of `(c)` either uses the old page table or triggers a page fault when visiting the empty PGD `x`. However, on RM hardware, the instruction `(b)` can be reordered before `(a)` since these two stores are not data or address dependent. Thus, there is a chance that the page walk of `(c)` on CPU 2 uses the updated PTE `y` before PGD `x` is unmapped and the physical page `p` is mistakenly accessed.

TLB management. A translation lookaside buffer (TLB) is a cache to speedup address translation. When maintaining page tables, OS kernels also need to maintain TLBs to be consistent with the page tables, i.e., the value in the TLB is either invalid, or the same as the value in the page table. Unexpected bugs may result from RM behavior during TLB management.

Example 6 (Out-of-order page table and TLB reads)**.** Consider two threads with a page table and TLB such that `0x10` is a physical page and `y` is an address in the virtual page `0x80`:

PRE: page mapping `0x80` ↦ `0x10`

CPU 1	CPU 2
`(a) pte[0x80] := EMPTY;`	`(c) r0 := [y];`
`(b) TLB invalidate 0x80`	⋯
	`(d) r1 := [y];`

RM EXECUTION: `(b)` → `(c)` → `(a)` → `(d)`
POST: CPU 2's TLB `0x80` ↦ `0x10`;
page mapping `0x80` ↦ `EMPTY`

On an SC model, after CPU 1 unmaps `0x80` and invalidates all the corresponding TLB entries, CPU 2 can no longer access the physical page `0x10` in future instructions such as `(d)`. However, on RM hardware, the execution order `(b)` → `(c)` → `(a)` can result in the TLB being inconsistent with the page table. In this order, `(b)` first invalidates TLB entries. When `(c)` is executed, CPU 2 finds that the TLB entry for `0x80` is invalid and will then get the mapping result from the page table, which is `0x10`. This step may insert the mapping `0x80` ↦ `0x10` back to CPU 2's TLB. Future instructions (e.g., `(d)`) executed on CPU 2 can then access physical page `0x10` through the TLB even after CPU 1 unmaps `0x80` in `(a)`.

Information flow between an OS kernel and user programs. OS kernels, and hypervisors, run with user programs, or VMs, that may run arbitrary code. Even if a kernel itself is well synchronized, it may still have unexpected behavior due to it accessing user programs' memory with RM behavior.

Example 7. Suppose CPU 1 and CPU 2 run the code in Example 1 as user programs, and then increase the value of `[z]` if `r0` or `r1` is 1. CPU 3 runs the following kernel code that has access to the user program's memory `[z]`:

PRE: `[x] = [y] = [z] = r0 = r1 = 0`

CPU 1 (User)	CPU 2 (User)	CPU 3 (Kernel)
`(a) r0 := [x];`	`(c) r1 := [y];`	`if [z] == 2:`
`(b) [y] := 1;`	`(d) [x] := r1`	`  r2 := 0;`
`  barrier;`	`  barrier;`	`else`
`  if r0 == 1:`	`  if r1 == 1:`	`  r2 := 1;`
`  [z] += 1;`	`  [z] += 1;`	`r3 := 1 / r2;`

RM EXECUTION: `(b)` → `(c)` → `(d)` → `(a)`
POST: Divide-by-zero error on CPU 3

`r2` is always set to 1 on an SC model because the branch condition can never be satisfied, but may become 0 on RM hardware and trigger a divide-by-zero error because both `r0` and `r1` can be 1. In other words, kernel code cannot be verified alone without considering how user program behavior may affect the kernel's execution.

3 The wDRF Conditions

As shown in Section 2, kernel code may misbehave on RM hardware even after being verified using an SC model. In other words, verifying kernel code with an SC model does not provide any meaningful guarantee of its behavior on real RM hardware, even though it may require far less proof effort than verifying the code directly using an RM model.

Our insight is to use the examples of buggy programs from Section 2 to define a set of synchronization and memory access conditions such that for kernel code satisfying these conditions, the guarantees verified using an SC model still hold on the Arm architecture. We can then reduce the proof effort on RM hardware to verifying that these conditions hold as opposed to having to directly verify all aspects of the code on RM hardware. To characterize this specific class of kernel programs, we introduce *weak data race free* (wDRF) conditions defined as the following six conditions:

1. (DRF-KERNEL) Shared memory accesses in the kernel are well synchronized (i.e., protected by synchronization methods) except for the implementation of synchronization methods and page table management.
2. (NO-BARRIER-MISUSE) Barriers are correctly placed in the kernel to guard critical sections and synchronization methods.
3. (WRITE-ONCE-KERNEL-MAPPING) If the kernel's own page table is shared, it can only be *written once*—only empty page table entries of the kernel can be modified.
4. (TRANSACTIONAL-PAGE-TABLE) Shared page table writes within a critical section are *transactional*. A series of shared page table writes is called *transactional* if, under arbitrary reordering of these writes, any page table walk can only see (1) the walking result before all page table writes, (2) the walking result after all page table writes occur in program order, or (3) a page fault.
5. (SEQUENTIAL-TLB-INVALIDATION) A page table unmap or remap must be followed by a TLB invalidation, with a barrier between them.
6. (MEMORY-ISOLATION) The memory space accessible by the kernel code should be partially isolated with user programs such that user programs cannot modify kernel memory, and either (1) the kernel code will not read user memory, or (2) the verification of the kernel on an SC model does not rely on the concrete implementation of user programs.

Section 4 proves that any behavior of a wDRF kernel implementation running over the Arm RM architecture can be captured by running the same kernel implementation over an SC model. To demonstrate that these conditions are useful for verifying real systems, Section 5 proves that an Arm implementation of the Linux KVM hypervisor satisfies the wDRF conditions, and can therefore be verified on an SC model. Before jumping to the proofs, we first discuss the intuition behind each condition and how these conditions can prevent RM behavior bugs.

The DRF-KERNEL condition is a commonly-used approach in writing kernel code to limit data races and RM behavior,

such as shown in Example 1. However, synchronization methods and page table modifications are often not DRF. The No-Barrier-Misuse condition is used to constrain synchronization methods. It guarantees that the code within a critical section cannot be reordered with the implementation of the synchronization method. This prevents well-synchronized programs from misbehaving on RM hardware, as shown in Example 2 and Example 3.

The Write-Once-Kernel-Mapping, Transactional-Page-Table, and Sequential-TLB-Invalidation conditions are used to constrain page table modifications. The Write-Once-Kernel-Mapping condition forbids RM behavior on a kernel's regular memory access due to out-of-order reads of the kernel's page table, as shown in Example 4. Kernel code for 64-bit systems often satisfies or approximates this condition by mapping all physical memory to the kernel page table at boot. The Transactional-Page-Table condition guarantees that page table writes will not result in any behavior on RM hardware that cannot be produced by SC models. Since data races over shared page tables are inevitable due to MMU hardware, page tables are often updated in practice according to this condition to ensure that any potential out-of-order page table writes cannot result in an undesirable page mapping that may leak secrets, as shown in Example 5. The Sequential-TLB-Invalidation condition precludes RM behavior in TLB management code. TLB invalidation is commonly performed according to this condition in practice to avoid undesirable TLB shootdown problems, such as shown in Example 6.

