ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 415

Systems Group, Department of Computer Science, ETH Zurich

Verified Paging for x86-64 in Rust

by

Matthias Brun

Supervised by

Prof. Timothy Roscoe,
Andrea Lattuada

October 2, 2022

D INFK

Abstract

Writing correct operating systems is hard because they interact with complex computer hardware. At the same time, their reliability is extremely important because they represent the foundation of the many other software systems relying on them.

Machine-checked formal verification can provide strong guarantees about a program's behavior in all possible situations. SeL4 and other projects have demonstrated that the complete verification of an operating system's kernel is feasible but time-consuming. More recently, the VeriBetrKV project has shown that SMT-based verification workflows can scale to large end-to-end verification projects of low-level systems software.

In this thesis we develop paging functions targeting x86-64 hardware. We implement the functions in Rust and verify them using the new SMT-based verifier Verus. Finally, we integrate the functions into a research operating system and evaluate their performance.

By verifying one subsystem of an OS, we demonstrate that an approach based on Verus and state machine refinement is effective for proving complex properties about operating systems.

Our work has influenced Verus' development in multiple ways and we found and reported a large number of bugs, including ones affecting the soundness of verification results.

Acknowledgements

First and foremost I would like to thank my supervisor Andrea Lattuada. His competent supervision of my thesis work has made the last six months a very enjoyable experience. Much of his advice and what I have learned during our discussions will still be useful to me long after handing in this thesis.

I would also like to thank Prof. Timothy Roscoe for the opportunity to do my thesis in his research group.

Regular discussions with Reto Achermann, Gerd Zellweger and Jon Howell significantly improved this project. Reto and Gerd's expertise helped getting the implementation and hardware specification right, while Jon's experience was invaluable for figuring out how to best structure the project and the state machine refinement.

Many Verus contributors — Andrea Lattuada, Chris Hawblitzel, Chanhee Cho, Travis Hance, Bryan Parno, Jay Bosamiya — helped, whether through suggestions, help with understanding Verus' features or with fixing issues in Verus that caused problems in this project.

Prof. Timothy Roscoe, Dr. David Cock and Ben Fiedler gave useful feedback after a preliminary presentation of the thesis plan.

Comments from Roman Meier and Gerd Zellweger on drafts helped improve the presentation of this thesis.

Contents

Part I

Introduction and Overview

Chapter 1

Introduction and Motivation

An operating system (*OS*) is the foundation of many modern computer systems, supporting all kinds of workloads that run on top of it. It abstracts away the complex details of the underlying hardware, providing user-space programs with the illusion of a much simpler environment. They can operate in their own private virtual memory and use the OS' interface to access persistent storage and other devices.

To provide these services, parts of the OS run at a higher privilege level than user-space software, meaning programming errors may have far-reaching consequences. They can compromise the correctness of the entire software system built on top of it. Further, modern hardware is very complex, which makes the OS' task — interfacing with the hardware — particularly difficult, thus increasing the likelihood of programming errors.

The OS' important role in the overall system in conjunction with its complex environment makes it very desirable, yet exceedingly difficult to obtain a high degree of confidence in its functional correctness.

Conventional techniques for improving software quality include testing and code reviewing and auditing. Testing can be an excellent, low-cost measure to discover defects in a program. However, testing can only ever exercise a small, finite part of a program's behaviors and is thus inadequate for the task of ensuring a complex program's correctness. Similarly, while reviewing and auditing of code does help in improving overall code quality and finding programming errors, it cannot guarantee full correctness of a complex program. Even if an audit does make correctness claims, it is not generally accompanied by an artifact that allows a third party to easily verify these claims for themselves.

Machine-checked formal verification is an alternative that can provide strong guarantees about a system's behavior in all possible situations. In particular, it can guarantee that a large, complicated code base adheres to a much smaller and simpler formal specification. Gaining confidence in the correctness of the verified system only requires manual inspection of this specification. As long as the specification accurately describes what is meant by "correctness" of the program, the verification process guarantees that the program is indeed correct.

Several projects [30, 19, 37, 60, 43] have demonstrated the feasibility of complete, machine-checked formal verification of operating systems. Many of these projects [30, 19, 37] rely on interactive theorem provers such as Isabelle/HOL [44] or Coq [55] to perform verification. Projects [60, 43] using automated theorem proving have also been successful, though more limited in the functionality supported by the kernel.

More recently, the IronFleet project [22] introduced a methodology using Dafny [34], a verification tool based on the SMT solver Z3 [11], to verify distributed systems. VeriBetrKV [20] expands on this by improving the methodology and demonstrating its applicability to complex systems software interacting with an asynchronous environment.

Verus [9] is a new automated verification tool in early development, aiming to provide a better alternative for verifying full functional correctness of systems-level code, such as that in VeriBetrKV. Verus is based on the Rust programming language [39], which supports manual memory management and is intended for low-level systems code. Dafny code can only be compiled to high-level languages such as C#, while interactive theorem provers tend to only support code extraction to functional programming languages such as Haskell. Projects verifying low-level code with these systems usually do not use code extraction and instead import code written in a more suitable language. Verus' tight integration with Rust reduces the complexity of the overall toolchain. Furthermore, Rust's advanced type system has unique advantages for verification. Ruling out the existence of mutable, aliased references simplifies the verifier's task significantly.

In this work, we implement page table management functions targeting x86-64 hardware in Rust and verify them using Verus. We integrate the functions into the NrOS [3] research operating system and evaluate their performance. To perform the verification, we also develop a set of specifications, including a model of relevant aspects of the x86-64 ISA. By verifying one subsystem of an OS, we demonstrate that an SMT-based verification workflow is effective for proving complex properties about operating systems.

The paging implementation is verified to be functionally correct in terms of its observable behavior from the perspective of user-space programs. We aim to surpass existing work in terms of fidelity of the modeled environment. A more accurate model makes it easier to convince oneself that the hardware model does in fact reflect the real hardware's behavior. For modern hardware this implies modeling a multiprocessor system with a realistic memory model and including the effects of translation caching by the TLB in the model. As a step towards this goal, we verify the implementation with respect to a hardware model that includes a TLB but assumes a uniprocessor. We do not yet verify the implementation's correctness for multiprocessor hardware but believe it to be correct when combined with a concurrency mechanism and TLB shootdown protocol such as those implemented in NrOS.

Insights gained over the course of this project have influenced Verus' development in several areas. We also uncovered and reported a large number of bugs in Verus, some affecting the soundness of verification results. In part thanks to this, Verus is now a more mature and more robust tool than at the start of this project. This work is the largest body of Verus-verified code to date and the first to apply Verus in the verification of a real system.

Gerd Zellweger, Reto Achermann and Jon Howell are collaborators whose input influenced the design and overall structure of this project at a conceptual level. Gerd and Reto also contributed a small amount of proof code.

This thesis is structured as follows: In Chapter 2 we discuss related work and other important background. We then introduce Rust and Verus in Chapter 3 and provide an overview of the modeling and verification techniques we use in this project. In Chapter 4 we study the structure of the overall artifact, discussing the purpose and the high-level goals for each component of the system as well as how these components are integrated to yield a complete system. In the subsequent chapters 5 through 11 we consider each component in more detail. We proceed by evaluating our artifact in Chapter 12 and discussing our experience with Verus in Chapter 13. Finally, we outline possible future work in Chapter 14 and conclude in Chapter 15.

Chapter 2

Background and Related Work

In this chapter we discuss background required to understand this thesis, specifically how paging works on the x86-64 architecture and a general overview of the research operating system NrOS. Further, we explore related work in the areas of verification of operating systems, verification of Rust code and end-to-end verification using SMT solvers.

2.1 Paging on x86-64

This section describes paging on the x86-64 ISA as specified in Chapter 4 of Volume 3 of the Intel Software Developer's Manual [10] (*SDM*). For the most part we skip details relating to configuring the processor to enable paging in the correct mode, since we are concerned with proving correct an implementation of the page table operations, not the correct initialization of the processor. We will, however, consider configuration details insofar as they are relevant to our goal.

2.1.1 Address Translation with Page Tables

We intentionally deviate from Intel's naming conventions in favor of a more straightforward naming scheme. We call the overall data structure the *page table* even though it is not implemented as an actual table. What Intel refers to as the *PML4 table*, *page directory pointer table*, *page directory* and *page table* are all called simply *directory* in this thesis. To refer to a specific one we simply call it the layer N directory, where 0 is PML4 and 3 is the page table.

Four different paging modes are supported: 32-bit, PAE, 4-level and 5-level paging. We are interested only in ordinary 4-level paging (cf. Section 4.5 in the SDM).

The purpose of paging is the translation of virtual addresses to physical ones. This is achieved with a multi-level page table, a tree data structure.

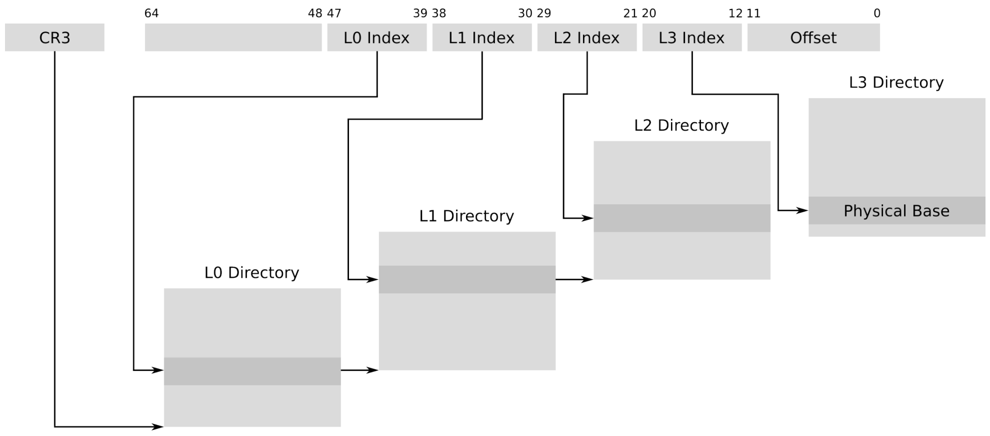


Figure 2.1: Multi-level paging on the x86-64 architecture

The root directory contains entries that reference (child-)directories. Child directories may contain either further recursive directories or denote a mapping. In the case of 4-level paging, this tree is at most four layers deep.

An address translation (called *page table walk*) in this mode is visualized in Figure 2.1. The CR3 register contains the physical address of the level 0 paging directory. Bits 47:39 are used as an index into this table to obtain the physical address of the level 1 paging directory. We then index with the subsequent nine bits 38:30 and iterate this process until we reach an entry containing an address translation rather than a directory pointer. In the usual case this means reaching the layer 3 directory, whose entries are by definition translations mapping a 4KiB page. An earlier termination of this page table walk happens if a *huge page* mapping is encountered. These can appear in layers one or two and have sizes of 1GiB or 2MiB, respectively but are not allowed in layer zero.

The bits used in the translation process are the 36 bits 47:12. The least significant twelve bits do not need to be translated. A normal page mapping maps a 4KiB, or 2^{12}B region, so these twelve bits are just an offset that can be copied to the physical address directly. For huge page mappings this concept extends to the least significant 21 or 30 bits, depending on the mapping's size.

This address translation process is implemented in the *memory management unit* (MMU) in the processor. However, setting up the page table with the correct translations and — when necessary — modifying it at runtime is the kernel's job. Necessary operations are the mapping and unmapping of pages. Often it's also desirable to be able to translate a virtual address to a physical one by implementing a page table walk in software. The goal of this thesis is to implement verified versions of these three operations. One key complication in these operations is that on a multiprocessor system, while the kernel is modifying the page table, it cannot prevent other cores' MMUs from accessing the page table concurrently and seeing intermediate states.

Note that addresses in page table entries referencing page directories are physical ones. However, the kernel typically runs in virtual memory, meaning any access to those addresses first undergoes address translation itself. To easily access physical addresses, kernels usually map the physical memory (or at least the part of it containing page table data) at a specific virtual address. Accessing a physical address in the kernel then becomes as easy as adding an offset and accessing the resulting address.

2.1.2 Page Table Management

Given this address translation mechanism, an operating system needs to be able to modify the page table, e.g. to set up a virtual address space for a newly spawned process, or to satisfy a process' request for increased or reduced virtual memory. Usually, this functionality is split across functions, with one function mapping new memory regions and another unmapping existing pages. Sometimes an operating system also includes further functions, such as one for adjusting existing mappings or for resolving a virtual address to the corresponding mapping.

In this thesis we implement functions for mapping, unmapping and resolving page table entries.

2.1.3 Translation Lookaside Buffers

Address translation means that any memory access will cause additional memory accesses during translation. With four-level paging this can mean that serving a program's memory access requires the processor to perform five memory accesses. Due to this, MMUs include so-called *Translation Lookaside Buffer*s (TLBs) to cache recent address translations.

While TLBs have profound implications for performance, what we are really interested in is how they can affect the correctness of our page table operations. They are not automatically kept coherent with the page table. Doing so is the responsibility of the kernel when modifying the page table.

Simply resolving addresses or mapping new entries is not a concern but when a mapping is removed, the TLB may still have that mapping cached. In that case, later memory accesses to that region would be successful rather than failing. If the backing physical memory is reused, all sorts of correctness and security issues may ensue. To ensure this does not happen, the kernel can either flush the TLB or perform targeted TLB invalidations for specific entries.

On a single-processor system this is a fairly straightforward process but on multiprocessor systems it is more complicated. We need to ensure that the TLBs of all cores also remain coherent. In a concurrent operating system this means we have to run a distributed protocol between the threads. Because we focus on the single-processor case we do not discuss this in more detail here.

Modern processors also include other caches to speed up translation. These are kept coherent with the caches on other cores by the hardware and thus are not a concern for our correctness arguments.

2.2 Operating Systems Verification

First attempts at formally verifying operating systems go as far back as the 1970s [56, 14]. The first complete, machine-checked formal verification of a general-purpose OS kernel's functional correctness was reported in 2009 with the seL4 project [30, 29]. Since then, a number of other projects [19, 37, 60] have reported similar results, though most of them for smaller or more functionally restrictive kernels. CertiKOS [19] was verified not just for uniprocessor hardware but for multiprocessors as well. Verve [60] is an operating system whose type safety and memory safety was verified using SMT-based verification methods, unlike seL4 and CertiKOS which were verified with the Isabelle/HOL [44] and Coq [55] proof assistants respectively. The SeKVM project [37] restructured the KVM hypervisor into a small, verified core with a set of unverified services, guaranteeing the confidentiality and integrity of virtual machine data.

The formal verification of an operating system is such an enormous task that any successful attempt has to make compromises. All the mentioned projects verify very small kernels and push a lot of complexity into user-space instead. Verification projects for operating systems and other software often assume simple hardware models. Some projects [30, 15, 42] assume a uniprocessor hardware model while others [19, 37] are verified for multiprocessors but assume a sequentially consistent memory model, while offering informal reasoning for why the proofs extend to realistic memory models. Under sequential consistency, memory operations on all CPU cores would have to be executed in some sequential order that respects program orders [31]. The memory models of contemporary hardware architectures provide far weaker guarantees [45, 49].

Recent work by Tao et al. [54] replaces SeKVM's sequential consistency assumption with the Promising Arm [48] model of Arm's relaxed memory semantics.

The TLB is another complication that is often left out of formal proofs. TLB shootdown protocols on multiprocessors in particular have significant potential for subtle errors. CertiKOS does not verify TLB shootdown, while it is not apparent from the SeKVM paper [37] whether or not it does. TLB invalidation on uniprocessors is much easier and for example Komodo [15] models the TLB and verifies TLB invalidation. Hoenicke et al. [26] verify the design of the Mach kernel's TLB shootdown protocol [4] with a pen-and-paper proof. Other work [53, 52] explores the effects of translation caching on a program logic for low-level programs, assuming a uniprocessor.

2.3 Rust Verification

With the rising popularity of Rust [39] as a systems programming language, there has also been a lot of interest from the verification community. Much of this interest is due to Rust's type system that makes Rust especially amenable to verification. RustBelt [27] is an early project that develops a formal semantics for a simplified version of Rust, including *unsafe* blocks, proves its soundness and verifies that Rust's standard library preserves the soundness.

Many other projects since then target the problem of verifying the correctness of programs written in certain subsets of the Rust language. Prusti [2] uses the Viper [41] infrastructure to verify Rust programs. The workflow intends the developer to annotate the source code with pre- and postconditions, invariants and assertions, which are then automatically checked by automated verifiers. Verus [9] adopts a similar approach but encodes directly to Z3 [11] and emphasizes the importance of small, efficient verification conditions. It also uses linear types to reduce the need for the verifier to perform alias reasoning [36].

Creusot [12] supports a large fraction of Rust's features and uses a semi-interactive approach to proof development, with the Why3 [16] backend.

Aeneas [24] offers part of a verification toolchain, that can translate a large subset of Rust programs into a pure lambda calculus, intended to enable verification with proof assistants.

2.4 SMT-Based End-to-end Verification

Many verification projects either verify only the abstract design of a system or only limited properties of concrete implementations. *End-to-end verification* is the complete verification of a system, which includes proofs to establish global properties of a model of the system and proofs that show the concrete implementation to correspond to that model. As a result, the global properties proved of the model extend to the implemented system.

Historically, many large end-to-end verified projects tended to use interactive theorem provers [30, 19, 37, 35]. Recently, Hawblitzel et al. [22] developed a methodology for verifying distributed systems with Dafny [34], a verifier based on SMT solvers. IronFleet's methodology is based on TLA-style state machine refinement [1, 32, 38]. The VeriBetrKV project [20] applied IronFleet's approach to the verification of a relatively large crash-safe key value store.

Sprenger et al. [51] propose combining interactive theorem provers with SMT-based tools for the end-to-end verification of distributed systems.

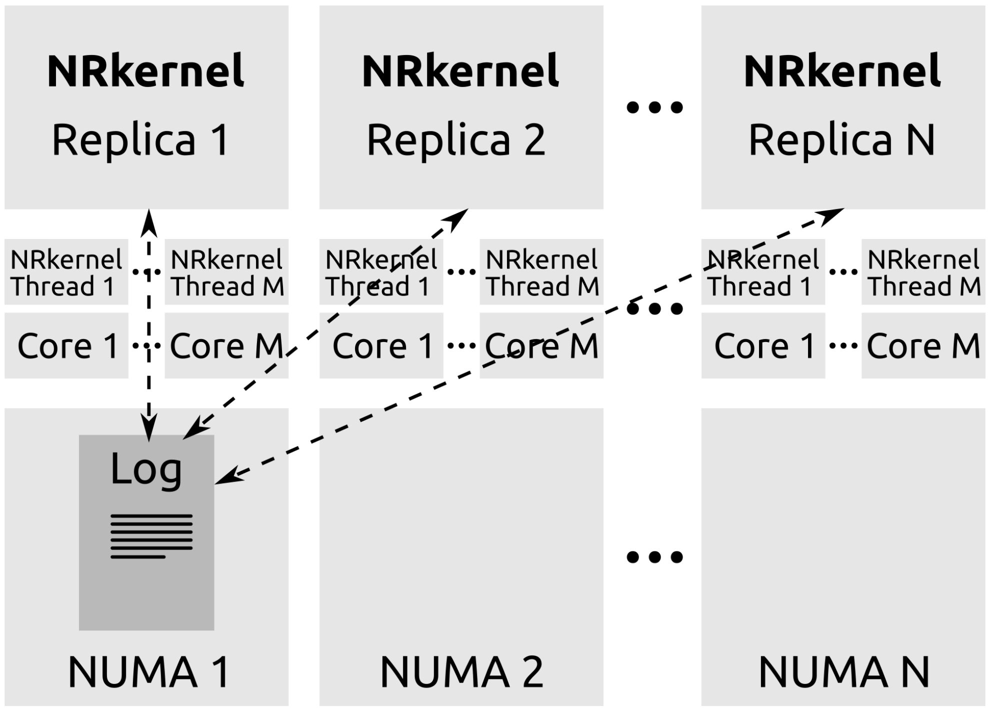


Figure 2.2: The structure of NrOS [a]

[a] adapted from Figure 1 in the NrOS paper [3]

2.5 NrOS

Node-replicated OS, or NrOS [3], is a research operating system whose concurrent kernel is constructed by primarily using sequential data structures and extending them to linearizable [23], concurrent data structures with a dedicated, efficient replication mechanism, node replication [7]. NrOS targets non-uniform memory access (NUMA) systems and scales similarly well or even better than other operating systems.

NrOS keeps a replica of the kernel structures on each NUMA node, thus avoiding expensive cross-NUMA memory accesses. Mutations of the kernel state are appended to a shared log, which the other replicas use to synchronize their states. This structure is visualized in Figure 2.2.

The approach of relying primarily on sequential data structures and a single replication mechanism simplifies reasoning about NrOS' correctness, making it more amenable to verification.

(Figure is adapted from Figure 1 in the NrOS paper, kernel thread and core numbering not ideal here)

Chapter 3

Verification with Rust and Verus

In this chapter we introduce the Rust programming language, the Verus toolchain and the verification techniques we use in this project.

3.1 Rust

Instead of boring the reader with long-winded exposition of each feature, we introduce Rust by writing a small program and using it to highlight Rust's differences to other imperative programming languages. The purpose of this section is to introduce features that are important for understanding this thesis. We do not cover many of Rust's advanced features and refer to *The Rust Programming Language* [28] for a more detailed introduction to the language.

The program we are going to implement is a binary search tree data structure and a function that checks whether a particular search tree contains a particular value.

3.1.1 Making a Tree out of Enums and Structs

We start out by specifying the data structure that encodes the search tree. We want to define a recursive `Tree` type, which can either be a `Leaf` or an intermediate `Node` that contains a value and has two child nodes. We can do so using an `enum` and a `struct`:

```
struct TreeNode {
  value: usize,
  left:  Tree,
  right: Tree,
}

enum Tree {
  Leaf,
  Node(Box<TreeNode>),
}
```

Similar to C, a struct in Rust is a type consisting of zero or more fields, each of which has a name and the type of the value it holds. Given a struct, we can access the values in its fields: `tree_node.value`.

