

Modal Abstractions for Virtualizing Memory Addresses

ISMAIL KURU and COLIN S. GORDON, Drexel University, USA

Virtual memory management (VMM) code is a critical piece of general-purpose OS kernels, but verification of this functionality is challenging due to the complexity of the hardware interface (the page tables are updated via writes to those memory locations, using addresses which are themselves virtualized). Prior work on verification of VMM code has either only handled a single address space, or trusted significant pieces of assembly code.

In this paper, we introduce a modal abstraction to describe the truth of assertions relative to a specific virtual address space: $[r]P$ indicating that P holds in the virtual address space rooted at r . Such modal assertions allow different address spaces to refer to each other, enabling complete verification of instruction sequences manipulating multiple address spaces. Using them effectively requires working with other assertions, such as points-to assertions about memory contents — which implicitly depend on the address space they are used in. We therefore define virtual points-to assertions to definitionally mimic hardware address translation, relative to a page table root. We demonstrate our approach with challenging fragments of VMM code showing that our approach handles examples beyond what prior work can address, including reasoning about a sequence of instructions as it changes address spaces. Our results are formalized for a RISC-like fragment of x86-64 assembly in Rocq.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Virtual memory**.

Additional Key Words and Phrases: program verification, virtual memory, modal logic

ACM Reference Format:

Ismail Kuru and Colin S. Gordon. 2025. Modal Abstractions for Virtualizing Memory Addresses. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 356 (October 2025), 29 pages. <https://doi.org/10.1145/3763134>

1 Introduction

Virtual memory management lies at the core of modern OS kernel implementation. It is deeply intertwined with most other parts of a typical general-purpose OS kernel design, including scheduling, hardware drivers, and even the filesystem buffer cache. In writing the authoritative reference on the internals of the Solaris kernel, McDougall and Mauro went so far as to claim that “*the virtual memory sub-system can be considered the core of a Solaris instance, and the implementation of Solaris virtual memory affects just about every other subsystem in the operating system*” [59]. This makes support for verification the virtual memory management subsystem of an OS kernel critical to the correctness of every other piece of an OS or any software running atop it.

At its core, the virtual memory functionality of modern CPUs is about *location virtualization*: the memory locations (addresses) seen by most code are not, in fact, the exact location in physical memory where data reside. Instead these are *virtual* addresses, which are mapped to actual physical resources by the cooperation of the hardware and OS. This is what enables separation of process memory resources: the OS manipulates hardware functionality to ensure program memory accesses only succeed if the kernel has granted the program access to the accessed address. But this is complicated by the fact that control over these mappings of virtual to physical addresses is itself mediated by *in-memory data structures*, which the kernel also accesses via virtual address, leading to indirect cycles.

Authors' Contact Information: Ismail Kuru, ik335@drexel.edu; Colin S. Gordon, csgordon@drexel.edu, Drexel University, Philadelphia, PA, USA.



This work is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART356

<https://doi.org/10.1145/3763134>

Further complicating matters, addresses themselves bear no information about which address space they originate from. This is not a problem for usermode code, which only accesses its own address space. But the kernel has (or can grant itself) access to all address spaces. Mixing up addresses from different address spaces leads to severe bugs. Proofs of kernel correctness must also track which *assertions* hold in different address spaces: some assertions should hold across all address spaces, while others hold in only one, and others may hold in multiple but still not all.

This kind of context-dependent assertion, where a fact may be true in one address space but not others, has a modal flavor. We propose tackling the verification of virtual memory subsystems (and kernels more broadly) by adapting ideas from hybrid modal logic, which can label assertions true under *other, named* circumstances (i.e., in another address space) with a modality indexed by a name for that space (in our case, the root of the page tables for an address space). This offers a *convenient* and *powerful* way to *modularly* isolate assertions specific to a particular address space, explicitly state when an assertion is true across address spaces, manipulate address spaces from within other address spaces, and reason about change in address spaces. This approach to reasoning about virtual memory is more flexible than prior program logic techniques [51, 52], which were only able to work with a single address space (the current address space on the CPU) because they were unable to speak directly *within the logic* about other address spaces, in addition to handling the *non-local* effects of page table updates whether within the current address space or across address spaces.

Contributions. We develop these ideas in the form of a logic for working with virtual-address-space-relative assertions, implemented as an embedded separation logic within Iris [48]. The result is a separation logic that lifts a number of major semantic restrictions present in the few prior logics tackling virtual address translation. The logic we develop covers core reasoning principles for reasoning about memory configurations and code reliant upon or manipulating those memory configurations in the presence of in-memory page tables, the primary memory protection mechanism across Intel/AMD’s x86-64 processors; ARM’s application class processors including AArch64 CPUs; as well as POWER, RISC-V, and other architectures. We prove the soundness of our vProp logic with respect to a RISC-like fragment of AMD64 instructions. We verify simplified versions of several critical virtual-memory-related pieces of OS functionality, including: switching address spaces, converting physical addresses to virtual addresses for software page table walks, and a complete software page table walk — the backbone of other essential functionalities of virtual-memory which we discuss for mapping and unmapping pages. These examples demonstrate the suitability and flexibility of a modal treatment of address spaces: each has a concise specification, each example either goes beyond the technical capabilities of prior logics, or revisits an example from prior work in much greater detail (e.g., verifying critical and challenging supporting code that was axiomatized or treated as out-of-scope in prior work). Our proofs are available [53].

2 Background

This section briefly recalls both fundamentals of page-table-based address translation in general-purpose OS kernels, high-level background on separation logics and the Iris [48] framework we build on, and some material on modal logic that informs our development.

2.1 Machine Model

In typical system configurations, all memory addresses seen by programs running on modern computers are *virtualized*: the address observed by a running program generally will not correspond directly to the physical location in memory, and may not even correspond to a physical location that *exists* in the machine. Instead, these *virtual* addresses are translated to *physical* addresses that correspond directly to locations in RAM. On most modern architectures, this translation is

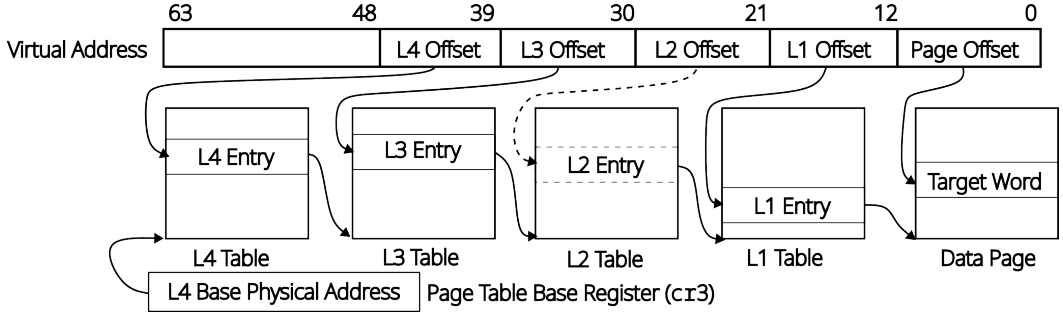


Fig. 1. x86-64 hardware address translation.

```

1  pte_t *pte_get_next_table(pte_t *entry) {
2      pte_t *next;
3      if (!entry->present) { // Check if entry needs initialization
4          pte_initialize(entry); // Alloc empty table for next level
5          entry->present = 1; // Mark valid
6      }
7      /* Convert phys. addr. from PTE to virt. addr. for access */
8      uintptr_t next_phys_addr = PTE_PFN_TO_ADDR(entry->pfn);
9      uintptr_t next_virt_addr = (uintptr_t) P2V(next_phys_addr);
10     next = (pte_t *) next_virt_addr;
11     return next;
12 }

1  pte_t *walkpgdir(pte_t *root, void *va){
2      pte_t *l4_entry = &root[L4Offset(va)];
3      pte_t *l3 = pte_next_table(l4_entry);
4      pte_t *l3_entry = &l3[L3Offset(va)];
5      pte_t *l2 = pte_next_table(l3_entry);
6      pte_t *l2_entry = &l2[L2Offset(va)];
7      pte_t *l1 = pte_next_table(l2_entry);
8      pte_t *l1_entry = &l1[L1Offset(va)];
9      return l1_entry;
10 }

```

Fig. 2. Most of a software page table walk in C, as used to map new pages.

performed through cooperation of the hardware and OS kernel: while executing an instruction that dereferences a (virtual) address, the CPU's *memory management unit* (MMU) hardware performs *address translation*, resulting in a physical address used to access the cache¹ and/or memory-bus.

On the x86-64 architecture, the MMU's address translation uses a sparse hierarchical set of tables: *page tables* (referring to pages of memory). As Figure 1 (based on Figure 5-17 of the AMD64 architecture manual [4]²) shows, address translation proceeds by repeatedly taking designated slices of the virtual address and indexing into successive tables of 512 8-byte entries (making each table 4KB in size). The final lookup in the page tables gives the base physical address of a 4KB page of physical memory accessible to the running program, to which the low-order 12 bits of the accessed virtual address are added to determine the actual physical address retrieved. On x86-64, standard configurations use 4 levels of page tables, labelled levels 4 through 1, with lookups in the level 1 page table resulting in the actual page of physical memory holding the requested data, and the low-order 12 bits being used to index into this page.³ The translation process or algorithm is often referred to as a *page-table walk*. While Figure 1 and most of our constants (how many levels, which virtual address bits index which table levels) are specific to common x86-64 CPUs, it is straightforward to adapt our approach to ARMv8/aarch64, RISC-V, or PowerPC, which use nearly-identical page table structures (only the order of flag bits in the entries differ), or to 5-level paging present in newer x86-64 or RISC-V designs (one step would be iterated an additional time).

¹Technically, for performance reasons most caches are indexed with parts of the virtual address, but tagged with the physical data addresses, so cache lookups and address translations can proceed in parallel.

²While x86 up through its 32-bit incarnation were due to Intel, the x86-64 architecture as a 64-bit extension to x86 was originally due to AMD. As a result, it is sometimes also referred to as the amd64 architecture.

³Technically levels 1–3 have explicit historical names, but for brevity and consistency, we simply number them, in keeping with the newer 5th level. Our formalization only deals with 4-level page tables, but is straightforwardly extensible to 5.

The entries (e.g., L4 Entry in Figure 1) of each table are 64 bits wide, but each points to a physical address aligned to 4KB (4096 byte) boundaries, leaving 12 bits to spare to control a validity bit (called the *present* bit), a read-write bit (enabling write access), and a range of additional bits which can be used to control caching, write-through, and more. This paper will only consider the present bit (0).

The page tables are managed by the OS kernel's *virtual memory manager* (VMM).⁴ Typically each process has its own page tables, which the OS registers with the CPU by storing the (page-aligned) physical address of the root of the page table tree (the start of the L4 table shown as root in Figure 1) in a specific register (cr3) as part of switching to a new process. Using different mappings, which map only disjoint portions of physical memory (with some exceptions in the next section) is how the OS ensures memory isolation between processes. If an instruction is executed that accesses a virtual address that either has no mapping, or does not have a mapping permitting the kind of access that was performed (e.g., the instruction was a memory write, but the relevant address range was marked read-only in the relevant page table entry), the hardware triggers a *page fault*, transferring control to a *page fault handler* registered with the hardware by the OS, allowing it to take corrective action. If no mapping was supposed to exist, this is a program bug (e.g., dereferencing virtual address 0 / NULL) and the faulting program should be terminated. But this can also be used for other purposes such as demand paging to save on IO and better-manage physical memory [34].

The key pieces of VMM functionality are adding a new page mapping (whether the mapped page contains zeros, file data, or swap data), and removing an existing page mapping. While this initially sounds like relatively modest functionality whose implementation may be complicated by hardware subtleties, correctness of even these basic operations are actually quite intricate. Notably, updates to the page tables are performed as writes to memory — *which are themselves subject to address translation*, and finding the correct page table to update requires converting between physical and virtual addresses. In the case of changing the mappings for the currently-active set of page tables, *the OS kernel is modifying the tables involved in its own access of the tables*. To get a sense of how subtle the required reasoning is, we can consider code such as Figure 2, used in our evaluation to locate the appropriate L1 entry to map a page into the current address space. `walkpgdir` (right) essentially mimics the hardware address translation up to the L1 entry (its caller will modify the entry to map a new page), indexing into each successive table (e.g., with `L4offset(va)` retrieving the L4 offset of Figure 1), with `pte_get_next_table` (left) fetching the base address of the next table from each entry. If, as shown in Figure 1, an entry such as the L2 entry is uninitialized (the *present* flag is not set), `pte_get_next_table` allocates the next table on Line 4 and marks it valid on Line 5. But there is a critical difference from hardware: every memory access in Figure 2 uses *virtual* addresses, as opposed to hardware's direct physical memory access. Thus `pte_get_next_table` uses the P2V (physical-to-virtual) macro on line 9 to convert physical addresses stored in page table entries into virtual addresses which the kernel can use to access the corresponding physical location. Proving it correct requires proving that it yields an address that is not only mapped, but known to map back to the original physical address! Verifying this code is beyond the reach of prior work, which either does not model address translation for the kernel [3, 40, 42, 49, 50] and would thus reason unsoundly about this code, or models address translation but lacks features required to reason about this code [51, 52]. We describe our proof of correctness for this code, based on the first formalization of critical VMM kernel invariants in an MMU-aware logic, in Section 5.