Finally, the Memory-Isolation condition forbids information flow from user to kernel via memory so that RM behaviors of user programs cannot be propagated to the kernel. The stronger version of the condition disallows kernel code from reading user memory, preventing the undesirable behavior shown in Example 7. This condition is simpler to prove, but may not strictly hold in practice for many systems. For example, the condition does not hold for an OS kernel that obtains arguments from user programs by reading their memory. The weaker version of the condition allows a kernel to read user programs' memory if the verification of the kernel on an SC model is independent of the user programs' implementation. If the proofs to verify the kernel on an SC model rely on the implementation of user programs, it may be the case that the user programs behave differently on RM hardware such that the guarantees proven on the SC model no longer hold on RM hardware. However, if the proofs are independent of the user programs' implementation, then we only need to show that the set of possible kernel behaviors on SC and RM models are the same. Note that the behavior of a kernel program P together with a user program Q on RM hardware may not be the same as the behavior of $P \cup Q$ on SC. However, this does not matter as long as the execution result can be captured by $P \cup Q'$ on SC for some other user program Q'. For example, in Example 7, the user program Q' can be a program that randomly sets `[z]` to be 0, 1, or 2. This weaker version of the condition is likely to hold for real kernels as they usually have no dependencies on user programs for their correct behavior. To distinguish between the stronger and weaker versions of the condition, we will use Memory-Isolation to refer to the former and Weak-Memory-Isolation to refer to the latter. We will initially use the Memory-Isolation condition, but then show in Section 4.3 how the Weak-Memory-Isolation condition can be used instead so that VRM can verify kernel programs that do not satisfy the Memory-Isolation condition.

Not all systems satisfy the wDRF conditions, but we hypothesize that these conditions are useful in practice for real systems. The wDRF conditions are sufficient but not necessary for a program to have the same behaviors on SC and RM models. It is possible to extend the proof to weaker conditions to accommodate a system design, as we discuss in Section 4.3. As for the systems that exhibit different behaviors on SC and RM models, VRM cannot be used to extend their proofs using an SC model to RM hardware. Instead, these systems must be reasoned about directly on a RM hardware model, which can be infeasible for real systems due to the huge proof burden.

4 The VRM Framework

The VRM framework enables proofs of wDRF programs constructed using an SC model to extend to the Promising Arm model [48], which has been proven equivalent to the Armv8 axiomatic specification [16, 47]. The key theorem of VRM is:

Theorem 1 (wDRF theorem)**.** *For any system that satisfies the wDRF conditions, for any piece of the kernel program P, any possible observable behavior of P on Armv8 RM hardware is also observable on an SC model.*

The guarantee. We first explain what Theorem 1 means before diving into its formal proof. This wDRF theorem guarantees that RM hardware will not introduce any "additional observable behavior" for a *kernel* program satisfying wDRF conditions with respect to the kernel's behavior on an SC model. Here, the observable behavior of the kernel program consists of (1) the execution results (i.e., the kernel states) of any piece of the kernel program and (2) the results of user programs' memory accesses that may be affected by the kernel program's update to the user's page table. Note that this theorem's guarantee only holds for the kernel program. User programs, or VMs when the kernel is a hypervisor, are allowed to contain arbitrary racy code, such that user programs, or VMs, may indeed have different behaviors on RM and SC hardware.

The formal model for Armv8. To prove Theorem 1, we first need to formally model Armv8 relaxed memory as the basis to specify programs' RM behavior. There is a long line of work [2, 4, 19, 47, 48] proposing formal memory models for RM hardware. Among them, the Promising model [27, 48] is the most "promising" one. In the Promising model, a program execution is represented by execution traces for each individual CPU, a global *promise list*, and read-from / fulfill

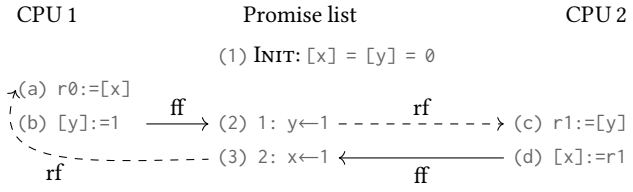

Figure 3. An execution trace for Example 1 in the Promising model.

relations between the traces and promise list. The promise list represents the global history of all writes. A *promise* can be denoted as "`c: a←v`" meaning that address `a` is set to value `v` by CPU `c`. Each memory access event in a CPU's execution trace must point to a promise in the promise list through either the read-from relation `rf` (for load events) or the fulfill relation `ff` (for store events). A load event gets the value of the promise it reads from. As for a store event, the value it writes must be the same with the value in the promise it fulfills.

Figure 3 shows how the RM behavior of Example 1 can be represented in the Promising model. The instruction `(a)` reads from the promise `(3)` that is fulfilled by the instruction `(d)`, while the instruction `(c)` reads from the promise `(2)` that is fulfilled by the instruction `(b)`. The model allows an instruction to read from a promise that will be fulfilled by a future write, making it possible to represent RM behavior.

To ensure the model is not more relaxed than the hardware, the Promising model can introduce constraints to limit the read-from and fulfill relations. The Promising Arm model implements the constraints for Armv8 hardware, which include: (1) a data dependency constraint which ensures that instructions from the same CPU that are data dependent cannot be reordered, (2) an address dependency constraint which ensures that instructions from the same CPU that are address dependent cannot be reordered, (3) a coherence constraint which ensures that memory accesses from the same CPU that access the same memory location cannot be reordered, and (4) a barrier constraint which ensures that memory accesses from the same CPU cannot be reordered across barriers.

While the Promising Arm model can precisely model Arm's RM behavior, it is hard to use for verifying real programs. To reason about a program's behavior using this model, one first needs to enumerate all possible promise lists and then validate each promise list to see if it can be fulfilled by the program. The validating step also has to deal with non-determinism since there may be more than one promise from which a load instruction can read. It is infeasible to verify every single line of a complex software system using this model. VRM addresses this problem by allowing most of a kernel program to be verified using an SC model and propagating the proven guarantees to Promising Arm.

Proof structure. Now we come to the proof. For simplicity, we decompose the proof into two stages. Section 4.1 first proves that, if a kernel program is running solely without interacting with any user programs, the kernel program has the same execution results on RM and SC models if it satisfies three wDRF conditions: DRF-Kernel, No-Barrier-Misuse, and Write-Once-Kernel-Mapping. Section 4.2 extends the proofs to accommodate running user programs, including considering the kernel's behavior on page tables for user programs, and proves Theorem 1 if the kernel program also satisfies the other three wDRF conditions: Transactional-Page-Table, Sequential-TLB-Invalidation, and Memory-Isolation. Section 4.3 weakens the Memory-Isolation condition to allow the kernel program to read user programs' memory under certain circumstances, so that VRM can be used for a broader class of systems that do not strictly satisfy the strong Memory-Isolation condition.

