A Method for the Verification of Memory Management Software in the Presence of TLBs

Yahya Sohail
The University of Texas at Austin
Austin, TX, United States of America
yahya@yahyasohail.com

Warren A. Hunt, Jr.
The University of Texas at Austin
Austin, TX, United States of America
hunt@cs.utexas.edu

***Abstract*—The correct management of translation lookaside buffers (TLBs) by memory-management software is critical to the security of modern computer systems. We develop a methodology for reasoning about software that modifies address translations executing on machines with a TLB, enabling the verification of the security-critical, memory-management code in operating system kernels at the binary level. We automate proving that the TLB does not contain entries that are inconsistent with the page tables which allows us to reduce the problem of reasoning about address translation in the presence of a TLB to the problem of reasoning about address translation by walking page tables. Our technique is independent of any particular ISA or TLB microarchitecture. To demonstrate the effectiveness of our approach, we add a TLB to the ACL2 model of the x86 ISA, implement this reasoning technique in the ACL2 theorem prover, and verify a page-table-altering program called Zero-Copy that copies a page of data in the virtual-address space by modifying the page tables. We were able to reuse and update substantial portions of a previous version of the proof of correctness that was performed with a version of the x86 ISA model lacking a TLB, confirming the effectiveness of our reduction-based method.**

***Index Terms*—Software Verification, Caching, Address Translation, x86, ACL2.**

I. Introduction

Contemporary microprocessors provide address-translation mechanisms to enable operating system kernels to implement process isolation and other desirable security properties. Without address translation, any running process could modify the memory of any other process or even the kernel. This would make it impossible for operating systems to implement process isolation or differentiate in any way between the permissions of different processes running on the system. The most commonly employed address-translation mechanism is paging, and paging-enabled microprocessors generally implement one or more address-translation caching mechanisms since paging would be prohibitively slow without them.

Since memory-management software is crucial to computer security, its verification has drawn the interest of the formal-methods community, and paging semantics have been specified formally for several ISAs. However, the behavior of memory-management software running on machines with address translation in the presence of address-translation caches is less well studied using formal methods. These caches have architecturally visible behavior, and memory-management software must correctly manage the contents of such caches to implement the security properties expected of modern operating systems. Thus, it is not sufficient to verify memory-management software using an ISA model lacking models of address-translation caches to ensure that the software will execute correctly on a machine with address-translation caches.

It is challenging to verify memory management software in the presence of a translation lookaside buffer (TLB), a common type of address-translation cache, because ISAs allow much flexibility in their implementation. On most ISAs, a TLB may cache any address translation that has ever been valid unless its corresponding TLB entry has been explicitly invalidated by software since it was last valid in the page tables. This is difficult to reason about because determining whether there may exist a TLB entry for a given translation may require considering the state of the page tables far before the current point in program execution. Furthermore, the page tables themselves form a tree structure in memory which can be very large, making the consideration of many previous states of the page tables even more difficult.

Our primary contribution is an ISA and microarchitecture-independent methodology for reducing the problem of reasoning about software in the presence of a TLB to the better studied problem of reasoning about software on a machine with paging but no TLB. This is not possible in general, due to TLBs having architecturally-visible behavior, but it is possible when the software correctly invalidates stale entries that may be present in the TLB after updating page-table entries. Fortunately, considering only this case is sufficient for verifying memory-management software because invalidating potentially stale TLB entries before they can be used to perform a translation is a property expected to be maintained by correct memory-management software. Our technique is largely automated and allows the user to easily reuse large parts of proofs about memory-management software that were performed assuming there is no TLB.

We implement our technique using the ACL2 theorem prover [1] on the ACL2, x86-ISA model [2] that we modified by adding a TLB. To our knowledge, this model is the most complete formal model of the x86 ISA; it is sufficiently complete to boot Linux and to run multi-programmed Linux programs. To demonstrate the efficacy of our reasoning strategy, we have used our implementation of the technique on

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_23

the x86-ISA model to verify a memory-management program, called Zero-Copy. The Zero-Copy program copies a 1 GB page in the virtual-address space by modifying the page tables to have both the source and destination memory regions translate to the same physical address; this avoids the need to actually copy the data in memory. We were able to reuse much of a correctness proof [2] written for a previous version of the ACL2, x86-ISA model that lacked a TLB.

Our implementation of the technique—including the ISA model, the proofs described here, and documentation—are publicly available as part of the ACL2 Community Books [3] under the 3-clause BSD license. The documentation [4] is available online in XDOC [5], ACL2's documentation system, and the source, including theorems and proofs, is distributed with ACL2 [1], in the `books/projects/x86isa` subdirectory of the ACL2 source tree available on Github at https://github.com/acl2/acl2.