An enum defines several variants where each variant can have some number of fields. A value of an enum type is exactly one of its variants. So, a value of our type `Tree` can either be a `Leaf` with no fields or a `Node` with one field containing a `TreeNode` struct. We are wrapping the `TreeNode` in the `Box` type so the variant only holds a reference to a `TreeNode`, not the `TreeNode` itself.

Note that our `TreeNode` and `Tree` types are mutually recursive. In Rust, we can use items before the location where they are defined.

The type describes a general binary tree. In the implementation we assume that we are working with *search trees*. A search tree is a binary tree where for any intermediate node containing a value n, all values in the left subtree are less than n and all values in the right subtree are greater than n.

3.1.2 Finding a Value in the Tree

Next, we define a function to operate on our new types. The function will take a `Tree` and a `usize` as arguments and check whether or not the tree contains the value.

```
impl Tree {
  fn contains_value(&self, m: usize) -> bool { .. }
}
```

Function definitions start with the `fn` keyword, followed by the name, the arguments with their respective types and, finally, after an arrow `->`, the function's return type and a block containing the function body. This function has a special first argument called `self` because it is defined in an `impl` block. With the `impl` keyword, we can define *methods*, i.e. functions that take a particular struct or enum as their first argument and which we can call with a convenient syntax, e.g. `tree.contains_value(x)` rather than `contains_value(tree, x)`. The first argument is always called `self` and has the type given after the `impl` keyword.

In this function we place an ampersand before the `self` argument to indicate that the argument is only *borrowed*. If we omitted the ampersand, `contains_value` would take ownership of the argument, meaning that calling the function consumes the value. Rust's linear type system ensures that any value is only used once. By borrowing the argument instead, we only get an immutable reference and the caller retains ownership. [1]

Our function's first argument, `self`, has type `TreeNode`. The first thing we need to do is check which variant the argument is.

```
match self {
  Tree::Leaf => { .. },
  Tree::Node(node) => { .. },
}
```

Rust's `match` statement allows us to do a case distinction between the possible variants. With this knowledge we can now implement the full function.

```
fn contains_value(&self, m: usize) -> bool {
  match self {
    Tree::Leaf => {
      false
    },
    Tree::Node(node) => {
      if m == node.value {
        true
      } else if m < node.value {
        node.left.contains_value(m)
      } else {
        node.right.contains_value(m)
      }
    },
  }
}
```

If the tree is a `Leaf`, it does not contain the value and we return false. We do not have to explicitly write `return false;`. An expression without a semicolon is treated as an implicit return in Rust, similar to functional programming languages.

If the tree is a `Node`, we check whether its value is the one we are looking for. If not, we do a recursive call on either the left or the right subtree. We rely on the search tree property to ensure that we do find the value if it is in the tree.

[1] Rust's linear type system is a complex topic, which we cannot do justice in this brief introduction. For more information refer to [57, 39].

3.1.3 Modifying the Tree

To illustrate how Rust supports mutability, we now implement a function that inserts a new value into the tree.

```
fn insert_value(&mut self, m: usize) { .. }
```

In contrast to `contains_value`, the first argument is `&mut self`, meaning we borrow the tree *mutably*. When our function returns, the caller will still have ownership of the tree but it may have changed because `insert_value` has permission to modify the contents. Rust guarantees at compile-time that only one mutable reference to any particular location can exist at a time.

As in `contains_value` we pattern match on `self`. If the tree is a `Leaf`, we replace it with a new node containing the value to be inserted.

```
Tree::Leaf => {
  let new_node = TreeNode {
    value: m,
    left:  Tree::Leaf,
    right: Tree::Leaf,
  };
  *self = Tree::Node(Box::new(new_node));
},
```

The `let` statement creates a new variable binding, to which we assign a new `TreeNode`. Variable bindings are immutable by default in Rust. To create a variable whose content can be modified we would have to use `let mut new_node = value;`.

We then replace the tree, which is a `Leaf`, with our new node by dereferencing the mutable reference `self` with the prefix asterisk (`*`) and assigning to it.

If the tree is an intermediate node instead, we recursively insert the value into either the left or right subtree, which completes this function definition, as shown in Figure 3.1.

3.2 Verus

Verus is developed by Andrea Lattuada (ETH Zurich), Chris Hawblitzel (Microsoft Research), Chanhee Cho, Travis Hance, Bryan Parno, Yi Zhou (Carnegie Mellon University), Jon Howell (VMware Research) with contributions by Jay Bosamiya, Matthias Brun, Baptiste Lepers, Reto Achermann, Gerd Zellweger and others. The Verus team has been exceptionally helpful during this project, explaining Verus' features, helping with encountered issues and prioritizing certain features and bug fixes based on the needs of this project.

```
fn insert_value(&mut self, m: usize) {
  match self {
    Tree::Leaf => {
      let new_node = TreeNode {
        value: m,
        left:  Tree::Leaf,
        right: Tree::Leaf,
      };
      *self = Tree::Node(Box::new(new_node));
    },
    Tree::Node(node) => {
      if m == node.value {
        // value already exists
      } else if m < node.value {
        node.left.insert_value(m);
      } else {
        node.right.insert_value(m);
      }
    },
  }
}
```

Figure 3.1: The complete definition of `insert_value`

In this section we introduce Verus by proving correct the `contains_value` function for binary search trees, that we developed in the previous section.

For the benefit of readers who skipped that section, we reproduce the implementation in Figure 3.2.

We now use this code in the context of Verus, which assigns a *mode* to every function. `Contains_value` does not specify otherwise, so it has the default mode *exec*, meaning it is intended to be executable and can carry specifications. We will encounter the other two modes — *spec* and *proof* — in the remaining chapter.

Verifying this implementation consists of two parts. First, we write a specification to express the property we wish to prove. Second, we develop a proof that the function satisfies that specification.

```
struct TreeNode {
    value: usize,
    left:  Tree,
    right: Tree,
}

enum Tree {
    Leaf,
    Node(Box<TreeNode>),
}

impl Tree {
  fn contains_value(&self, m: usize) -> bool {
    match self {
      Tree::Leaf => {
        false
      },
      Tree::Node(node) => {
        if m == node.value {
          true
        } else if m < node.value {
          node.left.contains_value(m)
        } else {
          node.right.contains_value(m)
        }
      },
    }
  }
}
```

Figure 3.2: Unverified `contains_value` implementation

3.2.1 Specification

Verus allows us to attach a specification to executable functions in the form of Floyd-Hoare-style [25, 17] pre- and postconditions, denoted with `requires` and `ensures`. Additionally, we can name the return argument and refer to it in the postcondition.

```
fn contains_value(&self, m: usize) -> (r: bool)
  requires .. // preconditions
  ensures  .. // postconditions
{ .. }
```

For the contains function, the precondition should state that the given tree satisfies the *search tree* property and the postcondition should somehow express that the return value *actually* tells us whether or not the tree contains the value.

Verus supports *spec* functions, which we can use to write complex pre- and postconditions. A spec function is defined mostly like a normal Rust function. Its signature is prefixed with the keyword spec, it can only use a purely functional subset of Rust and we have to prove that it terminates. Spec functions are erased before the code is compiled. They do not have to be executable and can use logical constructs not available in Rust, such as universal quantification.

For example, the following spec function is a predicate that checks whether all values in a tree are less than a given value.

```
spec fn all_values_less_than(self, m: usize) -> bool
  decreases self
{
  match self {
    Tree::Leaf => {
      true
    },
    Tree::Node(node) => {
      &&& node.value < m
      &&& node.left.all_values_less_than(m)
      &&& node.right.all_values_less_than(m)
    },
  }
}
```

The `decreases` clause must evaluate to a natural number that decreases on every recursive call, to ensure termination. Using `self` in that clause denotes the tree's height. The `&&&` syntax corresponds to the logical conjunction of the values. I.e. `&&& A &&& B &&& C` means the same as `A && B && C` but the syntax is more convenient for enumerating conditions on separate lines.

Precondition

Using `all_values_less_than` and a similarly defined `all_values_greater_than` function we can now express what it means for a tree to be a search tree.

```
spec fn is_search_tree(self) -> bool
  decreases self
{
  match self {
    Tree::Leaf => true,
    Tree::Node(node) => {
      &&& node.left.all_values_less_than(node.value)
      &&& node.right.all_values_greater_than(node.value)
      &&& node.left.is_search_tree()
      &&& node.right.is_search_tree()
    },
  }
}
```

A `Leaf` is a search tree. A `Node` with value n is a search tree if all values in its left subtree are less than n, all values in its right subtree are greater than n and both subtrees are search trees themselves. The spec function is a direct translation of this natural language definition.

Now we can write `contains_value`'s precondition as `requires self.is_search_tree()`.

Postcondition

To express the postcondition we define another spec function. In verification, we often define *view* functions, that interpret a concrete data structure as a more abstract one. A search tree implements a collection of items, i.e. a set. Verus' standard library provides a `Set` type we can use.

```
spec fn view(self) -> Set<nat>
  decreases self
{
  match self {
    Tree::Leaf => set![],
    Tree::Node(node) => {
      set![node.value as nat] + node.left.view() + node.right.view()
    },
  }
}
```

If the tree is a `Leaf`, we return an empty set; If it is a `Node` we call `view` recursively on each subtree and return the union of the subtree's sets and the singleton set containing this node's value. In spec functions we typically also use more abstract integer types. Instead of the concrete `usize` we use Verus' type for natural numbers, `nat`. Consequently, we convert the value to a `nat` using Rust's `as` keyword for type-casting. The angled brackets in the return type are also a Rust feature, denoting that `Set` is a generic type, which we instantiate with the `nat` type.

Using this view function, we can now write `contains_value`'s specification, including the postcondition:

```
fn contains_value(&self, m: usize) -> (r: bool)
  requires self.is_search_tree()
  ensures  r == self@.contains(m)
```

In the postcondition we write the abbreviation `@` instead of `.view()`. View functions tend to be ubiquitous in verified code and this syntax reduces the visual clutter they introduce.

3.2.2 Proof

If we add the new pre- and postcondition to `contains_value` and run Verus on the file, it will report that it failed to prove the postcondition. Proving the postcondition involves reasoning steps that are too large for the verifier to make automatically. We can insert `assert` statements in the code, which the verifier will check in addition to the postconditions. Assertions have two purposes. We can use them to locate the reasoning steps, where the verifier fails. The assertions also help the verifier by guiding its triggering and E-matching mechanisms [40, 13]. Sometimes, asserting an intermediate reasoning step is sufficient for the verifier to figure out the remaining steps automatically.

For the current function, we start by asserting the postcondition at all exit points of the function.

```
match self {
  Tree::Leaf => {
    assert(!self@.contains(m));
    false
  },
  Tree::Node(node) => {
    if m == node.value {
      assert(self@.contains(m));
      true
    } else if m < node.value {
      let res = node.left.contains_value(m);
      assert(res == self@.contains(m));
      res
    } else {
      let res = node.right.contains_value(m);
      assert(res == self@.contains(m));
      res
    }
  },
}
```

Verus informs us that the two assertions for the recursive cases fail, while the ones for `Leaf` and for `Node` with an equal value succeed. We now prove the assertion for the recursive case on the left subtree. The other case is symmetric to this one.

```
..
} else if m < node.value {
  let res = node.left.contains_value(m);
  assert(res == self@.contains(m));
  res
}
..
```

After the recursive call, we can assume the postcondition for that subtree. Thus, we know that `res == node.left@.contains(m)` and we have to prove that `res == self@.contains(m)`.

We can perform a case distinction on `res` with a simple `if`-statement.

```
let res = node.left.contains_value(m);
if res { assert(self@.contains(m)); }
  else { assert(!self@.contains(m)); }
```

Verus is able to automatically prove the `if`-branch but not the `else`-branch.

In the else-branch we know that `node.left@` does not contain `m` and we also know that the current node is not `m`. We need to use the search tree property to prove that `node.right@` also does not contain `m`.

The precondition `self.is_search_tree()` tells us that all values in the right subtree are greater than `node.value`. Since we know that `m` is less than `node.value`, all values in the right subtree must also be greater than `m`. And if all the values are greater than `m`, the subtree cannot contain `m`.

These two reasoning steps both include an inductive step because the involved definitions are recursive. Proofs involving induction generally require proving auxiliary lemmas, i.e. functions in `proof` mode. Inductive proofs are written as recursions. We prove the second property with the following lemma.

```
proof fn lemma_search_tree_greater(self, m: usize)
  requires
    self.is_search_tree(),
    self.all_values_greater_than(m),
  ensures
    !self@.contains(m as nat),
  decreases self
{
  match self {
    Tree::Leaf => { },
    Tree::Node(node) => {
      node.left.lemma_search_tree_greater(m);
      node.right.lemma_search_tree_greater(m);
    },
  }
}
```

We prove a similar lemma for the first property and call it `lemma_transitivity`. Back in the `contains_value` function we can call these lemmas inside a `proof` block to use their postconditions.

```
..
} else if m < node.value {
  proof {
      node.right.lemma_transitivity(node.value);
      node.right.lemma_search_tree_greater(m);
  }
  let res = node.left.contains_value(m);
  assert(res == self@.contains(m));
  res
}
..
```

```
fn contains_value(&self, m: usize) -> (r: bool)
  requires self.is_search_tree()
  ensures r == self@.contains(m as nat)
{
  match self {
    Tree::Leaf => {
      false
    },
    Tree::Node(node) => {
      if m == node.value {
        true
      } else if m < node.value {
        proof {
          node.right.lemma_transitivity(node.value);
          node.right.lemma_search_tree_greater(m);
        }
        node.left.contains_value(m)
      } else {
        proof {
          node.left.lemma_transitivity(node.value);
          node.left.lemma_search_tree_less(m);
        }
        node.right.contains_value(m)
      }
    },
  }
}
```

Figure 3.3: Verified `contains_value` implementation

Now Verus is able to verify that the assertion is satisfied.

By applying the same reasoning to the remaining recursive-call branch and removing superfluous assertions, we are left with just the definition shown in Figure 3.3. The complete code, including definitions and lemmas, is available online,[2] along with instructions for running Verus to verify it.

3.2.3 A Better Proof

Using a verifier effectively requires experience and an understanding of its strengths and weaknesses. In the previous section, we had to prove two lemmas because Verus was unable to perform the inductive steps by itself.

[2]https://github.com/matthias-brun/verified-paging-for-x86-64-in-rust

In this section we use quantifiers to write better, non-recursive definitions for `all_values_less_than` and its counterpart `all_values_greater_than`. As a result, we will not need any lemmas to prove `contains_value`'s postcondition.

Recall that we previously defined `all_values_less_than` directly as a recursive function on the tree structure. Shortly thereafter we defined a `view` function that interprets the tree as a set. This view function enables us to omit the recursion in other functions such as `all_values_less_than`.

```
spec fn all_values_less_than(self, m: usize) -> bool
{ forall|x: usize| self@.contains(x) ==> x < m }

spec fn all_values_greater_than(self, m: usize) -> bool
{ forall|x: usize| self@.contains(x) ==> x > m }
```

These functions use universal quantification and implication, well-known from standard predicate logic but not supported in Rust. Verus allows us to use them in spec functions. The universal quantifier `forall` repurposes Rust's closure syntax to universally quantify the names given as arguments: `forall|(arg1: type1, ...)| body`.

The resulting definitions are much more concise and easier to reason about. The branch in which we previously called two lemmas can now be proved almost automatically. We only need to provide one assertion to guide the verifier.

```
..
} else if m < node.value {
  let res = node.left.contains_value(m);
  if !res {
    assert(!node.right@.contains(m));
  }
  res
}
..
```

The complete verified code with these alternative definitions is also available online.[3]

3.3 Verification with State Machine Refinement

Data refinement [59] is a development approach whereby the developer initially designs an "abstract" program, operating on abstract data types and then develops the concrete program by *refining* the program, i.e. replacing

[3]https://github.com/matthias-brun/verified-paging-for-x86-64-in-rust

the abstract functions and data types with concrete ones. Often multiple refinement steps are used to gradually introduce more detail.

With respect to verification, data refinement can be used to prove that a program behaves the same way as a simpler one that abstracts away a lot of detail. For example, the abstract program may use a set data structure, which in the concrete program is implemented with a tree data structure. In fact, this is precisely what we did in the previous section. The proved postcondition establishes that `contains_value` on a search tree refines the `contains` function on a set.

However, data refinement is a relation between two programs. We cannot use it to prove any properties about how the program will behave in a given environment. For that, we have to turn to *state machine refinement* [1, 32, 38], where the environment can be modeled as a state machine (also called *transition system* or *event system*).

In this project we formulate all specifications as state machines. We model the behavior of the implementation in its intended environment as a state machine and prove that it refines another state machine, which describes the desired behavior.

3.3.1 State Machines

A state machine describes the behavior of some system as a starting state followed by a possibly infinite sequence of discrete state transitions. The *system* can be almost anything. It can describe the in- and output actions of a program, a communication network or even a physical system such as a traffic light.

The core of the state machine is a set of variables. An *init* predicate describes the permissible initial states and a *next* predicate describes the permissible state transitions between two states.

```
pub open spec fn init(s: Variables) -> bool
pub open spec fn next(s1: Variables, s2: Variables) -> bool
```

The `open` attribute specifies that the definition's body is visible outside of this module.

Transition predicates are generally a combination of two types of conditions. *Enabling conditions* tell us in which starting states the transition can be executed. *Updating conditions* tell us how the resulting state relates to the starting state.

In this project we use labeled state machines, meaning that every type of transition has an associated label, which contains the transition's parameters. For example, we will later define a *page table state machine* to describe page table operations. That state machine uses the following labels:

```
pub enum PageTableStep {
  Map     { vaddr: nat, pte: PageTableEntry, result: MapResult },
  Unmap   { vaddr: nat, result: UnmapResult },
  Resolve { vaddr: nat, result: ResolveResult },
  Stutter,
}
```

A `Map` transition's parameters are thus the virtual address, the page table entry to be mapped and the operation's result. The labels serve as a standalone interface of the transitions supported by the state machine. They have a special significance in refinement proofs.

We structure the state machine around these labels by defining a separate transition predicate for each label and combining them into a `next_step` predicate, which in turn is used to define `next`.

```
pub open spec fn next_step(
 s1: PageTableVariables,
 s2: PageTableVariables,
 step: PageTableStep) -> bool {
  match step {
    PageTableStep::Map { vaddr, pte, result }
      => step_Map(s1, s2, vaddr, pte, result),
    PageTableStep::Unmap { vaddr, result }
      => step_Unmap(s1, s2, vaddr, result),
    PageTableStep::Resolve { vaddr, result }
      => step_Resolve(s1, s2, vaddr, result),
    PageTableStep::Stutter
      => step_Stutter(s1, s2),
  }
}
pub open spec fn next(
  s1: PageTableVariables,
  s2: PageTableVariables) -> bool {
    exists|step: PageTableStep| next_step(s1, s2, step)
}
```

3.3.2 State Machine Refinement

Similar to data refinement, we use state machine refinement to show that a "concrete" state machine, encoding a program's behavior, refines a more abstract state machine. The abstract state machine is the specification, i.e. the behavior we *would like* the implementation to satisfy.

Fundamentally, a state machine refinement is relatively simple. We first need to define an *abstraction function*, which maps the concrete state machine's variables to the abstract machine's. Then we prove that the concrete machine's `init` predicate implies that of the abstract machine and that any step allowed in the concrete `next` also results in a valid abstract `next` step. In the following code, we denote the abstraction function `abs`.

```
proof fn init_refinement(s: concrete::Variables)
    requires concrete::init(s)
    ensures  abstract::init(s.abs())
{ .. proof .. }

proof fn next_refinement(
  s1: concrete::Variables,
  s2: concrete::Variables)
    requires concrete::next(s1, s2)
    ensures  abstract::next(s1.abs(), s2.abs())
{ .. proof .. }
```

These two lemmas establish a state machine refinement. But it is not yet quite sufficient for our purposes. The `next_refinement` lemma does not establish any correspondence between the exact steps of the two state machines. Suppose we implement a page table `unmap` function by *mapping* some arbitrary frame instead. If that implementation's behavior is encoded in the concrete machine, we could still prove the refinement because the incorrect unmap step can refine the abstract machine's map step.

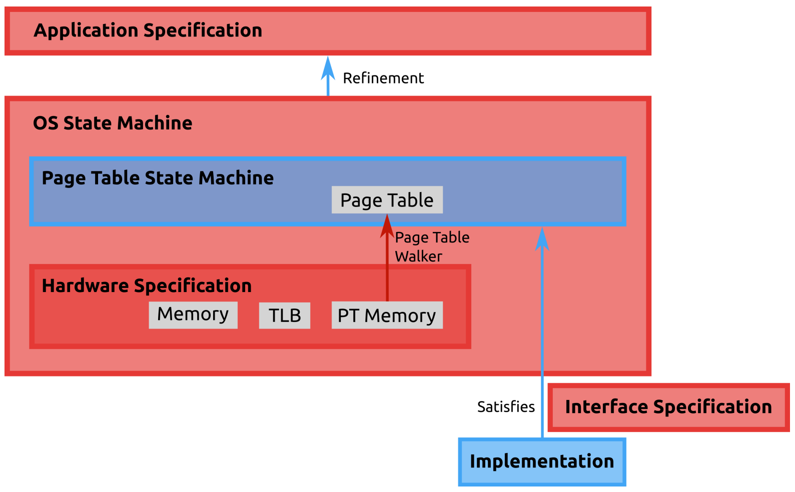
This problem is the reason we use labeled state machines. With a *label abstraction function*, we map concrete labels onto abstract ones and prove the refinement lemma with the `next_step` function instead.

```
proof fn next_refinement(
  s1: concrete::Variables,
  s2: concrete::Variables)
    requires concrete::next_step(s1, s2, step)
    ensures  abstract::next_step(s1.abs(), s2.abs(), step.abs())
{ .. proof .. }
```

This proof establishes refinement as well as the proper correspondence of transitions.

Chapter 4

Overview of the Artifact



The artifact produced in this project comprises a number of components, as visualized in the diagram above. Some of them are **trusted**, while others are **untrusted**. The red arrow denotes a trusted interpretation function, while the blue arrows denote proofs. Overall, the result is an implementation of page table functions with a specification that they are proved to satisfy.