Virtual memory affects the OS scheduler, which deals with multiple address spaces, so must track which virtual addresses are valid in which address spaces. Some virtual addresses are valid in only

⁴Not to be confused with Virtual Machine Monitor. We focus on non-hypervisor scenarios, but hardware virtualization extensions for both x86-64 and ARM make use of an additional set of page tables translating what a *guest* considers to be its (virtualized) physical memory to actual physical memory. Our contributions should offer value in this scenario as well.

a single address space (e.g., a code address for a particular usermode process), while others are valid in all address spaces (e.g., kernel data structure pointers). The VMM must maintain some of these assumptions on behalf of the rest of the kernel, for example by guaranteeing that a certain range of virtual addresses (corresponding to the kernel's code and data) are valid in every address space.

Out of Scope: Translation Lookaside Buffers (TLBs). CPUs with MMUs typically have an additional *translation lookaside buffer* (TLB), to cache the (successful) results of page table walks, rather than transforming every virtual memory access into 5 physical accesses. Any time a virtual address that was accessible becomes inaccessible (or has downgraded permissions), the TLB (or at least entries in affected virtual address ranges) should be flushed. In most kernels, this occurs in only a few well-known places, which is why existing verified kernels `SEL4` [49, 50] and `CERTIKOS` [40, 41] currently trust that TLB flushes are handled correctly rather than actually modeling TLB hardware and verifying. Eventual verification of TLB handling is a worthwhile long-term goal, but it is a challenging pursuit in its own right. Based on others' early progress on verifying TLB operations in isolation [70], we expect it to be possible to combine this paper's insights with that support. Section 6 elaborates briefly on the challenges involved. Even without TLB modeling, our logic already enables verification of virtual memory management functionality that prior verified kernel work either trusts completely (`SEL4`, `CERTIKOS`) or is incapable of reasoning about.

2.2 Separation Logic

Separation logic [66] is a descendant of classic Hoare logic [44], where in addition to pre- and postcondition assertions, assertions themselves pick up a *separating conjunction* operator $*$, such that an assertion $A * B$ means A and B are true of disjoint pieces of state. This allows for local reasoning about updates, because it articulates that updates to the state backing A do not affect the truth of B . This is classically demonstrated through the *points-to* assertion: $l \mapsto v$ asserts that the memory cell at address l holds value v : knowing $x \mapsto 3 * y \mapsto 4$ and writing through x means information about y is preserved: $\{x \mapsto 3 * P\} x := 5 \{x \mapsto 5 * P\}$ can be derived for any P .

We build on the `IRIS` [48] separation logic framework, an abstract separation logic embedded in `Rocq`, which is useful for both metatheoretical work and interactive correctness proofs using the logic. Given an operational semantics structured a certain way (in our case, semantics for a fragment of x86-64 assembly including address translation), if a small number of “glue” lemmas are proven, `IRIS` provides a ready-made separation logic with a number of advanced features, including higher-order ghost state and impredicative invariants, for no additional work.

We suppress some technical `IRIS` details for brevity, but briefly note a few recurring details used heavily in `IRIS` but not necessarily in traditional separation logics. Because `IRIS` is an embedded framework in `Rocq`, proofs in `IRIS`-derived logics often encapsulate raw `Rocq` assertions: $\ulcorner P \urcorner$ is an embedding of the `Rocq` assertion P into an `IRIS` assertion (used for things like equality and other general pure logical assertions), similar to prior `Rocq`-embedded program logics [19]. Newly-defined `IRIS` assertions are in fact `Rocq` terms of a particular type, rather than being drawn from a fixed vocabulary.

`IRIS` includes two forms of implication. The magic wand operator \multimap is an affine implication: $A \multimap B$ describes a resource which, if combined with a resource satisfying A , will satisfy B . Notably, this implication involves no changes to ghost state. `IRIS`, building on the `Views` framework [33], also includes a *view shift* operator \Rightarrow which models updates to ghost state: $A \Rightarrow B$ means resources satisfying A may be transformed into resources satisfying B , intuitively by updating only ghost state (a slight simplification of `IRIS`'s update modalities, but adequate intuition for non-`IRIS`-experts). View shifts are essentially logical entailment plus ghost updates.

An important limitation is that to date, every separation logic has assumed that all pointer addresses are meant for use in a single address space, which avoided the problem of tracking that certain points-to assertions are true only for certain address spaces, but not others.

2.3 Modal Logic

The problem of needing to keep track of things being true in some contexts and not in others is hardly unique to virtual memory management, and is the general insight behind most flavors of modal logic, which use unary operators to express that a logical claim P is *contingently* true in certain other circumstances, such as in other times [64] or places [39, 60].

Of particular interest for reasoning about virtual memory are modalities that permit *naming* the alternate circumstances, prominently *hybrid* modal logics [7, 12], which come equipped with assertions of the form $[\ell](P)$ indicating that P is true in the specific alternate circumstance (Kripke world) named by the term ℓ . Note that a distinctive property of hybrid logics is that, rather than hiding the points at which a modal assertion is evaluated inside the modality's definition, the choice of what world a modalized assertion should be true in is chosen *in the assertion itself*. This allows assertions to talk about not simply whether some other assertion is true in some possible future or past world related in a fixed way to the current world, but to talk about *arbitrary* other worlds. This is somewhat different from the accessibility relation modalities more common in verification, but also well-established, tracing its origin back to Arthur Prior [8], just as many temporal logics do.

This explicit naming of alternate worlds increases the power of propositional modal logics [12], and is necessary for completeness in classical separation logics [15]. However typically modal logics can always be “compiled” into logics with suitable quantifiers: the *standard translation* [13] of propositional modal logics into first-order logic has been thoroughly studied. IRIS's support for higher-order impredicative quantification means this applies to our modalities as well — they do not strictly speaking make IRIS more expressive. But while propositional modal logics are no more expressive than first-order logic (with just two variables!), the primary goal of a modal logic is not raw expressive power but intuitive specifications and proof rules. Temporal logics are used because they simplify specifications (vs. having explicit time variables in every base assertion) and reasoning principles (because the modal correspondents of quantification over time is used in a highly structured way). Our logic offers a similar value proposition: natural specifications and reasoning principles which are powerful enough for reasoning about virtual memory management, but without requiring pervasive tracking of and explicit abstraction over address spaces.

For our purposes, these are natural candidates to adapt for virtual memory management. We can reinterpret the notion of naming an alternate world slightly more loosely, and instead name *address spaces* by the physical address of the page table root, since these structures are the physical representations of page tables. Thus in this paper we develop the notion that we can represent contingent truth of an assertion via $[r](P)$ indicating that P holds in the address space rooted at physical address r . Because OS kernels create and destroy address spaces, it is sensible to use a hybrid-style logic that is not specialized to a fixed set of modalities, but this introduces some subtleties from the fact that the existence of certain modalities (address spaces) can change.

Interaction of hybrid modalities and substructural reasoning is relatively unexplored (see Section 6). Our development atop IRIS [48] needs to explore some additional subtleties that arise where the modality itself may entail ownership of resources, as well as interactions between our hybrid-style modality and substructural rules. Some prior IRIS-based work [27, 28] has constructed derived modalities in the style we propose, indexed by thread IDs. However their interpretation of those modalities was fully fixed ahead of time (to refer to essentially buffers in operationalized versions of C11 concurrency). In this setting, while our modalities will be indexed by page table roots, it is

possible to modify the address translation for an address space with root r — thus changing the interpretation of a modality, and even whether a modality is valid — *while assertions with that modality are active*. This is essential to updating page mappings for the current address space in use by a CPU.

3 Machine State and Semantics

To develop our core logical ideas, we instantiate IRIS with a simple language for streams of instructions and a logical machine model corresponding to execution of x86-64 assembly instructions with virtual memory enabled on the CPU. A register identifier, r , is chosen from a fixed finite set of register identifiers, greg .⁵ We use these identifiers r for register names such as `rax`, `r8`, or `cr3`. Our model includes all x86-64 integer registers (including stack and instruction pointers), as well as `cr3` (for page table roots) and `rflags` (for flags set by comparison operations and inspected by conditional jumps). For clarity and ease of representation, we use machine words, $w_n \in \mathcal{W}_n$, with the subscripts showing the number of bits in a word, for memory addresses, values, and offsets, rather than distinct location types that wrap machine words. For example, w_{12} is a 12-bit word, which can be obtained for example truncating away 52 bits of a 64-bit word (w_{64}). We represent the machine state mainly as a finite map of registers to register values and a map of word-aligned physical memory addresses to 64-bit physical memory values. Thus our states σ include register maps $\sigma.\mathcal{R} : \text{greg} \rightarrow_{\text{fin}} \text{regval}$ and memory maps $\sigma.\mathcal{M} : \mathcal{W}_{52} \rightarrow_{\text{fin}} (\mathcal{W}_{12} \rightarrow_{\text{fin}} \mathcal{W}_{64})$ which segment memory into page-sized increments. Of particular note, `cr3`, the page table register, is included in the machine state.

Programs in our logic are instruction sequences \vec{i} , which are formed by either a basic instruction skip, or prefixing an existing instruction sequence with an additional instruction ($i; \vec{i}$). We model (and later, give program logic rules for) instructions for register loads and stores, and reading and writing memory. The latter require page table walks. The most important instructions that we model are memory-accessing variants of the x86-64 `mov` instruction, which we format in Intel syntax (destination on the left, source on the right). Thus, a store to memory is `mov [rm] rr`, and a load from memory is `mov rr [rm]`. Operationally, each first translates the virtual address stored in the register r_m to a physical address (a page table walk), then either updates the memory at that physical location with the contents of r_r (for store) or updates r_r with the contents of that memory (for load). Our formalization includes additional `mov` variants (e.g., accessing memory at constant offsets from the base register, or moves between registers), basic integer and bitwise operations (`add`, `and`, bit shifts, etc.) with their effects on `rflags`, jumps and some (not all) conditional jumps, `call`, `ret`, `push`, and `pop`.

4 Program Logic for Location Virtualization

A program logic for reasoning about code which may work with (and possibly update) multiple address spaces requires dealing with several key challenges. It must ensure that reasoning about memory accesses only depends on assertions that hold in the active address space at the time of access. It must allow invariants for code or data structures to refer to other address spaces. These constraints mean it must also support reasoning about when the active address space changes, as this affects which memory assumptions are usable or not for memory access and thus which data structures are immediately directly accessible. Many approaches could handle these problems in principle, such as tagging pointers with the relevant address space. But such approaches introduce other complexities. Most code, even in an operating system kernel, only works with a single address space (the current one). Explicitly plumbing address space information through a simple linked list specification, simply because some other part of the program may manipulate other address spaces, adds significant specification burden. Specifications which need to talk about a particular invariant

⁵ r abbreviates *general register*, which includes integer registers (e.g., `rax`), control registers (e.g., `cr3`), and segmentation registers (which we do not discuss in detail, but are still used in a limited way in 64-bit mode on modern x86-64 processors).

holding in a specific other address space (or multiple other address spaces) would need to quantify over *functions* from address space identity to assertions rather than just assertions.

Using a modal approach resolves all of these challenges cleanly and uniformly. Specifications that are not *about* address space manipulation need not mention address spaces in assertions. One can use standard separation logic data structure specifications without restructuring or adding explicit address space tracking, but every assertion can be still stated for either the current or specific other address spaces as needed. In short, modalities make it possible for specifications to mention address spaces when it is important to the code, and not when it is unimportant to the code.

We describe a program logic (a separation logic) along the lines suggested above, where every assertion is relative to an address space in which it is interpreted, allowing us to define *virtual points-to* assertions that make claims about memory locations in a particular address space. Virtual addresses, and even virtual points-to assertions, are not tagged with their address spaces in any way. Memory access in this logic is validated through the use of virtual points-to assertions in preconditions, which guarantee that address translations succeed. This supports rules for updating not only typical data in memory that happens to be subject to address translation, but also manipulation of the page tables themselves via virtual addresses (as demanded by all modern hardware) and via virtual points-to assertions. To support specifications that deal with multiple address spaces, our logic incorporates a hybrid-style modality $[r](P)$ to state that an assertion is true in another (assertion-specified) address space rather than the address space currently active in hardware, which is not only useful for virtual memory manager invariants, but also critical to reasoning about change of address space. By developing this within the IRIS framework, we obtain additional features (e.g., fractional permissions) that allow us to verify some of the most subtle and technically challenging instruction sequences in an OS kernel (Section 5).

To support making assertions depend on a choice of address space, we work entirely in a pointwise lifting of IRIS's base BI logic, essentially working with separation logic assertions indexed by a choice of page table root as a \mathcal{W}_{64} , which we call $\text{vProp } \Sigma$.⁶ **Definition** $\text{vProp } \Sigma : \text{bi} := \text{word } 64 \text{ } \text{-b> } \text{iPropI } \Sigma$. This is the (RocQ) type of assertions in our logic. Most constructs in IRIS's base logic are defined with respect to any BI-algebra (of RocQ type bi), so they automatically carry over to our derived logic. However, we must still build up from existing IRIS primitives to provide new primitives that depend on the address space — primarily the notion of virtual points-to. To define and use virtual points-to assertions, we require two basic assertions that ignore the current address space:

Register points-to. The assertion $r \mapsto_r \{q\} \text{ rv}$ ensures the ownership of the register r containing the value of the register rv . The fraction q with value 1 asserts the unique ownership of the register mapping and grants update permission to it; otherwise, any value $0 < q < 1$ represents partial ownership, granting read-only permission on the mapping.⁷

Physical memory points-to. The soundness proofs for our logic's rules largely center around proving that page-table-walk accesses as in Figure 1 succeed, which requires assertions dealing with physical memory locations. We have two notions of physical points-to facts. The primitive notion closest to our machine model is captured by an assertion $\text{pfn} \sim \text{pageoff} \mapsto_a \{q\} v$, where pfn (a \mathcal{W}_{52} *page frame number*) essentially selects a 4KB page of physical memory, and pageoff (a \mathcal{W}_{12}) is an offset within that page. From this we can derive a more concise physical points-to when the split is unimportant: $w \mapsto_p \{q\} v \triangleq (\text{drop } 12 \text{ } w) \sim (\text{bottom } 12 \text{ } w) \mapsto_a \{q\} v$

⁶IRIS experts may notice our -b> resembles another pointwise lifting already in IRIS [27, 28]. This similarity is real, but the existing lifting does not appear to work with indexed RocQ types like our $\text{word } n$ as a domain.