We begin in Section II by reviewing related work. In Section III, we summarize relevant background information about address translation, address-translation caches, and their architecturally visible behavior. In Section IV, we describe our technique to reduce reasoning about software on a machine with a TLB to address translation on a machine without a TLB. In Section V, we describe how we added a TLB to our ACL2 x86-ISA model, and we implemented our reasoning technique in the ACL2 theorem prover. Then, in Section VI, we apply this work to verify the Zero-Copy binary program. Finally, in Section VII, we conclude and describe how this work may be employed in the future to verify operating-system, memory-management code.

II. Related Work

Modern formal ISA models include many details and subtle features, like segmentation, paging, TLB definitions, privilege levels, device configuration, and I/O. With such features, the formal-methods community is able to analyze supervisor-level, binary code that performs management of machine resources. This has resulted in ambitious projects to build formally verified operating systems, like seL4 [6], [7]. However, reasoning about the effects of caching on address translation has continued to prove difficult, so the correct management of the TLB is generally assumed when verifying software. For example, this is the approach employed by the seL4 verification effort.

In 2015, Goel et al. developed the ACL2-based, x86-ISA model [2] which we extend and employ to demonstrate our reasoning technique. This ACL2-based, x86-ISA model included a full specification of segmentation and paging in 64-bit mode, but it lacked interrupts and I/O peripherals. Goel used this model to verify the Zero-Copy program that copies data in the virtual address space by modifying address translations in the page tables, demonstrating that it could be used to reason about memory management software. However, this model did not capture the semantics of TLBs or any other address translation caches.

Syeda and Klein describe [8], [9] reasoning about a TLB model that they added to the Fox and Myreen Isabelle/HOL model [10] of the user-level, 32-bit Arm v7 ISA [11]. This extension concerned only the memory system, and their model did not sufficiently specify all of the ISA and memory semantics necessary to boot an operating system. They describe how they translate their model into more abstract memory models, such as a model where the TLB contains all of the address translation information for a particular address-space identifier (ASID). This approach was shown to be logically consistent with their lowest-level memory model.

Syeda and Klein proposed various abstractions of the TLB, and their work argues formally about cache models (and specifically, about the TLB). Their memory-management model captures the essential concepts required to support the ARM v7 [11] virtual-memory system. Their suite of memory models are elegant, and those models expose the essential issue of reasoning about programs that interact with the memory-management system. However, the ISA model lacks the detail required to support the booting of Linux or other operating systems on their ARM v7 model, making it difficult to gain confidence in the model's accuracy. Furthermore, their work was concerned with only the ARM v7 ISA, and they did not generalize their approach to reasoning about address translation to other ISAs.

In contrast, the work presented in this paper is an ISA and TLB microarchitecture independent method for reasoning about address translation with TLB caches and demonstrated on an x86-ISA model complete enough to boot Linux and run Linux programs. We have validated this model by booting Linux, running user programs under Linux, running programmatically generated tests, and cosimulation. By demonstrating our reasoning strategy on thsi model, we show that our technique can scale to the challenges of reasoning with large ISA models that capture the semantics and complexities of modern ISAs. Our reduction-based approach also simplifies updating proofs that were done with ISA models that lacked a TLB to accomodate TLB behavior, which we demonstrate by reusing substantial parts of Goel's Zero-Copy correctness proof, providing a path for updating existing software correctness proofs to account for TLB behavior.

III. Background

In a microprocessor that performs address translation, programs execute load and store instructions on memory locations specified by virtual addresses that are mapped, using a function specified by supervisor-level software, into physical addresses by a memory-management system. When address translation is successful, virtual addresses are translated into physical addresses which are used to access memory; otherwise, memory access is denied, and an exception is signaled. Most modern processors that support address translation use *paging* as the mechanism to specify the translation function. Paging allows supervisor software to specify the translation function for each *page*, a naturally aligned block of memory with a size that is

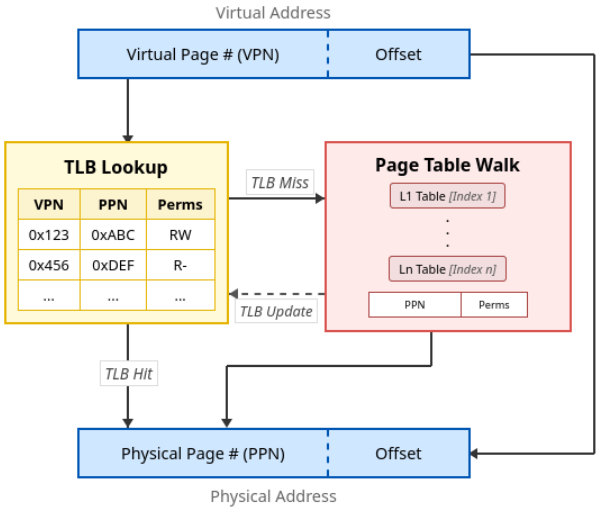

Fig. 1. The virtual address (top) is broken into two parts, the virtual-page number and the offset. The virtual-page number is used to index into the TLB (left). If there is a TLB hit, the VPN is found in the TLB and the corresponding physical page number is returned. Otherwise, a TLB miss occurs and the processor performs a page-table walk (right). The returned PPN and permissions from the page-table walk are used to update the TLB. The offset is concatenated onto the PPN to construct the physical address (bottom).

a power of 2, using memory-resident tables, known as *page tables*, that form a tree structure.