4.1 Proof for the solely running kernel program

We prove the following theorem:

Theorem 2. *Given a kernel program P running solely without user programs, or VMs, if P satisfies the DRF-Kernel, No-Barrier-Misuse, and Write-Once-Kernel-Mapping conditions, then for any execution of P over the Promising Arm model, we can find a corresponding execution of P over an SC model such that their execution results are the same.*

Since the kernel is running by itself, the Transactional-Page-Table, Sequential-TLB-Invalidation, and Memory-Isolation conditions are not needed for this theorem.

We first simplify the proof by removing the effects of virtual memory in the kernel program. The Write-Once-Kernel-Mapping condition requires that the kernel's own page table entries can only mapped once, enforcing that each virtual address of the kernel can only be mapped to at most one physical address during the entire execution of the kernel. Thus, we can replace all virtual addresses with physical addresses in the kernel program and do not consider the kernel's own virtual addresses in the rest of the proof. It is also the reason that we do not need to worry about TLB behavior in this setting.

Then, the key to prove Theorem 4.1 is to come up with a systematic approach to constructing an observably equivalent SC execution from an RM execution generated by a kernel program that satisfies the DRF-Kernel and No-Barrier-Misuse conditions. We achieve this by introducing a new push/pull Promising hardware model to encode these conditions into the RM execution and construct the SC execution from the RM execution that satisfies these conditions. We show that the program execution on Promising Arm is equivalent to its execution on push/pull Promising, and its execution on push/pull Promising is equivalent to its execution on an SC model.

The push/pull Promising model. The push/pull Promising model is based on Promising Arm, inheriting its data dependency, address dependency, coherence, and barrier constraints, but additionally encodes the DRF-Kernel and No-Barrier-Misuse conditions into *push/pull promises* that are

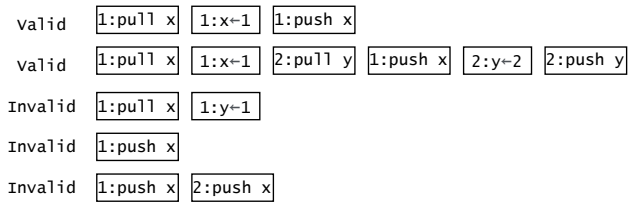

Figure 4. Examples of valid and invalid push/pull promise lists.

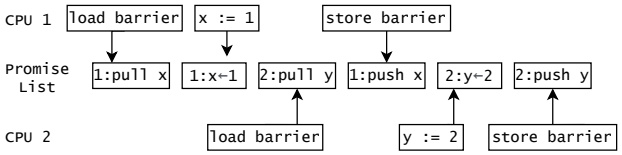

Figure 5. An example of a valid push/pull fulfill relation.

fulfilled using barriers. Push/pull promises are inspired by CertiKOS's push/pull semantics [23, 24], which have been used to reason about whether a program is DRF using an SC model. In push/pull semantics, to access a shared memory location, the CPU must acquire ownership of the location through a special `pull` primitive, which logically "pulls" the content from the shared memory to the CPU's local memory such that further accesses can be done as if the memory location were private. After the accesses, the CPU "pushes" back all modifications to the shared memory through a special `push` primitive. The hardware model panics when pulling a location that is currently owned by a CPU or pushing a location that is not owned by the current CPU. A program is DRF if all invocations of `pull` and `push` primitives in the program never cause a panic.

In our push/pull Promising model, besides promising a stored value, a CPU can also promise to push (a push promise) or pull (a pull promise) a memory location. The effects of push and pull promises are similar to those for push/pull semantics. After a CPU promises to pull a memory location, the location is owned by the CPU. After a CPU promises to push a memory location, the location is no longer owned by the CPU and considered free. The DRF-Kernel condition is satisfied if the push/pull promise list is valid, meaning only free locations are pulled, only owned locations are pushed by their owners, and only the respective owner can access an owned location. Figure 4 presents a few examples of valid and invalid push/pull promise lists. The No-Barrier-Misuse condition is satisfied by requiring push/pull promises to be fulfilled by a CPU by memory barriers in that CPU's execution. A load barrier (e.g., Arm's load acquire instruction) can fulfill a pull promise issued by the same CPU, and a store barrier (e.g., Arm's store release instruction) can fulfill a push promise. A full barrier can fulfill both push or pull promises issued by the same CPU.

The push/pull Promising hardware panics if any push/pull fulfill relation is not valid. A push/pull fulfill relation is valid only if (i) the push/pull promise list is valid, (ii) all push/pull promises are fulfilled by proper barriers, and (iii) the fulfill relations must be consistent with program order. Figure 5 presents an example of how to fulfill a push/pull promise list with barriers. If a valid push/pull fulfill relation exists, we know that share memory accesses are well synchronized by push/pull and are well protected by barriers. Note that, when a read gets the forwarded value from a local program-order-before write, it does not need to be protected by barriers because it does not read from shared memory locations. A kernel program satisfies the DRF-Kernel and No-Barrier-Misuse conditions on the push/pull Promising model if all push/pull fulfill relations in the program never cause a panic.

Constructing an execution on the push/pull Promising model. We first show that, for any program that satisfies the DRF-Kernel and No-Barrier-Misuse conditions, any possible execution over the Promising Arm model can be extended into a valid execution over the push/pull Promising model with the same execution results. Given a promise list of such a program, we can insert pull and push promises when entering and existing critical sections, respectively, and the resulting list must be a valid push/pull promise list because the program satisfies the DRF-Kernel condition. We can then prove that the inserted push/pull promises can be correctly fulfilled because the No-Barrier-Misuse condition ensures that the program contains proper barriers at the right places. In other words, the constructed execution on the push/pull Promising model is equivalent to the execution on Promising Arm.

Constructing an SC execution. We next construct an SC execution from an execution over the push/pull Promising model. For an SC model, the execution is represented as a global list of events generated one by one by all CPUs. For the push/pull Promising model, the execution is represented by a global promise list and per-CPU local execution traces such that the events from different CPUs lack a relative order. To construct an SC execution, we need to know the relative order of events generated by different CPUs. Only shared memory accesses are not local to each CPU, so we just need to determine the relative order of shared memory access events by different CPUs.