⁷We adopt the standard naming convention of q -related names representing fractional permission, with fractions sometimes appearing in braces or as subscripts in various assertions.


```

1  Definition  $va \mapsto_t \{q\} v : vProp \Sigma :=$ 
2     $\exists l4e \ l3e \ l2e \ l1e. \ulcorner aligned \ va \urcorner * L4\_L1\_PointsTo(va \ l4e \ l3e \ l2e \ l1e \ paddr) * paddr \mapsto_p \{q\} v.$ 
3  Definition  $L4\_L1\_PointsTo(maddr \ l4e \ l3e \ l2e \ l1e \ paddr : word \ 64) : vProp \Sigma := \lambda \ cr3val.$ 
4     $\ulcorner entry\_present \ l4e \wedge entry\_present \ l3e \wedge entry\_present \ l2e \wedge entry\_present \ l1e \urcorner *$ 
5     $(14M52 \ maddr \ cr3val) \sim (14off \ maddr \ cr3val) \mapsto_a \{q1\} \ l4e *$ 
6     $(13M52 \ maddr \ l4e) \sim (13off \ maddr \ l4e) \mapsto_a \{q2\} \ l3e *$ 
7     $(12M52 \ maddr \ l3e) \sim (12off \ maddr \ l3e) \mapsto_a \{q3\} \ l2e *$ 
8     $(11M52 \ maddr \ l2e) \sim (11off \ maddr \ l2e) \mapsto_a \{q4\} \ l1e * \ulcorner addr\_L1(va, l1e) = paddr \urcorner.$ 

```

Fig. 3. A Strong Virtual Points-to Relation

4.1 An Overly-Restrictive Definition for Virtual Memory Addressing

A natural definition for a virtual points-to asserting that virtual address va points to a value v would contain partial ownership of the physical memory involved in the page table walk that would translate va to its backing physical location — with locations existentially quantified since a virtual points-to should not assert *which* locations are accessed in a page table walk, as in Figure 3. It asserts the existence of four page-table entries, one at each translation level, and via $L4_L1_PointsTo$ asserts that the physical page table walk (per Figure 1) succeeds in reaching the L1 entry, which points to the page holding the physical memory backing the virtual address, which contains value v . Most of the definition lives directly in $vProp$, using the separation logic structure lifted from IRIS's $iProp$.

$L4_L1_PointsTo$ works by chaining together the entries for each level, using the sequence of table offsets from the address being translated to index each table level, and using the physical page address embedded in each entry.⁸ For example, the first-level address translation to get the L4 entry (l4e) uses the masks 14M52 with the current $cr3$ to get the physical address of the start of the L4 table and 14off with the virtual address being translated to compute the correct byte offset within that table just as in the first translation step of Figure 1.⁹ Thus Line 5 asserts that the physical address built from the table base and offset points to the L4 entry l4e. Subsequent levels of the page table walk assertion (Lines 6–8) work similarly. The statement of these assertions is simplified by the use of our split physical points-to assertions, since each level of tables is page-sized.¹⁰ This helper definition is also more explicit in $vProp$ which binds a value to $cr3$ and uses it to start the translation process.

This solution is in fact very close to that of Kolanski and Klein [51], who define a separation logic from scratch in ISABELLE/HOL, where the semantics of all assertions are functions from pairs of heaps and page table root values to booleans.¹¹ Our solution in the next subsection improves on theirs, removing some restrictions in this definition by further abstracting the handling of address translation.

4.2 Aliasing/Sharing Physical Pages

The virtual points-to definition shown in Figure 3 is too strong to specify some operations that a virtual memory manager may need to do, such as move one level of the page table to a different physical location while preserving all virtual-to-physical mappings. The use of $L4_L1_PointsTo$ in Figure 3's virtual points-to definition stores knowledge of the page table walk details with ownership of the backing physical memory. Updating any of these mappings (e.g., moving the page tables in physical memory, as in coalescing for superpages or hugepages) would require explicitly collecting all virtual points-to facts that traverse affected entries. It is preferable to permit the page

⁸The fractions $q1$ through $q4$ represent the fractional ownership of each entry based on how many word-aligned addresses might need to share the entry — $(\frac{1}{512})^n$ for each level n .

⁹Note the offsets mentioned in Figure 1 are 9-bit indexes into the 512 entries; the byte offset is that times 8.

¹⁰We do not address superpages and hugepages in this paper.

¹¹This was a typical explicit construction at the time; their work significantly predates IRIS.

$$va \mapsto_v \{q\} \text{ val} : v\text{Prop} \Sigma \triangleq \exists pa. \exists \delta. (\underbrace{\lambda cr3val. \exists \epsilon. cr3val \hookrightarrow_{\epsilon}^{\delta s} \delta}_{\text{Find addr. space invariant}}) * \underbrace{va \hookrightarrow_q^{\delta} pa}_{\text{Ghost translation}} * \underbrace{pa \mapsto_p \{q\} \text{ val}}_{\text{Physical location}}$$

Fig. 4. Virtual-Points-to for Sharing Pages

$$\frac{\text{GhostMap}(\gamma, \theta) * pa \hookrightarrow^{\gamma} va \Rightarrow \text{GhostMap}(\gamma, \theta[va \mapsto pa']) * va \hookrightarrow^{\gamma} pa'}{\text{GHOSTMAPUPDATE}}$$

$$\frac{\text{GhostMap}(\gamma, \theta) * va \hookrightarrow^{\gamma} pa \dashv \ulcorner \theta(va) = \text{Some}(pa) \urcorner}{\text{GHOSTMAPLOOKUP}}$$

Fig. 5. IRIS rules for ghost maps

$$IASpace(\theta, m) \triangleq ASpace_Lookup(\theta, m) * \underset{(va, pa) \in \theta}{*} \exists (l4e \ l3e \ l2e, l1e, pa). L4_L1_PointsTo(va, l4e, l3e, l2e, l1e, pa)$$

where $ASpace_Lookup(\theta, m) \triangleq \lambda cr3val. \exists \delta. \ulcorner m !! cr3val = \text{Some}(\delta) \urcorner * \mathcal{A}bsPTableWalk(\delta, \theta)$

Fig. 6. Per-address-space invariant with a fixed global map of address space names m

tables themselves to be updated independently of the virtual points-to assertions, so long as those updates preserve the same virtual-to-physical translations. But this is not possible with Figure 3's definition, which ties ownership of particular pieces of page table memory to the virtual points-to.

We separate the physical page-table walk from the virtual points-to relation, replacing it with a ghost state that merely guarantees that the address translation would succeed. IRIS includes a *ghost map* construction, which we use to track mappings from virtual addresses to the physical addresses they translate to as a piece of ghost state. The map includes, for each key in the map (i.e., each virtual address), a token $k \hookrightarrow^{\gamma} v$ whose ownership is required to update that key-value pair in the ghost map named γ . The existence of such a token implies that the actual map θ tracked by a corresponding $\text{GhostMap}(\gamma, \theta)$ resource indeed maps k to v . These properties are captured by key IRIS rules in Figure 5.¹² There are other rules, but these two are most important for explaining ghost maps. GHOSTMAPUPDATE says that ownership of the actual ghost map with ghost name γ and map contents θ , and a token witnessing that θ maps pa to va permits an update to the ghost map's state, changing the map and replacing the token to represent the new value. GHOSTMAPLOOKUP allows using the same information to simply conclude that the mapping indicated by the token is true.

The *virtual memory manager's invariant* ensures that for each $va \hookrightarrow^{\gamma} pa$ mapping in this map, there are *physical* resources sufficient to ensure that the address translation for va will resolve on the hardware to pa — via $L4_L1_PointsTo$. This kernel invariant turns out to be a key ingredient in supporting proofs of VMM functionality: in Section 5 we will see that separating the logical and physical virtual-to-physical mappings is what allows stating the global kernel invariants needed for software page traversals, which prior work did not (and could not) pursue.

For clarity, we refer to the specific ghost map summarizing virtual-to-physical translations by $\mathcal{A}bsPTableWalk(\delta, \theta) \triangleq \text{GhostMap}(\delta, \theta)$ (omitting δ for brevity when only one is in scope) and keep this in a per-address-space invariant described shortly. We then replace the physical traversal $L4_L1_PointsTo$ in Figure 3's virtual points-to definition with ownership of the token $va \hookrightarrow_q^{\delta} pa$, yielding Figure 4's definition. This new definition guarantees that the ghost map θ maps the virtual address (va) to a physical address (pa), and thus that the per-address-space invariant described next will contain the physical resources that guarantee that the hardware resolves the translation.

We place the authoritative ownership of the ghost map translation $\mathcal{A}PTableWalk$ in a per-address-space invariant $IASpace$ (Figure 6), allowing changes to the page tables that preserve overall virtual-to-physical translations in isolation, and also allowing changes to specific the virtual-to-physical translations when combined with the token stored in the relevant virtual points-to (Figure 4).

¹²IRIS ghost maps lack established notation; the syntax we use captures the details of `iris.base_logic.lib.ghost_map`.

$$\begin{array}{c}
\frac{}{[r](P) : \text{vProp } \Sigma \triangleq \lambda_{_} . P \text{ r}} \quad \frac{}{\text{Fact } P \triangleq \forall r . r' . P \text{ r} \dashv\vdash P \text{ r}'} \quad \frac{}{\text{Fact } [r](P)} \quad \frac{}{\text{Fact } (r \mapsto_r \{q\} \text{ rv})} \\
\frac{}{\text{Fact } (w \mapsto_p \{q\} \vee)} \quad \frac{}{(P \vdash Q) \vdash ([r](P) \vdash [r](Q))} \quad \frac{}{[r](P * Q) \dashv\vdash ([r](P) * [r](Q))} \\
\frac{}{[r](P \wedge Q) \dashv\vdash [r](P) \wedge [r](Q)} \quad \frac{}{[r](P \vee Q) \dashv\vdash [r](P) \vee [r](Q)} \quad \frac{}{\text{Fact } P \vdash [r](P) \dashv\vdash (P)}
\end{array}$$

Fig. 7. Other-space Modality and Its Laws

We must also ensure that different address spaces can have independent ghost maps — which we resolve with an additional unique global ghost map (with ghost name δs) from address-space identifiers (page table roots whose values are manipulated by the kernel code) to the IRIS ghost name for that address space’s ghost map. In Figure 4, the extra ghost map token for δs asserts that δ — which is existentially quantified — is the correct ghost name for the current address space. That is then the ghost map named in the ghost virtual-to-physical translation token of Figure 4. Just as the ghost name δ names the ghost map with contents θ , δs names a ghost map, whose contents appear as m in Figure 6 (the association of δs to m is a global invariant not shown). $\text{IASpace}(\theta, m)$ then performs 3 roles: it associates the current address space’s root with an appropriate IRIS ghost name δ ; it tracks authoritatively that δ ’s logical contents match θ ; and it stores the physical resources for the current address space mappings (via the iterated $\text{L}_4\text{L}_1\text{PointsTo}$).

4.3 Address Space Management

Real VMMs must handle more than one address space. Doing so requires a way to talk about other address spaces, and means to switch address spaces. Figure 7 gives the definition of our modal operator for asserting the truth of a modal (address-space-contingent) assertion *in another address space*, which we call the *other-space* modality. The definition itself is not particularly surprising — as our modal assertions are semantically predicates on a page table root (physical) address, the assertion $[r](P)$ is a modal assertion that ignores the (implicit) current page table root, and evaluates the truth of P as if r were the page table root. The novelty here is not in the details of the definition, but in recognizing that this is the right way to deal with multiple address spaces, and working out how to support interaction of multiple address spaces (discussed in the next section). The modal assertions, together with the other-space modality, mean we can give generic definitions of data structure assertions (e.g., linked lists, etc.) which do not need to track information about their own address space. In fact, *only* assertions that explicitly deal with multiple address spaces need to mention address spaces at all (via the other-space modality).

We can prove that this modality follows certain basic laws, showing that its truth is independent of the address space in which it is considered, that it distributes over various logical connectives, and that it follows the rule of consequence. We call vProp assertions whose truth is independent of the current address space Facts; these include other-space assertions, physical memory points-tos, and register assertions. Facts can freely move in and out of other address space modalities.

In general, per-address-space invariants should be collected in a larger VMM invariant, with individual address spaces’ invariants pulled out as needed, such as when proving soundness of an individual virtual memory access. However, such larger invariants would contain many kernel-specific properties that are orthogonal to the fundamental reasoning principles that are the focus of this paper. We leave such kernel-specific reasoning to future work, but our verification of task switching (Section 5) demonstrates support for managing multiple address spaces.