When a microprocessor needs to refer to the page tables to translate an address, it traverses the tree in a process known as a *page-table walk*. This is done by using the high-order bits of the virtual address, called the virtual-page number, to pick a subtree at each level of the page tables until the processor arrives at a leaf-page table entry. As it traverses the page tables, the processor collects information about the permissions of the page that are used to determine whether the processor should raise an exception, and the leaf-page table entry contains the physical page number corresponding to the virtual page number. If the access is allowed and the processor does not raise an exception, the processor translates the virtual address by concatenating the physical-page number to the page offset, the low-order bits of the virtual address that are not part of the virtual page number.

Without caching, paging is prohibitively slow because each translation requires multiple memory accesses. Consequently, microprocessors employ a number of caches to minimize the number of memory accesses necessary for translating addresses. Of these caches, the most common is the translation-lookaside buffer (TLB) that maintains an associative mapping from virtual-page numbers to their corresponding permissions information and physical page number as shown in Figure 1. Processors do not automatically invalidate TLB entries when corresponding page-table entries are updated since doing so would be prohibitively expensive. Thus, it is possible that a TLB contains *stale* entries, entries that do not agree with the page tables. To address this issue, ISAs specify how

```
CLC                                (!rip (+ 2 (rip x86))
STC                                      (!flgi :cf 1 x86))
```

Fig. 2. On the left is an x86 assembly program that first clears and then sets the carry flag. On the right is the result of symbolically executing the program on the ACL2 x86 model using the ACL2 rewriter under certain preconditions. It has been lifted into an expression in the ACL2 logic that is a composition of two operations: the first, `!flgi`, that updates the flag register—in this case, setting the carry flag to 1, and the second, `!rip`, that updates the instruction pointer—in this case by adding two to the current instruction pointer.

software may invalidate TLB entries; for example, in the x86 architecture, executing the instruction `INVLPG` invalidates any TLB entries associated with a specified virtual address and changing the value of the `PG` bit in the `cr0` register from 1 to 0 invalidates all TLB entries [12]. It is the responsibility of supervisor-system software to employ invalidation operations to invalidate stale TLB entries after updating the in-memory page tables.

While ISAs require that certain entries are invalidated by certain operations, they generally allow implementations to invalidate additional entries whenever they choose. They also allow implementations to perform speculative page table walks and choose whether or not to update address-translation caches after a page-table walk, speculative or otherwise, as they see fit. Thus, any translation which has been valid in the page tables and has not had a corresponding TLB entry explicitly invalidated since it was last valid in the page tables may or may not have a corresponding TLB entry. This makes it difficult to reason about what entries may be present in the TLB at a given point in time.

IV. Our Technique

The primary difficulty when reasoning about address translation with caching concerns the incomplete nature of ISA-level specifications. ISAs often describe weak guarantees about TLB contents, so the behavior of address translation in regions with stale TLB entries is not specified fully. Therefore, software should not rely on the results of address translation in such regions. Consequently, we reason about an address translation first by proving that there are no stale TLB entries corresponding to a desired translation or equivalently that the address translation is *TLB consistent* with respect to the machine state. In this case, the result of the translation is the same regardless of whether a TLB entry or the page tables are used to perform the translation. Thereafter, we can reason about the result of the translation by consulting the page tables.

To help prove that a *translation*—that is a tuple of the virtual address, access type (i.e., read, write, or execute), and the values of various processor control bits that affect address translation—is TLB consistent, we have developed a theory of how address translation is affected by different operations on the machine state. Our model of a machine is a function that takes an initial machine state and a number of instructions to execute as input and returns the machine state after executing the given number of instructions. We can execute this function

symbolically via rewriting under the assumption that the initial machine state contains a given program's machine code in the machine's memory at the address given by the instruction pointer. The result of symbolic execution is an expression in the logic that captures the effect of the program on the machine state as shown in Figure 2.

