Leveraging Rust for Lightweight OS Correctness

Ramla Ijaz
ramla.ijaz@yale.edu
Yale University

Kevin Boos
kevinaboos@gmail.com
Theseus Systems

Lin Zhong
lin.zhong@yale.edu
Yale University

Abstract

This paper reports our experience of providing lightweight correctness guarantees to an open-source Rust-based OS, Theseus. First, we report new developments in intralingual design that leverage Rust's type system to enforce invariants at compile time, trusting the Rust compiler. Second, we develop a hybrid proof approach that combines formal verification, type checking, and informal reasoning. By lessening the strength of correctness guarantees, this hybrid approach *substantially* lowers the proof burden. We share our experience of applying this approach to the memory subsystem of Theseus, demonstrate its utility, and quantify its reduced proof effort.

1 Introduction

Correctness is a desirable yet challenging property to achieve for complex systems software like an operating system (OS). A key technology for correctness is formal verification. In recent years, various formal verification approaches have emerged that make different tradeoffs between expressiveness and proof burden, i.e., what can be proven vs. how difficult it is to generate those proofs. However, even so-called *push-button* approaches [25, 26, 28, 29, 33] still suffer from significant proof burdens, even for very small software systems and for proving very limited invariants, mainly restricted to the decidable portion of first-order logic.

In this paper, we present our recent experience in exploring new ways to ensure OS correctness. Towards achieving high expressiveness with low proof burden, we relax the *strength of correctness guarantees*. We observe that while full formal verification is desirable, the type system of the implementation language, i.e., Rust, combined with informal reasoning can also be used to provide weaker yet distinctly useful guarantees.

In §3, we further develop the idea of intralingual design introduced by Theseus OS [8], extending its reach with new techniques. Intralingual design uses language-level features to enforce invariants with the compiler. We analyze its limits, showing that the invariants presented in [8] were based on an incomplete foundation. While Rust's ownership model guarantees that an instance of a linear type has a single owner, it cannot guarantee there is no unexpected overlap between the actual values of two separate instances of the same linear type. When a linear type instance represents a system resource, an overlap would let users access the same resource concurrently using different instances. This shortcoming led to an insidious bug in the memory subsystem of Theseus, for which we reported and contributed a solution.

Therefore, we advocate a hybrid approach (§4) that combines intralingual design, formal verification, and informal reasoning to achieve *lightweight* correctness for OSes. We deem it lightweight not only because it requires much less effort compared to conventional formal verification, but also because its strength of guarantee is weaker due to its use of informal reasoning and trust of the implementation language.

In §5, we report our experience applying this approach to a revision of the memory subsystem of Theseus, which provides stronger guarantees of the original invariants and eliminates several classes of insidious bugs. We also share our experience using burgeoning formal verification tools for Rust programs, specifically Prusti [6]. Our experience in §6 shows that the hybrid approach has low development burden and negligible performance overhead.

In summary, this paper makes three contributions:

- We present a new set of techniques to expand the scope and extend the reach of intralingual design.
- We describe a hybrid proof approach that combines type checking (as used by intralingual design), formal, and informal reasoning for improving OS correctness.
- We report our experience applying this low-effort approach to the memory subsystem of Theseus.

2 Background and Related Work

Singularity [14] popularized linear types in OS design and used them for zero-copy sharing of heap memory across software-isolated domains. Recent works have used Rust's linear type system[1] for features such as lightweight fault isolation [24], zero-copy communication [27], compiler-checked session types [15], and decentralized resource management [8]. The authors of [7] also discussed the potential of using Rust for static information flow control and automatic program state manipulation. Linear Dafny [19] used linear types as a complementary approach to SMT solving and removed memory reasoning from verification conditions. In contrast,

[1]Strictly speaking, Rust employs an *affine* type system, which relaxes the restrictions of true linear type systems to also permit weakening.

KISV '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0411-6/23/10.
https://doi.org/10.1145/3625275.3625398

our work designs and implements the OS such that the Rust type system can provide guarantees that go beyond safety, in collaboration with formal verification.