4.4 Selected Logical Rules

As common for assembly-level verification [62, 63], we define our logic using Hoare doubles:

$$\{\Phi\}_{\text{rtv}} e : \text{iProp } \Sigma := ((\text{cr3} \mapsto_r \text{rtv} * \Phi) \text{ rtv}) * \text{WP } e \{ _, \text{True} \}$$

$$\begin{array}{c}
\text{WRITEToREGFROMVIRTMem} \\
\frac{\left\{ \begin{array}{l} P * \text{IASpace} * \text{rip} \mapsto_r \text{iv} + \text{MovLen}(r_d, [r_a]) * \\ r_d \mapsto_r v * r_a \mapsto_r \{q\} \text{ vaddr} * \text{vaddr} \mapsto_v \{q\} v \end{array} \right\}_{\text{rtv}} \overline{is}}{\{P * \text{IASpace} * \text{rip} \mapsto_r \text{iv} * r_d \mapsto_r \text{rvd} * r_a \mapsto_r \{q\} \text{ vaddr} * \text{vaddr} \mapsto_v v\}_{\text{rtv}} \text{ mov } r_d, [r_a]; \overline{is}} \\
\\
\text{WRITEToVIRTMemFROMREG} \\
\frac{\{P * \text{IASpace} * \text{rip} \mapsto_r \text{iv} + \text{MovLen}(r_d, [r_a]) * r_s \mapsto_r \{q_1\} \text{ rvs} * r_a \mapsto_r \{q_2\} \text{ vaddr} * \text{vaddr} \mapsto_v v\}_{\text{rtv}} \overline{is}}{\{P * \text{IASpace} * \text{rip} \mapsto_r \text{iv} * r_s \mapsto_r \{q_1\} \text{ rvs} * r_a \mapsto_r \{q_2\} \text{ vaddr} * \text{vaddr} \mapsto_v \text{rvs}\}_{\text{rtv}} \text{ mov } [r_a], r_s; \overline{is}} \\
\\
\text{WRITEToREGCTLFROMREGMODAL} \\
\frac{\{[\text{rtv}](P * \text{IASpace}) * \text{rip} \mapsto_r \text{iv} + 4 * \text{IASpace} * R * r_s \mapsto_r \{q\} \text{ rvs}\}_{\text{rvs}} \overline{is}}{\{P * \text{IASpace} * \text{rip} \mapsto_r \text{iv} * [\text{rvs}](R * \text{IASpace}) * r_s \mapsto_r \{q\} \text{ rvs}\}_{\text{rtv}} \text{ mov cr3}, r_s; \overline{is}}
\end{array}$$

Fig. 8. Proof rules for selected AMD64 instructions

Our Hoare doubles $\{\Phi\}_{\text{rtv}}$ e state that the expression (i.e., sequence of instructions) e are safe to execute (will not fault) when executed with $v\text{Prop}$ precondition $\Phi * \text{cr3} \mapsto_r \text{rtv}$. WP is Iris’s own weakest precondition modality, unmodified [48]. Making rtv a parameter to the double (vs. a simple register assertion) makes it possible to ensure ownership of the cr3 register and its value is accounted for while avoiding some technical headaches with trying to enforce that Φ itself contains that.

The rest of this section describes specifications of three key AMD64 instructions in our logic. These rules and others (e.g., including accessing memory at an instruction-specified offset from a register value, which is common in most ISAs) can be found in our artifact. In general, we use metavariables r_s and r_d to specify source and destination registers for each instruction, and prefix various register value variables with rv . We sometimes use r_a to emphasize when a register is expected to hold an address used for memory access, though the figure also uses typical assembler conventions of specifying memory access operands by bracketing the register holding the memory address. Standard for Hoare doubles, there is a frame resource P in each rule for passing resources not used by the first instruction in sequence through to subsequent instructions. Our rules include tracking of each instruction’s memory address to track rip updates, which is critical for control transfer instructions. Our development also includes handling of the rflags register updates from arithmetic instructions. Most rules are otherwise standard (e.g., mov between registers, etc.), with Figure 8 showing the rules most unique to our development. As a reminder, in systems of Hoare doubles, an instruction’s precondition appears in the conclusion of the rule, and an instruction’s “postcondition” appears as the precondition to subsequent instructions in the antecedent of the rule.

4.4.1 Accessing Virtual Addresses. Figure 8 includes two rules for accessing memory at an address stored in a register r_a . Setting aside P , IASpace , and the instruction pointer rip , $\text{WRITEToREGFROMVIRTMem}$ and $\text{WRITEToVIRTMemFROMREG}$ are nearly-standard (assembly) separation logic rules for memory accesses [19, 63]. For example, $\text{WRITEToREGFROMVIRTMem}$ ’s specification reflects that it reads from the (virtual) memory address vaddr stored in the address register r_a — and thus requires register points-to and virtual points-to assertions describing that relationship and the assumed value v in memory in its precondition (the precondition of the rule’s conclusion). It reflects the load (mov) of that memory value into the destination register r_d , with the updated register points-to in the precondition for \overline{is} . P describes framed resources, which are passed along to the precondition of subsequent instructions, as in any system of Hoare doubles [19, 63]. $\text{WRITEToVIRTMemFROMREG}$ is analogous for writing to memory. There are only two changes specific to our approach.

First, because we split the physical resources for the page table walk from the virtual points-to itself (per the discussion of Section 4.2), the rule requires IASpace for the current address space to be carried through. The soundness proofs for these rules extract the token $(va \hookrightarrow_q^\delta pa)$ from the

virtual points-to, use that to extract the physical page-table-traversal points-to collection describing the page table walk for the relevant address ($L_4_L_1_PointsTo$) from the invariant ($IASpace$), prove that the page table walk succeeds and that memory or registers are updated appropriately, before re-packing the invariant and virtual points-to resources.

Second, the memory access rules — as with all rules in our logic — increment the instruction pointer `rip` by the length of the encoded instruction. `MovLen` returns how long the instruction encoding for the corresponding `mov` is; x86-64 instruction encodings are often longer for instructions using registers that are absent from the 32-bit x86 ISA that preceded x86-64.

Note that the use of a modal abstraction of address space simplifies these rules. The antecedents of `WRITEToREGFROMVIRTMem` and `WRITEToVIRTMemFROMREG` only mention the address space in the index of the Hoare double — not in P , or the (virtual) points-to assertions. There is no extra condition to discharge that the address being accessed is from the current address space.

4.4.2 Updating `cr3`. Unlike other rules, `WRITEToREGCTLFROMREGMODAL` updates the root address of the address space determining the validity of resources, from `rtv` before the `mov` to `rvs` afterwards. The global effects of this rule are reflected in moving assertions for the current address space (P and $IASpace$) under an other-space modality for the initial (outgoing) address space `rtv`, and moving the new address space's assertions out of the corresponding modality (since after the `mov`, those will hold in the new current address space). The *global* aspect is important. A naïve frame rule would be unsound for `cr3` updates: one could frame out assertions in one address space, switch address spaces, and bring those assertions from the *old* address space back into the *new* address space, where they may not hold. It is often said that the frame rule (below) is one of the key pieces of separation logic.

$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}} \text{FRAME} \quad (\text{unsound with address space changes})$$

Such a rule is normally recoverable from Hoare doubles (see, e.g., Chlipala [18, 19]). However, in the presence of address space changes, the traditional frame rule is unsound. Consider:

$$\frac{\dots}{\frac{\{Pre\} \text{mov } \%cr3, r \{Post\}}{\{a \mapsto_v x * Pre\} \text{mov } \%cr3, r \{a \mapsto_v x * Post\}} \text{FRAME}}$$

In this (broken!) hypothetical example, both the precondition and the postcondition assert that $a \mapsto_v x$ in the current address space, but the new address space may map a to another value. So, this derivation clearly leads to an unsound conclusion. The essential problem is that the frame rule is motivated by local reasoning about local updates, but a switch of address space is a *global* change that may invalidate information about virtual addresses. Thus, framing around arbitrary `cr3` updates is unsound — hence the *global* nature of `WRITEToREGCTLFROMREGMODAL` — though a variant ensures the same `cr3` value is installed before and after the framing can be recovered.

4.5 Soundness

Our rules from Figure 8 are proven to be sound in Iris against an assembly-level hardware model implementing a fragment of x86-64, including 64-bit address translation with 4-level page tables. Our rules for control transfers (`jne`, `call`, and `ret`) are currently axiomatized (with completely standard specifications [19, 63])¹³ because Iris's built-in machinery does not provide convenient ways to discard the current continuation; adaptation of others' approaches [29] is future work. Our soundness proofs for all other instructions (including, critically, all memory accesses) are axiom-free.

¹³An assertion that code at some address is safe to call with a given precondition [62] asserts that the address is mapped and that memory at that address decodes to an instruction sequence that is safe with that precondition.


```

1  {P * IASpace( $\theta, \Xi, m$ ) * [rtv'] (IASpace( $\theta', \Xi', m'$ ) * Pother) * rsi  $\mapsto_r$  restore * rdi  $\mapsto_r$  save * rbx  $\mapsto_r$  rbxv}rtv
2  {rsp  $\mapsto_r$  rspv * rbp  $\mapsto_r$  rbpv * r12  $\mapsto_r$  r12v * r13  $\mapsto_r$  r13v * r14  $\mapsto_r$  r14v * r15  $\mapsto_r$  r15v}rtv
3  {ContextAt(save, _) * ContextAt(restore, [rbxv', ..., rtv'])}rtv
4  mov 0[rdi], rbx ... ;; also save rsp, rbp, r12, r13, r14, to offsets 8, 16, ... and 40 from rdi
5  ... mov 48[rdi], r15
6  {P * IASpace( $\theta, \Xi, m$ ) * [rtv'] (IASpace( $\theta', \Xi', m'$ ) * Pother) * rsi  $\mapsto_r$  restore * rdi  $\mapsto_r$  save * rbx  $\mapsto_r$  rbxv}rtv
7  {rsp  $\mapsto_r$  rspv * rbp  $\mapsto_r$  rbpv * r12  $\mapsto_r$  r12v * r13  $\mapsto_r$  r13v * r14  $\mapsto_r$  r14v * r15  $\mapsto_r$  r15v}rtv
8  {ContextAt(save, [rbxv, ..., rtv]) * ContextAt(restore, [rbxv', ..., rtv'])}rtv
9  mov 56[%rdi], %cr3 {... * rdi+56  $\mapsto_v$  rtv * ...}rtv
10 mov rbx, 0[rsi]
11 mov rsp, 8[rsi] ;; Switch to new stack, which may not be mapped in the current address space!
12 ... ;; load rbp, r12, r13, r14, from offsets 16, 24, 32, and 40 from rsi
13 mov r15, 48[rsi]
14 {P * IASpace( $\theta, \Xi, m$ ) * [rtv'] (IASpace( $\theta', \Xi', m'$ ) * Pother) * rsi  $\mapsto_r$  restore * rdi  $\mapsto_r$  save * rbx  $\mapsto_r$  rbxv'}rtv
15 {rsp  $\mapsto_r$  rspv' * rbp  $\mapsto_r$  rbpv' * r12  $\mapsto_r$  r12v' * r13  $\mapsto_r$  r13v' * r14  $\mapsto_r$  r14v' * r15  $\mapsto_r$  r15v'}rtv
16 {ContextAt(save, [rbxv, ..., rtv]) * ContextAt(restore, [rbxv', ..., rtv'])}rtv
17 mov cr3, 56[rsi] ;; <-- Switch to the new address space
18 {[rtv] (P * IASpace( $\theta, \Xi, m$ ) * ContextAt(save, [rbxv, ..., rtv]) * ContextAt(restore, [rbxv', ..., rtv'])))}rtv
19 {IASpace( $\theta', \Xi', m'$ ) * Pother * rsi  $\mapsto_r$  restore * rdi  $\mapsto_r$  save * ...}rtv

```

Fig. 9. Basic task switch code that switches address spaces.

5 Experiments

In this section, we verify several critical and challenging pieces of VMM code. First, we formally verify a switch into a new address space as part of a task switch, the first such verification handling both old and new processes' assertions (in different address spaces) at the time of the switch. Then, in several stages, we work up to mapping a new page in the current address space, addressing significantly more of this process than prior work that included address translation in its hardware model. This requires a number of independently challenging substeps: dynamically traversing a page table to find the appropriate L1 entry to update; inserting additional levels of the page table if necessary (updating the VMM invariants along the way); converting the physical addresses found in intermediate entries into the corresponding virtual addresses that can be used for memory access; installing the new mapping; and collecting sufficient resources to form a virtual points-to assertion. Of these, only the second-to-last step (installing the correct mapping into the current address space) has previously been formally verified with respect to a machine model with address translation.

5.1 Change of Address Space

A critical piece of *trusted* code in current verified OS kernels is the assembly code to change the current address space; current verified OS kernels currently lack effective ways to specify and reason about this low-level operation, for reasons outlined in Section 6.

Figure 9 gives simplified code for a basic task switch, the heart of an OS scheduler implementation. This is code that saves the context (registers and stack) of the running thread, and resumes execution of a previously-suspended thread of execution. In C this code would be given the signature `void swtch(context_t* save, context_t* restore)`. Saving the context is a straightforward matter of storing each register into the save context. Restoring the restore context is the tricky bit, because both the stack pointer and address space must be restored. Confusingly, a single dynamic execution of this function begins execution in one thread, and returns in another thread — because the execution switches stacks, and thus returns on the second thread's stack. Hence this is used, for example, when the scheduler has chosen a new thread to execute for a voluntary (non-preempted)

context switch, and will call the code with *save* pointing to a reserved storage space for the current thread, and *restore* pointing to the context of the next thread to execute. We will ignore non-integer registers; others are handled similarly. In Figure 9, Lines 4–9 store the callee-save registers (per the System V AMD64 ABI calling conventions) of the calling thread into the context data structure pointed to by *rdi* (at virtual address *save*). This is justified by the $\text{ContextAt}(\text{save}, _)$ assertion in the precondition (Line 3), which expands into a full set of full-permission (thus writable) virtual points-to assertions for various offsets from *save*, one for each saved register. Verification up through line 9 is straightforward application of $\text{WRITE_TO_VIRT_MEM_FROM_REG}$ (Figure 8).