Given two shared memory access events from different CPUs, their relative order can be determined from the critical sections to which they belong. We first locate the corresponding push/pull promises using the barriers protecting these two events that fulfill the promises. We then compare the order of their corresponding push/pull promises in the global promise list. We say the first event is *before* the second event in the SC trace if, and only if, the push promise of the first event is before the pull promise of the second event. Together with the CPU-local program order of events from the same CPU, we can construct a *partial order* for all shared memory access events. Thus, we can construct an SC execution through a topological sort on the partial order. Figure 6 shows

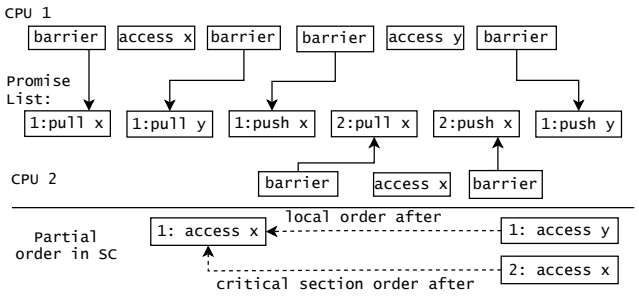

Figure 6. An example of constructing an SC execution trace. The critical section for the access to x by CPU 1 is before that by CPU 2, thus in the SC trace, the access event to x by CPU 1 will occur before the event by CPU 2. For the access to y by CPU 1 and the access to x by CPU 2, their critical sections overlap in the promise list, thus they have no relative order. Other promises and read-from/fulfill relations for memory accesses are not shown for simplicity.

an example of constructing an SC execution. We can show that all SC executions constructed from the same partial order must have the same execution results.

Proving the equivalence of execution results. We now prove that the SC execution constructed using the partial order captures all possible RM behaviors of the program. We just need to show that, if there is a partial order between two events, their execution results are the same on the SC and push/pull Promising models. We prove this by induction over the construction rules of the partial order. Since the induction step is trivial, we describe the proof for the base cases.

If the two events generated by different CPUs have a partial order, they must access the same memory location, so the first event's push promise must be before the second event's pull promise. Since push/pull promises can only be fulfilled by barriers, the events are separated by barriers. The Armv8 barrier constraint ensures that memory accesses cannot be reordered across barriers. Thus, these events cannot be reordered even over the push/pull Promising model and their execution result is the same on an SC model.

If the two events are generated by the same CPU, there are three cases. If they access the same memory location, the Armv8 coherence constraint ensures that two events cannot be reordered on the push/pull Promising model and their execution result is the same on an SC model. If they access different memory locations but are data or address dependent, the Armv8 data or address dependency constraint ensure that they cannot be reordered on the push/pull Promising model and their execution result is the same on an SC model. If they access different memory locations but are not data or address dependent, any order of these two events will have the same execution result, so any reordering on the push/pull Promising model will have the same result as on an SC model.

By induction, we conclude that the partial order used to construct the SC execution captures all the execution results over the push/pull Promising model for the kernel program satisfying the Write-Once-Kernel-Mapping, DRF-Kernel, and No-Barrier-Misuse conditions. Since the execution on the push/pull Promising model is equivalent to that on Promising Arm, the proof of Theorem 2 is complete.

4.2 Proof for the full system

Theorem 2 only discusses the execution of a solely running kernel program but, in real systems, the kernel runs with user programs, or a hypervisor runs with VMs, that may have arbitrary implementations. As shown in Example 7, this may cause unexpected behaviors. To limit the effects of user programs, we introduce the Memory-Isolation condition to enforce that the kernel code does not read user programs' memory and user programs cannot access kernel memory, such that user programs' execution will not affect the behavior of the kernel program, resulting the following theorem:

Theorem 3. *For any system satisfying the Memory-Isolation condition and for any piece of the kernel program P, suppose P is running along with the user program Q, then for any execution of $P \cup Q$, we can find an execution of P running solely without user programs such that the execution results of P are the same.*

By Theorems 2 and 3, we can show that the execution results of the kernel program are the same for SC and Promising Arm models if the DRF-Kernel, No-Barrier-Misuse, Write-Once-Kernel-Mapping, and Memory-Isolation conditions are satisfied. Besides the execution results, there is one more kind of observable behavior, namely a user program's memory access results via shared page tables. For a kernel program satisfying the Transactional-Page-Table and Sequential-TLB-Invalidation conditions, all page table writes within the critical sections are transactional such that any observable page table state, even considering TLB behavior, is either the state at the beginning of the critical section or the state at the end of the critical section. Since the above two states are observable in the SC trace constructed using the partial order, we can prove that, for any kernel program satisfying all six wDRF conditions, any of its observable behavior on the Promising Arm model is also observable over its SC trace constructed using the partial order, thus completing the proof of Theorem 1.

4.3 Weakening the Memory-Isolation condition

Theorem 1 requires a strong condition that kernel code must not read user memory, which may not hold for real systems. For example, the KVM hypervisor reads a VM's memory to create a VM snapshot. To accommodate a broader class of systems, we can use the Weak-Memory-Isolation condition, to allow kernel code to read the memory of the user program Q. We refer to the wDRF conditions with the Weak-Memory-Isolation condition used in lieu of the Memory-Isolation condition as the *weakened wDRF conditions*.

Under the Weak-Memory-Isolation condition, we know that the observable behavior of the kernel is independent with

the user programs' implementation. Thus, if the user memory is the same under SC and RM models, we can prove that the observable behavior of the kernel is also the same for SC and RM models, using a similar proof as for Theorem 1 based on the wDRF conditions without the Memory-Isolation condition. We can always find a user program Q' on an SC model that produces the same user memory state as Q on an RM model, for example, by having Q' simply write the required values into the user memory, thereby proving the following theorem:

Theorem 4. *For any system that satisfies the weakened wDRF conditions and for any piece of kernel program P, suppose P is running along with the user program Q, then for any execution of $P \cup Q$ in the Promising Arm model, we can find a user program Q' and an execution of $P \cup Q'$ in the SC model such that the observable behaviors of P are the same.*

Theorem 4 implies that the set of P's observable behavior on RM hardware across all possible user programs is the same as the set on an SC model. Therefore, if we have a proof of some kernel code verified using the SC model, the proof can be extended to the Promising Arm model if the system satisfies the weakened wDRF conditions.

5 Verifying a KVM Hypervisor on Arm

We have verified that a Linux KVM multiprocessor hypervisor implementation, SeKVM, guarantees the confidentiality and integrity of VMs [36]. SeKVM retrofits KVM [15] into a small core, KCore, and a set of untrusted services, KServ, so that the security properties of the entire KVM hypervisor can be proven by verifying KCore alone. The retrofit uses Arm Virtualization Extensions [7] to run KCore in Arm's EL2 privilege level, also known as hypervisor mode, so that it controls the hardware and can isolate its memory from KServ and VMs. KCore enables stage 2 page tables (Arm's nested page tables) for KServ and VMs to limit their access to physical memory and uses SMMU (Arm's I/O memory management unit) page tables for DMA protection.[1] Although SeKVM is implemented on Arm, it was verified on an SC model without showing how its proofs hold for RM hardware. Using VRM, we prove that SeKVM satisfies the weakened wDRF conditions, with KCore as the kernel. Based on Theorem 4, we therefore prove that SeKVM's security guarantees extend to Arm RM hardware.

