A Formal Specification of the NOVA Microhypervisor's ABI

HOANG-HAI DANG, BedRock Systems, Inc, Germany
DAVID SWASEY, BedRock Systems, Inc, USA
PAOLO G. GIARRUSSO, BedRock Systems, Inc, Germany
GREGORY MALECHA, BedRock Systems, Inc, USA

We present a formal specification for the NOVA microhypervisor that handles concurrency and architectural behaviors. Our specification combines an operational specification for unprivileged user code with a separation logic specification of privileged state and operations. We find that the small footprint and open world nature of separation logic makes the specification highly modular and reasonably high level. Furthermore, we describe several uses of the specification for verifying applications that use NOVA.

Additional Key Words and Phrases: Separation logic, Operating system specifications, Iris, NOVA

1 INTRODUCTION

NOVA [32, 31] is a microhypervisor that provides basic services for virtualization, isolation, scheduling, and the management of physical resources. NOVA's microkernel-based design provides highly-efficient, low-level mechanisms and leaves policy decisions and higher-level functionality to be developed on top of it in user mode. At BedRock Systems, we aim to build a modern, trustworthy computing stack atop NOVA through the pervasive use of formal methods. Achieving this goal requires a rich, formal specification of NOVA. Several key requirements of this specification include:

(1) **Two-sided** [1]. It must be usable not only as the top-level specification of NOVA but also as the low-level specification for user-mode applications running atop NOVA. In particular, we aim to prove strong safety properties of user mode libraries and programs including behavioral refinements.
(2) **Robustly Safe** [11]. It must enable fine-grained reasoning about user-mode applications running atop NOVA in the presence of *arbitrary*, *untrusted*, and *malicious* user-mode code (without looking into the NOVA implementation).
(3) **Modular**. It must account for the behavior of *arbitrary* hardware interacting with the system mediated by security-relevant input-output management units (IOMMUs/SMMUs)[1].
(4) **Realistic**. It must support highly concurrent and parallel hardware, properly capturing subtle interleavings exposed by the implementation. Crucially, NOVA hypercalls are rarely atomic, and interleaving hypercalls can lead to subtle behaviors that the specification must account for.[2]

In this technical report, we present our approach for specifying the behavior of NOVA in the presence of *arbitrary* user code. We use a combination of operational semantics and separation logic. While the individual components that feed into this setup are not novel, their combination for an operating system specification differs from previous operating system specification and verification approaches [18, 12].

We focus more on the NOVA object and capability systems, and less on the details of the memory subsystem, *e.g.*, page tables and memory accesses, which are more provisional at this point. We also leave the handling

[1]We assume that hardware devices account conform to standard specifications, e.g. for PCI devices, but otherwise make no assumptions about their underlying behavior.
[2]Ultimately we aim to account for weak memory behaviors that are pervasive on modern platforms [24, 29, 28], though we currently leave this as future work.

Authors' addresses: Hoang-Hai Dang, BedRock Systems, Inc, Germany, hai@bedrocksystems.com; David Swasey, BedRock Systems, Inc, USA, david@bedrocksystems.com; Paolo G. Giarrusso, BedRock Systems, Inc, Germany, paolo@bedrocksystems.com; Gregory Malecha, BedRock Systems, Inc, USA, gregory@bedrocksystems.com.

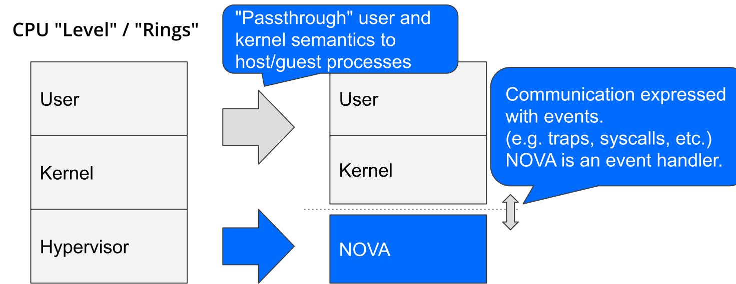

Fig. 1. The NOVA machine (CPU) abstracts and enriches physical CPU behavior by replacing the architectural specification of "hypervisor mode" with a "NOVA mode". The semantics of code running above NOVA is the same as it is on the architecture except when the architecture needs access to hypervisor mode state. When this access is necessary, the NOVA machine describes the semantics in terms of the NOVA state.