The code to restore the previously-saved context located at *restore* (accessed via *rsi*) in Lines 10–17 is where the proof becomes subtle, though our logic makes the construction of the proof feel similar to typical assembly-level verification because most instructions are verified with rules that work very similarly to standard proof rules while being proven sound against a machine model with address translation. Similar to the precondition for the *save* context, the *restore* context has a corresponding $\text{ContextAt}(\text{restore}, [\dots])$ assertion expanding to virtual points-to assertions – in the caller’s address space. The *mov* instructions prior to Line 17 are each verified with a fixed-register-offset variant of $\text{WRITE_TO_REG_FROM_VIRT_MEM}$, but Line 11’s implications are subtle because it switches stacks by updating *rsp*. Because the new stack pointer may only be valid in the address space of the restored context, stack accesses at this point are unsafe. Prior to Line 17, we can see in code and invariants that the local registers are updated to match the values populating the *restore* context, except for the page table root. Line 17 itself is verified with a rule similar to $\text{WRITE_TO_REG_CTL_FROM_REG_MODAL}$ (but obviously reading from a fixed offset of a register, as needed in Line 17). This rule also globally moves assertions into and out of other-space assertions, to reflect that assertions holding in the outgoing address space *rtv* generally will not hold in the incoming address space *rtv'* and vice versa. Thus the precondition has assertions for the new thread under an other-space modality for the new address space, and the postcondition has assertions for the old thread under an other-space modality for the old address space. Both ContextAt assertions end up under the old other-space modality, but in a real kernel would transfer out based on kernel-specific invariants.

The specification above does not directly discuss the relationship between instruction pointers and registers – and does not need to because *P* and *POther* can be instantiated to capture that relationship with additional information about stack contents. This code is meant to be called with a return address for the current thread stored on the current stack, and a return address for the target thread on the target thread’s stack. For a given call site, *P* would be instantiated to require that the initial stack pointer (before *rsp* is updated) has a return address expecting the then-current callee-save register values in the *current* (initial) address space to (together with other resources used to instantiate *P*) imply the precondition of the code at that return address. The situation for the target thread is similar, but using *POther*, and using the other-space modality because the other thread’s stack, code, and other relevant assertions may only be valid in the new address space. Our logic’s rules for updating the page table root, and thus moving assertions into and out of other-space modalities, neatly manage which assertions are *currently* valid, without the need to explicitly plumb address space labels through every assertion in the larger proof.

Although prior work has verified context switches within a single address space [63], and address space switches without any code before or after [70] (that is, not reasoning about the *impact* of address space change on what data were accessible), this is the first verification that handles both.

5.2 Traversing Live Page Tables

We build up to the main task of mapping a new page after traversing the page tables in the software. This algorithm is complex and corresponds to a significant amount of assembly code. To assist with readability, we present C code for this process, with assertions adjusted slightly to refer to C

program variables rather than registers. The actual verification was carried out on x86-64 assembly *generated from this source code*. Listings of the assembly fragments with inline assertions appear in our technical report [54]. Whether in C or assembly, the page table traversal involved in mapping a new page is very challenging functionality to verify. Loading the current table root from `cr3` is straightforward (a `mov` instruction). However, this produces the *physical* address stored in `cr3`, not a *virtual* address the kernel code can use to access that memory. This problem repeats at each level of the page table: assuming that the code has *somehow* read the appropriate L4 (or L3, or L2) entry, those entries again yield physical addresses, not virtual. The only prior work to verify page mapping ignored the traversal and only verified mapping assuming code *already* had an appropriate virtual address for the L1 entry, where a physical address could simply be stored. Our proof is the first to additionally deal with the critical code leading up to that point.

Code Overview. As described in Section 2, mapping a new page consists of simulating the hardware address translation of Figure 1, but in software. The code for this task takes three explicit parameters: the root pointer (read from `cr3` by earlier code), the page-aligned *virtual* address (`va`) at which to make a new piece of memory accessible, and the *physical* address (`fpaddr`) of the memory to map in that location. The function we ultimately verify, `vaspace_mappage` (Figure 15), relies primarily on a helper function already shown in Figure 2. `walkpgdir` finds the (virtual) address of the the correct L1 entry to translate `va`, by walking the page tables in software one level at a time. `vaspace_mappage` then uses the result to install the new entry. `walkpgdir` itself relies on its own helper function `pte_get_next_table`, also shown in Figure 2, which implements a single-level of traversal from level $n + 1$ to level n (and whose specification and proof are therefore parameterized by page table level), allocating additional levels as needed.

We organize our explanation of the proofs by essentially following execution from the start of `walkpgdir`, through execution of `pte_get_next_table`, and out to its callsite in `vaspace_mappage`. While slightly awkward because we start in the middle of the mapping execution, this ordering allows us to start with the simpler pieces of the proof, and incrementally explain the complexities of the proofs and kernel invariant, before concluding with the top-level verification.

5.2.1 From L_{n+1} Entries to L_n Tables. We discuss access to the level 4 table later (Section 5.2.6). However, for subsequent levels, the base address of level n must be fetched from the appropriate entry in the table of level $n + 1$. This is the role of `pte_get_next_table` (originally Figure 2, with proof details in Figure 10). It is passed the virtual address of the page table entry in level $n + 1$, and should return the *virtual* address of the *base* of the level n table indicated by that entry. If the entry is empty (i.e., this is a sparse part of the page table representation), the code also allocates a page for the level n table, installs it in the level $n + 1$ entry, and establishes appropriate invariants. Figure 10 presents the function with proof annotations that we will explain shortly, but we first explain the functionality. `pte_get_next_table` accepts a *virtual* address entry which points to a level- $n + 1$ table entry.

On Line 9, the code checks the present bit of the entry. If the bit is unset, there is no level- n table, so one must be allocated via `pte_initialize` (explained shortly, but it essentially allocates a fresh physical page, and initializes the memory pointed to by entry with that physical address) and marked present. By Line 20 the entry is known to be valid and contain the physical address of the base of a level n table. That address is then extracted (Line 20), converted to a virtual address (Line 21), and returned to the caller. We can now discuss `pte_get_next_table`'s proof of correctness. While at first glance this code may look like its subtlety is mostly care to distinguish physical and virtual addresses, it has a highly nontrivial correctness argument, which depends critically on detailed invariants on how access to page table entries is shared between parts of the kernel. No prior work has engaged with this problem.

```

1  /* @param entry: virtual address of level n+1 entry which should point to a level-n table base
2     @return next: a virtual pointer to the base of a valid level n table */
3  pte_t *pte_get_next_table(pte_t *entry) {
4      pte_t *next;
5      {P * IASpaceid( $\theta, \Xi \setminus \{entry \mapsto v\}, m\} * r_{tv} \hookrightarrow_{\delta^s} \delta * entry \mapsto_r entry + KERNBASE * (entry) \hookrightarrow_{id} level\}_{rtv}$ 
6      {entry+KERNBASE  $\mapsto_{vpte, qfrac} entryv * \ulcorner qfrac = 1 \leftrightarrow \neg(entry\_present\ entryv) \urcorner\}_{rtv}$ 
7      { $\ulcorner entry\_present(entryv) \urcorner \multimap \forall_{i \in 0..511} table\_root(entryv.pfn) + i * 8 \hookrightarrow_{id} v-1\}_{rtv}$ 
8      /* Reading the page table is entry justified by the virtual pte-points-to */
9      if (!entry->present) { // If the present bit is zero, need to allocate next level
10         {entry+KERNBASE  $\mapsto_{vpte, qfrac} entryv * \ulcorner qfrac = 1 \wedge \neg(entry\_present\ entryv) \urcorner\}_{rtv}$ 
11         pte_initialize(entry); // Allocate a zeroed physical page and install in entry
12         entry->present = 1; // Install newly-allocated level n table, mark present
13         /*Now we know that entry is initialized, and the condition to access children list holds*/
14         {entry+KERNBASE  $\mapsto_{vpte, qfrac} entryv pte\_initialized(pfn\_set(entryv\ nextpaddr)) * \ulcorner qfrac = 1 \urcorner\}_{rtv}$ 
15         { $\forall_{i \in 0..511}. ((table\_root(pte\_initialized(pfn\_set(entryv\ nextpaddr)))) + i * 8) \hookrightarrow_{id} v-1\}$ 
16     }
17     {P * IASpaceid( $\theta, \Xi \setminus \{entry \mapsto v\}, m\} * r_{tv} \mapsto_{id} \_ * r_{tv} \hookrightarrow_{\delta^s} \delta\}_{rtv}$ 
18     {entry+KERNBASE  $\mapsto_{vpte, qfrac} entryv (pte\_initialized(entryv.pfn)) \urcorner\}_{rtv}$ 
19     { $\forall_{i \in 0..511}. ((table\_root(pte\_initialized(entryv.pfn)))) + i * 8) \hookrightarrow_{id} v-1\}$ 
20     uintptr_t next_phys_addr = PTE_PFN_TO_ADDR(entry->pfn); // Fetch physical addr of next table
21     uintptr_t next_virt_addr = (uintptr_t) P2V(next_phys_addr); // Convert to virtual address
22     next = (pte_t *) next_virt_addr;
23     return next;
24 }

```

Fig. 10. Ensuring entry points to a valid next-level table, allocating if necessary, returning its virtual address.

For this C presentation of what is really an assembly-level proof, we abuse notation and use our register points-to for C-level program variables. So on Line 5, $entry \mapsto_r entry + KERNBASE$ means that the *register representing* the C program variable *entry* (per the calling convention, *rdi*) holds the sum on the right (a constant offset added to the physical address *entryp* of the entry). That particular value is one subtlety of the proof related to the aforementioned kernel invariant, and is explained in Section 5.2.3. The *virtual pte-points-to* from that virtual address (Line 6) indicates that it points to a value *entryv*, a (possibly-unpopulated) page table entry. A virtual pte-points-to is defined just like the normal virtual points-to of Figure 4, except the physical address (*entryp* on Line 6) is explicit in the assertion rather than existentially quantified:

$$va \mapsto_{vpte, q} pa \ v : vProp \ \Sigma \triangleq \exists \delta. (\lambda cr3val. cr3val \hookrightarrow_{\delta^s} \delta) * va \hookrightarrow_q pa * pa \mapsto_p v$$

This supports memory access rules much like Figure 8's rules (which are proven sound using the virtual pte-points-to rules as lemmas!) while exposing the physical location being modified. This is useful for page table modifications, which require knowing the physical location being changed. They are used throughout the software page table walk because entries in any level may be initialized.

5.2.2 Address Space Invariant: Identity Mappings and Conditional Page Table Ownership. Assembly-level verification of compiler output from Figure 10 is verbose, but largely similar to other assembly-level verification thanks to Section 4's logic (including virtual pte-points-to assertions), but only after resolving two critical challenges. Two key challenges stand out and end up affecting both the pre- and post-conditions, neither of which has been addressed by prior work. First, the update to the memory at (virtual) address *entry* depends on subtle ownership invariants: if the entry is present then its fractional ownership is shared with a large number of $L_4_L_1_PointsTo$ assertions from the address space invariant (Figure 6), but if the entry is absent the proof requires full ownership to

$$\begin{aligned}
IASpace_{id}(\theta, \Xi, m) &\triangleq ASpace_Lookup_{id}(\theta, \Xi, m) * \\
&\left(\begin{array}{l} * \exists (l4e, l3e, l2e, l1e, paddr). L4_L1_PointsTo(va, l4e, l3e, l2e, l1e, paddr) \\ (va, paddr) \in \theta \\ * \exists (qfrac, q, val, va). \ulcorner va = pa + KERNBASE \text{ level} > 1 \urcorner * \underbrace{va \hookrightarrow_q^\delta pa}_{\text{① Ghost translation}} * \underbrace{pa \mapsto_p \{qfrac\} val}_{\text{② Physical location}} \end{array} \right) * \\
&\underbrace{\ulcorner qfrac = 1 \leftrightarrow \neg \text{entry_present}(val) \urcorner}_{\text{③ Entry validity}} * \underbrace{\left(\ulcorner \text{present_L}(val, level) \urcorner * \forall_{i \in 0..511}. ((\text{entry_page } val) + i * 8) \hookrightarrow^{id} \text{level} - 1 \right)}_{\text{④ Indexing into next level of tables}}
\end{aligned}$$

where $\text{present_L}(val, level) \triangleq \text{entry_present}(val) \wedge \text{level} > 1$

$$ASpace_Lookup_{id}(\theta, \Xi, m) \triangleq \lambda \text{ cr3val}. \exists \delta, \delta'. \ulcorner m !! \text{cr3val} = \text{Some } \delta \urcorner * \mathcal{A}bsPTableWalk(\delta, \theta) * \mathcal{A}P2VMappings(\delta', \Xi)$$

Fig. 11. Address space invariant of Figure 6 extended with a ghost map bookkeeping identity mappings.

update it. We resolve this by extending the address space invariant to make the owned fraction of the entry's memory *dependent on its own contents*. Second, the conversion of physical addresses into a corresponding virtual address that can be used to modify the specific physical location relies on subtle, never-before-formalized kernel invariants. Since the key to solving these challenges is to extend the address space invariant, we first discuss that invariant and the kernel designs it supports, before returning to the subtle details of verifying lines 12 and 21. The key idea is to establish extra invariants on physical addresses that are part of a page table — but to do so in a way that meshes with the existing invariants (in the informal sense) already preserved in most unverified kernel designs. Each of the above problems requires its own extension to the invariant, but we will discuss physical-to-virtual conversion first, both because it dictates the organization of the invariant and because when Line 9's conditional check is false that is all that is necessary for the proof; correctness of the conditional branch deals with *both* extensions.