5.1 Write-Once-Kernel-Mapping

KCore has its own EL2 page table, which is the kernel page table in the Write-Once-Kernel-Mapping condition. When the system is booted, all physical memory is mapped to a contiguous virtual memory region in the kernel page table, similar to how the Linux kernel manages physical memory on 64-bit systems. The kernel page table is never changed after boot except for the `remap_pfn` hypercall used for secure VM boot. `remap_pfn` maps a physical page to a virtual address outside of the aforementioned contiguous virtual memory region in KCore's address space. Because physical pages allocated by KServ to store VM images may not be contiguous, `remap_pfn` maps those pages to a contiguous virtual memory region required by the integrated crypto library (Ed25519) to calculate a hash of the memory content for VM image authentication. The hypercall never unmaps or remaps a virtual page. Since unmap and remap are never needed, there is only one primitive, `set_el2_pt`, that handles the kernel page table, which is called by `remap_pfn`. We verify that `set_el2_pt` can never overwrite an existing mapping. Thus, we guarantee the Write-Once-Kernel-Mapping condition holds for SeKVM.

[1] On Arm hardware without an SMMU, SeKVM guarantees VM confidentiality and integrity assuming no DMA attacks.

```
1  void acquire(){
2      u32 my_ticket =acquire fetch_and_inc(ticket);
3      while (my_ticket !=acquire now) {};
4      pull();
5  }
6  void release(){
7      push();
8      now++release;
9  }
```

Figure 7. The ticket lock implementation in Linux. An `acquire` or `release` notation indicates that the memory access instruction is with a barrier. Push/pull primitives have been added.

5.2 DRF-Kernel and No-Barrier-Misuse

We verify that KCore satisfies the DRF-Kernel and No-Barrier-Misuse conditions by proving that (1) KCore's lock implementation is correct, and (2) KCore uses the lock correctly to protect shared memory accesses. For locks, KCore uses Linux's ticket lock implementation, shown in pseudocode in Figure 7.[2] The ticket lock implementation itself contains data races—the shared variables `ticket` and `now` can be accessed by multiple threads simultaneously—so we need to reason about the lock implementation directly on an RM model. Figure 7 shows that all memory reads (lines 2 and 3) use Arm's load-acquire instruction and the memory write (line 8) uses Arm's store-release instruction, which introduce barriers and forbid reordering. We then can just prove that following the in-order execution of `acquire` and `release`, (1) only the thread whose ticket is equal to `now` can hold the lock, and (2) each thread has a unique ticket number due to the atomicity of `fetch_and_inc`. Together, these properties guarantee mutual exclusion.

Now that we have verified that the ticket lock is implemented correctly, we need to prove that throughout KCore, locks are used correctly to protect shared memory accesses. KCore accesses memory based on the mappings in its own EL2 page table. As discussed in Section 5.1, all physical memory is mapped to KCore's EL2 page table at boot. Let us initially assume that KCore only accesses physical memory through these initial mappings, which we prove are one-to-one such

[2] The original code can be found at https://elixir.bootlin.com/linux/v4.18.20/source/arch/arm64/include/asm/spinlock.h#L30.

that each physical address is only mapped to one virtual address. We use the push/pull Promising model, as explained in Section 4.1, to prove that locks correctly protect shared memory accesses by verifying that all `push` and `pull` primitives never panic, thereby proving that the DRF-Kernel condition holds. Figure 7 shows that the `push` primitive is directly followed by a store barrier `release` and the `pull` primitive follows a load barrier `acquire`, such that critical sections protected by the lock are correctly bounded by barriers, thereby proving that the No-Barrier-Misuse condition holds.

In SeKVM, vCPU contexts are not directly protected by a lock but by a state variable, similar to Example 3. Before accessing a vCPU context, a physical CPU must check that the state of the vCPU context is `INACTIVE`, which means that it is not used by other physical CPUs. Then, it sets the state to `ACTIVE` and starts accessing the vCPU context. After finishing the access, it will set the state back to `INACTIVE`. We prove that the DRF-Kernel condition holds by placing the push/pull primitives for the vCPU context before setting `INACTIVE` and after setting `ACTIVE` and proving that only one physical CPU can pull the context at one time. We show that the No-Barrier-Misuse condition also holds because Arm's load acquire and store release instructions are used when setting `INACTIVE` and checking `INACTIVE`, which include barriers.

This proof would be complete if KCore never mapped any physical memory to its page table after booting. However, as discussed in Section 5.1, there is one case when the kernel page table is changed after booting, such that multiple virtual pages can map to a single physical page. We prove that KCore never writes to any of these virtual pages and, thus, will not cause data races, satisfying the DRF-Kernel condition.

5.3 Weak-Memory-Isolation

We verify that SeKVM satisfies the Weak-Memory-Isolation condition by first proving that VMs and KServ, which are considered user programs for the purposes of this condition, cannot write KCore's memory. Arm hardware ensures that when stage 2 paging is enabled for VMs or KServ, they cannot access any physical memory that is not mapped in their own stage 2 page tables. Similarly, DMA-capable I/O devices controlled by a VM or Kserv cannot access physical memory that is not mapped in their SMMU page tables. We prove that that the EL2 and SMMU registers that control stage 2 and SMMU page tables, respectively, are always enabled as invariants of the system, and that KCore's memory is never mapped to stage 2 or SMMU page tables. KCore tracks the owner of each 4 KB physical page of memory in an `s2page` data structure. A page can only have one owner at any given time, which can be KCore, KServ, or a VM. KCore will always check that it is not the owner of a physical page before mapping it to a stage 2 or SMMU page table. Therefore, no pages owned by KCore are ever mapped to any stage 2 or SMMU page table. Thus, KCore's memory is inaccessible to VMs and KServ, with the exception of the stage 2 and SMMU page tables themselves, which are allocated from KCore's memory and readable by MMU and SMMU hardware, respectively, but never written by MMU or SMMU hardware. Therefore, none of KCore's memory is ever written by VMs or KServ.

We then show that the SeKVM proofs do not rely on the implementation of user programs. In SeKVM, reading from VM or KServ memory is modeled by *data oracles*, a random number generator to mask the expected information flow [36]. Since data oracles can generate any possible user program memory contents, the SeKVM proofs hold for any possible user programs and are independent of any concrete implementation of user programs. Given any RM behavior of the VM or KServ, we can always find a data oracle to produce a sequence of numbers resulting in the same memory contents over an SC model. Therefore, SeKVM satisfies the Weak-Memory-Isolation condition.

5.4 Transactional-Page-Table

We prove that KCore satisfies the Transactional-Page-Table condition for all page tables that it manages other than its own, namely stage 2 page tables for VMs and KServ, and SMMU page tables. Since the proofs are similar, we first prove this for stage 2 page tables, then discuss how the proof applies to SMMU page tables.