In building the Verve OS [32], Yang and Hawblitzel creatively combined formal verification and a safe implementation language by dividing the OS into a lower core *Nucleus* and a higher kernel. They applied formal verification to the Nucleus, implemented in assembly, for safety and correctness, while relying on the implementation language (C#) for the kernel's safety. Our hybrid approach does not aim for a clear split between the verified and unverified portions of the OS. Instead, we use a combination of proof techniques in whichever subsystem we aim to prove an invariant about, pairing the correctness property with the proof technique best suited to it.

Many have leveraged Rust's type system to ensure correctness beyond just safety issues. We next overview these ideas before developing them further in §3.

Linear Types for Pairwise Operations: Many operations must always occur in pairs, e.g., memory allocation/deallocation, lock acquisition/release, and reference count increment/decrement. Manually keeping track of pairwise operations is difficult since the second operation can occur much later and in different parts of the code than the first. Mismatchings of such pairwise operations are common in the Linux kernel [20, 21, 31]. This problem can be solved by using linear types, placing the first pairwise operation in the constructor and the second in the destructor. Rust itself follows this design pattern when providing types for heap-allocated data structures (`Vec<T>` [4]), locks (`MutexGuard<T>` [3]), and reference-counted pointers (`Arc<T>` [2]).

Linear Types as Unforgeable and Unique Capabilities: In Rust, an instance of a linear type is a *unique* and *unforgeable* capability as long as it does not implement the `Clone` trait, meaning it cannot be duplicated [17, 24]. The ownership of such an instance automatically confers the right to use it without the need for runtime checks of its authenticity [10, 24]. Since the capability has a linear type, it has a single owner, and we can use Rust's built-in ownership rules to prevent data races and automatically insert resource cleanup code. In §3.1, we take inspiration from this idea by using linear-type instances as representations of OS resources.

Linear Types for Statically-Enforced State Machines: A linear type system can prevent incorrect state machine transitions at compile time by implementing the state machine using behavioral type techniques, e.g., typestates and session types. When combined with linear types, a typestate protocol can be statically validated [10], imposing no runtime overhead.

3 Intralingual Design

Intralingual design [8] aims to maximize the compiler's role in enforcing correctness by leveraging programming language features, namely type systems, to more precisely convey system requirements to the compiler. In other words, it *encodes the requirements* into the implementation itself, such that they can be enforced by the compiler. Many requirements cannot be so encoded because the type system is not expressive enough to convey them. We categorize these requirements as *extralingual*. Our objective herein is to build upon the foundation provided by recent works (§2) and introduce a *systematic methodology* for incorporating linear types and other type-based techniques into low-level system design.

```
// Pages is a representation of a range of virtual pages.
struct Pages<S: State> {
   range: RangeInclusive<usize>
}
// The possible states a Pages instance can be in.
enum State {
   Free,
   Allocated,
   Mapped,
   Unmapped
}
// Only Pages in the Mapped state have memory access functions.
impl Pages<Mapped> {
   pub fn write(&mut self, data: [u8]);
   pub fn read(&self) -> &[u8];
   fn unmap(self) -> Pages<Unmapped>;
}

impl<S: State> Drop for Pages<S> {
   fn drop(&mut self) {
      match S {
         State::Free => {
            // Re-take ownership of the pages by replacing it
            // with an empty range; return it to page allocator.
            let pages = replace(&mut self, Pages::empty());
            free_pages_list.insert(pages);
         }
         State::Mapped => {
            // PTE(s) have been cleared, so we transition
            // the Pages to the Unmapped state.
            let pages = replace(&mut self, Pages::empty());
            pages.unmap(); // Drop the returned Pages<Unmapped>
        } ...
      }
   }
}
```

Listing 1. An example of using an IRS to manage virtual memory. A `Pages` instance is a representation of a range of pages, which can be in one of four states. When a `Pages<Mapped>` instance is dropped, it is eventually returned to the `Free` state and stored in the list of free pages. The code has been simplified for brevity/readability.

3.1 Intralingual Representation System

We present an Intralingual Representation System (IRS) that shifts some of the responsibility of managing system resources from the OS at runtime into the compiler. Typically, an OS creates software objects to represent physical or abstract resources, e.g., objects of `struct page` representing physical frames in Linux. The IRS combines the representation of a resource with the authority to use it, via linear types: the ownership of the linear-type instance denotes the sole authority to use the resource. We call this instance a representation of the resource. Representations in an IRS are checked by the compiler, which means that (*i*) there is only ever one mutable reference to the representation at a time, and (*ii*) access to the representation is governed by the rules conveyed via the type system. Strongly-typed languages like Rust already use linear type instances as a limited form of representations for memory objects, but an IRS extends this to apply representation types to arbitrary system resources beyond just memory.