5.2.3 Physical-to-Virtual Mappings and P2V. Kernels need to convert between physical and virtual addresses, in both directions. Traversing the page tables in software is the simplest way to convert a virtual address to a physical address; this is the context we are working up to. However, implementing this virtual-to-physical (V2P) translation in software ironically requires physical-to-virtual (P2V) translation, because the addresses stored in page table entries are physical, but memory accesses issued by the OS code use virtual addresses. Because VMM operations are performance-critical for many workloads, most kernels maintain invariants that enable very fast P2V conversions (rather than adding another data structure). Specifically, many kernels maintain an invariant on their page tables that the virtual address of any page used for a page table is *a constant offset from the physical address* — a practice sometimes referred to as *identity mapping* (even though the physical-to-virtual translation is typically not literally the identity function, but adding a nonzero constant offset).¹⁴ Thus P2V on line 21 of Figure 10 is a macro for adding the fixed constant KERNBASE.

Figure 11 extends the per-address-space invariant to also track which addresses we can perform a P2V conversion on by adding a constant offset (i.e., the set of physical addresses which participate in page tables). Ξ is another ghost map, from physical addresses to the level of the page table they represent (1–4). *Only* physical addresses in Ξ can undergo P2V conversion. Section 5.2.3 describes the verification of an actual conversion, but this invariant must be *established* when adding a new page table level (notably on Line 11, hence the comment of Line 13).

For each $pa \mapsto v \in \Xi$, the invariant contains a virtual points-to justifying that virtual address $pa + \text{KERNBASE}$ maps to physical address pa (① in Figure 11); fractional ownership of the physical

¹⁴Some kernels do this for all physical memory on the machine, simplifying interaction with DMA devices. On newer platforms like RISC-V, this sometimes truly is an identity mapping — x86-64 machines are forced into offsets by backward compatibility with bootloaders that cannot access the full memory space of the machine.

memory for that page table entry (②, which together with ① is equivalent to a virtual points-to); and for valid entries (with the present bit set) above L1, ghost map tokens for Ξ for every entry in the table pointed to by the entry, which can be used to repeat the process one level down (④). ④ becomes part of the precondition to `pte_get_next_table`: Line 7 says that if the entry is valid (points to a next-level table) then there are tokens for accessing Ξ for every entry in the next-level table. By Line 20 the entry is guaranteed to be valid so all tokens for converting the next table level's physical addresses to virtual are available (in the form expressed by the assertion on Line 15).

As noted above, for the code path where the conditional does not execute (there was already a valid entry), this is all we need of the new invariant to verify the end of the function from Line 20 onward. By that point the invariant holds and applies to the definitely-valid entry, so we can the physical address of the next-level table to a corresponding virtual address via the identity mappings just described. Line 20 simply retrieves the physical address. Line 21 is the critical piece, and arguably corresponds to the most subtle verification of an `add` instruction (`add rax, KERNBASE`) that we are aware of, and something no prior work on verified OS kernels has dealt with.

After Line 20, it is already known that the present bit is set in the entry; Line 19's assertion reflects that the tokens for Ξ exist for each word-aligned physical address in the next-level table. However, note that no argument to this function specifies which virtual address is being accessed, so `pte_get_next_table` does not know which entry in the next table to retrieve. Even if that address were passed, this function is used for each step-down, so the slice of the virtual address (per Figure 1) is not fixed. Thus Line 21 computes the virtual address of the *base* of the next-level table, and the postcondition includes a renamed version of the assertion on line 19, for the *caller* — `walkpgdir` (discussed next) to perform the conversion for The caller determines which entry in that table must actually be accessed — by selecting the appropriate index into the 512 ghost map tokens returned in the postcondition, and using the ghost translation and physical location portions of the invariant to assemble a `vppte-pointsto` that justifies the caller's subsequent access to a particular entry of the returned table. The postcondition also passes back the per-address-space invariant with the identity mapping resources for entry still pulled out (it was removed by the caller).

5.2.4 Self-Conditional Fractional Ownership and Installing a New Table. The fractional ownership of the entry's physical memory is subtle. As noted above, a *valid* entry must coexist with the fractional ownership from `L4_L1_PointsTo` and therefore have less than full ownership, but in the case where the entry is *invalid*, Line 11 must have full permissions in order to populate the entry (i.e., to install a reference to a next-level table). Fortunately, the entry can only be in use if its valid bit is set; if the valid bit is not set, we know that no virtual points-to entry in δ or θ holds any partial ownership. But determining this requires reading the very memory whose ownership is being determined. We use the invariant portion annotated as "Entry validity" (③) in Figure 11 to capture this: if the entry is invalid the invariant holds full ownership of the entry, so it can be updated; while if the entry is valid, the invariant owns only a constant nonzero fraction sufficient to read but not modify the entry. Since the fractional ownership is always non-zero, Line 9 in Figure 10 can read the entry (using a rule similar to `WRITEToREGFROMVIRTMEM`, tailored to virtual PTE-points-to assertions), and if the entry is dynamically found to be invalid, the invariant is refined (Line 10) to indicate full ownership, allowing updates. Note that the caller is responsible for providing this conditional ownership, having pulled it out of the invariant earlier. This is why the precondition (Line 5) explicitly excludes the entry's physical address from the invariant ($\Xi \setminus \{\text{entry}\}$) — its relevant assertions have already been borrowed by the caller.

If the entry is not set, `pte_initialize` allocates a physical page for use as the next-level table. `pte_initialize` (Figure 12) calls `kalloc` to allocate a physical page (Figure 12 Line 4), and installs it into the entry (Line 5). The page-allocator's `kalloc` is the only unverified (trusted) code in our case

```

1  /* @param entry: virtual address of non-present entry to initialize with a new physical page*/
2  {IASpaceid( $\theta, \Xi, m$ ) *  $\neg(\text{entry\_present entryv}) * \text{entryp} + \text{KERNBASE} \mapsto_{\text{vpte},1} \text{entryp entryv} * (\text{entryp}) \hookrightarrow_{\text{id}} \text{level}$ }
3  void pte_initialize(pte_t *entry) {
4      pte_t *local = kalloc(); // Allocate a full zeroed page for 512 8-byte entries
5      entry->pfn = PTE_ADDR_TO_PFN((uintptr_t) local);
6      {IASpaceid( $\theta, \Xi, m$ ) *  $\neg(\text{entry\_present entryv}) * \text{entryp} + \text{KERNBASE} \mapsto_{\text{vpte},1} \text{entryp pfn\_set}(\text{entryv nextpaddr})$ }
7      {rtv  $\hookrightarrow^{\delta_s} \delta * \forall i \in 0..511. () \text{table\_root}(\text{pte\_initialized}(\text{pfn\_set}(\text{entry\_val nextpaddr}))) + i * 8 \mapsto_{\text{id}} \text{level}-1$ }rtv
8  }

```

Fig. 12. Allocating a physical page

study.¹⁵ Since we are using `pte_initialize` for page-table address allocation, we must relate this newly allocated physical address to the identity mapping map Ξ — see Line 11 in Figure 10, where `kalloc`'s specification guarantees it has returned memory from a designated memory pool that is already mapped¹⁶ and satisfies the offset invariants (trivially, as the new page is zeroed). The presence bit of the entry is *not* set during `pte_initialize`, but upon return to `pte_get_next_table`, where it will validate the conditional ownership discussed above. `pte_initialize` has a full-permission virtual pte-points-to in its precondition. Then the assertions that hold after Line 13 of Figure 10 are enough to establish the same page table invariants which hold in the case where the entry was already valid, by updating the current address space's entry.

5.2.5 The Specification of `pte_get_next_table`. Note that the specification does *not* assume a specific page table level and is used for all three level transitions (4 to 3, 3 to 2, 2 to 1). The logical parameter v represents the level of the entry passed as an argument (c.f. the token $(\text{entryp}) \hookrightarrow_{\text{id}} v$ witnessing that `entryp` is part of the page table invariant on Line 5). This comes into play with a key subtlety of `pte_get_next_table`'s specification: its precondition includes a virtual pte-points-to (discussed earlier, Line 6) but its postcondition does not yield new virtual points-to assertions! It merely computes the base virtual address of the next table, and returns adequate tokens (Line 19) for the *caller* to construct a vpte-points-to for any entry of the next table level.

5.2.6 Walking The Page Tables: Calling `pte_get_next_table` for Each Level. Implementing a software page table walk amounts to calling `pte_get_next_table` for each level as shown in Figure 13. `walkpgdir` traverses the page table anchored at L4 (the virtual address of the base of the L4 table) and returns the virtual address of the L1 entry that should map the virtual address va . For each level, `walkpgdir` locates the appropriate entry by using the level-specific slice of va to index into that table (simulating the hardware translation as in Figure 1), and passes the virtual address of that entry to `pte_get_next_table` to get the base of the next level down. For example, Line 5 uses `L4Offset` (a bit shifting and masking macro) to extract bits 39–47 of va , and uses that to find the address of the L4 entry that would map va in the address space. That is then passed to `pte_get_next_table` on Line 9, which returns the virtual address of the base of the L3 table. This process repeats for 3-to-2 (Lines 14–21), and 2-to-1 (Lines 22–23), after which Line 24 returns the virtual address of the appropriate L1 entry.

¹⁵This is an allocator for regions of pre-zeroed physical memory that is mapped, but not accessed by the allocator itself, as is typical for slab allocators [14]. Its verification would be similar to verifying a usermode malloc verification [19, 73], just with additional invariants on the memory pool.

¹⁶A reasonable reader might wonder where this pool initially comes from and how it might grow when needed. Typically an initial mapping subject to this identity mapping constraint is set up prior to transition to 64-bit kernel code (notably, a page table must exist *before* virtual memory is enabled during boot, as part of enabling it is setting a page table root). Growing this pool later requires cooperation of physical memory range allocation and virtual memory range allocation, typically by starting general virtual address allocation at the highest physical memory address plus the identity mapping offset. This reserves the virtual addresses corresponding to all physical addresses plus the offset for later use in this pool, as needed.

```

1  /* @param l4: the (virtual) address of the L4 page table tree
2     @param va: virtual address to be translated and mapped */
3  pte_t *walkpgdir(pte_t *l4, const void *va) {
4     {P * l4p = rtv⊥ * rtv + KERNBASE = l4⊥ * (l4p + 8 * L4Offset(va)) ⇐id 4 * IASpaceid(θ, Ξ, m) * rtv ⇐δs δ}rtv
5     pte_t *l4_entry = &l4[L4Offset(va)]; // Virtual address of L4 entry
6     {P * IASpaceid(θ, Ξ \ {l4p+8*L4Offset(va)}, m) * rtv ⇐δs δ * r qf4 = 1 ⇐ ¬(entry_present l4e_val)⊥}rtv
7     {entry_present(l4e_val) * ∀i∈0..511 (table_root(l4e_val) + i * 8) ⇐id 3}rtv
8     {l4p+KERNBASE+L4Offset(va) ↦vpte,qf4 (l4p+L4Offset(va)) l4e_val}rtv
9     pte_t *l3 = pte_get_next_table(l4_entry);
10    /*pte_get_next_table may have allocated a new page which updates entry value*/
11    {P * IASpaceid(θ, Ξ \ {l4p+8*L4Offset(va)}, m) * rtv ⇐δs δ * r qf4 = 1 ⇐ ¬(entry_present l4e_val)⊥}rtv
12    {∀i∈0..511 (table_root(l4e_val'.pfn) + i * 8) ⇐id 3 * l3-KERNBASE = table_root(l4e_val'.pfn)⊥}
13    {l4p+KERNBASE+L4Offset(va) ↦vpte,qf4 (l4p+L4Offset(va)) l4e_val}rtv
14    pte_t *l3_entry = &l3[L3Offset(va)]; /* Virtual address of L3 entry*/
15    {l3 + L3Offset(va) * 8 = l3p ∧ table_root(l4e_val'.pfn) = l3⊥}rtv
16    {P * IASpaceid(θ, Ξ \ {l4p+8*L4Offset(va), l3p+8*L3Offset(va)}, m) * rtv ⇐δs δ}rtv
17    {∀i∈0..511 \ {L3Offset(va)} (table_root(l4e_val.pfn) + i * 8) ⇐id 3}rtv
18    {l4p+KERNBASE+L4Offset(va) ↦vpte,qf4 (l4p+L4Offset(va)) l4e_val * r qf4 = 1 ⇐ ¬(entry_present l4e_val)⊥}rtv
19    {l3p+KERNBASE+L3Offset(va) ↦vpte,qf3 (l3p+L3Offset(va)) l3e_val * r qf3 = 1 ⇐ ¬(entry_present l3e_val)⊥}rtv
20    {entry_present(l3e_val) * ∀i∈0..511 (table_root(l3e_val.pfn) + i * 8) ⇐id 2}rtv
21    pte_t *l2 = pte_get_next_table(l3_entry);
22    pte_t *l2_entry = &l2[L2Offset(va)];
23    pte_t *l1 = pte_get_next_table(l2_entry);
24    return &l1[L1Offset(va)]; // Return virtual addr of L1 entry with a similar index computation
25 } {Rwalk * Rl1e}rtv