For the following discussion, we define an *operation* to be a function that may appear in the expression that results from symbolically executing a program using a machine model. In Figure 2, the functions `!rip` and `!flgi` are operations. While `!rip` and `!flgi` each only update a single field of the machine state, an operation can perform an arbitrarily complex update to the state of the machine. Any function that returns a new machine state given some parameters and the previous machine state may be an operation. There are two sets of translations associated with every operation that are relevant to address-translation caching behavior:

- *Maintained Translations* — These are translations that are TLB consistent after the operation if they were consistent before the operation. These correspond to translations whose corresponding page-table entries and TLB entries were not updated by the operation.
- *Consistent Translations* — These are translations that are guaranteed to be TLB consistent after the operation is performed, regardless of whether they were TLB consistent before the operation. These translations correspond to translations that have had any corresponding TLB entries invalidated by the operation.

For example, a "load 0 into a general-purpose register" operation's set of maintained translations is all translations and its set of consistent translations is the null set because loading a general-purpose register does not affect the TLB or page tables. On the other hand, an "invalidate all TLB entries" operation has all translations as its set of consistent translations since invalidating all TLB entries will necessarily invalidate all stale TLB entries.

For each operation, we prove theorems that show which translations, possibly in terms of the parameters to the operation and the previous state, are maintained and consistent. We can then use these theorems to show that a given translation is consistent after a program is executed. Showing that a given translation is in the set of consistent translations of the outer-most operation in an expression for the state of the machine proves the translation is TLB consistent in the state given by that expression. Showing that a translation is in the set of maintained translations of the outer-most operation in an expression allows us to "peel away" the outer-most operation and reduce the proof to showing that the translation is consistent in the state prior to the application of the outer-most operation. We can then recursively attempt to prove the translation is consistent in the prior state.

Without loss of generality, in the following discussion, we assume operations have a single parameter because if an operation has multiple parameters, one can define an equivalent operation with a single parameter that is a tuple of the original operation's parameters. Then, all programs on all machines lift into a sequence of operations that looks like

$$\begin{aligned} q_1 &\leftarrow \mathcal{O}_{i_1}(p_1, q_0) \\ q_2 &\leftarrow \mathcal{O}_{i_2}(p_2, q_1) \\ &\vdots \\ q_n &\leftarrow \mathcal{O}_{i_n}(p_n, q_{n-1}) \end{aligned}$$

where $\mathcal{O}_i$ are the operations of the machine, p_i are parameters to the operations (which may be dependent on the initial machine state), q_0 is the initial state, and q_n is the final state. Let $M_i(p, q)$ and $C_i(p, q)$ be the sets of translations that are maintained and consistent respectively by operation $\mathcal{O}_i(p, q)$. Then we can show translation t is TLB consistent with respect to q_k under hypotheses h using Algorithm 1, PROVE-CONSISTENT(t, k, h), which returns either true indicating that t is consistent with respect to q_k, or fails (in which case it is inconclusive). The PROVE-CONSISTENT algorithm assumes we have some procedure PROVE, for example the ACL2 prover, that can be used to attempt to prove a conjecture under given hypotheses. This procedure either succeeds in proving the conjecture and returns true, indicating the conjecture is a theorem, or it fails, returning false, in which case the conjecture may or may not hold true. In practice, we expect the base case that attempts to prove t is consistent in q_0 succeeds due to a hypothesis in h that states t is TLB consistent in q_0.

Algorithm 1 PROVE-CONSISTENT(t, k, h)

```
if k = 0 then
    if PROVE(t is consistent in q_0 under hypotheses h) then
        return true
    else
        return fail
    end if
end if
if PROVE(t ∈ C_{i_k}(p_k, q_{k-1}) under hypotheses h) then
    return true
else if PROVE(t ∈ M_{i_k}(p_k, q_{k-1}) under hypotheses h) then
    return PROVE-CONSISTENT(t, k − 1, h)
else
    return fail
end if
```

We demonstrate the correctness of PROVE-CONSISTENT(t, k, h), that is PROVE-CONSISTENT(t, k, h) returning `true` implies t is consistent in q_k, by a simple inductive argument that inducts over k. The base case is when $k = 0$. If we can prove t is consistent in q_0, it must be consistent in $q_k = q_0$, and we return true; otherwise, we fail, which is inconclusive. For $k > 0$, if we can prove $t \in C_{i_k}(p_k, q_{k-1})$, we return true, and t must be consistent in q_k because it is in the set of consistent translations of $\mathcal{O}_{i_k}(p_k, q_{k-1})$. If we can prove $t \in M_{i_k}(p_k, q_{k-1})$, we return PROVE-CONSISTENT(p_k, q_{k-1}, h), and this is correct because if the recursive call returns `true`, t must

be consistent since it is in the set of maintained translations of $\mathcal{O}_{i_k}(p_k, q_k)$ and, applying the inductive hypothesis, PROVE-CONSISTENT(p_k, q_{k-1}, h) returning true implies t is consistent in q_{k-1}. Finally, if we cannot prove either case, we fail, which is inconclusive.