Changing Access Rights via Typestates. In an IRS, the access rights of a representation are defined by the publicly-visible methods of its type. Each distinct set of access rights is represented by a typestate, and the access rights of the representation change upon state transitions. A state transition method takes a representation as input, *consumes* it, and changes its state. For example, in Listing 1, a `Pages` instance is a representation that can be in one of four states: `Free`, `Allocated`, `Mapped`, or `Unmapped` (L6); its methods transition the representation between these states, e.g., `unmap()` in L16. The compiler can enforce that the representation is accessed in accordance to the restrictions of its current state. For example, in the `Free`, `Allocated`, and `Unmapped` states, page table entries (PTEs) are not set up for the given pages, so the `Pages` representation cannot be used to access the underlying memory range. This is statically enforced by implementing the `read()` and `write()` methods *only* for `Pages` in the `Mapped` typestate (L14).

Delegation via Ownership Transfer, Sharing, Borrowing. In an IRS, a representation is a singleton instance with either one exclusive owner or multiple owners that can only mutably access it through a mutual exclusion mechanism. Both cases uphold the Rust invariant of only one mutable reference to a type instance existing at once. An owner can conveniently delegate authority by granting access to the representation in one of three ways: (*i*) transferring ownership to a new owner who gains full exclusive access rights, (*ii*) sharing ownership via a reference-counted smart pointer so multiple parties can jointly co-own the representation, or (*iii*) temporary (scoped) lending to a borrower that can access the representation through a reference.

Returning Representations via Automatic Destructors. We can shift the complex responsibility of inserting correctly-ordered cleanup sequences into the compiler by placing all cleanup code in a linear type's destructor (a Rust `Drop` handler). This is important for representations that represent physical resources (e.g., physical frames) that should never be destroyed: these representations must be returned to the OS for future use. With typestates, a representation can have multiple drop handlers, one for each state; each state's drop handler undoes any changes made when entering that state, reverting the representation to its previous state. The drop handler for the initial state finally returns the instance back to the OS for storage. For example, in Listing 1, the drop handler for `Pages` in the `Mapped` state removes the PTEs and converts it to the `Unmapped` state (L28). Then, each predecessor state's drop handler is iteratively invoked until the `Pages` instance returns to the `Free` state, upon which the `Pages<Free>` instance is returned to an redblack-tree of free page chunks maintained by the page allocator (L22).

Representation vs. Capability. The notion of a representation may remind readers of that of a capability. Like a capability, a representation is also *unforgeable* and *delegable*. Unlike a capability, a representation is *unique* in that no two representations exist in the system for the same resource. This precludes *derivation* in which multiple copies of a capability, with varying access rights, exist at the same time. Importantly, language-level mechanisms are not sufficient to implement all features of a capability system as reported by [23, 30]. Redleaf [24] uses instances of linear types to implement fine-grained access control between isolated domains, but doesn't attempt to build a full-featured capability system.

3.2 Linear Types as Proof of Work

The other main way we use linear types is to indicate that a certain function has been executed, by returning a dedicated type instance from the function. In this manner, a linear type instance no longer represents a *spatial* resource (as in typestates), but rather a proof of a *temporal* action having occurred. A linear type used for this purpose is simply a type that can only be instantiated by a single function that performs the required "work". To prevent instances of this type from being created anywhere, we make sure it is composed of a *private* inner type that is inaccessible outside of the type's module.

Combining *linear types as a proof of work* with strongly-typed function interfaces can statically enforce an order between "stages" of operations. That is, we can create a chain of functions where one function creates and returns an instance of a linear type to be consumed by the next function, effectively requiring each instance of a linear type to be used in the order it is created for progress to be made. In fact, this pattern of instantiation and then consumption lies at the core of many ways in which we use linear types (§2). The difference here is in what the type instance represents.