KCore only has two primitives that update stage 2 page tables, `set_s2pt` and `clear_s2pt`. `set_s2pt` establishes a new page mapping and `clear_s2pt` unmaps an existing mapping. KCore dynamically builds page tables in `set_s2pt`, by allocating free pages from a reserved page pool private to KCore. All bytes of a newly allocated page are guaranteed to be 0. KCore scrubs the pool of memory during initialization. When setting a page mapping with `set_s2pt`, KCore walks from the page table root to the target leaf table. During the page table walk, if the next-level page table does not exist, it allocates a new page from the reserved pool and inserts the page into the page table. At the leaf table, it will check and set a new page mapping only if it will not overwrite any existing mapping. The entire walk-allocate-set procedure is within the critical section protected by the respective page table lock. `clear_s2pt` always walks to the target leaf table and sets an existing `pte` to 0. It does not reclaim any empty table so no table at any level will be removed or substituted by other tables once inserted into the page table tree.

We prove that both `set_s2pt` and `clear_s2pt` satisfy the Transactional-Page-Table condition. `clear_s2pt` contains at most one write to the page table so the proof is trivial. Although `set_s2pt` may involve multiple writes to the page table due to inserting new tables into the page table tree, it is still transactional because (1) if it sets the mapping in an existing table, only one page table write is required so it is transactional; (2) if it sets the mapping in a newly inserted table, for any reordering of the writes, any page table walk will page fault unless it happens after all page table writes have occurred, in which case it will see the final result of the

table write to the last-level page table, so it is transactional. In other words, a page fault occurs on any reordering of the writes if not all the writes have occurred because either intermediate levels of the page tree will be missing, resulting in a page fault, or the update to the last-level page table will be missing, also resulting in a page fault. Therefore, since both `set_s2pt` and `clear_s2pt` satisfy the Transactional-Page-Table condition, SeKVM satisfies this condition.

For SMMU page tables, KCore has two primitives, `set_spt` and `clear_spt`. Other than allocating pages from a page pool reserved for the SMMU, they work the same as `set_s2pt` and `clear_s2pt`. The proof that SMMU page tables satisfy the Transactional-Page-Table condition is therefore the same as the proof for stage 2 page tables.

5.5 Sequential-TLB-Invalidation

We prove that all SeKVM page tables satisfy the Sequential-TLB-Invalidation condition. For stage 2 page tables, we verify that this condition holds over two primitives: `set_s2pt` and `clear_s2pt`. As discussed in Section 5.4, `set_s2pt` must operate on an empty entry so no TLB invalidation is needed. The `clear_s2pt` primitive unmaps a page table entry. We validate that such an unmap operation is directly followed by a barrier and a TLB invalidation. For SMMU page tables, the proofs for `set_spt` and `clear_spt` primitives are similar to the `set_s2pt` and `clear_s2pt` primitives except that an SMMU TLB invalidation instead of a TLB invalidation is used. For KCore's EL2 page table, no TLB invalidation is needed as only empty entries can be updated due to the Write-Once-Kernel-Mapping condition.

5.6 Verifying Multiple KVM Versions

By proving that SeKVM satisfies the weakened wDRF conditions, using Theorem 4 we know that SeKVM's security guarantees originally proved on an SC model also hold for Arm RM hardware. No changes to the original verified SeKVM implementation or proofs were required. However, the original verified SeKVM implementation was for a retrofitted KVM in the Linux 4.18 kernel that used 4-level stage 2 page tables, and we have since added support for 3-level stage 2 page tables, useful for improving performance on Arm CPUs with smaller TLBs because less intermediate page table entries will need to be cached in the TLB. We have verified the updated implementation in Coq with modest additional proof effort because it was only necessary to verify the page table changes rather than reverify all of the retrofitted KVM, thanks to the modular layered verification approach used by the proof. Furthermore, we have verified that the weakened wDRF conditions is satisfied for both 3-level and 4-level stage 2 page tables.

Taking advantage of this additional verified stage 2 page table support, we have further ported SeKVM across multiple hardware platforms and Linux kernel versions, which involved modest changes to KServ. Specifically, we have verified eight KVM versions in Linux 4.18, 4.20, 5.0, 5.1, 5.2, 5.3, 5.4, and 5.5 running across multiple Armv8 multiprocessor hardware configurations. This is the first proof of a commodity multiprocessor hypervisor on Arm RM hardware, and the first proof of multiple versions of a KVM hypervisor on Arm RM hardware.

Proof	Coq LOC
VRM sufficiency of wDRF conditions	3.4K
SeKVM satisfies wDRF conditions	3.8K
SeKVM's security guarantees on SC	34.2K

Table 1. LOC breakdown of the Coq proofs.

We formulate all proofs in the Coq proof assistant [55]. Table 1 lists the lines of code (LOC) in Coq required for proving: (1) the theorems used by VRM that the weakened wDRF conditions are sufficient to propagate systems' guarantees from SC to Arm RM hardware, (2) the whole SeKVM implementation indeed satisfies the weakened wDRF conditions, and (3) the original SeKVM guarantees of VM confidentiality and integrity on an SC model [36] with the additional proofs for 3-level page tables. The proof effort for VRM itself is modest and a one-time cost that can be reused to extend proofs of other systems on SC to Arm RM hardware. The proof effort that SeKVM satisfies the wDRF conditions is almost an order of magnitude less than the original proof for SeKVM on an SC model, demonstrating that only modest additional proof effort is required to extend the proofs to Arm RM hardware. Not only is VRM useful because it makes it possible to verify systems on Arm RM hardware for the first time, but also because the proof effort for VRM to reuse SC proofs for RM hardware is quite manageable.

6 Evaluation

Since SeKVM requires modest changes to KVM to make it possible to verify its security guarantees, we compared its performance against unmodified KVM across different software and hardware configurations to quantify the performance impact of the changes. We ran SeKVM and unmodified KVM in both Linux 4.18 and 5.4 on two different Armv8 hardware configurations: (1) an HP Moonshot m400 server with an 8-core 64-bit ARMv8-A 2.4 GHz Applied Micro Atlas SoC, 64 GB of RAM, a 120 GB SATA3 SSD, and a Dual-port Mellanox ConnectX-3 10GbE NIC, and (2) an AMD Seattle Rev.B0 server with an 8-core 64-bit ARMv8-A 2 GHz AMD Opteron A1100 SoC, 16 GB of RAM, a 512 GB SATA3 HDD for storage, and an AMD XGBE 10 GbE NIC. For client-server workloads, clients ran on another m400 machine when using the m400 server, and ran on an x86 machine with 24 Intel Xeon CPU 2.20 GHz cores and 96 GB RAM when using the Seattle server, in all cases connected via 10 GbE.

We used different software configurations across the servers to measure performance across multiple software and VM configurations. We used Ubuntu 18.04 and QEMU 3.0 for the m400 server and its VMs, and Ubuntu 16.04 and QEMU 2.3.50 for the Seattle server and its VMs. We used different SMP VM configurations, 2 CPUs and 256 MB RAM on the m400 server and 4 CPUs and 12 GB of RAM on the Seattle server. The smaller VM

Name	Description
Hypercall	Transition from a VM to the hypervisor and return to the VM without doing any work in the hypervisor. Measures bidirectional base transition cost of hypervisor operations.
I/O Kernel	Trap from a VM to the emulated interrupt controller in the hypervisor OS kernel, then return to the VM. Measures base cost of operations that access I/O devices supported in kernel space.
I/O User	Trap from a VM to the emulated UART in QEMU and then return to the VM. Measures base cost of operations that access I/O devices emulated in user space.
Virtual IPI	Issue virtual IPI from a VCPU to another VCPU running on a different CPU, both CPUs executing VM code. Measures time from sending virtual IPI until receiving VCPU handles it.