of weak-memory behaviors as future work. We aim for this document to be self-contained, but restricting our presentation to the kernel objects and hypercalls already precludes us from covering all components in detail. Therefore we focus on the approach and the specification idioms and reference concrete examples that illustrate them.

1.1 Approach

Modern hardware architectures are built around the notion of security levels with controlled behavior to privileged state. Some behaviors are freely programmable, *e.g.*, a fault from unprivileged code can result in a transfer of control to a hypervisor. Others behaviors are dictated by the hardware, *e.g.*, the address translation that occurs during memory accesses uses privileged state to convert virtual addresses to physical addresses. A system-level abstraction, such as NOVA, must unify both of these behaviors, providing a higher-level, uniform interface to the underlying hardware resources. Figure 1 demonstrates how NOVA effectively replace the architecture's state and behaviors in hypervisor mode with abstractions where instructions are run in privileged mode with higher-level state and behaviors.

Fully embracing the separation logic paradigm, *we build a specification of the NOVA abstract machine using ghost resources and weakest preconditions.* Foregoing any purely operational characterization of NOVA, we express the NOVA state as first-order predicates in separation logic. Leveraging separation, the specification is highly modular, allowing us to develop portions of the NOVA state incrementally and maintain them as the NOVA machine evolves.

We use separation logic weakest preconditions to describe the behavior of NOVA in terms of its logical state. The logical, as opposed to functional, characterization based on weakest preconditions builds in powerful programming features such as partiality, non-determinism, sequencing, and even general recursion "for free". Finally, powerful features such as invariants and logical atomicity [5, 33, 16] provide a pattern for expressing atomicity in an interleaving semantics.

To connect the NOVA behavior to threads that NOVA runs (called *execution contexts*, ECs), *we connect this separation model of NOVA to an operational model of the processor that is partitioned according to privilege levels.* Unprivileged steps in this partitioned model are codified as silent steps, and privileged state is accessed through events. The NOVA specification describes the interpretation of these privileged events using weakest preconditions expressed in separation logic. For example, when the user code performs a hypercall (syscall) event, the weakest precondition parses the event and dispatches to the WP for the appropriate hypercall. Similarly, memory access events are interpreted as WPs that use NOVA's Memory space predicates (NOVA's abstraction of page tables) to

translate virtual addresses to physical addresses. Note that while the trap handlers for hypercalls ultimately run NOVA code (which is compiled from its C++ and assembly sources), the "code" that runs on a memory access is dictated by the hardware architecture. Because these steps are encapsulated by the NOVA abstract machine, reasoning about these steps becomes part of the NOVA proof and gives rise to many interesting proof obligations about *concurrent* page table access.

To connect this event-handler-style specification to a machine semantics, we specify NOVA's entry point as a higher order function (§6). This specification, which is expressed as a simple entailment, can then be chained with other entailments to derive full system properties. Connecting with adequacy of the underlying machine logic enables us to extract properties that are independent of separation logic (§6.2). Further connecting to refinement proofs of user-mode applications enables extracting whole-machine refinement proofs (§7.1).

1.2 Structure

The structure of this technical report is as follows.

- §2 provides brief background on NOVA [32, 31, 30].
- §3 presents how we reflect NOVA's kernel objects into separation logic assertions.
- §4 presents our logically atomic specifications of hypercalls.
- §5 presents how we reflect user code semantics into separation logic weakest preconditions, `wp_nova_ec` and `wp_nova_dev`. We discuss how the weakest preconditions interact with the assertions in §3 and the specifications in §4, as well as with NOVA's abstractions for memory mappings (page tables) and device mappings.
- §6 sketches the specification of NOVA's "main" function, that is, how the NOVA's boot process turns raw physical resources into NOVA's abstract resources.
- §7 elaborates on user verification using the NOVA specifications, in particular, how one can prove robust safety and refinements.