For example, the Page Table Entry (PTE) `unmap` function of Theseus returns an `UnmapResult` instance which stores information about physical frames that have just been unmapped. `into_unmapped_frames()` recreates a representation for those frames by consuming the `UnmapResult` instance as a proof that those frames are now free. It can be invoked much later than when the PTE is cleared, and the frames representation will not be recreated until then, ensuring its uniqueness. `UnmapResult` here is not a representation to the newly unmapped physical frames. It does not exist in multiple states, and provides no access to any resource, memory in this example. Instead, it is a single-use type instance that represents the clearing of PTEs, and so the typestate pattern is not applicable here.

3.3 Intralingual Hardware Abstraction Layers

An intralingual Hardware Abstraction Layer (HAL) enforces datasheet-provided rules for communicating with a hardware device without any runtime checks. We use a `struct` to represent the layout of memory-mapped I/O (MMIO) registers and other I/O data structures, which can then be overlaid atop a region of memory. This ensures that every register and bitfield is always accessed in a type-safe manner at its correct offset (and alignment) within the underlying memory region. Our approach precludes the unsafe pointer arithmetic commonly used to access MMIO registers, which cannot be reasoned about by the compiler.

Using type-system techniques, we can further prevent incorrect reads and writes to MMIO, as listed below:

1. Type wrappers on `struct` fields can enforce volatile access and read-only or write-only restrictions.
2. Fields marked as reserved have private visibility in the `struct`, rendering them inaccessible outside the HAL.
3. Only fields where every bit is accessible, without any restrictions, are publicly visible.
4. Fields with restrictions on which values can be written to them are only accessible through `struct` methods. An `enum` encodes the set of valid values and must be passed as the method argument, forbidding arbitrary raw values at compile time.
5. Using linear types as a proof of work, we can statically enforce an order of operations between successive reads and writes of different registers or even between disjoint bitfields of the same register.

3.4 Limitations of Intralingual Design

Intralingual design, while powerful, has many shortcomings stemming from its reliance on the language's type system. First, it is not as expressive as many formal verification techniques due to the limited invariants that can be enforced by the type system. Generally, it is incapable of proving any algorithmic property, e.g., that a `sort()` function actually performs sorting.

Moreover, it also cannot reason about unrestricted types: where they originated from or the values they store. This limitation is fundamental in an OS, where in order to interact with hardware, the lowest layers must use built-in unrestricted data types, i.e., raw or primitive types, such as unsigned integers, e.g., `u32` in Rust. The type system alone cannot provide any guarantees about the correctness of values read from or written to the lowest layer of raw primitive types.

Finally, the IRS design uses a linear-type instance to represent an OS resource. *A linear type system itself cannot guarantee uniqueness of the resource represented.* This goes beyond the intralingual (type-level) uniqueness based on Rust's ownership model: no two variables of the same linear type can own the same value (memory object), but there is no guarantee that the resources represented by the values (memory objects) do not overlap. If there is an overlap, then multiple instances of this type can give access to the same resource, i.e., the overlapping parts. This limitation underlies our discovery of an important bug in Theseus's memory subsystem, discussed in §5.4.

4 Hybrid Approach for Correctness

To overcome the limitations of intralingual design and the high proof burden of formal verification, we argue for a hybrid approach that pairs a correctness property with a proof technique. When choosing the proof technique, we consider the tradeoff between proof effort, strength of guarantee, and performance. Our hybrid approach combines formal verification, type checking (used by intralingual design), manual code review, and prose proofs. We employ SMT-based formal verification for Rust, i.e., Prusti [6], for only select properties, while embracing informal reasoning, especially prose proofs, to prove higher-level invariants. Prose proofs can "stitch" together the invariants proven by different proof techniques (and those specified in different languages) without requiring a unified formal specification; avoiding this requirement significantly lowers the proof burden. Within the prose proof, we use natural language to express the invariants proven with each technique and justify how these invariants combine together to imply a high-level invariant.

Given that a linear type system itself cannot guarantee uniqueness of the resource represented by an instance, and that such uniqueness is the foundation of an IRS (§3.1), the uniqueness of a linear-type instance is an excellent candidate for formal verification. To formally verify that a linear-type instance is unique, we only need to formally verify that the type's constructors will never create instances that represent overlapping resources.