In the following sections we will explore the purpose of each component, as well as how they fit into the bigger picture of the whole artifact. Each section header includes the Rust module name containing the component.

4.1 Application Specification `spec_t::hlspec`

To prove an implementation's correctness we need to ask what we mean by *correctness*. The application specification is a state machine encoding our answer to that question. Since our paging functions are primarily interesting in how they affect a program's view of its virtual memory, this state machine models that view. Transitions in this specification expand and reduce (map, unmap) the memory's domain and allow reading and modifying it (memory loads and stores).

The application specification is trusted in the sense that its behavior defines what we consider to be the "correct" behavior. I.e. we need to convince ourselves that it reflects what we want to prove.

This specification represents the proof target. Our implementation running in the intended environment should refine it. This is demonstrated in part by the proof that the OS state machine refines this specification.

4.2 Hardware Specification `spec_t::{hardware,mem}`

Any refinement proofs of our implementation are made with respect to its intended environment. That environment is what the hardware specification encodes in the form of a state machine.

It needs to faithfully model all relevant behavior of the x86-64 ISA. For a page table implementation this must include a model of the physical memory and the address translation mechanisms, i.e. page table walker and TLB.

This hardware specification is combined with the implementation semantics in the OS state machine.

The hardware specification is trusted. The behavior encoded in this state machine must correspond to the behavior of the actual hardware. Stated formally, the actual hardware's behavior must be a refinement of this specification.

4.3 Page Table State Machine `impl_u::spec_pt`

All our specifications are written as state machines but our implementation is imperative code that mutates memory. Ultimately, we need to establish the implementation's correspondence to transitions in the application specification. The page table state machine takes care of part of this correspondence by describing the functions' behavior as a state machine operating on an abstract view of the page table. By implementing the interface specification, the functions prove that they behave as specified by the page table state machine.

4.4 Interface Specification `spec_t::impl_spec`

For every page table function, the interface specification prescribes a pair of pre- and postconditions, which guarantee that a function implementing this specification behaves as described in the page table state machine. E.g. when implementing the interface, the `map_frame` function gets to assume the enabling conditions for the map transition. It must then prove that it modifies the memory in such a way that the resulting initial and final abstract states represent a valid map transition.

This specification is trusted, as we have to make sure that it does in fact guarantee this correspondence.

4.5 Implementation `impl_u::l2_impl`

The implementation is a set of imperative functions implementing the page table operations.

We do not trust the implementation but prove that it implements the interface specification to connect it to the page table state machine.

4.6 OS State Machine `spec_t::os`

We aim to prove that *the behavior of the implementation running in its intended environment* refines the application specification. The OS state machine encodes this behavior by combining the implementation's behavior — the page table state machine — and the intended environment — the hardware specification. The way in which these are composed introduces assumptions on how the OS behaves outside of the paging functions, another part of the environment. Modifications to the page table memory are only allowed in the paging functions.

This state machine is proved to refine the application specification, which is one of the main steps in the overall correctness proof.

The OS state machine is trusted in how it composes the implementation and hardware state machines and in the OS assumptions it introduces. Even though the map and unmap transitions are defined in terms of the page table state machine, we do not need to trust the contents of those definitions. By implementing the interface specification, the implementation proves that the page table state machine does encode its behavior.

4.7 Refinement Proof `impl_u::os_refinement`

The refinement proof establishes the fact that the OS state machine refines the application specification. It is checked by Verus and thus untrusted.

4.8 Structural Correctness

The project structure guarantees the following proof goal:

The implementation running in its intended environment refines the application specification, assuming the correctness of the trusted components.

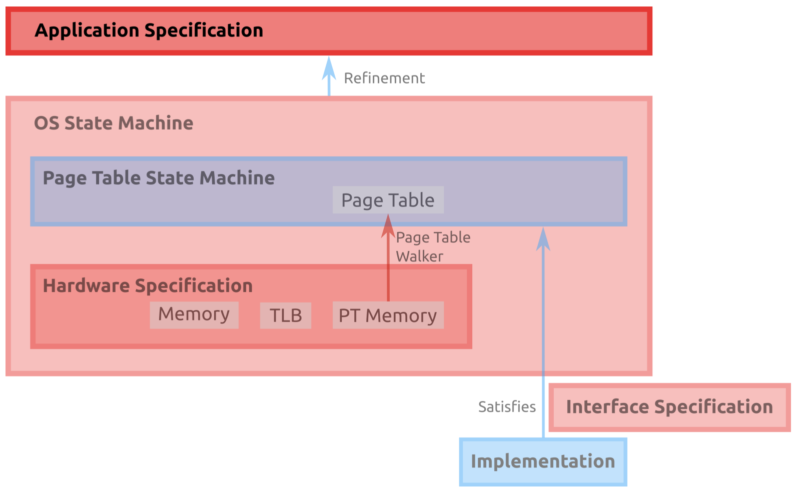
The untrusted implemented functions consist of imperative code that mutates memory. By implementing the trusted interface specification they prove their correspondence to transitions that operate on an abstract view of that memory, as defined in the untrusted page table state machine. The trusted OS state machine combines this semantics with the intended environment as modeled in the trusted hardware specification. The OS state machine is proved to refine the trusted application specification.

Part II

Specification

Chapter 5

Application Specification



The application specification serves as our proof target. All the proofs we construct have the ultimate purpose of enabling us to prove that the implementation, when run in its intended environment, refines this specification. Hence, our goals for the specification must be primarily informed by our goals for the overall project. The specification's behavior, described by a state machine, must allow us to conclude any properties we wish to prove about the implementation.

This is also the extent to which this specification is trusted. Trusting the specification means being convinced that it entails the desired properties.

It is not even necessary that we are confident an implementation exists which can satisfy the specification, as a proof of application correspondence for an implementation guarantees that it *does* satisfy the specification.

As a secondary goal, the specification should be suitable for use in the verification of a larger system that builds on top of this project, such as a verified database software.

5.1 State Machine Design

Our stated goal is to prove the functional correctness of the page table functions. The behavior of these functions is interesting insofar as it affects a user program's view of their virtual memory. Thus, a specification focusing on that view and the user program's interactions with it is adequate to achieve our goal.

Taken at face value, such a specification does not guarantee that the functions also correctly manage the kernel's virtual memory. However, for the purpose of this specification, the difference between the kernel and a user-space program are small. Mapping or unmapping memory in the kernel's address space can change the page table functions' semantics by either affecting the contents of the page table or by affecting the memory storing the functions' instructions. A kernel avoiding these behaviors looks much the same as an ordinary user-space program and can also benefit from the guarantees of this specification.

The specification's state must contain a representation of the program's virtual memory, i.e. a map from virtual addresses to machine words.

The relevant actions a program may then take are:

1. Loading a word from memory
2. Storing a word to memory
3. Mapping memory
4. Unmapping memory
5. Resolving a virtual address to a physical one

These actions correspond to the transitions that the state machine may take. We include a resolve operation to look up a mapping in the page table, since this function is used in NrOS.

5.2 Concrete State Machine

5.2.1 Variables

The type of the state machine's variables is shown in Figure 5.1.

The specification revolves around a representation of the virtual memory, the map `mem`, mapping virtual addresses to machine words, both of which we represent as natural numbers.

```
pub struct AbstractConstants {
  pub phys_mem_size: nat,
}
pub struct AbstractVariables {
  pub mem:      Map<nat,nat>,
  pub mappings: Map<nat,PageTableEntry>,
}
```

Figure 5.1: The application specification's variables

```
pub enum AbstractStep {
  ReadWrite { vaddr: nat, op: RWOp,
              pte: Option<(nat, PageTableEntry)> },
  Map       { vaddr: nat, pte: PageTableEntry,
              result: MapResult },
  Unmap     { vaddr: nat, result: UnmapResult },
  Resolve   { vaddr: nat, result: ResolveResult },
  Stutter,
}
```

Figure 5.2: The application specification's state machine transition labels

When specifying the transitions, we add *enabling conditions*, which must be satisfied in the initial state for the transition to take place. E.g. after mapping a 2 MiB huge page at address 0, a subsequent `Unmap` transition should be able to unmap the entire region but it should not be able to unmap just the first 4 KiB. Expressing such a condition requires keeping a record of the page table's state, which we do in the `mappings` variable.

The variables also include a constant. A program may map a physical memory location that does not exist. To ensure that memory accesses to such a location result in a page fault, we use `phys_mem_size` to limit the size of the physical memory.

Both `mem` and `mappings` are empty in the initial state.

5.2.2 Transitions

The state machine's transitions and their parameters are shown in Figure 5.2. We discuss the types together with the transitions that use them. Reading from and writing to memory shares a significant amount of logic. Consequently, we combine these operations in a single transition.

We suggested earlier, that the transitions represent actions taken by the program. However, they have a more general meaning. A memory write transition may represent a memory write by the program but it could also represent input from a device that communicates with the program via memory-mapped I/O or DMA. In reality, the device operates on a different address space than the program, either physical or virtualized through an IOMMU. However, if the memory is mapped in the program's address space, the `ReadWrite` transition expresses how that action changes the user program's view of its memory. Where an observed change in the memory originated is irrelevant for the specification.

When transitions do originate from the user program, the transitions' parameters are in fact just parameters, not input arguments. For example, the Map transition has a `PageTableEntry` parameter, which would suggest that the program gets to choose the physical memory region that is mapped. In the case of NrOS, the kernel API only allows user-space programs to specify the *virtual* memory region to be mapped, except for a special mechanism for mapping device memory. The selection of which physical memory to back it with remains at the kernel's discretion. It is not important that the transitions mirror any real interface but rather that any actions that may happen can be described with these transitions.

ReadWrite

```
ReadWrite { vaddr: nat, op: RWOp,
            pte: Option<(nat, PageTableEntry)> }

pub enum RWOp {
    Store { new_value: nat, result: StoreResult },
    Load { is_exec: bool, result: LoadResult },
}
pub enum StoreResult {              pub enum LoadResult {
    Pagefault,                          Pagefault,
    Ok,                                 Value(nat),
}                                   }
```

The ReadWrite transition allows reads from and writes to the virtual memory. Because they share a significant amount of logic, we use just a single transition to represent both.

The `RWOp` parameter carries information specific to the particular operation. Store operations have a new value to be stored and their result is either successful or a page fault. Load operations have a flag indicating whether the load is to fetch instructions and the result can either be a value or a page fault.

The general parameters are `vaddr` and `pte`. These are the virtual address to which the memory access is made and the relevant page table mapping, respectively. We require that the virtual address is word-aligned (but note that unaligned accesses can be represented by two successive aligned accesses). The page table mapping is wrapped in an `Option` type to indicate either the presence or the absence of a mapping for the given virtual address. If a mapping is given, it must be present in the page table. If none is given, the page table must not contain a mapping that can translate `vaddr` and the result of the operation must be a page fault.

If a mapping does exist, the result of the operation depends on the mapped region's flags and whether or not the translated physical address is larger than the available physical memory. We assume that the valid physical address space ranges from zero to some upper bound.

The page table mappings and the memory remain unchanged except in the case of a successful store operation.

The full definition implementing these constraints is shown in Figure 5.3.

```
pub open spec fn step_ReadWrite(
  c: AbstractConstants,
  s1: AbstractVariables, s2: AbstractVariables,
  vaddr: nat, op: RWOp, pte: Option<(nat, PageTableEntry)>)
  -> bool
{
  let vmem_idx = vaddr / 8;
  &&& aligned(vaddr, 8)
  &&& s2.mappings === s1.mappings
  &&& match pte {
    Some((base, pte)) => {
      &&& s1.mappings.contains_pair(base, pte)
      &&& between(vaddr, base, base + pte.frame.size)
      let paddr = (pte.frame.base + (vaddr - base)) as nat;
      let pmem_idx = paddr / 8;
      &&& match op {
        RWOp::Store { new_value, result } => {
          if pmem_idx < c.phys_mem_size &&
             !pte.flags.is_supervisor && pte.flags.is_writable {
            &&& result.is_Ok()
            &&& s2.mem === s1.mem.insert(vmem_idx, new_value)
          } else {
            &&& result.is_Pagefault()
            &&& s2.mem === s1.mem
          }
        },
        RWOp::Load { is_exec, result } => {
          &&& s2.mem === s1.mem
          &&& if pmem_idx < c.phys_mem_size &&
                 !pte.flags.is_supervisor &&
                 (is_exec ==> !pte.flags.disable_execute) {
            &&& result.is_Value()
            &&& result.get_Value_0() == s1.mem.index(vmem_idx)
          } else {
            &&& result.is_Pagefault()
          }
        },
      }
    },
    None => {
      &&& !mem_domain_from_mappings(c.phys_mem_size, s1.mappings)
            .contains(vmem_idx)
      &&& s2.mem === s1.mem
      &&& match op {
        RWOp::Store { new_value, result } => result.is_Pagefault(),
        RWOp::Load { is_exec, result }  => result.is_Pagefault(),
      }
    },
  }
}
```

Figure 5.3: The ReadWrite transition

Map

```
Map { vaddr: nat, pte: PageTableEntry, result: MapResult },
```

```
pub struct PageTableEntry {
  pub frame: MemRegion,
  pub flags: Flags,
}
```

```
pub enum MapResult {
  ErrOverlap,
  Ok,
}
```

The Map transition maps a new page table entry at a given virtual address or fails with an error. It has a small number of enabling conditions that must be satisfied for the transition to take place. Most of these are simple checks that can (and must!) be ensured by the caller, i.e. the kernel. They include alignment of the virtual and physical base addresses with the frame size, a check that the mapping is in bounds of the virtual address space and a check that the size is either the page size (4KiB) or one of the valid huge page sizes (2MiB, 1GiB).

The only more complicated enabling condition is that we cannot map any physical memory in multiple locations in the virtual address space. In other words, we disallow synonymous mappings. With synonyms, a memory write could change the `mem` map at several distinct addresses, thus making the `ReadWrite` transition more complex. The benefits are questionable. We can easily argue the correctness of the implemented functions with the simpler specification. In future iterations of this specification we may eventually want to allow synonyms to make it more useful as a component of larger verified systems.

These conditions guarantee that the only possible error is the existence of another mapping that would overlap the new one. In Section 5.3 we imagine an alternative application specification in which enabling conditions are replaced by errors.

The definition of the map-transition itself is then relatively straightforward. If an overlapping mapping exists, the result must be an error and the state unchanged. Otherwise the new mapping is inserted. Since the memory is a map, its domain changes to account for the changed page table but its contents are unchanged for anything that was mapped already. The mappings in the page table serve as the source of truth for what the memory's domain is. Figure 5.4 shows the full definition.

```
pub open spec fn step_Map_enabled(
  map: Map<nat,PageTableEntry>,
  vaddr: nat,
  pte: PageTableEntry) -> bool {
    &&& aligned(vaddr, pte.frame.size)
    &&& aligned(pte.frame.base, pte.frame.size)
    &&& candidate_mapping_in_bounds(vaddr, pte)
    &&& {
        ||| pte.frame.size == L3_ENTRY_SIZE
        ||| pte.frame.size == L2_ENTRY_SIZE
        ||| pte.frame.size == L1_ENTRY_SIZE
    }
    &&& !candidate_mapping_overlaps_existing_pmem(
          map, vaddr, pte)
}

pub open spec fn step_Map(
  c: AbstractConstants,
  s1: AbstractVariables,
  s2: AbstractVariables,
  vaddr: nat,
  pte: PageTableEntry,
  result: MapResult) -> bool
{
  &&& step_Map_enabled(s1.mappings, vaddr, pte)
  &&& if candidate_mapping_overlaps_existing_vmem(
          s1.mappings, vaddr, pte)
  {
    &&& result.is_ErrOverlap()
    &&& s2.mappings === s1.mappings
    &&& s2.mem === s1.mem
  } else {
    &&& result.is_Ok()
    &&& s2.mappings === s1.mappings.insert(vaddr, pte)
    &&& (forall|idx:nat| s1.mem.dom().contains(idx)
                  ==> s2.mem[idx] === s1.mem[idx])
    &&& s2.mem.dom() === mem_domain_from_mappings(
                         c.phys_mem_size, s2.mappings)
  }
}
```

Figure 5.4: The `Map` transition and its enabling conditions

```
pub open spec fn step_Unmap_enabled(vaddr: nat) -> bool {
  &&& between(vaddr, PT_BOUND_LOW, PT_BOUND_HIGH)
  &&& {
    ||| aligned(vaddr, L3_ENTRY_SIZE)
    ||| aligned(vaddr, L2_ENTRY_SIZE)
    ||| aligned(vaddr, L1_ENTRY_SIZE)
  }
}

pub open spec fn step_Unmap(
  c: AbstractConstants,
  s1: AbstractVariables, s2: AbstractVariables,
  vaddr: nat, result: UnmapResult) -> bool
{
  &&& step_Unmap_enabled(vaddr)
  &&& if s1.mappings.dom().contains(vaddr) {
    &&& result.is_Ok()
    &&& s2.mappings === s1.mappings.remove(vaddr)
    &&& s2.mem.dom() === mem_domain_from_mappings(
                          c.phys_mem_size, s2.mappings)
    &&& (forall|idx:nat| s2.mem.dom().contains(idx)
                      ==> s2.mem[idx] === s1.mem[idx])
  } else {
    &&& result.is_ErrNoSuchMapping()
    &&& s2.mappings === s1.mappings
    &&& s2.mem === s1.mem
  }
}
```

Figure 5.5: The `Unmap` transition and its enabling conditions

Unmap

```
Unmap { vaddr: nat, result: UnmapResult },

pub enum UnmapResult {
  ErrNoSuchMapping,
  Ok,
}
```

The `Unmap` transition is defined analogously to `Map`. It requires its enabling conditions to be satisfied. The only possible error is that no corresponding mapping exists. If the mapping does exist, it is removed. The memory's domain changes accordingly and the contents of still mapped memory locations remain unchanged. The full definition is shown in Figure 5.5.

```
pub open spec fn step_Resolve_enabled(vaddr: nat) -> bool {
  &&& aligned(vaddr, 8)
}

pub open spec fn step_Resolve(
  c: AbstractConstants,
  s1: AbstractVariables, s2: AbstractVariables,
  vaddr: nat, result: ResolveResult) -> bool {
  &&& step_Resolve_enabled(vaddr)
  &&& s2 === s1
  &&& match result {
    ResolveResult::Ok(base, pte) => {
      &&& s1.mappings.contains_pair(base, pte)
      &&& between(vaddr, base, base + pte.frame.size)
    },
    ResolveResult::ErrUnmapped => {
      let vmem_idx = word_index_spec(vaddr);
      &&& !mem_domain_from_mappings(
            c.phys_mem_size, s1.mappings)
              .contains(vmem_idx)
    },
  }
}
```

Figure 5.6: The `Resolve` transition and its enabling conditions

Resolve

```
Resolve { vaddr: nat, result: ResolveResult },

pub enum ResolveResult {
  ErrUnmapped,
  Ok(nat, PageTableEntry),
}
```

Resolving a virtual address does not modify the page table, so the corresponding transition is read-only and does not modify the state. If there is a mapping in the page table containing the given address, the result should be that mapping, i.e. a `(nat, PageTableEntry)` pair. Otherwise, the result is an error. We use a custom result type `ResolveResult` as for the other operations. The full definition is shown in Figure 5.6.

Stutter

The stutter step always allows a transition where nothing changes:

```
pub open spec fn step_Stutter(
  c: AbstractConstants,
  s1: AbstractVariables, s2: AbstractVariables) -> bool
{
  s1 === s2
}
```

This allows an implementation to take additional "internal" steps. An example of one such step is the caching of a page table entry in the TLB. Since this specification has no concept of the TLB, that step does not lead to any visible change and therefore should refine the stutter step.

5.3 Simplifying the Application Specification

A user space program's virtual memory is the centerpiece of the application specification. In a perfect world that would be the entire state. We do not care about the internal state of the page table, only that the OS, using our implementation, is able to supply a working virtual memory. We include a view of the page table only to express enabling conditions for some of the transitions. The page table is arguably an implementation detail of the lower levels, leaking into the application specification. In future work, we may be able to improve this specification.

Without knowledge of the page table's state, we have to weaken the enabling conditions for the transitions, possibly using *always-enabled* transitions, similar to IronFleet [22]. This transformation is a straightforward task but without further work it weakens the specification significantly.

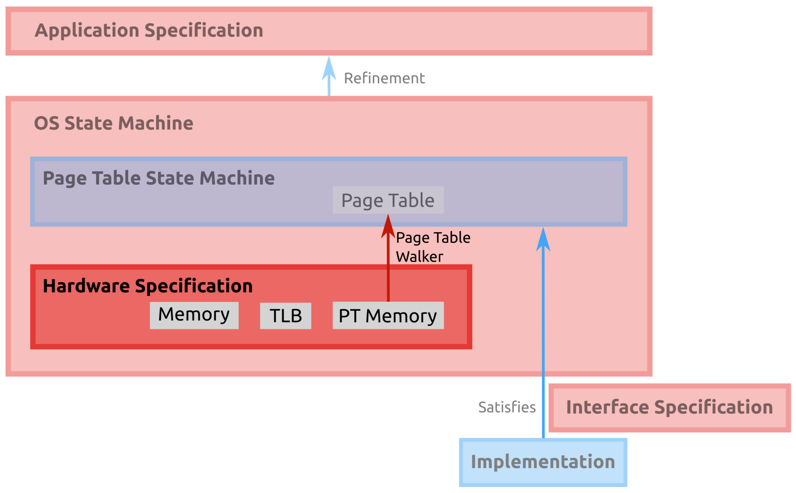
An implementation may fail when these conditions are not satisfied, so we have to allow it to return different errors, e.g. `ErrPartialUnmap`, when attempting to unmap part of a huge page. But now `return ErrPartialUnmap` is a valid implementation of `Unmap`. To prevent this, we have to explicitly specify when any particular error is allowed. Without the page table state, we cannot express these conditions in the application specification. Instead, we have to introduce these conditions in a lower level state machine, where we do have knowledge of the page table.

The result is a simpler application specification, which is easier to inspect. However, its correctness now also depends in part on error conditions in a *different* specification being sufficiently constrained.