1.3 The Coq Development

This technical report accompanies the Coq development of the NOVA formal specification. The released formal specification focuses on the most stable parts that are crucial for verifying user-mode programs running on top of NOVA. We will refer to various definitions with their Coq counterparts, in order to help with finding them in the Coq development. The development comes with a README file (`README.md`) that describes the structure in detail. There is also extensive documentation in the Coq files.

To facilitate exploration, the specification is organized using the following directory setup.

- `machine_logic/` (`bedrock.nova_interface.machine_logic`) describes the assertions from the machine-level logic that the NOVA formal specification assumes, such as `byte_at` for ownership of a single byte in physical memory (§5.2).
- `model/` (`bedrock.nova_interface.model`) encodes various NOVA types in Coq, for example kernel object types and information, permissions, status codes, hypercall arguments, and so on.
- `opsem/` (`bedrock.nova_interface.opsem`) contains the interface of CPU architectural semantics that the NOVA formal specification assumes (see §5.1).
- `predicates/` (`bedrock.nova_interface.predicates`) contains the assertions of the NOVA state and their properties (see §3.1).
- `hypercall/` (`bedrock.nova_interface.hypercall`) contains the hypercall specifications (see §4).
- `wp_nova/` (`bedrock.nova_interface.wp_nova`) provides the weakest-preconditions for user code, *i.e.*, `wp_nova_ec` (§5.2) and `wp_nova_dev` (§5.3).

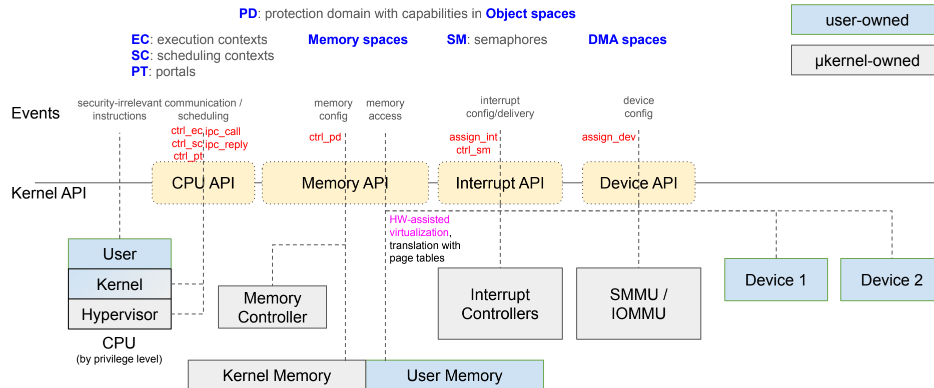

Fig. 2. NOVA takes ownership of the security critical components of the system and exposes them to userspace through kernel objects. Other components of the system, *e.g.*, hardware devices and memory regions, are passed through to userspace to configure and manage.

It is worth reiterating that NOVA itself is under active development, and so is the formal specification. Many features are still work-in-progress, including the specifications of address translations and page table updates (*i.e.*, ctrl_pd for NOVA Memory spaces), the specifications for device mappings and lookups, as well as the handling of weak-memory behaviors. In some places in the code we provide provisional sketches to show how these holes could be filled, but we caution readers that these specifications are often built around general abstractions, and will therefore fail to account for many of the lower-level details in the real system.

2 BACKGROUND ON NOVA

Following the microkernel design philosophy, NOVA only takes control of critical system resources and re-exposes them to clients under restricted APIs. Figure 2 gives an overview of this strategy. Once booted up (by the bootloader), NOVA will control

- all of the machine's cores (CPUs) in hypervisor mode,[3]
- a subset of the main memory (DRAM) for its kernel operations (such as its own disk images, and 2nd-stage page tables), and
- several system devices including memory management units (MMUs), interrupt controllers, and input-output controllers (SMMUs/IOMMUs).