4.1 Minimize Proof Burden

Because formal verification is the main source of proof burden in our hybrid approach, we leverage the type system to bootstrap formally-verified results in the following ways: (*i*) An instance of a composite type is unique if its members are unique. Therefore, it is unnecessary to formally verify the instance uniqueness of every representation in an IRS. Instead, by verifying the instance uniqueness of select basic types in the system, e.g., `Pages` in Listing 1, we can rely on the type system to ensure the instance uniqueness of a type composed from these basic types. For example, a representation of a device's MMIO registers is composed of a `Pages<Mapped>` instance, and so is proven unique without formal verification using this property. (*ii*) By formally verifying generic code, the verified property is also true for its specific instances. In addition to generic data structures, we can also use generics when verifying methods of types that represent similar resources.

4.2 Intralingual Specifications

Both intralingual designs and prose proofs may make type-based assumptions. We leverage Rust's language mechanisms to specify these assumptions so that they can be checked at compile time. The type-based assumptions that occur frequently, in our experience, include but are not limited to: (*i*) a type does not implement certain traits, (*ii*) a type is composed of another type, (*iii*) a function consumes an instance of a type, (*iv*) a type's inner fields are private. For example, instances of the `Pages` type in Listing 1 need to be unique. A proof of this uniqueness requires that the type is linear and doesn't expose its inner fields. These requirements can be coded by not implementing the `Clone` and `DerefMut` traits for `Pages`, and by setting its `range` field to private, respectively. A change in the codebase, e.g., a heedless developer implementing `Clone` for `Pages`, could inadvertently break the proof of uniqueness. Since these changes do not violate Rust typing rules, the compiler cannot catch them.

To check such type-based assumptions at compile time, we employ static assertions that can be organized in a separate file from the code base. These assertions generate code that aborts compilation if the assertion is not satisfied. For example, to prevent implementation of traits for the `Pages` type, we can add this line to Listing 1:

```
    assert_not_impl_any!(Pages: DerefMut, Clone);
```

These static assertions constitute an intralingual specification of type-based assumptions.

4.3 Performance vs. Verification Effort

Because type checking happens at compile time, most intralingual design ideas do not incur any runtime overhead. However, an IRS design can cause runtime overhead if we push intralingual design to its extreme so that the maximum amount of code is written such that its correctness can be reasoned about by the compiler. In such a design, *all* physical resources are represented at their finest granularity using linear type instances that are never destroyed (dropped), and the OS stores these instances (representations) in a data structure. This design is not always practical due to the additional time to store and search for a representation, and because a representation may need to be destroyed if the resource it represents no longer exists, e.g., hot-plugging of memory. In the former case, the overhead can be avoided if we decide the additional proof effort to recreate a representation after it is dropped is acceptable. In the latter case, we would additionally verify that a representation's drop handler updates system bookkeeping state so that the representations can be recreated if a device is plugged back in.

For example, a completely intralingual page table design would use a linear `PTEs` type, an instance of which owns the memory that hosts a set of page table entries (PTEs) and also the `Frames` instance representing the frames mapped by those PTEs. The `map/unmap` functions would search a data structure for the `PTEs` instance using type-safe code understood by the compiler. This design increases memory use due to storing all representations as fine-grained software objects and incurs extra cycles due to searching for `PTEs`.

The design we settled on for Theseus avoids these overheads at the cost of additional verification effort. We represent a page table as a single linear-type instance and not as a collection of PTEs; the extralingual page table walk to find the PTE is part of the HAL. The mapping procedure forgets `Frames<Mapped>` instances to avoid the overhead of searching for them when unmapping. Instead, the function that unmaps and clears a PTE returns an `UnmapResult`, which acts as *proof of work* that the frames have been unmapped; then, a verified function recreates the `Frames` instance from the `UnmapResult`.

5 Case Study: Theseus Memory Management

We next report our experience of applying intralingual design and the hybrid approach to the memory subsystem of Theseus.

5.1 Intralingual Design of the Memory Subsystem

We redesigned portions of the memory subsystem of Theseus in accordance with intralingual design patterns (§3). The functions to walk the page table and update PTEs are categorized as part of the intralingual HAL of the memory subsystem; a PTE is always manipulated through a type-safe interface. We use an IRS to create a `Frames` and `Pages` type, instances of which are the unique representation of a region of physical or virtual memory, respectively. With typestate programming, we create four possible states for instances of the `Frames` and `Pages` types: `Free`, `Allocated`, `Mapped`, and `Unmapped`. `Pages<Mapped>` instances (which represent accessible memory) are used according to the rules of the Rust type system: one mutable reference or multiple immutable references, which corresponds to a single writer or multiple readers to the underlying memory, but not both at the same time. A detailed code example of the `Pages` type is given in Listing 1.