Table 2. Microbenchmarks.

Benchmark	m400		Seattle	
	KVM	SeKVM	KVM	SeKVM
Hypercall	2,275	4,695	2,896	3,720
I/O Kernel	3,144	7,235	3,831	4,864
I/O User	7,864	15,501	9,288	10,903
Virtual IPI	7,915	13,900	8,816	10,699

Table 3. Microbenchmark performance (cycles).

configuration was also used to show results for running many SMP VM instances given the RAM limits of the m400 server. We also measured performance natively on the servers with the host OS capped at using the same number of CPUs and amount of RAM as the respective VM configuration. Full disk encryption (FDE) was enabled for Seattle VMs but not m400 VMs given the limited memory assigned to m400 VMs. VMs were configured to use paravirtualized I/O, typical of cloud infrastructure deployments, with standard VHOST networking and `cache=none` for block storage devices [13, 14, 38].

Microbenchmarks. We first ran KVM unit tests [26] to measure the cost of common micro-level hypervisor operations listed in Table 2. Table 3 shows the microbenchmarks measured in cycles for unmodified KVM and SeKVM in Linux 4.18 for each hardware configuration. SeKVM overhead compared to unmodified KVM is much higher on the m400 server versus the Seattle server because the m400 CPUs have a tiny TLB [46] compared to Seattle CPUs. Although KCore supports huge pages for stage 2 page tables for VMs, the current implementation maps regular 4 KB pages in KServ's stage 2 page table so microbenchmark workloads that spend most of their time running in KServ require more TLB entries to cache address translations, increasing TLB capacity misses. Newer Arm CPUs have more reasonable TLB sizes similar to or greater than the Seattle CPUs, so the Seattle measurements are more reflective of typical Arm server performance. For Seattle, SeKVM only incurs 17% to 28% overhead over KVM, with the added benefit of verified VM protection.

Application benchmarks. We next ran application benchmarks listed in Table 4 to measure performance on more realistic workloads. Figure 8 compares the performance for running the workloads in a VM on unmodified KVM versus SeKVM, with performance normalized to native execution on the respective hardware configuration without full disk encryption. Results are shown for both KVM and SeKVM in Linux 4.18 and 5.4 on both m400 and Seattle hardware. In all cases, SeKVM performance on real application workloads is comparable to unmodified KVM, yet provides the added benefit of verified VM protection. Worst case overhead for SeKVM is less than 10% versus unmodified KVM, even when running on the m400 server with its small TLBs. There is no substantial change in relative performance when running 2 CPU VMs versus 4 CPU VMs.

Name	Description
Hackbench	`hackbench` [49] using Unix domain sockets and process groups running in 500 loops; m400 used 20 groups, Seattle used 100 groups.
Kernbench	Compilation of the Linux kernel using `allnoconfig` for Arm; m400 compiled v4.18 with GCC 7.5.0, Seattle compiled v4.9 with GCC 5.4.0.
Apache	`Apache` server handling concurrent requests via TLS/SSL from remote `ApacheBench` [54] v2.3 client, serving the `index.html` of the GCC manual; m400 used v2.4.29 serving 7.5.0 manual, Seattle used v2.4.18 serving 5.4.0 manual.
Mongodb	`MongoDB` server handling requests from a remote `YCSB` [11] v0.17.0 client running workload A with 16 concurrent threads; m400 used v3.6.3 with readcount=10000 and operationcount=50000, Seattle used v4.0.20 with readcount=500000 and operationcount=100000.
Redis	`Redis` v4.0.9 server handling requests from a remote `YCSB` v0.17.0 client running workload A; m400 used v4.0.9, Seattle used v3.0.6.

Table 4. Application benchmarks.

We evaluated the performance scalability of KVM versus SeKVM by running the application benchmarks in Table 4 with multiple concurrent VMs running on the m400 server. Figure 9 shows the measurements for Linux 4.18 from 1 to 32 VMs normalized to native execution of one instance of the workload running; the maximum number of VMs was only limited by the number of VM images we could store on the server's SSD. The measurements for 1 VM in Figure 9 are the same as the m400 results in Figure 8. As expected, running more concurrent VM instances of the application benchmark results in slower performance as the number of instances increases, but the results show a similar slowdown for both KVM and SeKVM for all application benchmarks. In all cases, even when running 32 concurrent VMs, SeKVM has no worse than 10% overhead compared to unmodified KVM, demonstrating that SeKVM has similar performance scalability as unmodified KVM.

These results indicate that the use of locks in SeKVM to protect shared memory accesses and make its proofs tractable and, more generally, the fact that SeKVM satisfies the weakened wDRF conditions so that its proofs hold on RM hardware do not adversely affect SeKVM's performance scalability in running multiple multiprocessor VMs on Arm RM hardware. The results show that VRM can be used to verify real systems on RM hardware without adversely affecting their performance.

7 Related Work

RM models. Every modern multiprocessor architecture employs an RM model, allowing some effects of out-of-order and speculative execution to be programmer-visible, including

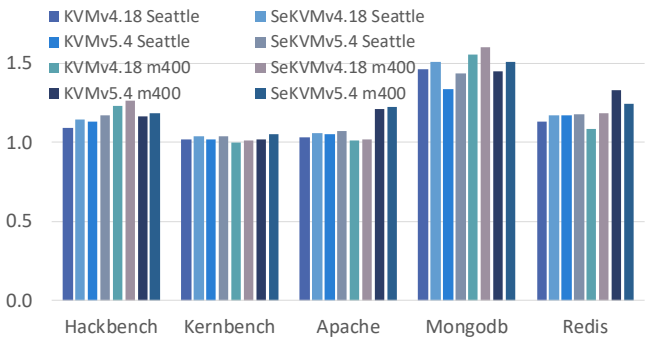

Figure 8. Single-VM application benchmark performance.