Only future exploration will show where exactly the sweet spot in this tradeoff lies.

Chapter 6

Hardware Specification



A significant complication in verifying (parts of) an operating system is the necessity of an accurate but sufficiently abstract formal specification for the targeted hardware. Manufacturers tend to only publish informal, English-language specifications such as Intel Corporation's Software Development Manual [10].

In this chapter we develop the *hardware specification*, a formal specification of our target ISA x86-64, expressed as a state machine. We deliberately keep the specification as simple as we can. We model in as much detail as necessary those components that are important to the correctness of our page table implementation. This includes for example the physical memory and the translation lookaside buffer. However, we do not model any components that have no direct impact on our implementation's correctness. Our hardware specification has no concept of data caches and many other processor implementation details.

For this specification to be "correct", the target hardware must be a refinement of the specification.

6.1 State Machine Design

The physical memory as well as address translation mechanisms — page table walker and translation lookaside buffer (cf. Section 2.1.3) — are the primary interesting components when it comes to verifying our paging implementation.

With respect to paging, the memory plays two roles. Physical memory regions are what is mapped by the page table but simultaneously the page table itself is stored in physical memory.

For the correct operation of our page table functions we need to make assumptions on the behavior of the operating system's unverified parts. One assumption is that it does not modify the page table memory other than through our verified implementation. To ensure this, the OS must not (writably) map any of the page table memory in user-space programs' virtual address space.

Instead of encoding this assumption as an axiom, we encode it in the specification's structure by making the page table memory and the application-mappable memory distinct (and disjoint) components of the state. Given the assumption stated above, the more accurate single-memory model is a refinement of this simplified view.

The page table walker semantics is implemented as a function that interprets the page table memory as an abstract page table: `Map<nat,PageTableEntry>`.

The same type is used to model the translation lookaside buffer (*TLB*), which is the final component in the state machine's state.

The state machine's transitions then are the ones affecting any of these components:

- Filling a page table entry into the TLB
- Evicting an entry from the TLB
- Modifying the page table memory
- Reading from or writing to application memory

6.2 Memory Semantics

An especially tricky part of correctly specifying a hardware architecture is the memory semantics. Often, the guarantees provided by the ISA are relatively weak and can cause counter-intuitive behavior. However, in the single-processor case, much of this complex behavior does not matter very much. Even on a multiprocessor system, the memory semantics only comes into play when two concurrent threads of execution access shared memory.

For the purpose of modeling the application memory (`mem`) on a uniprocessor, we can make the much stronger assumption that the memory behaves like a map. Stored values immediately overwrite any earlier values and any future loads are guaranteed to read that new value. There are no concurrent accesses to the application memory.

However, concurrent accesses do matter in the page table memory. While the page table functions cannot interfere with themselves, we cannot prevent the hardware page table walker from concurrently accessing the page table memory. We need to take into account the memory semantics when considering that possibility.

6.3 Concrete State Machine

With these considerations of the state machine design and memory semantics in mind, we can now understand the concrete state machine's variables and transitions.

6.3.1 Variables

```
pub struct HWVariables {
    pub mem:    Seq<nat>,
    pub pt_mem: mem::PageTableMemory,
    pub tlb:    Map<nat,PageTableEntry>,
}
```

The variables contain the application-mappable memory `mem`, the page table memory `pt_mem` and the TLB `tlb`. We will discuss the page table memory's type in Section 6.4. To define the transitions, we only need its interpretation of type `Map<nat,PageTableEntry>`, as defined by the page table walker (discussed in Section 6.5) function `interp_pt_mem`.

The TLB and the page table interpretation of `pt_mem` are empty maps in the initial state.

6.3.2 Transitions

```
pub enum HWStep {
    ReadWrite { vaddr: nat, paddr: nat, op: RWOp,
                pte: Option<(nat, PageTableEntry)> },
    PTMemOp,
    TLBFill  { vaddr: nat, pte: PageTableEntry },
    TLBEvict { vaddr: nat},
}
```

ReadWrite

The `ReadWrite` transition is very similar to the one in the application specification, so we do not discuss it in detail here but focus on the differences.

Instead of directly looking up a value with the virtual address, the value is looked up in physical memory with the physical address `paddr`. As in the application specification, `pte`'s option type encodes the presence or absence of a suitable page table entry. However, in the hardware specification, we require its *presence in the TLB* or its *absence in the page table*. A successful memory access corresponds to a `TLBFill` transition followed by a `ReadWrite` transition.

PTMemOp

The *PTMemOp* transition allows us to modify the page table memory. We could model this in much the same way as the *ReadWrite* transition. Page table modifications are indeed made by writing to virtual memory. However, such a fully detailed model is immensely complex. Instead, we choose to pretend that modifications of the page table memory behave like accesses directly to physical memory, meaning there is no address translation or TLB caching involved. This model is still correct. Early in the kernel's initialization, a linear mapping of the physical memory is created and not modified at any later point. Hence, the address translation is fairly simple and there is never a risk of accessing incorrect locations due to not invalidating TLB entries.

As a result, `PTMemOp` is much simpler than the `ReadWrite` transition. We simply allow the page table memory to change arbitrarily. We also allow TLB entries to be evicted during this transition. Our implemented functions refine this step's counterpart in the OS state machine and to unmap an entry we also need to evict it from the TLB.

```
pub open spec fn step_PTMemOp(
  s1: HWVariables,
  s2: HWVariables) -> bool
{
  &&& s2.mem === s1.mem
  &&& forall|base: nat, pte: PageTableEntry|
        s2.tlb.contains_pair(base, pte)
          ==> s1.tlb.contains_pair(base, pte)
  // pt_mem may change arbitrarily
}
```

While the page table memory may change arbitrarily in this transition, the OS state machine restricts the `PTMemOp` transition to the semantics of the actual implemented functions (cf. Section 7.2.2).

TLBFill and TLBEvict

Few things can happen to the TLB. It can cache new address translations from the page table or it can evict entries that it currently has cached. The `TLBFill` and `TLBEvict` transitions reflect these two possibilities. The `tlb` field may change accordingly, while `pt_mem` and `mem` must stay unchanged.

```
pub open spec fn step_TLBFill(
  s1: HWVariables, s2: HWVariables,
  vaddr: nat, pte: PageTableEntry) -> bool
{
  &&& interp_pt_mem(s1.pt_mem).contains_pair(vaddr, pte)
  &&& s2.tlb === s1.tlb.insert(vaddr, pte)
  &&& s2.pt_mem === s1.pt_mem
  &&& s2.mem === s1.mem
}

pub open spec fn step_TLBEvict(
  s1: HWVariables, s2: HWVariables,
  vaddr: nat) -> bool
{
  &&& s1.tlb.dom().contains(vaddr)
  &&& s2.tlb === s1.tlb.remove(vaddr)
  &&& s2.pt_mem === s1.pt_mem
  &&& s2.mem === s1.mem
}
```

Note the assumption that the TLB always caches *full* page table entries. In practice, a processor may implement translation caching for huge pages by caching multiple smaller, fictional entries. However, invalidating any address within the region causes the processor to invalidate all these entries, as if it was a single large one [10, Section 4.10.2.3]. Hence, our specified behavior is a safe approximation.

6.4 Page Table Memory

While the application-mappable memory only exists as an abstract view of type `Seq<nat>`, the page table memory is a component which we need to access in our implementation. As a result, the page table memory needs to have a concrete interface for executable code. That interface is defined in the struct `PageTableMemory`. The state machine's transitions are defined in terms of the page table walker's interpretation of that struct.

The struct allows the implementation to access physical memory to read or modify the page table. A view function on the struct gives us an abstract view in terms of which we can define the page table walk semantics. The executable functions are axiomatized in terms of this view, e.g. writing a new value to an address changes the view accordingly.

To access a physical address, we offset from a reference to the beginning of a linear mapping of the physical memory:

```
unsafe { phys_mem_mapping_ptr.offset(word_offset).read() }
```

We use these `unsafe` blocks in the `read` and `write` methods. Rust's type system helps us ensure that the implementation cannot use these methods to access offsets from any other references than the linear mapping. The reference is contained in a private struct field, which makes it impossible for Rust code in another module to construct its own instance of it. Instead, the implementation will take an instance of the struct as an argument and use that to access memory.

```
#[verifier(external_body)]
pub struct PageTableMemory {
    phys_mem_ref: *mut u64,
}
```

The `external_body` atribute marks the struct as trusted. Verus ignores the struct's fields and trusts its values to be coherent with the function specifications.

We define the abstract view of the memory as two unspecified functions.

```
pub spec fn regions(self) -> Set<MemRegion>;
pub spec fn region_view(self, r: MemRegion) -> Seq<u64>;
```

`regions` returns the set of memory regions that are "managed" by this struct, i.e. those regions that the read and write functions allow us to access. `region_view` returns the contents of a region. This is the *intended* semantics for these functions, which we achieve by axiomatizing the other functions accordingly.

The `write` function has the following signature:

```
pub fn write(&mut self, paddr: usize, region: Ghost<MemRegion>,
              value: u64);
```

It mutably borrows the memory struct, takes a physical address, a value to write and a memory region. The memory region is a ghost argument, i.e. is erased from the final code. We can use the postfix `@` operator to unwrap the ghost argument and obtain the contained value. We use the ghost argument to express the function's preconditions:

```
requires
  aligned(paddr, 8),
  old(self).regions().contains(region@),
  region@.contains(paddr),
  (paddr - region@.base) / 8 < 512,
  old(self).inv(),
```

The physical address must be aligned, the provided region must be in the memory's `regions()` and the region must contain the physical address. We assume that the regions are always page-sized. Additionally, the memory maintains the invariant that the regions in its domain do not overlap. This invariant is also a precondition.

The implementation is straightforward; We simply write the value to the given address:

```
unsafe { self.phys_mem_ref.offset(paddr / 8).write(value); }
```

The postconditions express what has changed in terms of the view. The `regions()` set remains the same, the region containing `paddr` has the expected location changed to the provided value and other regions are unchanged:

```
ensures
  self.region_view(region@)
    === old(self).region_view(region@)
                 .update((paddr - region@.base) / 8, value),
  forall|r: MemRegion| r !== region@
    ==> self.region_view(r) === old(self).region_view(r),
  self.regions() === old(self).regions(),
```

The `read` function is axiomatized similarly. Both definitions are shown in Figure 6.1. The `#[verifier(external_body)]` annotation indicates to Verus that these functions are trusted and should not be checked.

```
impl PageTableMemory {
  #[verifier(external_body)]
  pub fn read(
      &self,
      paddr: usize,
      region: Ghost<MemRegion>) -> (res: u64)
    requires
      aligned(paddr, 8),
      self.regions().contains(region@),
      (paddr - region@.base) / 8 < 512,
    ensures
      res == self.region_view(region@)[(paddr - region@.base) / 8]
  {
    unsafe { self.phys_mem_ref.offset(paddr / 8).read() }
  }

  #[verifier(external_body)]
  pub fn write(
      &mut self,
      paddr: usize,
      region: Ghost<MemRegion>,
      value: u64)
    requires
      aligned(paddr, 8),
      old(self).inv(),
      old(self).regions().contains(region@),
      region@.contains(paddr),
      (paddr - region@.base) / 8 < 512,
    ensures
      self.region_view(region@)
        === old(self).region_view(region@)
                     .update((paddr - region@.base) / 8, value),
      forall|r: MemRegion| r !== region@
        ==> self.region_view(r) === old(self).region_view(r),
      self.regions() === old(self).regions(),
  {
    unsafe { self.phys_mem_ref.offset(paddr / 8).write(value); }
  }
}
```

Figure 6.1: `PageTableMemory` read and write functions

The `PageTableMemory` struct has two more executable functions, whose implementations must be provided by the user (i.e. the kernel that uses our paging functions).

The `cr3` function returns the memory region containing the root page table.

```
#[verifier(external_body)]
pub fn cr3(&self) -> (res: MemRegionExec)
  ensures res === self.cr3_spec()
{ unreached() }
```

`MemRegionExec` is similar to the `MemRegion` struct but it uses `usize` instead of natural numbers, i.e. it can appear in executable code.

```
pub struct MemRegionExec { pub base: usize, pub size: usize }
```

The `cr3_spec` function returns the same values in spec mode and the (trusted) lemma `cr3_facts` lets us assume that the memory region is page-sized and page-aligned. These assumptions need to be ensured by the operating system.

```
pub open spec fn cr3_spec(&self) -> MemRegionExec;

#[verifier(external_body)]
pub proof fn cr3_facts(&self)
  ensures
    aligned(self.cr3_spec().base, PAGE_SIZE),
    self.cr3_spec().size == PAGE_SIZE;
```

The other remaining function `alloc_page` allocates one page of memory to use when creating new page directories. We assume that this function never fails and axiomatize it accordingly. In practice, the allocator may run out of memory and fail. We can easily implement the infallible allocation function in terms of this allocator. Only `map_frame` uses the allocator. It never needs to allocate more than three pages in one invocation, since at most one new directory is needed per level. Thus, it is sufficient to keep a buffer of three pages before calling `map_frame`. In the future, we intend to make this argument formal and directly allow a fallible allocator.

```
#[verifier(external_body)]
pub fn alloc_page(&mut self) -> (r: MemRegionExec)
  requires
    old(self).inv()
  ensures
    r@.size == PAGE_SIZE,
    r@.base + PAGE_SIZE <= MAXPHYADDR,
    aligned(r@.base, PAGE_SIZE),
    !old(self).regions().contains(r@),
    self.regions() === old(self).regions().insert(r@),
    self.region_view(r@) === new_seq::<u64>(512nat, 0u64),
    forall|r2: MemRegion| r2 !== r@
      ==> self.region_view(r2) === old(self).region_view(r2),
    self.inv()
{
  unreached()
}
```

MemRegionExec defines a view() function, which allows us to convert it to a MemRegion with the @ operator.

6.5 MMU Page Table Walk

The MMU translates addresses by performing a page table walk to look up the right entry. In a specification we can think of its semantics as instantaneously constructing an abstract map from virtual addresses to page table entries:

```
pub open spec fn interp_pt_mem(pt_mem: mem::PageTableMemory)
  -> Map<nat, PageTableEntry>;
```

Actually writing an implementation of such a function is somewhat involved. Since we already implement this semantics in the implementation as part of the three-level refinement (cf. Section 11.1), we reuse that definition. We axiomatize interp_pt_mem to have the same meaning:

```
#[verifier(external_body)]
pub proof fn axiom_page_table_walk_interp()
  ensures forall|pt: PageTable|
    pt.inv() ==>
      pt.interp().interp().map === interp_pt_mem(pt.memory);
```

While this approach saves some work, the drawbacks are quite significant. The size of the trusted code base (*TCB*) is increased considerably. The interpretation function consists of more than 200 lines of code.

Additionally, the interpretation is only defined when the implementation invariant is satisfied. We do prove that the functions re-establish the invariant but intermediate states do not necessarily satisfy the invariant. Our current interface specification does not allow interleavings where the MMU could observe these states (cf. Section 8.3). To obtain stronger correctness guarantees, we will want to allow these interleavings as well in the future.

Replacing the current definition with a smaller-TCB one in the future is reasonably easy. The implementation interpretation's complexity mainly stems from the fact that it is constructive. It is a recursive function, interpreting the memory as a page table. An alternative, simpler way of specifying the semantics of a page table walk is to write it as a predicate:

```
pub spec fn valid_pt_walk(
  pt_mem_view: Seq<nat>, cr3: nat,
  addr: u64, pte: PageTableEntry) -> bool
{
  let l0_idx = l0_bits(addr);
  let l1_idx = l1_bits(addr);
  [...]
  let l0_entry = pt_mem_view[cr3][l0_idx];
  let l1_pd_addr = pd_addr(l0_entry);
  let l1_entry = pt_mem_view[l1_pd_addr][l1_idx];
  [...]
  let l3_entry = pt_mem_view[l3_pd_addr][l3_idx];
  [...]
  pte_interp(l3_entry) == pte
}
```

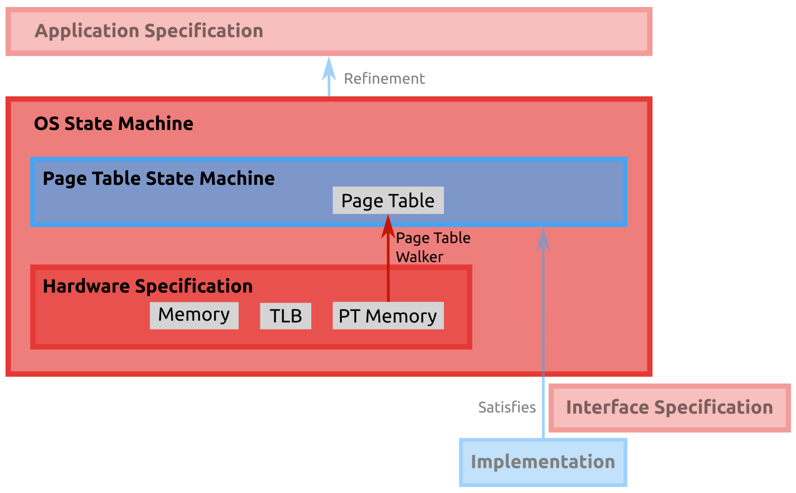
The real predicate, accounting for details such as huge pages and page directory flags, would be more complex but still far simpler than a direct, constructive definition. Because this interpretation does not need to be executable, we can then still construct a map via Hilbert choice:

```
pub open spec fn interp_pt_mem(pt_mem: mem::PageTableMemory)
  -> Map<nat, PageTableEntry>
{
  Map::new(
    |addr: nat|
      exists|pte: PageTableEntry|
        valid_pt_walk(pt_mem@, mem.cr3_spec().0, addr, pte),
    |addr: nat|
      choose|pte: PageTableEntry|
        valid_pt_walk(pt_mem@, mem.cr3_spec().0, addr, pte))
}
```

The first argument to the `Map::new` function is a predicate determining its domain, while the second one returns values for any arguments in the domain.

Chapter 7

OS State Machine



We aim to show that the implementation, when running in its intended environment, refines the application specification. The OS state machine describes the *implementation, when running in its intended environment* part of that sentence. The hardware specification is one part of the environment. The OS state machine represents the rest of it, unverified parts of the OS, and integrates it with the implementation's behavior.

7.1 State Machine Design

The goal is to construct a state machine that uses definitions from both the hardware specification's state machine and the page table state machine, which we have yet to discuss in Chapter 9. The page table state machine is an untrusted part of the implementation. We refer to its transitions in such a way that only if their semantics is "correct", do they enable us to complete the refinement proof in Chapter 10.

The page table state machine's transitions are defined on variables containing just an abstract page table map. To integrate them with the hardware specification, we instantiate that map with the page table walker's interpretation of the hardware's page table memory. The OS state machine's variables only contain the hardware specification variables.

Making assumptions on the rest of the OS does not require an explicit model of it. We assume that the remaining OS does not modify the page table memory, except through our implementation. This is encoded by only allowing the page table memory to change during transitions defined in terms of the page table state machine. We also assume that the OS takes care of TLB invalidation. This is encoded as an additional condition in the `step_Unmap` transition.

7.2 Concrete State Machine

7.2.1 Variables

```
pub struct OSVariables {
  pub hw: hardware::HWVariables,
}
```

The variables are exactly the hardware specification's variables, as discussed. To more easily refer to transitions in the page table state machine we define a method that constructs the corresponding page table state machine variables, using the page table walker's interpretation.

```
impl OSVariables {
  pub open spec fn pt_variables(self) -> spec_pt::PageTableVariables {
    spec_pt::PageTableVariables {
      map: hardware::interp_pt_mem(self.hw.pt_mem),
    }
  }
}
```

Note that this definition only uses the page table memory but not the application memory or the TLB.

7.2.2 Transitions

```
pub enum OSStep {
  Map     { vaddr: nat, pte: PageTableEntry, result: MapResult },
  Unmap   { vaddr: nat, result: UnmapResult },
  Resolve { vaddr: nat, result: ResolveResult },
  HW      { step: hardware::HWStep },
}
```

We compose the hardware state machine with the page table state machine. Each transition is a combination of one step in each state machine and possible additional conditions. We define explicit transitions for the page table operations. Each of these corresponds to a PTMemOp transition in the hardware state machine. In the HW transition we allow any of the other hardware transitions, which correspond to a stutter step in the page table state machine.

Since we rely on the other state machines, the definitions of these transitions are very succinct.

Map

```
pub open spec fn step_Map(
  s1: OSVariables, s2: OSVariables,
  base: nat, pte: PageTableEntry, result: MapResult) -> bool {
    &&& hardware::step_PTMemOp(s1.hw, s2.hw)
    &&& spec_pt::step_Map(
          s1.pt_variables(), s2.pt_variables(),
          base, pte, result)
}
```

Recall that the hardware state machine's PTMemOp prevents us from changing the application memory and the TLB but allows us to arbitrarily change the page table memory. The page table state machine's step_Map is defined using pt_variables(), so it can restrict how the page table memory may change.

We have not yet discussed the page table state machine and do not know how it defines step_Map. It bears repeating that its exact definition is not something we need to manually check or trust. The interface specification (discussed in Chapter 8) guarantees that it describes the actual implementation's behavior and the state machine refinement proof (discussed in Chapter 10) guarantees that that behavior is the correct one.

Unmap

```
pub open spec fn step_Unmap(
  s1: OSVariables, s2: OSVariables,
  base: nat, result: UnmapResult) -> bool {
    &&& !s2.hw.tlb.dom().contains(base)
    &&& hardware::step_PTMemOp(s1.hw, s2.hw)
    &&& spec_pt::step_Unmap(
          s1.pt_variables(), s2.pt_variables(),
          base, result)
}
```

The Unmap step is defined similarly to the Map step, with the additional complication of TLB invalidation. We assume that TLB invalidation is handled by the OS, i.e. we require that in the post-transition state, the TLB does not contain an entry for the address that was unmapped.