5.2 Bijective Mapping Invariant

The bijective mapping invariant of Theseus states:

Invariant 1 (Bijective Mapping)**.** *Each page in the virtual address space of the system can only be mapped to one frame of the physical address space, and vice versa.*

This invariant can be equivalently restated as "a frame can only be present in a single PTE". This prevents extralingual aliasing, which is necessary to provide memory isolation and safety for *all* system memory, not just the built-in Rust support for safe heap and stack memory. *Isolation in Theseus without the use of hardware address spaces relies upon this invariant always being upheld.*

The memory subsystem of Theseus only adds a PTE when creating a `Pages<Mapped>` instance, and only removes it when dropping the same instance. Thus, proving the bijective mapping invariant necessitates proving the correct construction and destruction of a `Pages<Mapped>` instance.

5.3 Proof Sketch

We use our hybrid approach for correctness (§4) to present a prose proof of the bijective mapping invariant, wherein we explicitly state where each proof technique is used. We list the Lemmas required to prove the invariant, and next to each Lemma we list the techniques used to prove it: formal verification (**F**), the type system (**T**), or manual checks (**M**). We use a prose proof (**P**) to tie multiple techniques together. Our proof [1] is based on the following four Lemmas.

Lemma 1. `Frames` and `Pages` instances are unique. **[F, T, M, P]**
Lemma 2. A `Pages<Mapped>` instance can only be created by taking ownership of *both* a `Pages<Allocated>` instance and a `Frames<Allocated>` instance. **[T]**
Lemma 3. PTEs can only be manipulated through the memory subsystem's `map` and `unmap` functions. **[T, M, P]**
Lemma 4. Creating a `Pages<Mapped>` instance adds PTEs *only* for the given pages and frames; dropping it removes them. **[M]**

Lemma 1 ensures that, at the time of creation of a `Pages<Mapped>`, the representation of the pages and frames that are about to be mapped are unique. Lemma 2 ensures that the pages/frames cannot be used for any other mapping while this mapping exists. Lemma 3 ensures that a mapping cannot be created in a way the circumvents the guarantees provided by `Pages<Mapped>`. Lemma 4 ensures that software state accurately reflects hardware state, such that the compiler can enforce invariants about page table contents. We do not reason about TLB shootdowns in our proof, though `unmap()` does send them to prevent the caching of invalid mappings in hardware that are not represented in software (Lemma 4).

5.4 Proof of Lemma 1: Bug Revealed

The proof of Lemma 1 greatly increases the strength of guarantee of the bijective mapping invariant. To prove this Lemma, we introduce the linear type `Chunk`, a non-`Clone`able wrapper around `RangeInclusive<usize>`, and formally verify all methods that create or mutate a `Chunk` to ensure any two `Chunk` instances from the same allocator cannot have overlapping ranges. We make its inner field private to prevent mutable access outside of its methods, and manually check that a `Chunk` instance is only mutably accessed in its verified methods. These joint properties of `Chunk` prove that every instance is unique. Both `Frames` and `Pages` are composed of only the `Chunk` type; a static assertion checks that this composition relationship always holds. Consequently, `Frames` and `Pages` are unique via composition, ensured by the Rust language – an example of how the type system can extend formally-verified invariants to other types (§4.1).

The original Theseus relied on manual checks in place of formal verification to prove Lemma 1 because uniqueness is beyond the scope of intralingual design. In the frame allocator bookkeeping code, we discovered a bug that led to the instantiation of overlapping `Frames` instances, violating the bijective mapping invariant. This bug only manifested in a very particular code path that had not occurred in the four years that the frame allocator code had been in use. It stalled network driver development for over one month, motivating us to explore ways to incorporate formal verification into an intralingual system.

5.5 Coping with Prusti Limitations

We chose Prusti [6] as our verification tool. Compared to other tools, such as Verus [18], Prusti is more mature in development and documentation [5].