NOVA's API is built around five types of kernel objects.

- A Protection Domain (PD) is a unit of protection and isolation for kernel objects. A PD manages the capabilities to create and control other kernel objects.
- Execution Contexts (ECs), Scheduling Contexts (SCs), and Portals (PTs) are used to execute, schedule, and support communications of user applications.
- Semaphores (SMs) are used to configure interrupts (and as a concurrency primitive).
- Memory spaces and DMA spaces manage the capabilities to set up permissions in page tables, repsectively for main memory and devices.

[3]NOVA also owns kernel mode for host execution contexts.

These kernel objects are accessible to user programs through capabilities [7] that are stored in NOVA spaces. These capabilities include fine-grained permissions and can be delegated between spaces to enable highly modular, least privilege designs.

NOVA APIs provide hypercalls (syscall) to create (create_{pd,ec,sc,sm,pt}) and interact with (ctrl_{pd,ec,sc,sm,pt}) kernel objects as well as operations to configure hardware (*e.g.*, assign_int and assign_dev). Other non-critical resources, such as the CPUs running in user mode, user memory (which can be used to store 1st-stage page tables), and non-criticial devices are passed through by NOVA and hence are controlled directly by user code. Consequently, a user application running atop NOVA sees that the semantics of physical machine cores (in user mode) are extended with a set of hypercalls that impose strong access control to critical physical resources.

3 REFLECTING THE NOVA STATE

We present some of the core predicates that we use to expose NOVA state in the formal specification, assuming familiarity with various concepts from separation logic and the Iris framework [15, 16]. We derive these predicates from the NOVA API (informal) documentation [31] which describes the various kernel objects, their state, and the relationships between them. These assertions can be collectively referred to as simply “NOVA state” and can be partitioned into three main categories:

(1) assertions for reflecting the properties of a kernel object (§3.1),
(2) capabilities for managing permissions on kernel objects (CAP_{OBJ}, §3.1.1), and
(3) capabilities for managing physical system resources exposed by NOVA (§3.1.2): for memory (CAP_{MEM}), for input-output (CAP_{PIO}), and for Model-Specific Registers (CAP_{MSR}).

As a guiding philosophy, our NOVA specification focuses on the mechanisms that NOVA provides, not the idiomatic usages of these mechanisms. Following this philosophy means that our predicates generally reflect the first-order state described in the NOVA documentation [31]. We will discuss some of the benefits of this approach in more detail in §7.

3.1 Kernel Objects

The NOVA state is comprised of *logically disjoint* kernel objects each bundling its own data and supporting its own operations. Each kernel object is given a distinct object identifier (`kobj_id`) and its data is represented as a group of separation logic assertions indexed by the identifier. For example, if `sm` is the kernel object identifier of a semaphore, then the state of the semaphore is captured by disjoint assertions for its value and wait queue which is expressed by the formula `nova.sm.value sm n ∗ nova.sm.queue sm ecs`. In the following, we use meta-variables *pd*, *ec*, *sc*, *pt*, and *sm* in place of *obj* if we know the object's type already.

Table 1 shows a selection of the predicates that the NOVA interface exposes for each type of kernel object. The data attached to each kernel object can be classified as configuration, i.e. immutable data initialized at object creation time, and state, which is mutable over the lifetime of the object.

Aside: Destroying Kernel Objects. As with other separation logics for garbage collected systems, the NOVA specification treats kernel objects as persistent, *i.e.*, once a kernel object is created it is never *logically* destroyed. This choice simplifies reasoning about immutable properties of objects; however, it does not preclude the NOVA implementation from reclaiming these resources. The obligation of the NOVA interface is to ensure that every operation is implementable, but once objects are no longer reachable, the underlying resources can be safely released. In the proof, the ghost state assertions exposed in the NOVA specification would still be available to clients, but the NOVA proof will decouple the physical resources of the objects from their logical representation thus permitting the implementation to reclaim the resources.