```

Fig. 13. Walking page-table via calls to pte_get_next_table in Figure 10

$$R_{\text{walk}} = \left(\begin{array}{l}
\lceil l3 + 8 * L3Offset(va) = l3_entry \wedge l2 + 8 * L2Offset(va) = l2_entry \wedge l1 + 8 * L1Offset(va) = l1_entry \\
\wedge \text{table_root}(l4e_val_{pfn}) = l3 \wedge \text{table_root}(l3e_val_{pfn}) = l2 \wedge \text{table_root}(l2e_val_{pfn}) = l1 \rceil * \\
P * IASpace_{id}(\theta, \Xi \setminus \{l4p+8*L4Offset(va), l3p+8*L3Offset(va), l2p+8*L2Offset(va), l1p+8*L1Offset(va)\}, m) * \\
(l4p + 8 * L4Offset(va)) \hookrightarrow^{id} 4 * (l3p + 8 * L3Offset(va)) \hookrightarrow^{id} 3 * \\
(l2p + 8 * L2Offset(va)) \hookrightarrow^{id} 2 * (l1p + 8 * L1Offset(va)) \hookrightarrow^{id} 1 * \\
rtv \hookrightarrow^{\delta s} \delta * (l4p + KERNBASE + 8 * L4Offset(va)) \mapsto_{vpte,qf4} (l4p + 8 * L4Offset(va)) l4e_val * \\
(l3p + KERNBASE + 8 * L3Offset(va)) \mapsto_{vpte,qf3} l3p + 8 * L3Offset(va) l3e_val * \\
(l2p + KERNBASE + 8 * L2Offset(va)) \mapsto_{vpte,qf2} l2p + 8 * L2Offset(va) l2e_val \\
R_{l1e} = (l1p + KERNBASE + 8 * L1Offset(va)) \mapsto_{vpte,qf1} l1p + 8 * L1Offset(va) l1e_val * r qf1 = 1 \leftrightarrow \neg(\text{entry_present } l1e_val)^{\perp}
\end{array} \right)$$

Fig. 14. R_{walk} : Resources obtained from invariant during a software page table walk in Figure 10

In Figures 13 and 14, there are four related concepts for each level. $l4p$ is the physical address of the L4 table base; $l4$ is the corresponding virtual address (using the same name for the value and the program variable name for brevity, since the variable is not reassigned); $l4_entry$ is the virtual address of the L4 entry used to translate va ; and $l4e_val$ is the value of that table entry. Other levels are named consistently. For each of the three level transitions, the main challenges for the proof are to construct a virtual pte-points-to assertion for the entry in that level's table, and pass the conditional assertion discussed in Section 5.2.3 that if that entry is present then there are identity map tokens for the the physical address of each entry in the subsequent level's table. For traversing the L4 table, this proceeds by exchanging the relevant identity map token provided in the precondition (Line 4) and pulling the resources for physical address $l4p + 8 * L4Offset(va)$ out of the identity map invariant: parts ① and ② of Figure 11 give a virtual pte-points-to, and ③ and ④ satisfy other parts of pte_get_next_table's precondition. This justifies the call on Line 9, which returns the virtual address of the base of the appropriate L3 table and whose postcondition includes

```

1  /* @param l4: the virtual address of the root of the page table tree
2     @param va: virtual address to be translated and mapped
3     @param fpaddr: physical address of a zeroed page to map if va is not already mapped */
4  void vaspacemappage(pte_t *l4, void *va, uintptr_t fpaddr) {
5      {P * (l4p+8*L4Offset(va))  $\hookrightarrow$ id 4 * IASpaceid( $\theta, \Xi, m$ ) * rtv  $\hookrightarrow$  $\delta^s$   $\delta$  *  $\neg \theta$  !! va = None $\neg$ }rtv
6      pte_t *pteaddr = walkpgdir(l4, va);
7      {Rwalk * Rl1e}
8      if (!pteaddr->present){
9          /*The entry is not present*/
10         pteaddr->pfn = PTE_ADDR_TO_PFN(fpaddr); /*Store updated entry back to L1 entry */
11         {l1_entry+KERNBASE  $\mapsto$ vpte,1 l1_entry (set_pfn(l1e_val, ADDR_TO_PFN(fpaddr))) *  $\neg$  (entry_present l1e_val) $\neg$ }
12         pteaddr->present = 1; // Set the present bit in entry
13     }
14     { (l1_entry+KERNBASE  $\mapsto$ vpte,qfrac l1_entry l1e_val *  $\neg$  (entry_present l1e_val) $\neg$ )  $\vee$  ( $\neg$  (entry_present l1e_val) $\neg$ ) *
15       { *l1_entry+KERNBASE  $\mapsto$ vpte,1 l1_entry (set_pfn(l1e_val, ADDR_TO_PFN(fpaddr))) } }

```

Fig. 15. Specification of updating L1 entry to reference a new page (fpaddr).

512 identity map tokens for each of those entries (anchored to the returned virtual address minus KERNBASE). That invariant (Line 12) is analogous to the one that justified the 4-3 step (Line 4), and the next two steps proceed the same way.

Even given the slight adaptation of our assembly-level proof for the C-level presentation in Figure 13, the proof outline in the figure omits some repeat intermediate assertions for readability. But by Line 19, it should be clear that the proof accumulates a set of similar assertions for each level. Figure 14 expands the abbreviated postcondition to the full set of facts that are accumulated in this way. R_{walk} together with R_{l1e} nearly entail $L4_L1_PointsTo$ (Figure 3) within the logic, forming the basis of the construction of a new virtual points-to for virtual address va . walkpgdir 's execution observes most evidence of an address translation for va , at least down to a possibly-invalid L1 entry (which walkpgdir 's caller, vaspacemappage , will check). Each virtual pte -points-to in R_{walk} internally contains the physical points-to portion of one page table walk step for $L4_L1_PointsTo$, and the pure assertions in R_{walk} ensure the address arithmetic works. R_{l1e} includes the self-conditional fractional ownership of the L1 entry (Section 5.2.4) for the caller to initialize the entry if it is empty, so the caller can complete a virtual points-to assertions for a newly-mapped page, as we discuss next.

5.3 Mapping a New Page

Finally we come to the top-level routine for mapping a new page of memory into an address space by updating page tables — the vaspacemappage function in Figure 15. Again, our verification was carried out at the assembly level, but presented as on the original C for readability, and the proof outline omits all relevant facts in favor of the most critical assertions involved in the key parts of the proof. vaspacemappage is typically called by a page fault handler, to map a previously-reserved but lazily-allocated page. It is passed the virtual address of the L4 table base (l4), the virtual address to map (va), and the *physical* address of an empty memory page which should be used as backing memory for va and its surrounding page. It begins by calling walkpgdir (Line 6) to return the virtual address of the L1 entry which corresponds to va (allocating intermediate tables as needed). It then checks if the entry is already initialized. If not, fpaddr is installed into the L1 entry, which is then marked valid (setting the present bit), and the page is mapped. Unmapping is the reverse of the logic we discuss here.

For brevity this example is specialized to the case where va is known to not be mapped: the precondition on Line 5 includes θ !! va = None; generalization to returning an error if it is already mapped is straightforward. The precondition on Line 5, directly entails the precondition of the

walkpgdir call.¹⁷ walkpgdir, as just discussed, returns the virtual address of an allocated L1 entry and its postcondition contains almost all of the information needed to construct a virtual points-to for va — except information about the L1 entry being present and pointing to a data page. We already discussed for the upper level page-tables how the entry-present checks are handled, and Line 8 is similar: R_{1e} includes the self-conditional fractional permission for the L1 entry, so as it is not present, by Line 10 it is known that full permission is held to update that entry. Lines 10 and 12 are verified using the pte-points-to memory store rule. In the assembly proof, this is a bitwise-and of the word-aligned fpaddr with 1 (setting the present flag), yielding a single store.

6 Related Work

We will discuss two streams of related work: OS verification, and program logics with modalities.

OS Kernel Verification. There has been relatively little prior work on formal verification of virtual memory. Most OS verification work minimizes reasoning about virtual memory management. The original VERISoft project [1–3, 3, 26, 43, 68] relied on custom hardware which always ran kernel code with virtual memory disabled, removing the circularity that is a key challenge of verifying VMM code for real hardware: at that point page tables become a basic partial map data structure to represent user program address translations, with an idiosyncratic format. Subsequent OS verification work targets real hardware that uses virtual addressing in the kernel, but unsoundly uses hardware models that do not. Thus they *trust* that the particular page table manipulations do not, for example, unmap kernel code (which can crash the machine even if done “temporarily”). This is true for SEL4 [49, 50, 67], whose formal machine model omits address translation, and CERTKOS [17, 40–42], whose refinement proofs rely on CompCert’s usermode-oriented memory model [56, 57] which assumes updates to one memory address are independent of updates to another — which is not true of page table updates. Other work on OS verification either never progressed to VMM verification (VERISoft XT [22–25]), or uses memory-safe languages for process isolation instead of address translation (SINGULARITY [9, 36, 46, 47], VERVE [74], and Tock [58]), ensuring memory safety, but ignoring other functional uses of virtual memory hardware, like swapping [30] or exploiting copy-on-write techniques for dynamic migration of virtual machines [21].

These limitations motivate work like ours on reasoning soundly about virtual memory management. As discussed earlier, Kolanski and Klein [51] are the only other researchers to study VMM verification against a realistic hardware model, where page table updates are performed through virtual memory accesses (later adding C-level type modeling [52]). As noted in Section 4.1, Kolanski and Klein’s virtual points-to is similar to that in Figure 3, with the attendant problems discussed earlier and lifted by our model. Their approach had modal elements, but did not tackle evaluations that would benefit from modality. Our work improves significantly on the technical capabilities of their logic and evaluates on kernel code that is more complete and more challenging than theirs.

Our modal approach makes it possible to specify address space changes cleanly, which their logic cannot do at all. Our use of virtual pte-points-to assertions enables nearly the same proof rules as standard memory accesses, and constructing virtual points-to information within the logic (c.f. the logical entailment between R_{walk} and $L4_L1_PointsTo$) whereas Kolanski and Klein must reason semantically about when the model state supports new virtual points-to assertions.

Kolanski and Klein verify the critical step of updating an already-located L1 entry to map a new page (ARM assembly corresponding roughly to Lines 10 and 12 in our Figure 15), but ignore the essential code preceding that step — which as our walkpgdir and pte_get_next_table verifications demonstrate, side-steps a significant amount of complexity and critical reasoning tasks.

¹⁷The proof of `vaspace_mappage`’s caller would extract this single identity map token for the specific L4 entry from a set of 512 that are part of the kernel invariant, as `walkpgdir`’s proof does for lower levels.

We have verified the entirety of the software page table walk up to mapping a new page, aside from a trusted physical memory allocator resembling `malloc` [19, 73]. As a consequence of tackling this larger verification challenge, our work is the first to formally specify large segments of the self-referential portion of an OS kernel’s virtual memory management invariants (per Sections 5.2.3 and 5.2.4), and to reason about converting from physical addresses to virtual addresses efficiently.

Due to a lack of fractional permissions in their formalism, they incidentally pick up other limitations orthogonal to their foundational focus: by requiring a virtual points-to have *full* ownership of the page table walk memory, they limit their system to having only as many virtual points-to assertions as there are entries in the top-level table (512) because they cannot share access to entries. We inherit fractional permission support from IRIS, and use it extensively (the overly-restrictive Figure 3 is already an improvement in this way). While we cannot claim credit for IRIS’s extensive feature set, the fact that the model of our assertions is based on a classic algebraic tool (pointwise lifting) makes our approach compatible with other logical bases as well, such as PULSECORE [35] or VST [5].

Kolanski and Klein prove that updates to any memory locations that are not part of the page tables do not affect the interpretation of other memory addresses, just like on real hardware. This implies that programs that do not modify memory mappings could be reasoned about without concern for mappings. An analogous result should hold of our model (though we have not proven it). Informally this is visible in the rules for `mov` instructions, which are nearly identical to rules in a VM-ignorant logic [19, 63]. In principle both our approach and Kolanski and Klein’s could be extended to account for pageable points-to assertions by adding disjunctions to an extended points-to definition, which would be the appropriate next step to extend reasoning to usermode programs running with a kernel that may demand-page the program’s memory.

As noted in Section 2.1, we do not formally model or reason about translation lookaside buffers (TLBs). TLB flushes are necessary when a page is *unmapped*, or when switching address spaces. This occurs in few places in uniprocessor kernels (in some, only 3 locations), so full verified kernels including `seL4` [49, 50] and `CERTIKOS` [40, 41] trust TLB management. Neither of the aforementioned systems has a hardware model including a TLB, so neither is able to verify TLB management in any form — they *must* trust its operation. This is true of multicore verified kernels as well [41, 71], though there the situation becomes much more challenging: when unmapping pages, the running CPU must invalidate the relevant TLB entries locally, but also send an inter-processor interrupt (IPI) to all other cores to ensure they also invalidate affected entries on their TLBs. No formal hardware model currently exists for IPIs on any architecture, or even for the memory-mapped IO used to trigger those IPIs.