When we initially wrote the verified portions of the frame and page allocators, we had to overcome certain limitations of Prusti to better balance verification effort and system performance, as discussed in §4.3. One such limitation was that Prusti did not support comparison operations for user-defined types in function specifications. This posed an issue because a major strength of Rust lies in its zero-cost abstractions, often realized through the newtype wrapper pattern. Rather than maintain two sets of identical data, one as a primitive type we could use in verified code and the other as a newtype used within the rest of Theseus, we implemented our own comparison operators for the newtype. This increased our code size, but we were able to verify code at a higher level of abstraction and without the overhead of maintaining two copies of the same data. Another limitation of Prusti was its incomplete support for Rust generics. Our first attempts to implement generic data structures that could be used within both the page and frame allocator resulted in Prusti panics. We had to duplicate code within our allocators, increasing our code size and verification times.

After completing a first implementation with our workarounds, we reported our findings to the Prusti developers, who then fixed these issues in a timely manner. This confirmed our understanding that these limitations were not fundamental to our approach or to Rust-based verification tools, but rather a result of using a tool still under active development. After applying the provided fixes, we were able to easily use newtypes and generic data structures within our code, allowing for verified code reuse and reducing our verification effort. For future work, we will further consolidate the verified code to create a generic chunk allocator that can be used by both the `Pages` and `Frames` allocation code.

6 Evaluation

We evaluate the performance of the new memory subsystem of Theseus and quantify the verification effort. Our objective is to find any performance differences caused by code

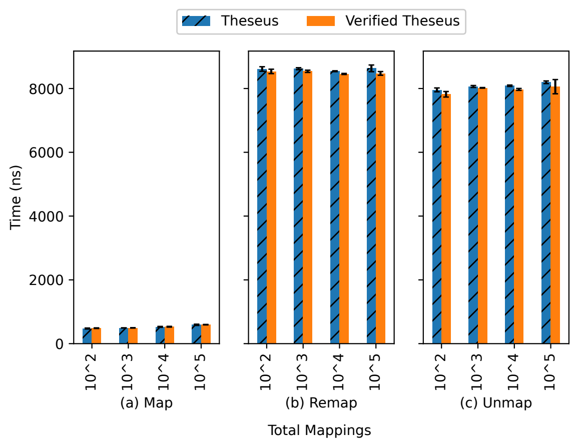

Figure 1. The individual time to map, remap, and unmap a 4 KiB page does not increase when verification is added to Theseus. The results presented are the mean times for 1 page, with the error bars representing the standard deviation.

changes resulting from the verification process. Additionally, we assess the proof effort needed to formally verify sections of the frame/page allocator to gauge the "lightweightedness" of the hybrid approach.

6.1 Performance comparison

We found that Theseus with formally-verified code (*Verified Theseus*) performed similarly to the original Theseus that had no verified code. We ran two memory subsystem microbenchmarks on an Intel(R) Xeon(R) Gold 6252N CPU at 2.30 GHz with hyperthreading disabled. The first microbenchmark was a Rust version of LMBench's [22] memory map. In this benchmark, a 4 KiB page is mapped, written, and unmapped 100,000 times. Both versions of Theseus showed identical performance, a mean time of 1.99 μs with a standard deviation less than the timer period (42 ns). The second microbenchmark was taken from the original Theseus paper [8], which separately measures the time to map a page, remap it, and then unmap it, with an increasing number of mappings. Figure 1 shows no significant difference between the two versions.

6.2 Verification Effort

Table 1 reports the size (in SLOC) and verification times for the Theseus memory subsystem described in §5. We find that the proof effort is magnitudes lower than end-to-end formally verified systems, and verification times are within minutes. All verification times were measured using the Prusti 2023-08-22 release running on Ubuntu 20.04 on an Intel Core i7-1260P CPU at 2.5 GHz.

The formally-verified portion of the memory subsystem consists of four parts: *(i)* Prusti external specifications for types from the Rust core library, *(ii)* generic data structures, *(iii)* portions of the frame allocator code that include methods

	Spec	**Impl**	**Proof**	**Verification Time**
		(SLOC)		(s)
External Spec				
`Option`	48	0	0	0
`Result`	33	0	0	0
`PartialOrd`	41	0	0	0
`RangeInclusive`	24	0	0	0
Data Structures				
Linked List	38	167	0	16.695
Static Array	22	91	3	18.22
Frame Allocator	120	520	7	48.428
Page Allocator	117	520	7	52.162
Total	443	1298	17	135.505

Table 1. Code size and verification times for formally-verified portions of the frame and page allocator. The proof SLOC are the loop body invariants that are manually added.

to create and modify a `Chunk`, *(iv)* portions of the page allocator code, nearly identical to that of the frame allocator. We can further reduce our code size by consolidating the page and frame allocator using generics as discussed in §5.5.