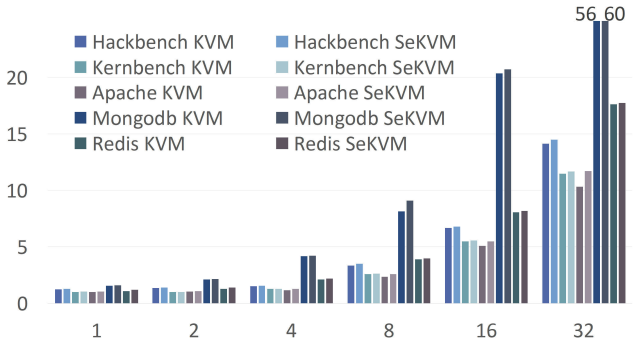

Figure 9. Multi-VM application benchmark performance.

x86 [45, 52], RISC-V [5], IBM Power [2, 20, 41, 50, 51], and Arm [2, 4, 19, 47, 48]. For Arm specifically, the Flowing model was proposed to precisely specify the architecturally allowable behavior of Armv8 [19]. The Armv8 axiomatic model was formalized after Armv8 was revised to be multicopy-atomic [47], and a Flat operational model was also formalized, which was mostly equivalent to the axiomatic model. The Promising Arm operational model [48], which we build upon, is simpler than the Flat model in that instructions are executed in atomic steps and mostly in order, yet has been shown to be equivalent to the Armv8 axiomatic model. All of these models only handle user-level code; they exclude system features such as MMU hardware that have been modeled by our VRM framework. Until now, the only use cases of these models were to exhaustively test programs with less than 500 lines of assembly code after compilation [48]. In contrast, by verifying SeKVM, we are the first to show the feasibility of formal verification of concurrent systems software on Arm RM models.

Previous work has also explored formally specifying concurrency behaviors at the C/C++ [6, 27, 32] or Linux [3] level. They cannot handle hardware-specific assembly code. For C/C++, there is no verified compiler that preserves Arm RM semantics, leaving no guarantee on the final machine code.

Verification of concurrent systems. Concurrent OS kernels and file systems have been verified on multiprocessor hardware, including CertiKOS [22–24, 28], AtomFS [56], and Perennial/Mailboat [8]. All these works assume an SC hardware model. Their verification results do not extend to any RM hardware except, in some cases, x86-TSO. We believe that VRM could be applied to these systems to propagate their SC-only proofs to RM hardware.

Recent progress has been made on verifying small concurrent programs directly on RM models [31, 40, 42]. Armada can verify user code on the x86-TSO memory model using a TSO-elimination technique [40]. Vsync [44] uses model checking to automatically verify the correctness of many synchronization primitives in RM models including Armv8. None of these approaches have been shown to scale to verify real systems such as KVM.

8 Conclusions

VRM is the first framework for formally verifying systems software on Arm relaxed memory hardware. VRM introduces the wDRF conditions, a set of synchronization and memory access conditions which ensure that kernel code, as used in operating systems and hypervisors, will have the same behavior on sequentially consistent and relaxed memory hardware. These conditions account for data races in lock implementations, read/write races in page table implementations due to MMU hardware, as well as arbitrary relaxed memory behavior in user programs. By proving that kernel code satisfies the wDRF conditions, we can extend verification results on sequentially consistent models to relaxed memory hardware without doing proofs directly on relaxed memory hardware.

We have implemented VRM in Coq and used it to extend the verified security guarantees of a retrofitted KVM hypervisor to hold on Arm relaxed memory hardware without any modifications to the original implementation or proofs. The only additional proofs required were to show that the wDRF conditions hold for the verified KVM implementation. Those proofs were only a few thousand lines of Coq code, roughly an order of magnitude less than the original KVM proofs. This is the first proof of commodity systems software on relaxed memory hardware. We further prove that the security guarantees hold for multiple versions of verified KVM on multiple Arm hardware configurations. Experimental results show that verified KVM performs similar to unmodified KVM on Arm multiprocessor hardware when concurrently running dozens of multiprocessor VMs with real application workloads.

Acknowledgments

Xuheng Li ported SeKVM to Linux 5.1, 5.2, 5.4, and 5.5. Michael Jan, Hans Montero, and Zachary Schuermann ported SeKVM to Linux 4.20, 5.0, and 5.3, respectively. Christoffer Dall and Chris Hawblitzel provided helpful comments on earlier drafts. This work was supported in part by two Amazon Research Awards, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. The last author is a co-founder of and has an equity interest in CertiK Global Ltd.

References

[1] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering-A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. 2–14.

[2] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming (DAMP '09)*. 13–24.

[3] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. 405–418.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 1–74.

[5] Krste Asanovic and Andrew Waterman. 2019. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. RISC-V Foundation.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. 55–66.

[7] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization.* Morgan and Claypool Publishers.

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 243–258.

[9] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. 431–447.

[10] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. 18–37.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[12] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI'16)*. Santa Barbara, CA, 648–664. https://doi.org/10.1145/2908080.2908100

[13] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. 304–316.

[14] Christoffer Dall, Shih-Wei Li, and Jason Nieh. 2017. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 221–234.

[15] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. 333–347.

[16] Will Deacon and Jade Alglave. 2020. The ARMv8 Application Level Memory Model. http://diy.inria.fr/www/?record=aarch64.

[17] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. *ACM SIGPLAN Notices* 53, 4 (June 2018), 242–255.

[18] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. 287–305.

[19] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. *ACM SIGPLAN Notices* 51, 1 (Jan. 2016), 608–621.

[20] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An Integrated Concurrency and Core-ISA Architectural Envelope Definition, and Test Oracle, for IBM POWER Multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO '15)*. 635–646.

[21] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, and Haozhong Zhang. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*. Mumbai, India, 595–608.

[22] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building Certified Concurrent OS Kernels. *Commun. ACM* 62, 10 (Sept. 2019), 89–99.

[23] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 653–669.

[24] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. 646–661.

[25] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. 1–17.

[26] Andrew Jones, Paolo Bonzini, and Thomas Huth. 2020. KVM Unit Tests. http://www.linux-kvm.org/page/KVM-unit-tests.

[27] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. 175–189.

[28] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS '17)*. 273–297.

[29] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS '07)*. 225–230.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. 207–220.

[31] Ori Lahav and Udi Boker. 2020. Decidable Verification Under a Causally Consistent Shared Memory. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. 211–226.

[32] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI '17). 618–632.

[33] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sept. 1979), 690–691.

[34] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Proceedings of the International Symposium on Formal Methods (FM '09)*. 806–809.

[35] Shih-Wei Li, John S. Koh, and Jason Nieh. 2019. Protecting Cloud Virtual Machines from Commodity Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. 1357–1374.

[36] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P '21)*. 1782–1799.

[37] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. 3953–3970.

[38] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. 2017. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*. 201–217.

[39] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2019), 1–31.

[40] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. 197–210.

[41] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*. 495–512.

[42] Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness Against a C11-Style Memory Model. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–33.

[43] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 225–242.

[44] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS '21)*. 530–545.

[45] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better X86 Memory Model: X86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*. 391–407.

[46] Igor Pavlov. 2015. Applied Micro X-Gene. https://www.7-cpu.com/cpu/X-Gene.html.

[47] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017).

[48] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. 1–15.

[49] Rusty Russell. 2008. Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

[50] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 311–322.

[51] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 175–186.

[52] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97.

[53] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 287–305.

[54] The Apache Software Foundation. 2020. ab - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.4/programs/ab.html.

[55] The Coq Team. 2021. The Coq Proof Assistant. http://coq.inria.fr.

[56] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. 2019. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 259–274.