Notice our correctness proof above only requires that PROVE-CONSISTENT(t, k) does not erroneously return true. An algorithm that always fails satisfies this correctness criteria, but is obviously not useful. We show that our algorithm is useful by implementing it, as described in Section V, and using it to verify a memory-management program described in Section VI.

V. IMPLEMENTATION FOR X86 IN ACL2

We added a TLB and implemented our reasoning strategy for the ACL2 model of the x86 ISA [2]. We chose ACL2 [1] and the aforementioned x86 model for several reasons: The x86-ISA model supported and had a mature theory for reasoning about address translation by walking page tables. ACL2's capacity for handling large proof terms and the high degree of automation facilitated by ACL2's primary proof procedure, its rewriter, makes it well suited for working with the large terms that arise from the symbolic execution of programs. The efficient executability of the ACL2 logic makes it possible to validate and debug the ISA model through testing.

Our ACL2-based model of the x86 ISA is written as an interpreter of x86 machine code. At its core, is a function that, given a machine state, decodes the instruction pointed to by the current instruction pointer, and then executes it by returning an updated machine state as specified by the x86-ISA documentation [12]. This function can be invoked repeatedly, producing an interpreter that executes a sequence of instructions. Since the interpreter is a function in the ACL2 logic, we can use the ACL2 theorem prover to prove theorems about its behavior. It is also a program written in the ACL2 programming language, so in addition to being a formal specification, the interpreter can be executed. It has support for about 500 x86 instructions, supports exceptions, interrupts, and address translation and is complete enough to boot Linux and execute the GNU C compiler as a Linux process. The executability of the model makes it possible to run tests to validate the accuracy of our ACL2 x86-ISA model and use cosimulation, executing our interpreter instruction by instruction in lock step with a trusted x86 implementation, to find bugs in our model.

The ACL2 theorem prover's primary proof procedure is its rewriter that applies conditional rewrite rules to simplify terms. This rewriter can be programmed by an ACL2 user by proving theorems which are stored in the rewriter's database as rewrite rules. When the rewriter finds a term that matches the left-hand side of a rewrite rule, it recursively attempts to rewrite the hypotheses of the rule to true; if it succeeds, the matched term is rewritten as specified by the rewrite rule. This allows a skilled ACL2 user to develop rewrite rules that can automatically be applied to simplify large proof terms without user intervention, greatly automating the proof process. This, along with the rewriter's ability to manipulate large proof terms efficiently, has made ACL2 a good choice for proofs involving large terms, like those that arise when executing programs symbolically. This is one reason why ACL2 has found use in industry for the verification of hardware designs at corporations like Centaur Technology [13] and Intel.

Goel developed a library of theorems to aid with the verification of x86 software using the x86-ISA model. These included many lemmas about address translation behavior that Goel used to verify a memory-management program called Zero-Copy [14]. This work was done on a model lacking a TLB, but because our approach reduces address translation in the presence of a TLB to address translation by walking the page tables, we were able to reuse much of Goel's work and re-verify an updated version of the Zero-Copy program on the ACL2, x86-ISA model with a TLB. We discuss this further in Section VI.

A. TLB Implementation

Our TLB is modeled as a mapping from virtual-page numbers, various control bits that control address translation, and the access type (i.e., read, write, or execute) to physical-page numbers. Our TLB treats all memory addresses as being contained within 4KB pages and may use multiple entries corresponding to naturally aligned 4KB regions to cache translations for larger page sizes. The x86 ISA requires implementations to invalidate one or more TLB entries on certain events. For example, when executing the `INVLPG` instruction all TLB entries corresponding to the page containing the given virtual address must be invalidated. Whenever an event occurs that requires one or more TLB entries to be invalidated, our model clears the TLB in its entirety. While our choice to cache large pages with multiple 4KB region entries and invalidate the entire TLB whenever we are required to invalidate any TLB entry may seem odd, our implementation is allowed by the Intel x86 Software Developer's Manual [12].

This design was chosen in part to maximize the speedup it gave in the execution performance of our x86-ISA model because when executing x86 programs using the model as an interpreter, a significant portion of the runtime is spent translating addresses. With this design, we can represent TLB mappings using the ACL2 *fast-alists* mechanism [15]. In the ACL2 logic, a fast-alist is an association list (or alist), a list of key-value pairs, but in execution, ACL2 associates with the alist a hash-table. To speed up execution, ACL2 maintains the invariant that looking up a key in the hash-table yields the same value as the value associated with the key in the alist. This allows us to have hash-table performance while retaining the reasoning simplicity of a list of key-value pairs, but prevents us from removing key-value pairs from the alist. Thus, we choose to clear the TLB by replacing the TLB alist with an empty one whenever we need to invalidate entries.