Proof Effort: Our proof-to-implementation ratio is `1:76`. This is lower than the `10:1` proof-to-implementation ratio of an end-to-end verified page table implementation written in Rust [9], by two orders of magnitude. Other fully verified works reported proof-to-implementation ratios ranging from five to twenty [11–13, 16].

This minimal proof effort is a direct result of our hybrid approach, and comes at the cost of lower strength of guarantee and a larger TCB. We only use SMT verification for one property (uniqueness), and use other techniques to reason about the correctness of writes to the page table.

Maintainability: We only need to re-verify our code if a change occurs within the chunk allocator module. In our experience, it is easy to experiment with the design of the chunk allocator since the verification time is so low, around two minutes. Most importantly, changing any code outside of the chunk allocator module *does not require re-verification*.

Acknowledgments

This work is supported in part by NSF Awards #2130257 and #2112562. The authors are grateful for the constructive feedback from the reviewers and the support from the Prusti team.

References

[1] Prose Proofs for Invariants in the Theseus OS. https://github.com/Ramla-I/theseus_proofs. Accessed: 2023-10-07.

[2] Struct std::sync::Arc. https://doc.rust-lang.org/std/sync/struct.Arc.html. Accessed: 2023-08-02.

[3] Struct std::sync::Mutex. https://doc.rust-lang.org/std/sync/struct.Mutex.html. Accessed: 2023-08-02.

[4] Struct std::vec::Vec. https://doc.rust-lang.org/std/vec/struct.Vec.html. Accessed: 2023-08-02.

[5] Verus Tutorial and Reference. https://verus-lang.github.io/verus/guide/. Accessed: 2023-08-04.

[6] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. Leveraging Rust types for modular specification and verification. In *Proc. ACM OOPSLA*, 2019.

[7] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in Rust: Beyond safety. In *Proc. Workshp. Hot Topics in Operating Systems (HotOS)*, 2017.

[8] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *Proc. USENIX OSDI*, 2020.

[9] Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. Beyond isolation: Os verification as a foundation for correct applications. In *HotOS*, 2023.

[10] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[11] 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 *Proc. USENIX OSDI*, 2016.

[12] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proc. ACM SOSP*, 2015.

[13] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. USENIX OSDI*, 2014.

[14] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 2007.

[15] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *Proc ACM SIGPLAN Wrkshp. Generic Programming*, 2015.

[16] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proc. ACM SOSP*, 2009.

[17] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter:bare-metal extensions for multi-tenant low-latency storage. In *Proc. USENIX OSDI*, 2018.

[18] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. In *Proc. ACM OOPSLA*, 2023.

[19] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. Linear types for large-scale systems verification. In *Proc. ACM OOPSLA*, 2022.

[20] Jian Liu, Lin Yi, Weiteng Chen, Chengyu Song, Zhiyun Qian, and Qiuping Yi. LinKRID: Vetting imbalance reference counting in linux kernel with symbolic execution. In *Proc. USENIX Security*, 2022.

[21] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun Shi. RID: finding reference count bugs with inconsistent path pair checking. In *Proc. ACM ASPLOS*, 2016.

[22] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.

[23] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[24] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *Proc. USENIX OSDI*, 2020.

[25] 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 *Proc. ACM SOSP*, 2019.

[26] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proc. ACM SOSP*, 2017.

[27] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. USENIX OSDI*, 2016.

[28] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button verification of file systems via crash refinement. In *Proc. USENIX OSDI*, 2016.

[29] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *Proc. USENIX OSDI*, 2018.

[30] Thorsten Von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A capability-based operating system for Java. *Secure Internet programming*, 1999.

[31] Chao Xu, Xiaozhu Lin, Yuyang Wang, and Lin Zhong. Automated OS-level device runtime power management. In *Proc. ACM ASPLOS*, 2015.

[32] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. ACM PLDI*, 2010.

[33] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *Proc. ACM SOSP*, 2019.