Resolve

```
pub open spec fn step_Resolve(
  s1: OSVariables, s2: OSVariables,
  base: nat, result: ResolveResult) -> bool {
    &&& hardware::step_PTMemOp(s1.hw, s2.hw)
    &&& spec_pt::step_Resolve(
          s1.pt_variables(),
          s2.pt_variables(),
          base, result)
}
```

Resolve's definition is analogous to that of Map.

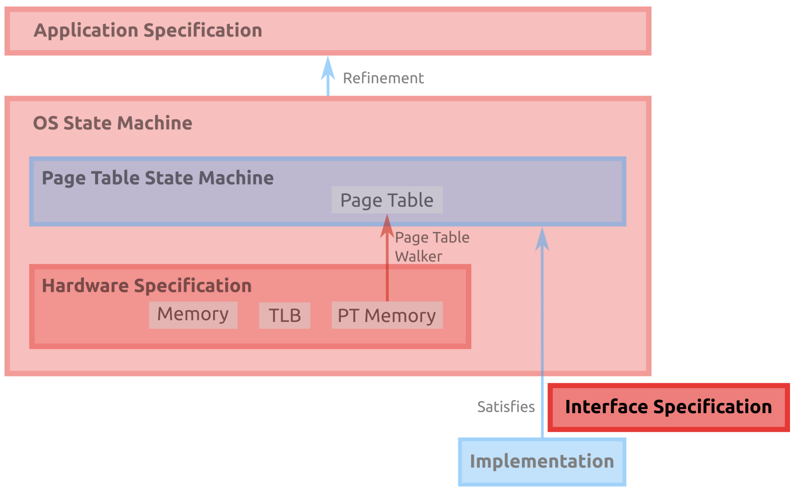
HW

```
pub open spec fn step_HW(
  s1: OSVariables, s2: OSVariables,
  system_step: hardware::HWStep) -> bool {
    &&& !system_step.is_PTMemOp()
    &&& hardware::next_step(s1.hw, s2.hw, system_step)
    &&& spec_pt::step_Stutter(
          s1.pt_variables(),
          s2.pt_variables())
}
```

The HW transition allows any non-PTMemOp transition from the hardware state machine and only a stutter step in the page table state machine. The stutter step is defined to be always-enabled and to not change the variables. The specific HW transitions we allow here are memory accesses to application memory and TLB operations.

Chapter 8

Interface Specification



We write specifications as state machines but the implementation consists of imperative code. The interface specification bridges this gap by connecting the implementation to the page table state machine. The goal is for each function — map, unmap, resolve — to guarantee that their behavior corresponds to the right step in this state machine. The interface specification prescribes a set of pre- and postconditions for each function intended to ensure this correspondence.

What exactly *is* the interface specification? In principle, the interface specification could simply be the pre- and postconditions of our implemented functions. We use *traits* instead because they allow us to cleanly separate the trusted interface specification from the untrusted implementation.

8.1 Traits in Verus

Traits are a Rust feature for polymorphic code. A trait is a collection of functions defined for some unknown type `Self`. The trait can then be implemented for any type. Verus extends this functionality by allowing the addition of pre- and postconditions to the defined functions.

```
trait T {
  fn abs(&self, n: i64) -> (r: i64)
    requires -1000 < n < 1000
    ensures r >= 0 && (r == n || r == - n);
}
```

When implementing the trait for some type, Verus requires the implementation to prove the postconditions specified in the trait definition.

```
struct S {}

impl T for S {
  // Verifies successfully
  fn abs(&self, n: i64) -> (r: i64) {
    if n < 0 { -n } else { n }
  }
}
```

Note that we implement the trait for a dummy type, an empty struct `S`. We are using the trait for its ability to enforce a certain interface, not for its usual Rust use case of enabling polymorphism.

Verus automatically verifies the above example but, as we would expect, fails to verify if we change the implementation to not satisfy the trait's postcondition.

```
struct S {}

impl T for S {
  // Fails to verify
  fn abs(&self, n: i64) -> (r: i64) {
      0
  }
}
```

Consequently, we can use a trait to write the interface specification and keep it separated from the untrusted implementation of the trait.

8.2 Interface Specification as a Trait

The interface trait defines three exec mode functions: `ispec_map_frame`, `ispec_unmap` and `ispec_resolve`. The implemented functions rely on an implementation invariant, which they are proved to preserve. To allow for this, the trait also defines a spec function `ispec_inv`, which the implementer can instantiate with an arbitrary predicate. Each function gets to assume the invariant in the precondition but has to prove that it is preserved. The trait also includes a proof mode function, i.e. lemma that requires the implementer to show that the invariant is satisfied initially, i.e. in an empty page table.

```
spec fn ispec_inv(&self, memory: mem::PageTableMemory) -> bool;

proof fn ispec_init_implies_inv(
  &self, memory: mem::PageTableMemory)
  requires
    memory.inv(),
    memory.regions() === set![memory.cr3_spec()@],
    memory.region_view(memory.cr3_spec()@).len() == 512,
    (forall|i: nat| i < 512 ==>
        memory.region_view(memory.cr3_spec()@)[i] == 0),
  ensures
    self.ispec_inv(memory);
```

`Spec_pt` refers to the module implementing the page table state machine, discussed in Chapter 9. For the present discussion we only need to know that it defines the usual state machine functions `init` and `next` and that its state contains a map representing the page table. We use the page table walker semantics `interp_pt_mem` to map the concrete memory to that abstract page table.

With the implementation invariant taken care of, we next define the paging functions.

Map_frame

`Map_frame` takes a memory struct as an argument to manipulate the page table memory as well as an address and page table entry to map. Its result is a `MapResult`, i.e. either success or an error if the new mapping would overlap an existing one. Additionally, it returns the changed memory struct. This pattern could also be expressed as the function mutably borrowing the memory but having ownership of the memory is more convenient for the implementer.

```
fn ispec_map_frame(&self,
  memory: mem::PageTableMemory, vaddr: usize,
  pte: PageTableEntryExec)
  -> (res: (MapResult, mem::PageTableMemory))
```

The function may assume the enabling conditions of the corresponding page table state machine transition and the invariant.

```
  requires
    self.ispec_inv(memory),
    spec_pt::step_Map_enabled(interp_pt_mem(memory), vaddr, pte@),
```

It must prove that the invariant is preserved and that it changes the memory in such a way that the interpretations of the pre and post states are a valid `Map` transition.

```
  ensures
    self.ispec_inv(res.1),
    spec_pt::step_Map(
      spec_pt::PageTableVariables { map: interp_pt_mem(memory) },
      spec_pt::PageTableVariables { map: interp_pt_mem(res.1) },
      vaddr,
      pte@,
      res.0);
```

Unmap

`Unmap` is very similar to `map_frame`, so we define it accordingly.

```
fn ispec_unmap(&self,
  memory: mem::PageTableMemory, vaddr: usize)
  -> (res: (UnmapResult, mem::PageTableMemory))
  requires
    spec_pt::step_Unmap_enabled(vaddr),
    self.ispec_inv(memory),
  ensures
    self.ispec_inv(res.1),
    spec_pt::step_Unmap(
      spec_pt::PageTableVariables { map: interp_pt_mem(memory) },
      spec_pt::PageTableVariables { map: interp_pt_mem(res.1) },
      vaddr,
      res.0);
```

`Unmap` has no proof obligations regarding TLB invalidation because we currently leave the responsibility of TLB management to the rest of the operating system. The assumption that the OS handles TLB invalidation correctly is introduced in the `step_Unmap` transition in the OS state machine (cf. Section 7.2.2). Once we extend this work to support multiprocessors we intend to also verify TLB shootdown, which would result in an additional postcondition here, certifying that the implementation correctly invalidated the TLBs.

Resolve

`Resolve` is defined similarly to `map_frame` and `unmap`. Because it is a read-only transition we additionally require that the returned memory is unchanged

```
fn ispec_resolve(&self,
  memory: mem::PageTableMemory, vaddr: usize)
  -> (res: (ResolveResultExec, mem::PageTableMemory))
  requires
    spec_pt::step_Resolve_enabled(vaddr),
    self.ispec_inv(memory),
  ensures
    res.1 === memory,
    spec_pt::step_Resolve(
      spec_pt::PageTableVariables { map: interp_pt_mem(memory) },
      spec_pt::PageTableVariables { map: interp_pt_mem(memory) },
      vaddr,
      res.0@
    );
```

Note that this type would in principle allow an implementation to modify the memory as long as it changes the values back to their original ones. In the future, we intend to axiomatize the memory to keep a log of the memory operations. Although the intended use is to allow more granular transitions, that change will also fix the problem of resolve being able to modify memory.

8.3 Trust and Assumed Atomicity

Our stated goal for the interface specification is simple enough: It must guarantee that executing each function corresponds to the right step in the page table state machine. Checking this to be true by inspecting the definitions is a straightforward task.

However, trusting this specification requires us to consider a few more subtleties.

Since these definitions represent the final specifications for the functions, we need to make sure that their preconditions are not too strong. While the invariant assumption is unproblematic, we have to ensure that the transition enabling conditions allow us to call the function in any situation we wish to use the functions in. The page table state machine's enabling conditions are identical to those in the application specification. Nearly all of them are straightforward restrictions on the parameters, e.g. when mapping a frame, the frame size must be 4KiB, 2MiB or 1GiB. The only more complicated restriction is that we cannot map any physical memory region twice.

Another consideration is that `map_frame` and `unmap` can make many modifications to the memory. By showing correspondence to a single transition, we treat all these modifications as a single, atomic modification of the memory. In practice, this is not the case. Page table walks in the MMU may observe intermediate states, where only a subset of the modifications is visible.

We ensure that in any of these intermediate states, the page table will appear to be in either the old or the new state. Tao et al. [54] aptly name a paging implementation with this behavior a *transactional page table*.

We make the argument for transactionality of `map_frame`. A similar line of reasoning applies to `unmap` but is omitted.

The `map_frame` function only maps a single frame. It only modifies the memory in two ways:

1. In a page directory, it may write to a currently-empty entry to point to a new, empty directory.
2. In the final page directory, it may write to a currently-empty entry to map the frame.

Mapping new, empty directories does not affect the page table walker's view of the page table. A page table walk is only aware of the new mapping if all writes are visible. If the page table walk observes any subset of writes, it either cannot see the frame mapping or part of the path to the mapping yet.

Hence, `map_frame` is transactional. Disallowing interleavings by assuming atomic transitions does not hide any observably different intermediate states.

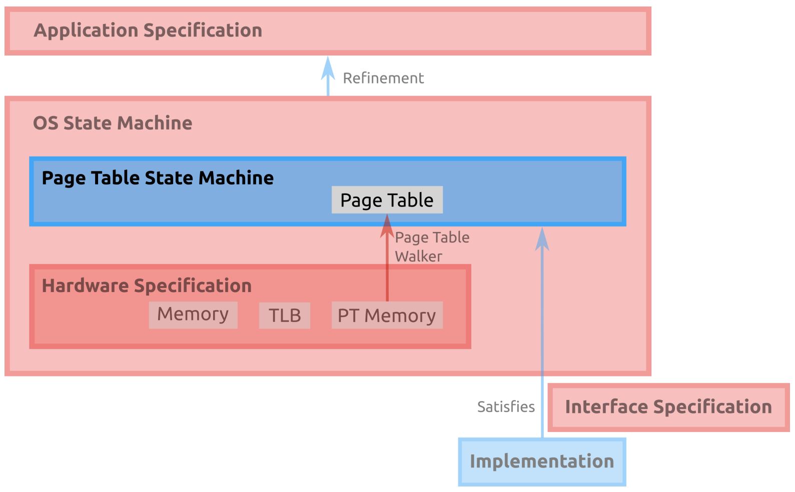
In future work we would like to make this argument formally, by treating all memory accesses in the page table as individual ones. A prerequisite for this is extending the page table memory to allow us to recover the individual operations. We discuss this extension in Section 14.4.1.

Part III

Implementation and Proofs

Chapter 9

Page Table State Machine



The page table state machine describes the effects that the implemented functions have on an abstract view of the page table. The implementation proves its adherence to this state machine by implementing the interface specification. The page table state machine is integrated in the OS state machine to combine the functions' semantics with the hardware and OS environment.

The page table state machine's definition is straightforward. As in the application specification, it contains an abstract map view of the page table.

```
pub struct PageTableVariables {
    pub map: Map<nat, PageTableEntry>,
}
```

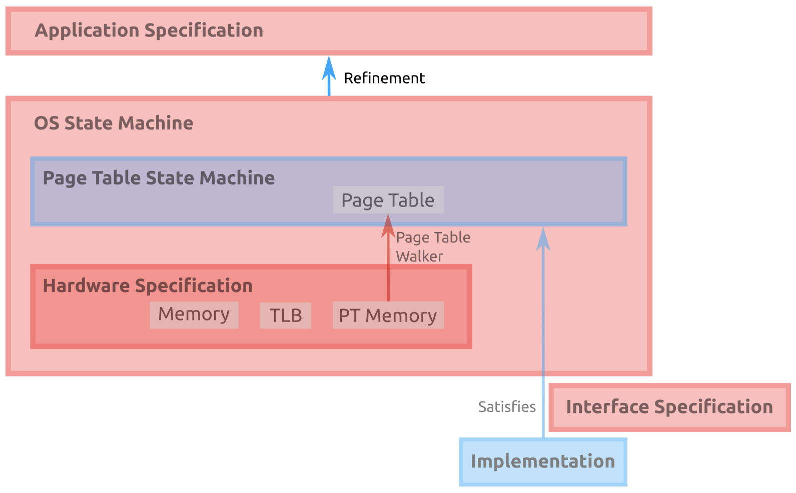
The supported transitions are identical to those of the application specification, except that the page table state machine has no concept of the application memory and does not have the `ReadWrite` transition.

```
pub enum PageTableStep {
    Map     { vaddr: nat, pte: PageTableEntry,
              result: MapResult },
    Unmap   { vaddr: nat, result: UnmapResult },
    Resolve { vaddr: nat, result: ResolveResult },
    Stutter,
}
```

We do not show any of the page table state machine's transitions here and refer to the application specification's transitions in Section 5.2.2 instead. The only difference in their definitions is that the page table state machine's transitions make no mention of the application memory.

Chapter 10

OS State Machine Refinement



One part of proving the implementation's refinement of the application specification is to show that the OS state machine, with the paging function semantics from the page table state machine, refines the application specification. In this chapter we discuss how this proof proceeds.

10.1 Interpretation Functions

For the refinement, we define two interpretation functions, which map the OS state machine's variables and labels to their application specification counterparts. These interpretation functions are technically part of the OS state machine and they are *trusted*. However, we discuss them in this chapter because they are only directly used in the state machine refinement proof.

We do not show the definitions but describe them in words. The definitions can be found in the `spec_t::os` module.

The label interpretation function maps the OS state machine's transition labels to those of the application specification. `ReadWrite`, `Map`, `Unmap` and `Resolve` are mapped to their equivalents, while the hardware `TLBFill` and `TLBEvict` transitions are mapped to the application specification's `Stutter` step.

The variable interpretation function's main task is to construct the virtual memory, which it does by constructing a map whose domain are all those virtual addresses that are mapped by the current address translation state. By address translation state, we mean the combination of translations in the TLB and in the page table, with TLB entries taking precedence.

10.2 Invariant

The refinement proof relies on a state machine invariant, which guarantees that none of the entries overlap in virtual or physical memory, that the mapped entries have a valid size and are word-aligned and that any entry in the TLB is also contained in the page table.

```
pub open spec fn inv(self) -> bool {
  &&& self.pt_mappings_dont_overlap_in_vmem()
  &&& self.pt_mappings_dont_overlap_in_pmem()
  &&& self.pt_entry_sizes_are_valid()
  &&& self.pt_entries_aligned()
  &&& self.tlb_is_submap_of_pt()
}
```

Note that the last conjunct is only true because our state machines do not currently expose intermediate states (cf. Section 8.3).

The proof that this invariant holds in the initial state and is preserved during transitions is straightforward.

10.3 Proof Strategy

The full refinement proof is contained in the module `impl_u::os_refinement`. Here, we just sketch the proof structure and informally justify the reasoning steps.

The main refinement theorem is the following one.

```
proof fn next_step_refines_hl_next_step(
  s1: OSVariables, s2: OSVariables, step: OSStep)
  requires
    s1.inv(),
    next_step(s1, s2, step)
  ensures
    hlspec::next_step(
      s1.interp_constants(),
      s1.interp(), s2.interp(),
      step.interp())
{ .. }
```

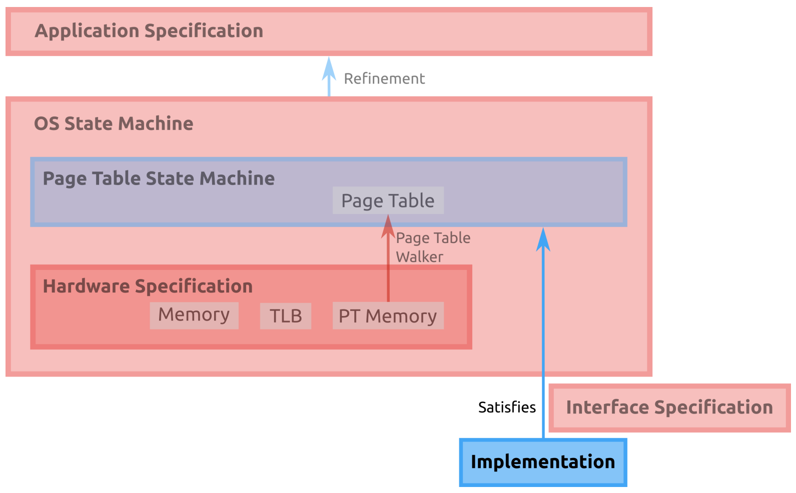
The `interp` functions correspond to the variable and label interpretation functions, respectively.

The proof is structured as a pattern match on the label `step`. We prove a lemma stating that the interpreted application specification's page table is equivalent to the one in the OS state machine. This follows from the facts that the TLB's entries are a subset of the page table's entries and that the interpretation uses these two to construct the interpreted page table.

This lemma makes the refinement proof fairly straightforward for most transitions. Only the `ReadWrite` transition relies on some more manual proofs that characterize the correspondence between the virtual and physical memories.

Chapter 11

Implementation



In this chapter we take a detailed look at the implementation and the corresponding proofs. The implementation is developed in a two-step refinement, so we start by considering the refinement structure. Then we are going to have a closer look at each of the three refinement layers, what they implement and which major proof steps they contain.

The implementation and its proofs make up the largest part of this project in terms of code size. Instead of studying it in detail, we will focus on the general structure and significant design decisions. Each section on the specific refinement layers indicates the Rust module containing it.

Finally, we prove that the resulting implementation satisfies the interface specification.

11.1 Refinement Structure

In principle, we could directly write an implementation and then prove that it satisfies the pre- and postcondition pair dictated by the interface specification. While technically possible, this would result in intractably large proof steps. In constructing the implementation by data refinement we hope to decompose this proof into many smaller ones, which can be feasibly proved by semi-automatic methods.

The implemented operations will manipulate memory and write entries into paging directories whose bits encode addresses and metadata. The proof goal is to prove that the result of these memory manipulations, as interpreted by the MMU, behaves like an abstract map from virtual addresses to page table entries.

At least two difficult steps separate these concepts. The step from mapping concrete memory manipulations to changes in an abstract data structure and the step from a hierarchical tree data structure to a simple, non-hierarchical map.

We choose the refinement layers accordingly. The topmost refinement layer, *RL0* is a purely functional definition of the operations on a simple map, closely mirroring what is required by the proof goals. Below that, we have the layer *RL1*, again purely functional but now using a tree data structure mimicking the page table hierarchy. Finally, the layer *RL2* at the very bottom is the *real* implementation, consisting of imperative code.

We prove:

- RL1 refines RL0
- RL2 refines RL1
- Thus: RL2 refines RL0

Finally, when we instantiate the interface specification with the real implementation, we use the refinement theorem to show that our operations behave like the ones on an abstract map. This is sufficient to complete the proof with very few additional steps.

11.2 Generic Architecture

Across all refinement layers we make the implementations generic to the exact hardware architecture. We keep these parameters contained in the `Arch` struct shown in Figure 11.1. The struct specifies the exact structure of the multi-level page table, i.e. how many entries each directory contains and how much address space is mapped by a page mapping or directory at a certain level.

```
pub struct ArchLayer {
    /// Size of address space mapped by one entry at this layer
    pub entry_size: nat,
    /// Number of entries at this layer
    pub num_entries: nat,
}
pub struct Arch {
    pub layers: Seq<ArchLayer>,
}
```

Figure 11.1: The Arch struct

We define helper methods `entry_size(self, layer: nat)` and `num_entries(self, layer: nat)`, which return the entry size and number of entries at a given layer, respectively.

In a meaningful architecture struct, each layer should have an entry size equal to the next layer's entry size times the next layer's number of entries. This is a condition that we add to the invariants at each refinement layer.

This approach provides a sufficient amount of flexibility that, for example, it would be straightforward to adapt the implementation for x86 five-level paging rather than four-level paging. Adapting it to another architecture such as Arm would be a much more involved task, not least because the different memory semantics would require changes in trusted specifications as well.

The objective is not to make the implementation reusable across architectures but to abstract away unnecessary detail.

11.3 RL0: Abstract Map Implementation `impl_u::l0`

The topmost refinement layer RL0 is defined entirely in Verus' spec mode. It will be erased when our code is compiled. Further, it allows only a purely functional subset of Rust.

We define the paging operations on the following struct.

```
pub struct PageTableContents {
  pub map: Map<nat, PageTableEntry>,
  pub arch: Arch,
  pub lower: nat,
  pub upper: nat,
}
```

The `map` field matches the variables in the page table state machine, which will make it easier for us to prove adherence to the interface specification.

```
pub open spec fn map_frame(self, base: nat, pte: PageTableEntry)
  -> Result<PageTableContents,PageTableContents> {
  if self.accepted_mapping(base, pte) {
    if (forall|b: nat|
          self.map.dom().contains(b) ==> !overlap(
            MemRegion { base: base, size: pte.frame.size },
            MemRegion { base: b,
                        size: self.map.index(b).frame.size }))
    {
      Ok(PageTableContents {
        map: self.map.insert(base, pte),
        ..self
      })
    } else { Err(self) }
  } else { arbitrary() }
}
```

Figure 11.2: The `map_frame` function on refinement layer 0

The `upper` and `lower` fields limit the domain for which this page table maps entries.