The only work to tackle TLB code verification was Syeda and Klein [69, 70], who are also the only prior work on verifying address-space-aware context switching. However, they verified only the 4 instructions to switch address spaces and update the TLB on an ARM processor, in isolation (i.e., not the full context switch including changing stacks with address spaces). The specification they proved for those instructions did not address program invariants that may be valid in one address space and not the other, so is not flexible enough to extend directly to a full context switching primitive as in Figure 9. Their logic adds an assertion tracking known-inconsistent addresses (i.e., recently-unmapped by a table update or change in page table root) and their memory access rules require accessed memory to lie outside that set. However in their logic, reasoning about updates to the inconsistent set (from a page table update) requires direct reasoning directly about the physical memory heap, which limits modularity. The right general solution would be to combine our work and an extension of theirs, a substantial project which we leave to future work. No other prior work has considered address space changes during context switching. `XCAP` [63] and `Bedrock` [18–20] deal only with usermode threading (in a single process). `CERTIKOS` and `seL4` trust assembly primitives for context switches, and do not model address translation for executing code.

Program Logics with Modalities. Modalities have long been a staple of program logics, at least since Dijkstra’s weakest precondition calculus [32] and Pratt’s observation the Hoare triples could be decomposed using the weakest-precondition modality of dynamic logic [65], in a form quite similar to what Iris uses today [48]. Variants of Nakano’s later modality [61] have long been used to deal with step-indexing for impredicative and recursive features of logics and type systems [6, 10, 11, 45, 48].

As noted earlier, our other-space modality derives from hybrid logic [7, 12, 37, 38], where modalities are indexed by *nominals* which are names for specific individual states in a Kripke model. Readers mostly familiar with modalities in prominent program verification approaches [6, 10, 11, 45, 48, 61, 64, 65] may not recognize hybrid logics, but as we discuss in Section 6, they (like temporal logics) trace their roots back to Arthur Prior in the 1950s. Little prior work combines these ideas with program logics. Brotherston and Villard [15] show that traditional nominals extends the expressive power of separation logic. Gordon [39] uses nominals to refer to states of other actors in an actor language. In parallel with our work, Wagner et al. [72] use a hybrid-logic-inspired modality to abstract reference-counting specifics from specifications of a low-level application binary interface (ABI) — their $@_l(P)$ indicates that l is the location of a reference count for resources satisfying P . Beyond these, there is limited work on the interaction of hybrid logic with general substructural logics, in restricted forms that do not affect expressivity [16, 31].

Some logics for weak memory models [27, 28] have been formalized in Iris using pointwise lifting, parameterizing by thread-local views of events (an operationalization of the release-acquire + nonatomic portion of the repaired C11 memory model [55]). There modalities $\Delta_\pi(P)$ and $\nabla_\pi(P)$ represent that P held before or will hold after certain memory fences by thread π . The definitions of those specific modalities existentially quantify over other views, related to the “current” view (the one where the current thread’s assertions are evaluated), and evaluate P with respect to those other views. This approach to parameterizing assertion semantics by a point of evaluation, and evaluating modalized assertions at other points quantified in the definition of a modality, is the classic notion of modal assertions, whereas hybrid logics expose the choice of evaluation point in assertions, allowing statements of more properties. In these weak memory examples this additional expressive power would not be useful, because any relevant points of evaluation (thread views) are intimately tied to memory fences performed by the program, whereas for virtual memory management the kernel must be able to choose or construct arbitrary other address spaces.

7 Conclusions

This paper advances the state of the art in formal verification of programs manipulating virtual memory mappings. We treat assertions about virtual memory locations explicitly as assertions in a modal logic, where the notion of context is a particular address space, named by the page table root. We improved the modularity of our reasoning about virtual address translation and virtual points-to assertions to permit page table modifications that preserve mappings without collecting all affected virtual points-to assertions. To specify of code involving other address spaces, we adapt hybrid logic’s notion of modalities explicitly naming alternative conditions. We implemented these ideas in a derived separation logic within Iris, and proved soundness of the rules for essential memory- and address-space-change-related x86-64 instructions against a hardware model of 64-bit 4-level address translation. Finally, we used our rules to verify the correctness of key VMM instruction sequences, including the first assembly-level proof of correctness for a change of address space expressing which assertions hold in which address space, the first physical-to-virtual translation proof, and the first verification of a software page table walk, all of which are beyond reach of prior work.

Acknowledgments

This work was supported in part by US NSF Award #CCF-1844964.

References

- [1] Eyad Alkassar, Mark A Hillebrand, Dirk Leinenbach, Norbert W Schirmer, and Artem Starostin. 2008. The Verisoft approach to systems verification. In *Verified Software: Theories, Tools, Experiments*. Springer, 209–224.
- [2] Eyad Alkassar, Wolfgang J Paul, Artem Starostin, and Alexandra Tsyban. 2010. Pervasive verification of an OS microkernel. In *Verified Software: Theories, Tools, Experiments*. Springer, 71–85.
- [3] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. 2008. Formal pervasive verification of a paging mechanism. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 109–123.
- [4] AMD. 2023. AMD64 Architecture Programmer's Manual, Volume 2: System Programming. Revision 3.40.
- [5] Andrew W Appel. 2014. *Program logics for certified compilers*. Cambridge University Press.
- [6] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) (POPL '07). ACM, New York, NY, USA, 109–122. doi:10.1145/1190216.1190235
- [7] Carlos Areces, Patrick Blackburn, and Maarten Marx. 2001. Hybrid logics: Characterization, interpolation and complexity. *The Journal of Symbolic Logic* 66, 3 (2001), 977–1010.
- [8] Carlos Areces and Balder ten Cate. 2006. Hybrid Logics. In *Handbook of Modal Logic*. Elsevier. <https://carlosareces.github.io/content/papers/files/hml-arecestencate.pdf>
- [9] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (June 2011), 81–91. doi:10.1145/1953122.1953145
- [10] Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional type theory with guarded recursive types qua fixed points on universes. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*. IEEE, 213–222. doi:10.1109/LICS.2013.27
- [11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. *ACM sigplan notices* 46, 1 (2011), 119–132.
- [12] Patrick Blackburn and Jerry Seligman. 1995. Hybrid languages. *Journal of Logic, Language and Information* 4, 3 (1995), 251–272.
- [13] Patrick Blackburn and Johan van Benthem. 2006. Modal Logic: A Semantic Perspective. In *Handbook of Modal Logic*. Elsevier. <https://carlosareces.github.io/ml18/downloads/hb.pdf>
- [14] Jeff Bonwick et al. 1994. The slab allocator: An object-caching kernel memory allocator.. In *USENIX summer*, Vol. 16. Boston, MA, USA.
- [15] James Brotherston and Jules Villard. 2014. Parametric completeness for separation theories. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 453–464.
- [16] Kaustuv Chaudhuri, Joëlle Despeyroux, Carlos Olarte, and Elaine Pimentel. 2019. Hybrid linear logic, revisited. *Mathematical Structures in Computer Science* 29, 8 (2019), 1151–1176.
- [17] Hao Chen, Xiongnan (Newman) 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* (Santa Barbara, CA, USA) (PLDI '16). ACM, New York, NY, USA, 431–447. doi:10.1145/2908080.2908101
- [18] Adam Chlipala. 2011. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) (PLDI '11). ACM, New York, NY, USA, 234–245. doi:10.1145/1993498.1993526
- [19] Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) (ICFP '13). ACM, New York, NY, USA, 391–402. doi:10.1145/2500365.2500592
- [20] Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) (POPL '15). ACM, New York, NY, USA, 609–622. doi:10.1145/2676726.2677003
- [21] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.
- [22] Ernie Coehn, Michal Moskal, Wolfram Shulte, and Stephan Tobies. 2010. *Local Verification of Global Invariants in Concurrent Programs*. Technical Report MSR-TR-2010-9. Microsoft Research.
- [23] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*. Springer, 23–42.

- [24] Ernie Cohen, Wolfgang J. Paul, and Sabine Schmaltz. 2013. Theory of Multi Core Hypervisor Verification. In *SOFSEM 2013: Theory and Practice of Computer Science, 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26–31, 2013. Proceedings*. 1–27. doi:10.1007/978-3-642-35843-2_1
- [25] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. 2009. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering-Companion Volume, 2009. ICSE-Companion 2009*. IEEE, 429–430.
- [26] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. 2005. On the verification of memory management mechanisms. In *Correct Hardware Design and Verification Methods*. Springer, 301–316.
- [27] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.
- [28] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Thuan Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 792–808.
- [29] Paulo Emílio de Vilhena and François Pottier. 2023. A type system for effect handlers and dynamic labels. In *European Symposium on Programming*. Springer Nature Switzerland Cham, 225–252.
- [30] Peter J. Denning. 1970. Virtual Memory. *ACM Comput. Surv.* 2, 3 (Sept. 1970), 153–189. doi:10.1145/356571.356573
- [31] Joëlle Despeyroux and Kaustuv Chaudhuri. 2014. A hybrid linear logic for constrained transition systems. In *Post-Proceedings of the 9th Intl. Conference on Types for Proofs and Programs (TYPES 2013)*, Vol. 26. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 150–168.
- [32] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.
- [33] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Rome, Italy) (POPL '13)*. ACM, New York, NY, USA, 287–300. doi:10.1145/2429069.2429104
- [34] Thomas W. Doepfner. 2010. *Operating Systems in Depth: Design and Programming*. Wiley.
- [35] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1516–1539.
- [36] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. 2006. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (Leuven, Belgium) (EuroSys '06)*. ACM, New York, NY, USA, 177–190. doi:10.1145/1217935.1217953
- [37] George Gargov and Valentin Goranko. 1993. Modal logic with names. *Journal of Philosophical Logic* 22, 6 (1993), 607–636.
- [38] Valentin Goranko. 1996. Hierarchies of modal and temporal logics with reference pointers. *Journal of Logic, Language and Information* 5, 1 (1996), 1–24.
- [39] Colin S Gordon. 2019. Modal assertions for actor correctness. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 11–20.
- [40] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Mumbai, India) (POPL '15)*. ACM, New York, NY, USA, 595–608. doi:10.1145/2676726.2676975
- [41] 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 *OSDI*. 653–669.
- [42] 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 (Philadelphia, PA, USA) (PLDI 2018)*. ACM, New York, NY, USA, 646–661. doi:10.1145/3192366.3192381
- [43] Mark Hillebrand. 2005. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. Ph.D. Dissertation. PhD thesis, Saarland University, Computer Science Dept.
- [44] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.
- [45] Aquinas Hobor, Robert Dockins, and Andrew W Appel. 2010. A theory of indirection via approximation. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 171–184.
- [46] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. 2007. Sealing OS Processes to Improve Dependability and Safety. In

- Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) (*EuroSys '07*). ACM, New York, NY, USA, 341–354. doi:10.1145/1272996.1273032
- [47] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. doi:10.1145/1243418.1243424
- [48] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.
- [49] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages. doi:10.1145/2560537
- [50] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) (*SOSP '09*). ACM, New York, NY, USA, 207–220. doi:10.1145/1629575.1629596
- [51] Rafal Kolanski and Gerwin Klein. 2008. Mapped Separation Logic. In *Verified Software: Theories, Tools and Experiments*, Natarajan Shankar, Jim Woodcock (Ed.). Springer, Toronto, Canada, 15–29.
- [52] Rafal Kolanski and Gerwin Klein. 2009. Types, Maps and Separation Logic. In *International Conference on Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Ed.). Springer, Munich, Germany, 276–292.
- [53] Ismail Kuru and Colin S Gordon. 2025. Modal Abstractions for Virtualizing Memory Addresses (Artifact). doi:10.5281/zenodo.16896150
- [54] Ismail Kuru and Colin S Gordon. 2025. *Modal Abstractions for Virtualizing Memory Addresses (Technical Report)*. Technical Report arXiv:2307.14471. arXiv.
- [55] 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*. 618–632.
- [56] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.
- [57] Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41, 1 (2008), 1–31.
- [58] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 234–251.
- [59] Richard McDougall and Jim Mauro. 2006. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education.
- [60] Tom Murphy VII, Karl Crary, and Robert Harper. 2008. Type-safe distributed programming with ML5. In *Trustworthy Global Computing: Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers 3*. Springer, 108–123.
- [61] Hiroshi Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*. IEEE, 255–266.
- [62] Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) (*POPL '06*). ACM, New York, NY, USA, 320–333. doi:10.1145/1111037.1111066
- [63] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. 2007. Using XCAP to certify realistic systems code: Machine context management. In *Theorem Proving in Higher Order Logics*. Springer, 189–206.
- [64] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*. IEEE, 46–57.
- [65] Vaughan R Pratt. 1976. Semantical consideration on Floyd-Hoare logic. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*. IEEE, 109–121.
- [66] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*. IEEE, 55–74.
- [67] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (*PLDI '13*). ACM, New York, NY, USA, 471–482. doi:10.1145/2491956.2462183
- [68] Artem Starostin. 2010. *Formal verification of demand paging*. Ph. D. Dissertation. PhD thesis, Saarland University, Computer Science Dept.
- [69] Hira Taqdees Syeda and Gerwin Klein. 2018. 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 9*. Springer, 542–559.
- [70] Hira Taqdees Syeda and Gerwin Klein. 2020. Formal reasoning under cached address translation. *Journal of Automated Reasoning* 64, 5 (2020), 911–945.
- [71] Michael von Tessin. 2013. *The clustered multikernel: An approach to formal verification of multiprocessor operating-system kernels*. Ph. D. Dissertation. PhD thesis, School of Computer Science and Engineering, UNSW, Sydney, Australia, Sydney, Australia.
- [72] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1249–1278.
- [73] John Wickerson, Mike Dodds, and Matthew Parkinson. 2010. Explicit stabilisation for modular rely-guarantee reasoning. In *European Symposium on Programming*. Springer, 610–629.
- [74] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-safe Operating System. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) (*PLDI '10*). ACM, New York, NY, USA, 99–110. doi:10.1145/1806596.1806610

Received 2025-03-25; accepted 2025-08-12