Predicate		Description
Common (PD)		
`nova.spc.info obj i`	P	the common configuration of `obj` is `i`
Protection Domains (PD)		
`nova.pd.owner pd pd_parent`	P	`pd_parent` is the parent PD of `pd`
`nova.spc.pd.{obj,hst,pio} pd s`	P+	`s` is the corresponding Object, Host or PIO space of `pd`
Execution Contexts (EC)		
`nova.ec.cpu ec c`	P	the `ec` is tied to the physical CPU number `c`
`nova.ec.kstate ec k`	Q	the continuation of `ec` is `k`
`nova.ec.call_stack ec ecs`	Q	the call stack of `ec` is `ecs`
`nova.ec.utcb_pa ec pa`	P	the physical address of `ec`'s UTCB is `pa`
`nova.ec.recall ec b`	Q	a recall exception flag is set on `ec` with value `b`
`nova.ec.reply ec c Q`	Q	`ec` needs to reply (in an IPC) with the post-condition `Q`
`nova.ec.scs ec scs`	E	records the list of SCs that are currently bound to `ec`
`nova.ec.regs is_guest ec q r`	Q	represents architecture-specific register state of `ec`
`nova.ec.ctrl_regs ec q r`	Q	represents architecture-specific control register state of `ec`
`nova.spc.ec.{obj,hst} ec s`	P	`s` is the Object or Host space of `ec` (host ECs only)
`nova.spc.ec.{gst,pio,msr} ec q s`	Q	`s` is the Guest, PIO, or MSR Space of `ec` (VCPUs only)
Scheduling Contexts (SC)		
`nova.sc.time sc t`	P	the `sc` has run for *at least* `t` ticks
Portal (PT)		
`nova.pt.id pt q id`	Q	the `pt` has ID `id`
`nova.pt.mtd pt q m`	Q	the `pt` has MTD `m`
`nova.pt.ec pt ec`	P	the `pt` is bound to `ec`
Semaphores (SM)		
`nova.sm.kindI sm k`	P	the `sm` has type `k`
`nova.sm.value sm n`	E	the `sm` has value `n`
`nova.sm.queue sm ecs`	E	the `ecs` are waiting for the `sm`

Table 1. Selected predicates that represent ownership of kernel object state. The middle column shows whether the assertion is exclusive (E), fractional (Q), persistent (P), or persistent after initialization (P+).

State. The NOVA interface exposes mutable state through fractional [4] or exclusive predicates. For example, the predicate `nova.spc.ec.gst ec q s` records the current Guest Memory space `s` of `ec`, which can change over time.

```
gst_fractional : ∀ ec s, Fractional (fun q ⇒ nova.spc.ec.gst ec q s)
```

When all uses of an assertion in NOVA require the ability to update the assertion, we have opted to expose an assertion as an exclusive assertion rather than a fractional one. For example, `nova.sm.queue sm ecs` is only accessed when the list of waiting ECs is modified, so we expose it exclusively.

```
queue_exclusive : ∀ sm ecs, Exclusive (nova.sm.queue sm ecs)
```

This is not limiting because fictional separation allows clients to derive fractional assertions on top of exclusive ones using a simple construction.

Configuration. The NOVA specification exposes the configuration of an object using *persistent* assertions, *i.e.*, those that, once established, hold universally [15]. For example, at creation time, a portal (PT) is permanently bound to an execution context (EC). The NOVA interface exposes the binding through the persistent assertion `nova.pt.ec pt ec`.

```
pt.ec_pers : ∀ pt ec, Persistent (nova.pt.ec pt ec)
```

The duplicable nature of persistent assertions makes them easy to share among threads.