As an example of this layer's definitions, we show the `map_frame` function in Figure 11.2. The `accepted_mapping` predicate checks some basic properties of the arguments, such as alignment of the virtual address. For arguments that do not satisfy these properties, we do not define the function, returning an arbitrary value instead. `Accepted_mapping`'s conditions line up with the enabling conditions of the page table state machine. Thus, the functions will never be called with the "wrong" arguments. For valid arguments, the function either returns an updated struct with the new mapping or, if the mapping would overlap with an existing one, an error with the unchanged state.

The other operations follow the same pattern as `map_frame`.

The only proofs we need on this layer are to show that these operations preserve the simple invariant, that all mappings in the state satisfy the conditions ensured in `map_frame`. These proofs are straightforward. The definitions and proofs for RL0 only make up $\sim$ 8% of the overall implementation code.

11.4 RL1: Abstract Tree Implementation `impl_u::l1`

The intermediate refinement layer is again defined purely in spec mode. It features paging functions that operate on an abstract representation of the page table's tree structure.

11.4.1 Data Structure

The data structure we chose in RL0 is largely dictated by the structure we use in the page table state machine. On RL2, we will have to use some representation or other of the actual memory. RL1 is an intermediate layer and does not need to conform to any other interfaces, giving us more freedom in the choice of data structure.

We can represent the abstract tree structure in a variety of ways. To illustrate the benefits of our current structure we start out with a different structure that we initially intended to use and then show its problems and how the new structure addresses them.

We first tried to use the following structure.

```
pub struct Directory {
  pub map: Map<nat, NodeEntry>,
  pub layer: nat,
  pub arch: Arch,
}
```

```
pub enum NodeEntry {
  Directory(Directory),
  Page(PageTableEntry),
  Empty(),
}
```

A page directory is on a particular layer (i.e. depth). As on RL0, it maps addresses to entries but the entries can not only be a mapped `Page` but also a nested `Directory` or it can be `Empty`. This is just RL0's structure with the added tree hierarchy.

However, to ensure that a given `Directory`'s hierarchy meaningfully encode an actual page directory, we need to heavily constrain what mappings are allowed. A directory's entries cannot overlap. The directory's entries must all be mapped at virtual addresses that are aligned to its layer's entry size. A nested directory should only contain entries within a particular range. All these constraints would become part of the invariant.

Even with all these constraints, the structure is still quite far removed from the concrete structure, which will make the refinement proof between RL1 and RL2 harder.

We solve both of these problems by changing the `Directory` struct.

```
pub struct Directory {
  pub entries: Seq<NodeEntry>,
  pub layer: nat,
  pub base_vaddr: nat,
  pub arch: Arch,
}
```

We replace the map with a sequence and store the base_vaddr, i.e. the address starting at which the directory maps entries. Instead of looking up a virtual address directly, we compute an index and use that to look up the corresponding entry. This more closely matches what we are going to do in RL2 and it directly encodes some of the problematic constraints. The invariant needs to ensure that the length of entries matches what is specified by arch and that the base_vaddr of an entry at index i is equal to parent.base_vaddr + i * parent.entry_size(parent.layer). If these conditions are satisfied, the regions mapped by any two entries do not overlap.

Paging implementations typically compute the index for a virtual address by shifting and masking off specific bits, which we cannot do here because we use natural numbers. However, because we keep track of the base virtual address for each directory, we can use that to compute the index instead.

```
pub open spec fn index_for_vaddr(self, vaddr: nat) -> nat {
  (vaddr - self.base_vaddr) / self.arch.entry_size(self.layer)
}
```

Computing the base address of an entry is also a common operation, so we add a definition for it as well.

```
pub open spec fn entry_base(self, idx: nat) -> nat {
  self.base_vaddr + idx * self.arch.entry_size(self.layer)
}
```

11.4.2 Invariant

We discussed the following constraints.

```
pub open spec fn directories_are_in_next_layer(&self) -> bool {
  forall|i: nat|
    (i < self.entries.len()
      && self.entries.index(i).is_Directory())
    ==> {
      let directory = self.entries.index(i).get_Directory_0();
      &&& directory.layer == self.layer + 1
      &&& directory.base_vaddr
            == self.base_vaddr + i * self.entry_size()
    }
}
```

The invariant ensures these constraints as well as a number of others.

```
pub open spec fn inv(&self) -> bool
  decreases self.arch.layers.len() - self.layer
{
  &&& self.well_formed()
  &&& self.pages_match_entry_size()
  &&& self.directories_are_in_next_layer()
  &&& self.directories_match_arch()
  &&& self.directories_obey_invariant()
  &&& self.directories_are_nonempty()
  &&& self.frames_aligned()
}
```

Most of these predicates assert fairly basic properties. `Directories_obey_invariant` ensures that the invariant holds recursively. The `decreases` clause guides the termination proof.

One of the more interesting conjuncts is `self.directories_are_nonempty()`. When attempting to map a huge page, we may discover that a directory is already mapped in that location. If that directory is empty, we should remove it and map the page instead; If it is not empty, we should return an error. To simplify that case, we eagerly unmap directories when they become empty as the result of an unmap operation. Any directories contained in the page table are thus non-empty.

11.4.3 Paging Functions

We again discuss just the `map_frame` function on this refinement layer. Similar to the recursive datatype and the recursive invariant, we define `map_frame` recursively as well.

The code is shown in Figure 11.3. At location one we see that the function is defined when some basic preconditions are met but also only on states where the invariant is satisfied. The invariant is in fact necessary to complete the termination proof, indicated at location two. The `decreases_by` clause points to a separate `proof` function, in which we can write assertions to help the verifier with the termination proof. In this case the termination argument is sufficiently complex that this is necessary.

The body of the function is relatively straightforward. At location three, we compute the index of the entry that contains the given virtual address. At location four, we pattern match on the entry and then implement the correct semantics. For example, at location five, if the entry is a directory and the region to be mapped is not the right size for this layer, we do a recursive call and, after it returns, replace the old entry with the result, preserving the `Ok` or `Err` error code.

```
pub open spec fn map_frame(self, vaddr: nat, pte: PageTableEntry)
  -> Result<Directory,Directory>
  decreases self.arch.layers.len() - self.layer
{                                                      2
  decreases_by(Self::check_map_frame);

  if self.inv() && self.accepted_mapping(vaddr, pte) {         1
    let idx = self.index_for_vaddr(vaddr);                   3
    match self.entries.index(idx) {
      NodeEntry::Page(p) => Err(self),
      NodeEntry::Directory(d) => {
        if self.arch.entry_size(layer) == pte.frame.size {
          Err(self)
        } else {
          match d.map_frame(vaddr, pte) {
            Ok(d)  => Ok(self.update(idx,
                                NodeEntry::Directory(d))),
            Err(d) => Err(self.update(idx,                  5
                                NodeEntry::Directory(d))),
          } } },                                              4
      NodeEntry::Empty() => {
        if self.entry_size() == pte.frame.size {
          Ok(self.update(idx, NodeEntry::Page(pte)))
        } else {
          let new_dir = self.new_empty_dir(idx);
          Ok(self.update(idx,
              NodeEntry::Directory(
                new_dir.map_frame(vaddr, pte).get_Ok_0())))
        }
      },
    }
  } else { arbitrary() }
}
```

Figure 11.3: Map_frame on refinement layer 1

11.4.4 Proofs

As in RL0, we prove that each operation preserves this refinement layer's invariant but now we additionally show that the operations refine the ones in RL0.

The interpretation function `interp` used for the refinement is based on the function `interp_of_entry`.

```
pub open spec fn interp_of_entry(self, idx: nat)
  -> l0::PageTableContents
  decreases [...]
{
  if self.inv() && idx < self.entries.len() {
    let (lower, upper) = self.entry_bounds(idx);
    l0::PageTableContents {
      map: match self.entries.index(idx) {
        NodeEntry::Page(p)      => map![self.entry_base(idx) => p],
        NodeEntry::Directory(d) => d.interp().map,
        NodeEntry::Empty()      => map![],
      },
      arch: self.arch,
      lower,
      upper,
    }
  } else { arbitrary() }
}
```

It constructs the RL0 struct for a particular entry. If the entry is a page, the corresponding RL0 map contains just one entry; If the entry is a directory, it contains all the entries in that directory's interpretation; If the entry is empty, it contains no entries.

The full `interp` function combines `interp_of_entry`'s results for all entries in a directory.

We prove a refinement lemma like the following one for each operation.

```
pub proof fn lemma_map_frame_refines_map_frame(
  self, base: nat, pte: PageTableEntry)
  requires
    self.inv(),
    self.accepted_mapping(base, pte),
  ensures
    self.map_frame(base, pte).map(|d| d.interp()) —— 1
        === self.interp().map_frame(base, pte),
  { .. }
```

The map method indicated at location one applies the `interp` function to either variant of the result (i.e. `Ok` or `Err`). Thus, the postcondition guarantees that `map_frame` refines the `map_frame` function on RL0. The `Result` variant must match and the returned state must be the same, modulo the interpretation function.

The definitions and proofs on this refinement layer make up approximately 40% of the overall implementation code.

An important aspect that reduces the complexity of the proofs are our efforts to isolate reasoning about nonlinear arithmetic into few lemmas.

11.4.5 Indexing Calculus

Early in the development of this refinement layer we discovered that Verus offers a subpar experience when it comes to proofs that contain reasoning steps, which require support for nonlinear arithmetic. The actual problems and their impact are discussed later in Section 13.1.1. Nonlinear arithmetic in Z3 (the SMT solver employed by Verus) is essentially any arithmetic involving multiplication, division or modulo.

As shown earlier we use the `entry_base` and `index_for_vaddr` functions for indexing purposes. These functions contain multiplication and division respectively. Initially, we did not define these functions but did the indexing arithmetic directly where needed. This caused reasoning about nonlinear arithmetic to come up in many different proofs.

After defining the functions, we were able to isolate virtually all nonlinear reasoning into one lemma for each function, allowing us to focus on other reasoning steps in most proofs.

11.5 RL2: Imperative Implementation `impl_u::l2_impl`

RL2 is the bottom refinement layer, in which we write the concrete implementation in imperative code. The key step between RL1 and RL2 is moving from a functional data structure to its encoding in a flat memory.

For this layer's discussion we largely refrain from including code snippets, as many of the definitions are large and include a lot of detail that is not particularly interesting but necessary to understand the definitions. However, all the definitions and proofs are available in the module `impl_u::l2_impl`.

11.5.1 Data Structure

On RL2 we no longer have a recursive data structure. Instead we rely on an instance of the page table memory struct discussed in Section 6.4. We also use `ArchExec`, a variant of the `Arch` struct that uses the concrete type `usize` instead of natural numbers.

```
pub struct PageTable {
    pub memory:   mem::PageTableMemory,
    pub arch:     ArchExec,
    pub ghost_pt: Ghost<PTDir>,
}
pub struct PTDir {
    pub region:       MemRegion,
    pub entries:      Seq<Option<PTDir>>,
    pub used_regions: Set<MemRegion>,
}
```

A primary concern in the proofs at this level is to show that modifications in the memory only locally affect the page table. To aid these arguments we use the ghost field `ghost_pt`, which contains metadata of the memory regions used to store page directories. Note that the used `PTDir` type is recursive. It is essentially a copy of only the page table's *structure*, with the `region` field identifying the physical memory region where that directory is stored and `entries` doing the same for any nested directories. The `used_regions` field contains the reflexive, transitive closure over the region field. I.e. the field contains all memory regions used by directories appearing recursively in this tree.

11.5.2 Invariant

The meaning we assigned to the `ghost_pt` field is our interpretation of it but largely not guaranteed by its structure. Consequently, we need to establish these properties in the invariant, making this layer's invariant significantly more complex than those for RL0 and RL1. Two thirds of the invariant's conjuncts are used for this purpose.

It is possible that a better structure would allow us to use a simpler invariant, as it did in RL1.

11.5.3 Paging Functions

We implement the paging functions on this refinement layer by closely matching the structure of RL1's functions but using the `memory` struct to access page directory entries in memory.

Matching RL1's structure includes the recursion. We could replace the recursion by a while loop with relative ease but do not see any particular reason for doing so. The function definitions are small enough to fit in a processor's instruction cache and the number of branching instructions would remain the same, so this should not have any noticeable implications for performance.

We do not show code listings for any of the functions here because of the size if we were to include the necessary auxiliary definitions. The complete code is available online.[1] The branch `erased` contains a version of the code with all specification and proof code erased.

11.5.4 Proof

We prove that the RL2 paging functions preserve the invariant and refine the corresponding functions in RL1.

The invariant preservation proofs are complete. The refinement proofs for `resolve` and `map_frame` are largely complete but contain a small number of unproved reasoning steps, mostly relating. Many but not all of the missing proof steps are related to showing that calculating the memory addresses we access to modify the page table, does not cause arithmetic overflow.

The refinement proof for `unmap` is still very incomplete. However, its structure closely matches that of RL1's `unmap` and the near-complete proofs of the other functions give us confidence in the definitions on this refinement layer. Hence, we consider it very likely that `unmap` does refine RL1 `unmap`.

This refinement layer is the most complex one. Its definitions and proofs make up more than half of the implementation code. More than a third of this layer's code is related to `map_frame`'s proofs. We expect that refactoring could reduce this amount significantly.

[1]https://github.com/matthias-brun/verified-paging-for-x86-64-in-rust

11.6 Interface Implementation Proof

impl_u::l2_refinement

Proving that the implementation satisfies the interface specification is the true goal of all the work we have done in the refinement proofs. We do so by defining an empty struct and implementing the interface specification trait for it.

```
pub struct PageTableImpl {}
impl InterfaceSpec for PageTableImpl {
  ...
}
```

We will show the implementation process for map_frame. Recall that our concrete implementation is defined on the PageTable struct.

```
pub struct PageTable {
    pub memory:   mem::PageTableMemory,
    pub arch:     ArchExec,
    pub ghost_pt: Ghost<PTDir>,
}
impl PageTable {
  pub fn map_frame(&mut self,
    vaddr: usize, pte: PageTableEntryExec)
    -> (res: MapResult)
}
```

However, we need to implement the following function signature in the trait.

```
fn ispec_map_frame(&self,
  memory: mem::PageTableMemory,
  vaddr: usize, pte: PageTableEntryExec)
  -> (res: (MapResult, mem::PageTableMemory))
```

We use a particular instance of the ArchExec struct, containing the correct parameters for x86-64 4-level-paging but we notice that the ghost_pt ghost state our implementation's proofs rely on is missing in the interface specification.

To have the ghost state available, we assert the existence of *some* suitable ghost state in the invariant, whose definition we are free to choose as long as it is satisfied initially and preserved with each function.

```
spec fn ispec_inv(&self, memory: mem::PageTableMemory) -> bool {
  exists|ghost_pt: l2_impl::PTDir| {
    let page_table = l2_impl::PageTable {
      memory: memory,
      arch: x86_arch_exec_spec(),
      ghost_pt: Ghost::new(ghost_pt),
    };
    &&& page_table.inv()
    &&& page_table.interp().inv()
  }
}
```

With this invariant, implementing `ispec_map_frame` becomes reasonably straightforward.

```
fn ispec_map_frame(&self,
  memory: mem::PageTableMemory,
  vaddr: usize, pte: PageTableEntryExec)
  -> (res: (MapResult, mem::PageTableMemory))
{
  let ghost_pt: Ghost<l2_impl::PTDir> = ghost(
    choose|ghost_pt: l2_impl::PTDir| { .. }
  );
  let mut page_table = l2_impl::PageTable {
    memory:    memory,
    arch:      x86_arch_exec_v,
    ghost_pt:  ghost_pt,
  };
  let res = page_table.map_frame(vaddr, pte);
  (res, page_table.memory)
}
```

We use `choose` to obtain the ghost state, create a mutable `PageTable` struct and call our RL2 `map_frame` function. When it returns, we simply return its result and the possibly modified memory.

The proof for `map_frame` is complete, except for one proof step that likely fails due to an incompleteness bug in Verus. The proofs for `unmap` and `resolve` are incomplete.

Part IV

Evaluation and Future Work

Chapter 12

Evaluation

With our evaluation we aim to answer three questions:

- Does our development approach scale to verifying larger systems? To what degree is that attributable to either Verus or our proof structure?
- Do our verified functions *actually work*?
- Is our paging functions' performance competitive with the implementations in other operating systems?

To answer the first question, we make measurements of proof latency and proof overhead, which show our approach to be scalable.

The second question is answered by integrating the verified functions into NrOS and running its test suite.

The third question is addressed by running NrOS' micro-benchmarks for the map function with the now-integrated functions. By comparing the results to those of NrOS' default implementation we conclude that our implementation scales similarly well. The overall performance is limited by the node replication overhead.

To get a more direct comparison we also run experiments to compare the implementations' single-threaded performance, which show that the map functions perform similarly well, while the verified unmap function performs worse than the unverified one. We identify the reason for the reduced performance to be the inefficient emptiness checking.

The data for all results reported in this chapter, as well as instructions for how to reproduce most of it are available online.[1]

[1]https://github.com/matthias-brun/verified-paging-for-x86-64-in-rust

12.1 Scalability

To estimate the scalability of our development approach, we measure and analyze two proxy metrics — verification latency and proof overhead — and discuss their impact on development, as well as how we perceived their impact in this project.

Low verification latency is an important part of an effective verification workflow, allowing the developer to obtain quick feedback, as opposed to just spending time waiting on the tool.

Proof overhead, i.e. the number of lines of proof per line of code, is another, albeit less reliable, indicator for the effectiveness of a verification workflow. Usually, low overhead is preferable, although when modifying verified code, explicit proof steps tend to be helpful in locating failing proof steps. Reducing the amount of *necessary* overhead, while retaining the ability to express technically unnecessary proof steps, is desirable. However, measuring only necessary overhead is virtually impossible, so we estimate it by measuring the effective proof overhead instead.

12.1.1 Verification Latency

Verus' verification latency consists of three components: the Rust compiler's type checking time, Verus' query generation time and lastly, the time taken for Z3 to check the queries.

We perform a number of measurements of the verification latency and its components. All these measurements were made on the hardware used for the majority of the development, a laptop computer with an Intel i7-10510 CPU and 16GiB of memory.

Rust Phase

For the full, current code base, this phase runs in 6 seconds (±15%). Verus does not currently support incremental compilation, meaning the phase is always executed for the entire project. Consequently, this time cost of 6 seconds is paid on every Verus invocation, even if verifying just one module or function. This latency represents a lower bound for the total latency of any verification task in this project.

Though noticeable, the latency introduced in this phase has remained small enough to not present a significant obstacle to an interactive verification experience. As the size of a project increases, that is likely to change, unless Verus gains support for performing the Rust compiler phase incrementally. The Rust phase currently consists of two invocations of the Rust compiler.

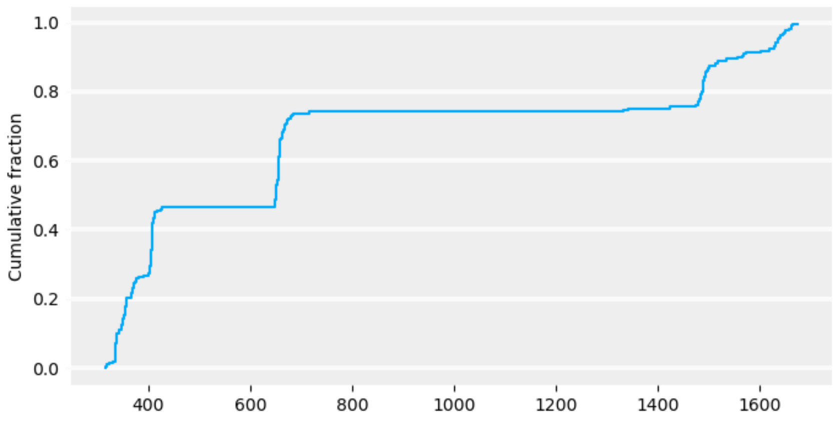


Figure 12.1: Cumulative distribution of Verus phase latency in milliseconds

A new feature in upstream Rust[2] will enable Verus to reduce this to one invocation, which is likely to cut this phase's latency nearly in half. There is also likely significant room for optimization in Verus' custom syntax macro, which causes a sizeable part of this phase's latency. Additionally, support for crates is on Verus' roadmap and may provide a workaround by allowing the developer to modularize a project and verify only one crate at a time.

Verus Phase

The Verus phase takes approximately seven seconds for the whole project, though this phase only runs for the requested modules or functions. During development, we generally worked on one function at a time and constrained the verifier accordingly. Hence, this is the granularity at which we measure latencies now. Verus always generates queries for functions in exec and proof mode but for spec functions only if they have the `checked` attribute. Queries are also generated for constants (to check for overflow).

Figure 12.1 shows the distribution of latencies in the Verus phase. For many functions, the latencies are too small to notice in practice. All functions exhibit latencies of less than two seconds in this phase.

Z3 Phase

The latency of automated verification tools tends to be dominated by the actual verification phase, i.e. waiting for Z3 to check the query.