When our x86-ISA model translates a virtual memory address to a physical address, it takes the virtual-page number of the address being translated, the current state of the processor

control bits that affect address translation, and the memory-access type, and it searches for a corresponding virtual-page number in our TLB. If it finds an entry, it uses the corresponding physical-page number to create a translated address by concatenating the lower 12 bits from the virtual address to the physical-page number found in the TLB. If no entry corresponding to the translation is found in the TLB, our x86-ISA model performs a page-table walk to translate the virtual address. If this translation fails, it triggers a page fault and does not update the TLB. Otherwise, it updates our TLB by adding an entry corresponding to the translation that maps to the physical-page number returned by the page table walk. Unlike a hardware TLB, our TLB model can grow to contain an arbitrary number of entries.

B. Reasoning Implementation

We implement PROVE-CONSISTENT implicitly by proving theorems that characterize the maintained and consistent set of translations for each operation of our x86 model. Then, when we attempt to prove a theorem that states a translation is TLB consistent after a sequence of operations, the ACL2 rewriter will attempt to apply our rewrite rules in a manner that implements the PROVE-CONSISTENT algorithm. If the sequence of operations is empty, it will attempt to prove the translation is consistent with respect to the initial state. If the translation under consideration can be proven to be in the set of maintained translations of the outer-most operation the ACL2 system, due to the rewrite rule associated with the theorem that describes the set of maintained translations of the outer-most operation, will rewrite the conjecture to prove that the address translation is TLB consistent with respect to the previous state. If the translation can be proven to be in the set of consistent translations of the outer-most operation, it will confirm the validity of the conjecture. This is exactly the PROVE-CONSISTENT algorithm.

We defined TLB consistency for the x86-ISA model in ACL2, shown in Figure 3. Recall, we informally defined TLB consistency to mean the TLB does not contain any stale entries corresponding to a given translation. Our formal definition checks that performing the given translation by first consulting the TLB, and falling back on the page tables if there is no relevant TLB entry, yields the same result as performing the translation by walking the page tables directly, without consulting the TLB. Since a stale TLB entry is by definition a TLB entry that is inconsistent with the page tables, our formal definition of TLB consistency captures our informal definition.

We define rewrite rules that describe the set of maintained translations and the set of consistent translations of the operations of the machine. For example, Figure 4 shows a theorem that states that all translations are maintained by the address translation operation. Since this theorem has no hypotheses it is stored as an unconditional rewrite rule by the ACL2 theorem prover. This rule programs the rewriter to rewrite terms that are calls to `tlb-consistent` composed with the address translation operation. Such terms are rewritten to calls of `tlb-consistent` on the state the address translation operation was applied to, i.e., the state before the address translation operation was applied. It is also possible for us to define the set of maintained operations in terms of the parameters to the operation or the machine state, as shown in Figure 5; this theorem states that a translation remains TLB consistent after a write to physical memory if that write does not change any page table entry used when performing the translation. This lemma is stored as a conditional rewrite rule in the ACL2 prover; when the ACL2 rewriter matches a term to the left-hand side of the equality, it attempts to recursively rewrite the hypotheses to true. If it succeeds, it will rewrite the term to the right-hand side.

In our x86-ISA model, we have only a single operation that has a non-empty set of consistent translations because clearing the TLB is the only way we invalidate TLB entries. Figure 6 shows a theorem which states that a machine state with empty TLB is consistent for all translations. While we could have instead proven all translations are consistent after the clear TLB operation, this theorem is more general, and the rewriter will easily be able to apply it in the same way when proving a translation is TLB consistent after the clear TLB operation. When the rewriter sees any `tlb-consistent` term, it will attempt to prove that the TLB in the given machine state is empty. If it succeeds, it will rewrite the `tlb-consistent` term to true.

VI. CASE STUDY: ZERO-COPY

We demonstrate the effectiveness of our TLB reasoning technique by proving the correctness of a supervisor memory-management program. Zero-Copy is a program that copies a page of data in the virtual-address space to another page in the virtual-address space. It does this not by copying the data but instead by modifying the page tables so that the destination virtual addresses translate to the same physical addresses as the source virtual addresses. Goel proved the correctness of this program on the x86-ISA model without a TLB [14]. Since Goel's program was written for a version of the x86-ISA model that lacked a TLB, it did not attempt to invalidate TLB entries. Under certain assumptions, Goel proved the following theorems to verify Zero-Copy:

Theorem 1 (Source Unmodified): After the program runs, the data in the source region is identical to the data in the source region prior to the run.

Theorem 2 (Destination Updated): After the program runs, the data in the destination region is the same as the data in the source region before the program runs.

Theorem 3 (Program Unmodified): After the program runs, the program in memory is the same as the program in memory before the program runs.

We added code to Zero-Copy that invokes the x86 `INVLPG` instruction to invalidate TLB entries in the destination region after updating the page tables to avoid the TLB containing stale entries, which is necessary for the correctness of the program. The only TLB-related change we made to the statements of Goel's correctness theorems was the addition of hypotheses that required translations used by the program to be

```
(define tlb-consistent ((lin-addr canonical-address-p)
                        (r-w-x    :type (member  :r :w :x))
                        x86)
  ...
  (b* ((lin-addr (mbe :logic (logext 48 (loghead 48 lin-addr))
                      :exec lin-addr))
       ((mv flg phys-addr &) (ia32e-la-to-pa lin-addr r-w-x x86))
       ((mv flg2 phys-addr2 &) (ia32e-la-to-pa-without-tlb lin-addr r-w-x x86)))
      (and (equal flg flg2)
           (equal phys-addr
                  phys-addr2)))
  ...)
```

Fig. 3. Function `tlb-consistent` is the ACL2 definition of TLB consistency for our x86-ISA model. `ia32e-la-to-pa` performs an address translation by consulting the TLB first. If it finds a relevant TLB entry, it uses the entry to perform the translation. If no relevant TLB entry was found, it uses the page tables to perform the translation. `ia32e-la-to-pa-without-tlb` performs an address translation using only the page tables. `tlb-consistent` returns true if and only if the given translation has the same result when translated with both of the aforementioned functions. Some details not relevant to the logical definition have been elided.

```
(defthm translating-addresses-maintains-consistency
          (equal (tlb-consistent lin-addr r-w-x
                                 (mv-nth 2 (ia32e-la-to-pa lin-addr2 r-w-x2 x86)))
                 (tlb-consistent lin-addr r-w-x x86))
          ...)
```

Fig. 4. ACL2 definition of `translating-addresses-maintains-consistency`, a theorem that states a translation is TLB consistent after the address translation operation if and only if it was TLB consistent before the address translation operation. This is a stronger statement than the address translation operation's set of maintained translations is all translations because it is biconditional instead of only an implication. This theorem is stored as a rewrite rule that can be used by the ACL2 rewriter to rewrite appropriate terms. Hints to the prover to help it prove this theorem have been elided.

```
(defthm writing-non-page-table-memory-maintains-tlb-consistency
          (implies (disjoint-p (list index)
                               (xlation-governing-entries-paddrs (logext 48 lin-addr) x86))
                   (equal (tlb-consistent lin-addr r-w-x (xw :mem index val x86))
                          (tlb-consistent lin-addr r-w-x x86)))
          ...)
```

Fig. 5. ACL2 definition of `writing-non-page-table-memory-maintains-tlb-consistency`, a theorem that states that writing to physical memory does not change the consistency of a translation if the write is to an address that is not part of a page table entry that may be consulted when performing the translation. Like the theorem in Figure 4, it is stronger than showing that the translation is in the set of maintained translations of the physical memory write operation since it states that translation's consistency is unchanged by the memory write, rather than stating that the translation is consistent after the write if it was consistent before the write. Hints to the prover to help prove this theorem have been elided.

```
(defthm empty-tlb-is-consistent
        (implies (atom (xr :tlb nil x86))
                 (tlb-consistent lin-addr r-w-x x86)))
```

Fig. 6. ACL2 definition of `empty-tlb-is-consistent`, a theorem that states when the TLB is empty all translations are consistent.

TLB consistent in the machine's initial state. The correctness theorems do not hold without these hypotheses. We also added minor hypotheses unrelated to the TLB to accomodate other changes made to the x86-ISA model between when Goel initially verified the Zero-Copy program and when we verified the updated Zero-Copy program with our TLB enhanced version of the x86-ISA model.

Using our methodology, we were able to reverify Zero-Copy without significant changes to the structure of the proof. Our approach involves showing that the translations the program uses are TLB consistent and then using that fact to reason about the results of address translation as though the translation was performed using the page tables. This allowed us to avoid having to make major modifications to Goel's proofs since, when a translation is TLB consistent, the results of the translation are the same as they would have been in the model Goel used that lacked a TLB.

Since we implemented our reasoning strategy using rewrite rules, we were able to exploit ACL2's rewriter and reprove many theorems automatically. While most of the lemmas about

address translations and virtual memory that Goel used in her proof were not theorems in the updated x86-ISA model with a TLB, they were theorems after adding hypothesis that require the translations relevant to the lemma to be TLB consistent. After adding such hypotheses to the various lemmas in Goel's proof, they could often be re-proven automatically. Other theorems required additional theorem-prover hints. Many of these updates were done to the library Goel developed for x86-ISA, machine-code verification, and these updated lemmas can be used directly to verify other programs.