Z3's latency can vary significantly due to small changes to the code. Nevertheless, measuring the verification times in the project's current state gives

[2]https://github.com/rust-lang/lang-team/issues/161

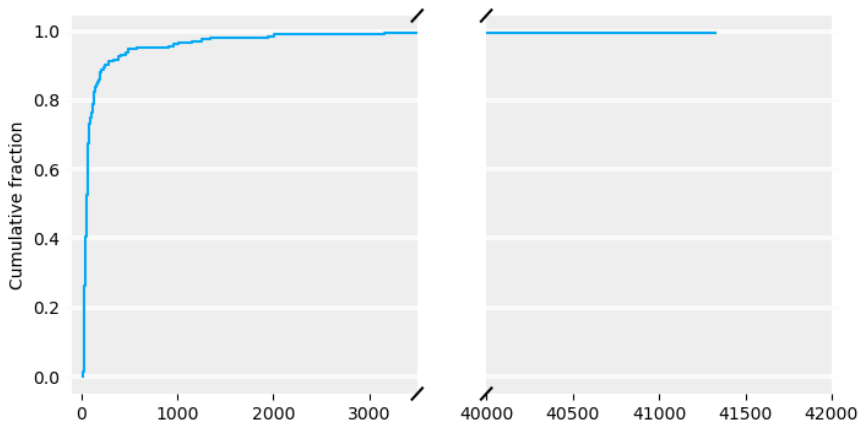


Figure 12.2: Cumulative distribution of Z3 phase latency in milliseconds

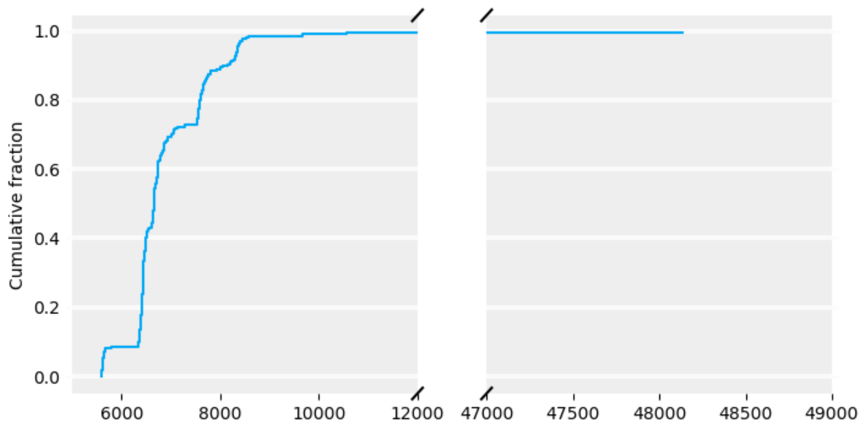


Figure 12.3: Cumulative distribution of total verification latency in milliseconds

us a good estimate of the typical Z3 latencies experienced during development. Again, we measure the latencies for each definition or lemma.

Figure 12.2 shows how the Z3 latencies for each verification unit are distributed. The latencies are strongly skewed towards very low numbers. 96% of queries verify in less than one second. Only one takes longer than four seconds to verify. Notably, this outlier is the only large verified function in exec mode.

Total Latency

In Figure 12.3 we show the distribution of total verification latencies. With 99% of all queries verifying in less than 10 seconds, these numbers support the hypothesis that our approach scales to larger projects.

VeriBetrKV [20] is a larger (6.4k implementation LoC) systems software verification project, whose authors perform a similar analysis to ours in their paper, making it an excellent candidate for comparisons.[3] Their reported numbers are similar to ours: 98.3% of verification units verify within ten seconds. Few outliers take a long (up to 140s) time to verify. Although our project appears to have fewer outliers, we do not draw conclusions on that basis; Our single outlier is the largest verified exec mode function, which suggests that a larger project with more such functions may also see more outliers.

VeriBetrKV's developers followed a policy of deliberately improving definitions and lemmas with long-running verification times, attempting to keep them below 20 seconds. In this project, we did not consciously follow any such policy, only deliberately improving proofs in very few cases. Nevertheless, the verification latencies in this project compare somewhat favorably to VeriBetrKV's, suggesting that our approach is likely to scale at least as well.

Accurately attributing the low verification latency to either Verus or our project structure is difficult. In all likelihood, Verus' efficient encodings and conservative triggering strategies played a very significant role in keeping the latencies low. We suspect that two key decisions in the project structure had a big impact as well. Due to Verus' lackluster performance with respect to nonlinear arithmetic (cf. Section 13.1.1), we took care to contain nonlinear reasoning in a select few lemmas (cf. Section 11.4.5), thus avoiding slow nonlinear reasoning in other lemmas. Additionally, constructing the implementation by refinement allowed us to keep the size of the proof steps manageable, resulting in simpler queries to Z3. It may even have been worthwhile to include an additional refinement step, which does not yet mutate memory but already includes a functional representation of the page table memory.

In summary, the vast majority of lemmas and definitions show very low verification latency, which is mostly dominated by the time spent in preprocessing steps. Support for incremental compilation in Verus could reduce the observed latencies significantly. Current verification latencies are low enough to support an effective verification workflow. We attribute this performance largely to Verus and two key decisions in our project structure.

[3]The paper gives no information on the used hardware. However, based on a conversation with its authors, we believe the performance of the used hardware is likely comparable to ours.

12.1.2 Proof Overhead

The entire page table project consists of approximately 7000 lines of code, excluding comments. These can be split as follows:

- Specifications, $\sim$ 600 lines
- Implementation, $\sim$ 500 lines
- Proofs, $\sim$ 5000 lines
- Imports, modules and other, remaining lines

The effective proof overhead, i.e. the ratio of proof code to executable code, is about 10:1. A large majority of the proof code deals with the implementation proofs. Only about 600 lines of proof are necessary to show that the OS state machine refines the application specification.

This overhead is somewhat larger than the 7:1 proof overhead reported for VeriBetrKV [20]. This increased overhead is not unexpected, as VeriBetrKV is implemented in Dafny, which employs more aggressive triggering strategies than Verus. Aggressive triggering can allow the verifier to take bigger proof steps without guidance. However, it is also more likely to cause problems due to excessive triggering. Part of VeriBetrKV's contributions are strategies for keeping Dafny's automation under control, while this has hardly been a concern in this project.

CreuSAT [50] represents another interesting data point in the design space. It is a SAT solver written in Rust and verified with Creusot [12]. At about 1000 lines of executable code, its implementation is roughly twice the size of ours. The reported proof overhead of 3:1 is significantly lower than ours. Verification of the whole project in a single-threaded configuration requires manual interaction and takes approximately 53 minutes while our proofs can be verified automatically within 70 seconds. However, CreuSAT is very different from this project both in application domain and the used verifier's mode of interaction. Hence, although the comparison is interesting, we refrain from drawing any conclusions.

Our proof overhead is not large enough to prevent our development approach from scaling to bigger projects. Other verified systems [30, 19], especially ones using interactive theorem provers, report overheads well north of ours.

12.2 Integration and Testing in NrOS

To integrate the verified functions into NrOS, we first erase all spec and proof code manually. Verus does not currently support outputting erased source code and how exactly Verus should support interfacing with verified code is an ongoing discussion[4].

The subsequent work of integrating the erased functions and adjusting them was done by Gerd Zellweger.

By adding a `Cargo.toml` manifest file, the verified paging project can be integrated into NrOS as a library. The functions were adjusted slightly so they match the behavior of the unverified functions. Specifically, the implementations differ in the following ways:

1. Default-`unmap` returns information on the unmapped mapping, verified-`unmap` does not.
2. Default-`resolve` returns flags for the translated address, verified-`resolve` does not.
3. Default-`unmap` unmaps a region, even if it is called with an address that is not the base address of the mapping. Verified-`unmap` returns an error.
4. Default-`map` succeeds when attempting to create the same mapping twice, without changing anything on the second invocation. Verified-`map` returns an error.

In the future, we will also adjust the verified implementation to match the first two behaviors. This should require minimal verification effort. For the third and fourth items, while currently required in certain places in NrOS, we have yet to decide how to change the verified implementation. Changing it to match these behaviors or writing alternative versions that support these behaviors are both viable options.

Subsequent testing uncovered two bugs in the implementation. The supervisor flag set by the `map` function was inverted. User-space mappings became supervisor-only and vice-versa. Additionally, `map` would always set new page directories to be supervisor-only. These behaviors are described by the page table walker specification, which is currently axiomatized to be equivalent to the implementation's interpretation function. A proper page table walk specification, as discussed in sectionSection 6.5, would have made it much easier to notice this misspecification. Notably, NrOS' test suite only caught the first of these bugs. The second one only became apparent due to it causing unexpected behavior. The test suite uses property-based

[4]https://github.com/verus-lang/verus/discussions/105

testing [8, 6], which is effective at checking that the page table's externally observable behavior matches that of a map but not as good at testing properties of its internal representation, which is necessary to ensure that the MMU's interpretation is what we expect.

Another issue encountered in testing was that the test suite attempted to map a memory region of size 512MiB. Unverified-`map` would create multiple mappings to cover this area, whereas verified-`map`'s preconditions require the region size to be 4KiB, 2MiB or 1GiB. By disregarding this precondition, the test suite's call caused verified-`map` to panic on an out-of-bounds array access. Mapping of entries that are not page- or huge-page-sized is used by NrOS in practice. To support this, a wrapper for verified-`map` now splits regions into smaller ones and maps them with individual calls, when necessary.

We draw two conclusions from this incident. First, whenever possible the interface exposed by a verified implementation should not rely on preconditions but rather check them when possible and return an explicit error if they are not satisfied. Using a verified system under conditions it was not verified for is a common failure mode [18]. Second, by verifying more of NrOS, we could extend correctness guarantees to a larger fraction of it and make this type of error less likely.

Except for these issues, the implementation passed all tests and is working correctly.

12.3 Performance

To evaluate our implementation's performance, we first compare the results of the existing NrOS benchmarks. To get a more direct comparison that excludes the node replication overhead, we then also do a direct comparison with a benchmark that measures the single-threaded performance of the verified and NrOS' unverified implementation.

12.3.1 NrOS Benchmarks

Gerd Zellweger ran NrOS' existing benchmarks and provided us with the data for the original, unverified implementation and the data for our verified implementation.

NrOS reported results for microbenchmarks of the paging functions. We compare the performance of verified-`map` and verified-`unmap` with that of NrOS' unverified-`map` and unverified-`unmap`. The NrOS paper includes comparable benchmarks for several other operating systems (Linux, Barrelfish, sv6). NrOS' unverified implementation compares favorably against Linux and Barrelfish but does not scale as well as sv6.

The map benchmark repeatedly maps the same 4KiB frame at sequential, page-aligned virtual addresses and measures the latencies for each of 100’000 operations.

The unmap benchmark repeatedly maps a 4KiB frame at the same address and then unmaps it, measuring the latency of the unmap operation, collecting 100’000 samples.

The benchmark was run on a system configured with two Xeon Gold 5120 processors with 14 cores each and a total of 192GiB of memory.

The latency in NrOS’ paging benchmark is dominated by the node replication mechanism. Hence, we expect to observe comparable numbers between the verified and unverified functions unless one implementation is considerably less performant. Our goal is not to outperform the unverified implementation on a single-threaded benchmark but to achieve sufficient performance to be able to replace it without degrading NrOS’ performance.

Figure 12.4(a) compares the latencies of the map functions, while (b) compares the unmap latencies. The minimum represents the first percentile, the maximum the 99th percentile, with the 25th, 50th and 75th percentiles shown as horizontal bars.

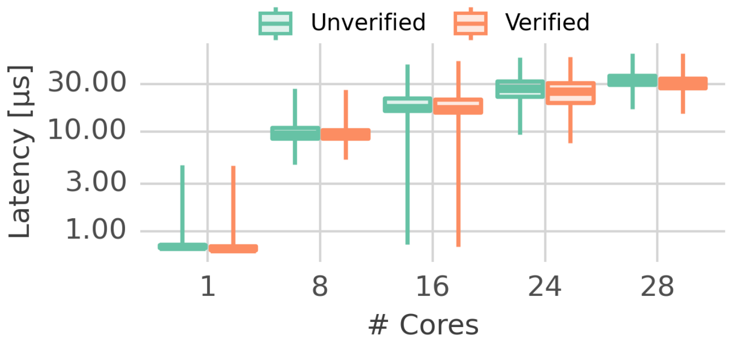
The verified map implementation appears to show marginally lower latency while that of verified unmap is marginally higher. Both operations show similar tail characteristics to their unverified counterparts, which is consistent with the observation that the latency is dominated by node replication.

The surprisingly low p1 latency in the 16-core benchmark is a result of NrOS’ architecture and the hardware’s characteristics. With 14 cores per socket, the 16-core benchmark causes the replicas to be imbalanced, one has 14 cores, while the other only has two, resulting in much lower latencies on the two-core replica.

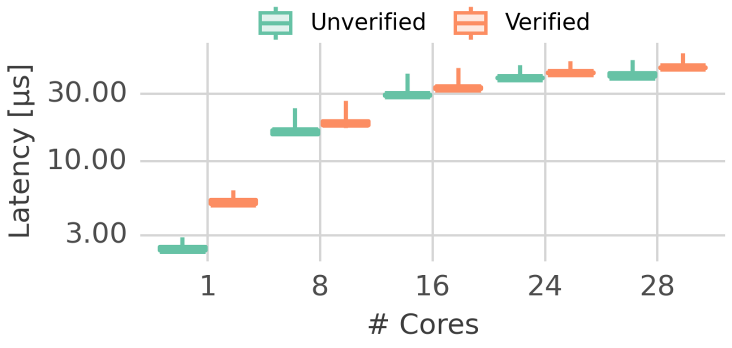
Verified-unmap’s slightly worse performance is likely due to it greedily checking directories for emptiness. The resulting difference is somewhat prominent in the single-core benchmark but largely disappears in the node replication overhead on higher core counts. This benchmark exercises verified-unmap’s worst-case behavior, as each measured unmap operation leaves the directories on every layer empty. We discuss a possible improvement in Section 14.3.

The benchmarks support the hypothesis that the verified implementation performs similarly well as the unverified one.

[5]The latency plots were generated with code provided by Gerd Zellweger.



(a) Map Latency



(b) Unmap Latency

Figure 12.4: Latency of unverified-`map` vs. verified-`map`[5]

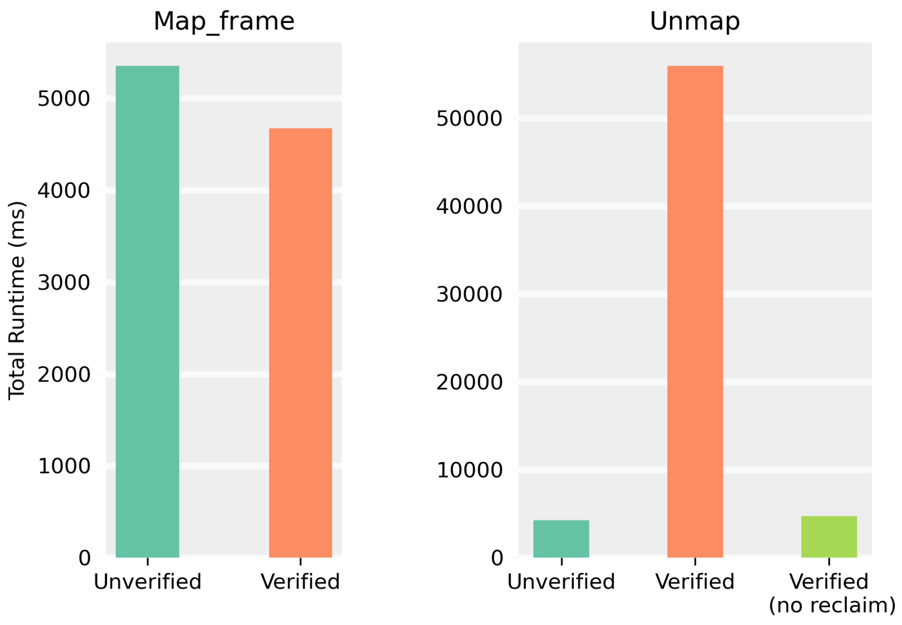


Figure 12.5: Runtimes of `map` and `unmap`

12.3.2 Single-Threaded Benchmarks

To compare single-threaded performance and to confirm or reject our hypothesis regarding the source of verified-`unmap`'s worse performance, we run the following three experiments.

1. Measure the time taken for verified-`map` and unverified-`map` to sequentially map a 4KiB frame 100'000'000 times.
2. Measure the time taken for verified-`unmap` and unverified-`unmap` to sequentially unmap 100'000'000 4KiB frames.
3. Modify verified-`unmap` to avoid checking directories for emptiness and reclaiming them. Then measure the time needed to sequentially unmap 100'000'000 4KiB frames with that function.

The first two experiments allow us to compare the performance of the verified and unverified implementations, while the third experiment tests our hypothesis regarding the performance impact of emptiness checking in verified-`unmap`.

We run these experiments on a Xeon E-2224G CPU with 8GiB of memory. We disable Intel Turbo Boost prior to running the experiments.

We show the results of these experiments in Figure 12.5. For unverified-`map` and verified-`map` we observe runtimes of 5341ms and 4661ms, corroborating our suspicion that the verified implementation is slightly faster.

At 55787ms, verified-`unmap`'s runtime is significantly higher than that of unverified-`unmap` at 4144ms. While this difference is larger than expected, we do expect verified-`unmap` to perform somewhat worse, due to its emptiness checking. As expected, verified-`unmap` with this mechanism removed exhibits a runtime of 4580ms, only barely slower than unverified-`unmap`.

These results suggest that it would be worthwhile to implement the optimization outlined in Section 14.3. However, the results of the NrOS benchmark indicate that in practice, even in its current state, verified-`unmap`'s performance is sufficient.

Chapter 13

The Verus User Experience

Part of this thesis' goal is to try out Verus in a reasonably large verification project. In this chapter we make some general observations on what we perceive Verus' strong and weak points to be with respect to this work. These observations are substantiated with supporting anecdotal evidence. We particularly focus on weak points and identify potential solutions to some of them. However, this chapter is *not* a comprehensive evaluation of Verus.

13.1 Background

We briefly explain the terms *nonlinear arithmetic* and *resource limit*, which are used in later sections.

13.1.1 Nonlinear Arithmetic

Z3 considers anything involving multiplication, division or modulo to be nonlinear arithmetic (*NLA*). Support for NLA can be enabled or disabled. With NLA disabled, Z3 is often unable to prove even the most trivial facts, such as `0 * a == 0`. With NLA enabled, Z3's behavior becomes less stable. This means that small, unrelated changes can significantly affect the time needed to verify an NLA query or even cause it to time out. For this reason, Verus normally runs Z3 without support for NLA. The NLA support is only enabled when the developer specifically requests it, e.g. by adding `by(nonlinear_arith)` after an assertion.

13.1.2 Resource Limit

The resource limit is a parameter that allows Verus to set a "timeout" on Z3 queries. The limit is not based on wall-clock-time but on an internal Z3 metric, which avoids the problem of non-deterministic verification failures.

13.2 Verification Latency

In its current form, Verus is able to verify the entire page table project in approximately 70 seconds on a laptop computer with an Intel i7-10510U CPU and 16GiB of memory. Over the course of the whole project, the required time for verification has never exceeded 120 seconds. However, setting Verus' resource limit to a higher value may have allowed Verus to verify the project rather than timing out in some cases.

We generally used the default resource limit, which is tuned to cause Z3 to time out within around two to ten seconds. Only when proving the postconditions of `map_frame`'s implementation did we feel the need to increase this limit.

A vast majority of functions and lemmas are verified very quickly, with only a few functions taking up the majority of the overall 70 seconds. Performance tends to be very good as long as no non-linear arithmetic is involved and the proofs are only concerned with spec mode definitions.

We perform a detailed analysis of the experienced verification latency in the evaluation chapter, in Section 12.1.1.

13.3 Caching

Verus does not currently support any caching of verification results. This means that all functions have to be re-verified on every invocation of Verus. In practice, the two command-line options `--verify-module` and `--verify-function` somewhat alleviate this problem by restricting verification to a single module or function.

However, managing these options is tedious and can easily lead one to accidentally not re-verifying everything necessary, when changing a definition. While this type of mistake is eventually caught during later re-verification of the whole project, it wastes the developer's time because they rely on a broken definition without realizing it.

A better solution would be to implement proof caching. During the course of this project we wrote a proposal for what such caching may look like [33].

The core of the idea is to only cache *exact* queries to Z3, e.g. by keeping hashes of queries and caching these with their results. However, this does not work out well with Verus' current approach to query construction.

When Verus generates queries for Z3, it includes all of the verified module's definitions as context, even if that context is not actually used in the query itself. Consider for example the following Rust code:

```
struct A {          struct B {
  a: usize,           b: usize,
}                   }

exec fn f_A(sa: A) -> (res: bool)
  ensures res == sa > 0
{ sa > 0 }

exec fn f_B(sb: B) -> (res: bool)
  ensures res == sb > 0
{ sb > 0 }
```

Verus generates two queries for this file, one to verify `f_A`'s postcondition, the other to verify `f_B`'s postcondition. Each query includes the full context, i.e. both struct definitions, as well as the function signature, pre- and post-conditions for the respectively other function. Any changes to `f_A` would also cause `f_B`'s query to change and thus exact caching would do nothing in this file.

Our proposed change is to perform a dependency analysis and only include the definitions a query (transitively) depends on. In the previous example, this would mean `f_B`'s query no longer includes either the `struct A` or `f_A` definitions and thus exact caching would work much better.

With this strategy, Verus could cache exactly those queries whose results cannot depend on any changes we made, except by inadvertently influencing Z3's heuristics.

13.4 Instability

Verus uses the heuristics-based Z3 SMT solver as a backend to verify its queries. As with any tool relying on heuristics, some degree of unpredictability is unavoidable. As a result, verification tools like Verus tend to experience *instability*.

We speak of instability when a previously succeeding proof step fails to verify or is verified significantly more slowly, due to seemingly unrelated changes.

SMT solvers use non-deterministic heuristics. To still guarantee deterministic verification results, the randomness seed is generated from the input file. Modifying the file in any way can change the seed and thus influence the heuristics unpredictably, thereby causing instability.

Instability often indicates that a proof step is too large. Adding assertions for intermediate proof steps will typically resolve the problem and make future instability of that proof step less likely.

During the work on this project, we rarely observed instability on "normal" proof steps. However, we found queries with NLA support enabled to be quite unstable, often finding it difficult to stabilize the proofs with the addition of more assertions. This state of affairs has improved significantly when Verus moved to a new iteration of Z3's nonlinear solver but some problems remain. Especially modular arithmetic causes very unstable behavior in Z3.

In one particular case, we also noticed that an unused lemma caused the verification of an unrelated proof step to slow down by 6x, seemingly being influenced by the unused lemma's triggers. It is not clear whether this may be a bug in either Z3 or Verus.[1] We provide detailed instructions to reproduce this behavior in Appendix A.1.

13.5 Proof Development

In this section we collect notes on missing or existing features of Verus that impact the efficiency of the proof development process.

13.5.1 Nonlinear Arithmetic and Bitvector Queries

As explained earlier, Verus disables NLA by default. The same is true for bitvector reasoning, which requires Z3 to be run with a different theory.

Both modes can be enabled for single assertions with `by(nonlinear_arith)` or `by(bit_vector)`. Unlike with normal assertions, the queries for such assertions do *not* import the surrounding context, as this would often cause Z3 to be overwhelmed and time out. The developer is intended to use these queries to prove implications, that allow the proofs in the surrounding context to succeed without any specialized reasoning. Assertions support the following precondition syntax to more easily write such implications.

```
assert(proof_goal) by(nonlinear_arith)
  requires
    precondition1,
    precondition2,
{ .. proof .. };
```

The careful minimization and isolation of reasoning within Z3's less stable or more specialized fragments means that verification outside of these queries is generally faster and more stable. The downside is that the developer has to manually identify the specific proof steps and the required context.

[1]cf. https://github.com/verus-lang/verus/issues/307

13.5.2 Finding and Using Suitable Lemmas Automatically

An important part of proof development is to factor out common proof steps into lemmas and making them reusable, thereby making future proofs easier. Most verification tools support some mechanism for automating the use of suitable lemmas.

Verus does not currently have any such mechanism, though one is planned.[2] Part of the proposed mechanism can be approximated by creating a lemma and calling it at the beginning of every proof or exec function.

The absence of such mechanisms has a detrimental impact on the developer experience. Proving a lemma in Verus generally consists of iteratively adding more assertions to narrow down a failing proof step. The result of this iterative process generally falls in one of three categories.

1. The proof failed due to a complex proof step. The developer proves an additional lemma or guides the verifier with assertions.
2. The proof failed due to a trivial proof step. Few additional assertions help it pass.
3. The proof failed due to a trivial NLA or bitvector reasoning step. A lemma call or inline specialized assertion helps it pass.

The first category is unavoidable in any verification work. The second and third categories are problematic precisely *because* it is a trivial proof step that fails. Tracking down the exact step requires the developer to manually perform time-consuming step-by-step reasoning without any direct insight into the proof state. A mechanism to automate the use of particular lemmas could help reduce the frequency with which outcomes two and three are encountered.

Verus also does not currently support searching for lemmas or even searching for proofs involving uncalled lemmas.

Tools similar to Isabelle/HOL's *find_theorems* [58] and *sledgehammer* [47, 5] could greatly improve a proof developer's efficiency. *Find_theorems* allows the developer to search for lemmas by their type. *Sledgehammer* attempts to automatically find a proof from lemmas available in the context, often with impressive effectiveness.

Verus does have some support for automatic proof failure localization with the `--expand-errors` feature. It also has a profiler, which can help in locating problematic triggers.

[2]https://github.com/verus-lang/verus/discussions/37

13.5.3 Generic Lemmas

Verus does not currently allow calling generic lemmas without specifying a type. For example the following lemma's postcondition asserts that some property `P` is true for all sets with elements of any type.

```
proof fn lemma<T>()
  ensures forall|s: Set<T>| P(s)
  { .. }
```

However, we can only call this lemma with a particular type, making the postcondition far weaker.

```
lemma();          // Error
lemma::<nat>(); // No error
```

We experienced the impact of this limitation when modifying existing proofs. In one particular case, we changed the return type of the paging functions. After the change, a somewhat large lemma failed to verify at a seemingly innocuous proof step. The problem turned out to be an earlier call to a generic lemma, instantiated with the old return type.

13.5.4 Mixed Triggering

SMT solvers rely on mechanisms called *triggering* and *E-matching* [40, 13] to decide when and how to instantiate quantifiers. For example, to prove `P(a)`, given the assumption `∀x. P(x)`, the solver needs to instantiate the bound variable `x` with the concrete variable `a`. If we specify the trigger `P(x)` for the quantified term, the solver will instantiate it whenever it encounters the term `P(x)` for some `x` in its context (i.e. in the known facts or the proof goal).

Verus attempts to automatically select triggers but warns the user when it is unable to do so or if it suspects the automatic trigger to be of questionable quality. The user can always manually annotate a trigger to be used. Selection of the best possible triggers is an essential aspect of good verification performance. Overly general triggers cause many instantiations.

Either function calls or arithmetic operations can be used as triggers, however, Verus does not allow triggers that contain both. For example, the trigger `P(x,Q(y))` is valid and the trigger `(x + y)` is valid, while the trigger `P(x + y)` is not. In this project, we have come up against this limitation a number of times. Sometimes we were forced to choose suboptimal triggers, other times we were unable to use certain quantified terms at all because they contained no valid, "pure" triggers.

There is a significant, technical reason for this limitation to exist. However, with the following examples we want to argue, that the limitation is problematic and that it should be a priority to find a solution for the problems it causes.

Examples

All the examples listed below are ones we have encountered during development, where we had to resort to unsatisfactory workarounds. Sometimes the only available workaround is to state the properties as lemmas with arguments and tediously instantiate the variables manually when calling the lemma.

Example 1: entry_base

In Section 11.4.1 we introduce the `entry_base` function.

```
spec fn entry_base(self, idx: nat) -> nat {
  self.base_vaddr + idx * self.arch.entry_size(self.layer)
}
```

For this function we would like to prove the following property.

```
forall|i: nat|
  self.entry_base(i + 1)
    == self.entry_base(i) + self.arch.entry_size(self.layer)
```

The trigger for this term must contain the quantified variable `i`. Because mixed triggers are not allowed, the only valid trigger is `self.entry_base(i)`. However, this trigger causes a triggering loop. Whenever the term is instantiated for some variable `i`, the resulting fact contains the term `self.entry_base(i + 1)`, which again matches the trigger and causes an instantiation.

The invalid trigger `self.entry_base(i + 1)` would be highly specific and unlikely to cause performance problems.

Example 2: Triggering Loop in Alignment Lemma

```
forall|a: nat, b: nat, c: nat|
  aligned(a, c) && aligned(b, c) && c > 0 ==> aligned(a + b, c)
```

This term's only valid trigger is the combination `[aligned(a, c), aligned(b, c)]`. As in the previous example, this causes a triggering loop and the mixed trigger `aligned(a + b, c)` would be very specific.

Example 3: Impossible Alignment Lemma

The following property cannot be stated at all because it contains no valid triggers.

```
forall|a: nat, b: nat, c: nat|
  aligned(a, b * c) && c > 0 ==> aligned(a, b)
```

A trigger would have to cover all three quantified variables.

Chapter 14

Future Work

In this chapter, we outline some areas of future work.

14.1 Complete Proofs

Currently, a small number of proofs still rely on unproved reasoning steps. Most notably, the fact that the concrete `unmap` implementation refines the corresponding function on the intermediate refinement layer RL1, is largely unproved.

We intend to complete the outstanding proofs of these reasoning steps. We believe most of these proof steps to be reasonably straightforward.

Before proving the refinement of `unmap`, we will want to reconsider the proof strategy we used for the `map_frame` refinement, as that function has been the only one where we started regularly running into Z3 timeouts. We discovered that verifying just one branch of the outermost if-statement at a time greatly improves the performance,[1] suggesting that (possibly optional) changes to Verus' encoding of if-statements may alleviate the problem.

Further, we may experiment with using linear types instead of explicit framing arguments to simplify the proofs [36].

14.2 Improved Specifications

14.2.1 Application Specification

In Section 5.3, we discuss the possibility of simplifying the application specification by exposing less implementation detail in it.

[1]See Appendix A.2 for instructions to reproduce the observed behavior.

14.2.2 Page Table Walks

In Section 6.5, we outline an approach to specify the page table walk semantics in a much simpler way. Currently, we reuse the fairly large page table interpretation functions from the implementation. A smaller specification of this semantics would make it easier to audit. When testing our implementation we discovered two issues in the specification; Both of them were in the complex interpretation function.

14.3 Efficient Entry Count in Paging Directories

To check whether a directory is empty, our implementation iterates over its entries until it finds a non-empty entry. This approach is inefficient and likely responsible for the slightly worse performance of the verified `unmap` function compared to its unverified counterpart.

We intend to replace this with a more efficient check, implemented by storing the number of non-empty entries for a given page directory in unused bits. Specifically, per the Intel SDM [10], entries in directories at any layer ignore bits 9, 10 and 11. Thus, we can store a directory's entry count in these bits of its first three entries.

14.4 Concurrency

Our specifications and proofs assume a uniprocessor model and do not account for concurrency. The implementation is constructed to be correct on a multiprocessor, assuming the addition of a TLB shootdown protocol and a concurrency mechanism such as the node replication in NrOS.

Hance et al. [21] have ported this node replication mechanism to Dafny and verified its correctness. We intend to port this verified implementation to Verus and integrate it with our verified paging to create a paging implementation that is verified to be correct on multiprocessor hardware.

This task will require significant changes to the specifications and proofs. In the following sections, we sketch some of the changes this will require and the challenges it entails.

14.4.1 Atomic Transitions

By implementing the interface specification, we prove that each of the implemented functions corresponds to the right transition in the page table state machine. However, the implementation may write to memory multiple times in that transition. Before moving to a fully concurrent model, we may relax this assumption of atomic transitions to first introduce concurrency with the MMU's page table walks.

Relaxing this assumption requires us to adjust the page table memory interface. The current page table memory struct is axiomatized as having a view of its managed memory. The transitions are defined in terms of the view before a paging function is executed and the view after the function returns. The modified page table memory's view will include a history of the performed memory operations, allowing us to expose them at sufficient granularity. Due to the transactionality of our paging implementation (cf. Section 8.3), we will be able to show that the operation becomes visible to an observer as soon as all constituent memory writes are visible to them.

At this point we may also want to assume a realistic memory semantics for the page table memory and extend our proofs.

14.4.2 Memory Semantics

Our current model assumes an oversimplified memory semantics that treats the memory as if it was a map. When extending this memory semantics to a realistic one, we have two choices. We can either integrate a model of the memory semantics such as x86-TSO [45, 46] or we can underspecify the semantics, only allowing conclusions about the memory contents to be made after the use of particular synchronization primitives such as memory barriers or serializing or atomic instructions.

Because we split the page table memory and application memory into two separate components, we also have the option of choosing to use different memory semantics for each one. Our primary goal is to verify the paging functionality. Thus, an accurate memory semantics for the page table memory is important to ensure we allow all possible interactions with the MMU's page table walks. The application memory's semantics is less important. If we kept the current map semantics for the applications, *technically* our proofs would only provide guarantees if all memory accesses were atomic. However, this is sufficient to make the argument that our implementation correctly manages the page table data structure. For use in larger verified systems built on top of this specification, an accurate memory semantics would likely be important.

Chapter 15

Conclusion

In this thesis, we present an implementation of paging functions for the x86-64 architecture, programmed in Rust and formally verified for functional correctness using the Verus Rust verification tool. Insights due to this project and bugs discovered and reported during its development have made Verus a more robust and mature tool. This work is the largest body of Verus-verified code to date and the first to apply Verus in the verification of a real system.

Our proof development is carefully separated into trusted and untrusted components, making our claims verifiable. The proofs are complete, except for a few small assumed proof steps that we postpone due to lack of time.

We evaluate the typical proof latencies experienced during development as well as the final proof overhead and compare it to other projects' reported data. We find that the evidence supports the hypothesis that our development approach will scale to larger projects, though we identify some aspects of Verus where there is still room for improvement.

Integration of our verified paging functions into the NrOS research operating system shows that, except for two cases where we set inverted flags, the paging functions function correctly. The flag inversion bugs were not caught in verification due to a mistake in a part of the specification that we know to be unnecessarily complex and intend to replace in future work. We encountered a further bug due to the OS violating a precondition of the `map_frame` function, which leads us to conclude that in a future iteration we should explicitly check preconditions in functions that are exposed to interaction with unverified code. We use NrOS' existing benchmarks to compare the paging functions' performance to NrOS' unverified implementation. The two implementations perform similarly well, making the future replacement of the unverified implementation a realistic outcome.

In future work, we will resolve the few remaining proof steps and we intend to make a number of improvements to the existing specifications and implementation. Further, we hope to extend the specifications and proofs to the weaker assumption of multiprocessor hardware with realistic memory semantics.

The whole artifact discussed in this thesis is available online.[1]

[1]https://github.com/matthias-brun/verified-paging-for-x86-64-in-rust

Appendices

Appendix A

Instructions for Reproducing Unexpected Verifier Behavior

This appendix contains instructions for reproducing certain verifier behaviors discussed in other parts of this thesis.

The necessary code is available online. [1] We refer to this repository as the *artifact repository*.

We use the variable `$RUST_VERIFY` to refer to the path of Verus' `rust-verify.sh` script and `$VNRK_PATH` to the location of the artifact repository.

The results we show here resulted from running the instructions on a laptop computer with an Intel i7-1051U processor and 16GiB of memory.

A.1 Unused Lemma Causing Slow Verification

We use Verus at commit 5356d52d3bae0e8abd9e1e2aea6102d77b24d7b5, the artifact repository at the newest commit on the branch *timeout-lemma* and Z3 version 4.10.1.

We first run the command

```
$RUST_VERIFY --time --arch-word-bits 64
  --verify-module pt_impl::l1
  $VNRK_PATH/page-table/main.rs
```

and observe the following output:

```
verification results:: verified: 50 errors: 0
total-time:           37246 ms
    rust-time:             2544 ms
```

[1] https://github.com/matthias-brun/verified-paging-for-x86-64-in-rust

```
        init-and-types:         1458 ms
        lifetime-time:          1086 ms
        compile-time:              0 ms
    vir-time:                427 ms
        rust-to-vir:              82 ms
        verify:                  336 ms
        erase:                     3 ms
    air-time:               1266 ms
    smt-time:              33004 ms
        smt-init:                  0 ms
        smt-run:               33004 ms
```

Then we delete lines 43–48 in the file `$VNRK_PATH/page-table/pt_impl/l1.rs`, which should be the following ones:

```
#[verifier(external_body)]
proof fn slow_down_everything()
    ensures
        forall|d: Directory, base: nat, pte: PageTableEntry|
               d.inv() && #[trigger] d.accepted_mapping(base, pte) ==>
               d.interp().accepted_mapping(base, pte);
```

Then we rerun the previous command and observe that the verification time is much shorter:

```
verification results:: verified: 50 errors: 0
total-time:           18637 ms
    rust-time:               2526 ms
        init-and-types:         1445 ms
        lifetime-time:          1081 ms
        compile-time:              0 ms
    vir-time:                399 ms
        rust-to-vir:              87 ms
        verify:                  302 ms
        erase:                     3 ms
    air-time:               1240 ms
    smt-time:              14466 ms
        smt-init:                  0 ms
        smt-run:               14466 ms
```

The single largest contributor to the changed verification time is a specific function. With the following command we verify just that one function.

```
$RUST_VERIFY --time --arch-word-bits 64
  --verify-module pt_impl::l1
  --verify-function Directory::lemma_interp_of_entries_disjoint
  $VNRK_PATH/page-table/main.rs
```

With lines 43–48 present, we see the following output:

```
verification results:: verified: 3 errors: 0
total-time:           21466 ms
    rust-time:             2567 ms
        init-and-types:        1481 ms
        lifetime-time:         1086 ms
        compile-time:             0 ms
    vir-time:               288 ms
        rust-to-vir:             83 ms
        verify:                 196 ms
        erase:                    2 ms
    air-time:               382 ms
    smt-time:             18224 ms
        smt-init:                 0 ms
        smt-run:              18224 ms
```

With the lines removed, we see this output:

```
verification results:: verified: 3 errors: 0
total-time:            5844 ms
    rust-time:             2555 ms
        init-and-types:        1474 ms
        lifetime-time:         1081 ms
        compile-time:             0 ms
    vir-time:               250 ms
        rust-to-vir:             84 ms
        verify:                 157 ms
        erase:                    3 ms
    air-time:               347 ms
    smt-time:              2687 ms
        smt-init:                 0 ms
        smt-run:               2687 ms
```

A.2 Faster Separate Verification of If-else Branches

We use Verus at commit 5356d52d3bae0e8abd9e1e2aea6102d77b24d7b5, the artifact repository at the newest commit on the branch *slow_if_split* and Z3 version 4.10.1.

We first run the command

```
$RUST_VERIFY --rlimit 100 --time --arch-word-bits 64
  --verify-module impl_u::l2_impl
  --verify-function PageTable::map_frame_aux
  $VNRK_PATH/page-table/main.rs
```

and observe the following output:

```
verification results:: verified: 1 errors: 0
total-time:           84402 ms
    rust-time:             4732 ms
        init-and-types:        2647 ms
        lifetime-time:         2084 ms
        compile-time:             0 ms
    vir-time:               650 ms
        rust-to-vir:            152 ms
        verify:                 480 ms
        erase:                    5 ms
    air-time:               341 ms
    smt-time:             78672 ms
        smt-init:                 0 ms
        smt-run:              78672 ms
```

We then insert the line

```
assume(false);
```

after line 956 in the file `$VNRK_PATH/page-table/impl_u/l2_impl.rs`.

We then rerun the previous command and see the following output:

```
verification results:: verified: 1 errors: 0
total-time:           13656 ms
    rust-time:             4749 ms
        init-and-types:        2698 ms
        lifetime-time:         2050 ms
        compile-time:             0 ms
    vir-time:               493 ms
        rust-to-vir:            153 ms
        verify:                 322 ms
        erase:                    5 ms
    air-time:               385 ms
    smt-time:              8021 ms
        smt-init:                 0 ms
        smt-run:               8021 ms
```

We then remove the line we added and instead insert the line

```
assume(false);
```

after line 1176 in the file `$VNRK_PATH/page-table/impl_u/l2_impl.rs`.

We then rerun the previous command and see the following output:

```
verification results:: verified: 1 errors: 0
total-time:            6600 ms
    rust-time:             4679 ms
        init-and-types:        2626 ms
        lifetime-time:         2053 ms
        compile-time:             0 ms
    vir-time:               420 ms
        rust-to-vir:            154 ms
        verify:                 248 ms
        erase:                    5 ms
    air-time:               296 ms
    smt-time:              1198 ms
        smt-init:                 0 ms
        smt-run:               1198 ms
```

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA):147:1–147:30, 2019.

[3] Ankit Bhardwaj, Chinmay Kulkarni, Reto Achermann, Irina Calciu, Sanidhya Kashyap, Ryan Stutsman, Amy Tai, and Gerd Zellweger. NrOS: Effective replication and sharing in an operating system. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 295–312. USENIX Association, 2021.

[4] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation lookaside buffer consistency: A software approach. In Joel S. Emer and John L. Hennessy, editors, *ASPLOS-III Proceedings - Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, April 3-6, 1989*, pages 113–122. ACM Press, 1989.

[5] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016.

[6] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon S3. In Robbert van Renesse and Nickolai Zeldovich, editors,

SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 836–850. ACM, 2021.

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for NUMA architectures. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 207–221. ACM, 2017.

[8] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[9] Verus contributors. Verus. https://github.com/verus-lang/verus. [Online; accessed 2022-09-22].

[10] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2022.

[11] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[12] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verication of rust programs. In *ICFEM 2022 - 23th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, Madrid, Spain, October 2022. Springer.

[13] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[14] Richard J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (psos). *AFIPS Conference Proceedings*, 1979.

[15] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating*

Systems Principles, Shanghai, China, October 28-31, 2017, pages 287–305. ACM, 2017.

[16] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[17] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics*, 19:19–32, 1967.

[18] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 328–343. ACM, 2017.

[19] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 653–669. USENIX Association, 2016.

[20] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 99–115. USENIX Association, 2020.

[21] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Title tbd. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023*. USENIX Association, 2023. to appear.

[22] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[23] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[24] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *CoRR*, abs/2206.07185, 2022.

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[26] Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. Thread modularity at many levels: a pearl in compositional verification. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 473–485. ACM, 2017.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018.

[28] Steve Klabnik and Carol Nichols. The rust programming language. `https://doc.rust-lang.org/stable/book/`. [Online; accessed 2022-09-24].

[29] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, 2014.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[31] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[32] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[33] Andrea Lattuada. Discussion 147, incremental proofs. https://github.com/verus-lang/verus/discussions/147. [Online; accessed 2022-09-29].

[34] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[35] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, 2009.

[36] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–28, 2022.

[37] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux KVM hypervisor. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1782–1799. IEEE, 2021.

[38] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[39] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.

[40] Michal Moskal, Jakub Lopuszanski, and Joseph R. Kiniry. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.*, 198(2):19–35, 2008.

[41] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[42] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In Tim Brecht and Carey

Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 225–242. ACM, 2019.

[43] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 252–269. ACM, 2017.

[44] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[45] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009.

[46] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO (extended version), 2009.

[47] Lawrence C. Paulson. Sledgehammer: some history, some tips. https://lawrencecpaulson.github.io/2022/04/13/Sledgehammer.html. [Online; accessed 2022-09-29].

[48] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. Promising-arm/risc-v: a simpler and faster operational concurrency model. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1–15. ACM, 2019.

[49] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. Relaxed virtual memory in armv8-a. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 143–173. Springer, 2022.

[50] Sarek Høverstad Skotåm. CreuSAT: Using Rust and Creusot to create the world's fastest deductively verified SAT solver. Master's thesis, University of Oslo, 2022.

[51] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.*, 4(OOPSLA):152:1–152:31, 2020.

[52] Hira Taqdees Syeda. *Low-level program verification under cached address translation*. PhD thesis, University of New South Wales, Sydney, Australia, 2019.

[53] Hira Taqdees Syeda and Gerwin Klein. Formal reasoning under cached address translation. *J. Autom. Reason.*, 64(5):911–945, 2020.

[54] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 866–881. ACM, 2021.

[55] The Coq Development Team. The coq proof assistant. https://doi.org/10.5281/zenodo.5846982, January 2022.

[56] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.

[57] Aaron Weiss. Oxide: The essence of rust. *CoRR*, abs/1903.00982, 2019.

[58] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, 2021.

[59] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[60] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 99–110. ACM, 2010.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verified Paging for x86-64 in Rust

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Brun	Matthias

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Winterthur, 1.10.2022	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.