The most involved changes were in the proof of a lemma that stated if a leaf 1 GB page table entry is updated and then values are read from the virtual-memory region corresponding to the page table entry, the values read will be equal to the values in the page at the physical address specified in the updated page table entry. After adding the condition that TLB consistency must hold for all the read translations in the virtual-memory region after the page table entry is written, the proof of this particular lemma was redone with significant changes to how it was structured. In principle, with the old line of reasoning it should have been possible to update using our method, but ACL2 could not automatically re-prove the lemmas, and we decided to redo much of the proof to make it easier to understand and maintain.

Using our approach minimized the complexity of reasoning with the x86-ISA model that was introduced by the TLB's semantics. Proofs that concern code that does not update the page tables was largely done in the same fashion as proofs done on the model without a TLB because our technique allowed the ACL2 rewriter to often automatically discharge TLB-consistency hypotheses, which were the primary complication introduced by the TLB when reasoning about such code. Proofs regarding code that updates the page tables required more involved changes in reasoning, but these changes were localized to only the small pieces of reasoning concerning page table updates. Future work could abstract some of these pieces of reasoning into a library that can be shared between proofs to minimize the amount of work needed for reasoning about page table updating code.

VII. Conclusion & Future Work

We described a procedure to reduce theorems about address translation in the presence of a TLB cache to address translation without caching. We added a TLB to the ACL2 x86-ISA model and implemented our algorithm implicitly using ACL2 rewrite rules. Our reduction is performed by proving that the translations we are considering are TLB consistent, meaning if there is a relevant entry in the TLB it agrees with the page tables. We then successfully applied our methodology to re-verify the Zero-Copy program that had been verified by Goel with a previous version of the x86-ISA model that did not have a TLB. We found that our methodology allowed us to reuse much of her proof. We have released our implementation and proofs under the 3-clause BSD license, and they are available, along with documentation [4], in the ACL2 Community Books [3].

There is much more to be done with respect to reasoning about memory-management software for contemporary microprocessors. To comprehensively verify security and correctness of operating systems, it will be necessary for the formal-methods community to continue the formalization of the instruction set architectures that are employed by contemporary computer systems. In particular, we would like to see address translation cache models extended to other address translation caches beyond TLBs, like page table entry caches. Additionally, we would like to see our reasoning technique implemented for a TLB model which does not make any assumptions about the TLB microarchitecture beyond that it is consistent with the ISA.

Acknowledgment

The authors would like to thank the ACL2 developers and community for their help over the 15+ years we have been pursuing this project. This work was supported, in part, by Intel Corporation.

References

[1] M. Kaufmann and J. S. Moore, "ACL2," https://www.cs.utexas.edu/users/moore/acl2/, 2025.

[2] M. K. Shilpi Goel, Warren A. Hunt Jr., "Engineering a formal, executable x86 isa simulator for software verification," in *Provably Correct Systems (ProCoS)*. Springer International Publishing, 2017, pp. 173–209.

[3] The ACL2 Community, *The ACL2 Community Books*. https://github.com/acl2/acl2/tree/master/books, 2024.

[4] ——. (Accessed, August 2025) X86isa. [Online]. Available: https://www.cs.utexas.edu/~moore/acl2/manuals/latest/?topic=ACL2____X86ISA

[5] ACL2, *User manual for the ACL2 Theorem Prover and the ACL2 Community Books*, accessed 2025-08-13. [Online]. Available: https://www.cs.utexas.edu/~moore/acl2/manuals/latest

[6] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[7] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2560537

[8] H. Syeda and G. Klein, "Reasoning about translation lookaside buffers," in *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, T. Eiter and D. Sands, Eds., vol. 46. EasyChair, 2017, pp. 490–508. [Online]. Available: https://doi.org/10.29007/c2f1

[9] ——, *Program Verification in the Presence of Cached Address Translation*. Springer, Lecture Notes in Computer Science, July 2018, pp. 542–559.

[10] A. Fox and M. O. Myreen, "A trustworthy monadic formalization of the armv7 instruction set architecture," in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–258.

[11] ARM Corporation. (Accessed, May 2025) Arm architecture reference manual armv7-a and armv7-r edition. [Online]. Available: https://developer.arm.com/documentation/ddi0406/latest/

[12] Intel Corporation, *Intel 64 and IA-32 Architecture Developer Manuals*. Intel Corporation, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[13] W. A. Hunt Jr., M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of the Royal Society of London Series A*, vol. 375, no. 2104, September 2017.

[14] S. Goel, *Formal Verification of Application and System Programs based on a Validated x86 ISA Model*. The University of Texas, 2016. [Online]. Available: https://repositories.lib.utexas.edu/server/api/core/bitstreams/858b2f9b-5532-4b2a-bed1-fb889a265f6c/content

[15] B. Boyer and W. A. H. Jr. (Accessed, May 2025) Fast-alists. [Online]. Available: https://www.cs.utexas.edu/~moore/acl2/v8-5/manual/index.html?topic=ACL2____FAST-ALISTS