Late Initialized Predicates. While rare, some NOVA predicates are "late initialized" meaning that they are created in a partially initialized state, and subsequent kernel calls are needed to finish the construction of the object. This state primarily occurs with protection domains (PDs) and arose during the decomposition of PDs in NOVA release 22.35.0 (August 2022). When a PD is created with the hypercall create_pd, it does not yet have an associated Object or Memory space. These spaces must be explicitly created and attached (also by more create_pd hypercalls) to the PD after the fact. However, once attached, these spaces are permanently bound to the PD. Consequently, assertions such as `nova.spc.pd.obj` and `nova.spc.pd.hst` are "two-stage predicates". The parameter `s` optionally carries the object ID of a space bound to the PD. `nova.spc.pd.obj pd None` is an exclusive assertion and represents the fact that an Object space is not yet bound, and, implicitly, the right to bind one to `pd`. On the other hand, `nova.spc.pd.obj pd (Some o)` is persistent, and represents that the Object space of `pd` is `o`. Formally, this theory is exposed as:

```
obj_agree : ∀ pd s1 s2, nova.spc.pd.obj pd s1 ∗ nova.spc.pd.obj pd s2 ⊢ ⌜ s1 = s2 ⌝
obj_excl  : ∀ pd, Exclusive (nova.spc.pd.obj pd None)
obj_pers  : ∀ pd o, Persistent (nova.spc.pd.obj pd (Some o))
```

3.1.1 Object Spaces & Capabilities. Kernel objects are accessed through *selectors* [31, §4.2], which are indices into Object spaces which map them to object capabilities [31, §4.1.2]. An object capability is an optional pair of a kernel object identifier and a set of permissions. The NOVA formal specification exposes this mapping as a fractional separation logic assertion $sel \overset{\text{cap}}{\longmapsto}{}^{o}_{q} \mathsf{Some}(perms, obj)$ (in Coq, `nova.spc.cap_at sel o q (Some (perms, obj)))`) which says that, in the Object space o, the selector sel has the permissions $perms$ on the object with ID obj. The NULL capability [31, §4.1.1] does not point to any object, has empty permissions, and is represented by None.

The permissions of a capability depends on the object type that the capability refers to. For example, an Object space capability [31, §4.1.2.1] supports TAKE and GRANT permissions, while a semaphore supports CTRL_UP, CTRL_DN, and ASSIGN [31, §4.1.2.11]. While the permissions for each object type are logically disjoint, NOVA's ABI exposes how these permissions are exposed as bitmasks so that object delegation, implemented by `ctrl_pd`, can operate across objects of different types.

3.1.2 Hardware Capabilities. Beyond Object capabilities, NOVA also provides capabilities for managing hardware resources:

- Memory capabilities ($\mathsf{CAP}_{\mathsf{MEM}}$) [31, §4.1.3] to manage permissions on memory page frames that belong to a Host, Guest, or DMA space;
- MSR capabilities ($\mathsf{CAP}_{\mathsf{MSR}}$) [31, §4.1.5] to manage permissions on registers of an MSR space (x86-only); and
- PIO capabilities ($\mathsf{CAP}_{\mathsf{PIO}}$) [31, §4.1.4] to manage permissions on I/O ports of a PIO space (x86-only).

While configuration of these capabilities occurs through NOVA hypercalls, direct interaction with the underlying resources is dictated largely by hardware. For example, NOVA code manages the virtual memory page tables

through Memory spaces, but the address translation process that uses those tables is performed exclusively by the MMU hardware.

The NOVA specification itself provides minimal information for the hardware behavior, delegating that responsibility to the ISA manuals. Because of this, and the current lack of formal modeling around system-level weak memory, the predicates that encapsulate these resources are still provisional. As an example of the difficulty here, while all cores have a consistent view of the contents of an Object space, the weak memory behavior of page tables and TLBs means that different cores can see different memory mappings from the same Memory space at the same time. We believe that the view-based approach from weak-memory logics [17, 21, 6] can be used to abstract the details of how these views might differ between cores. This approach does introduce some degree of ambiguity into the NOVA machine's model, but this ambiguity may, effectively, be necessary to accommodate the broad range of behaviors that are possible on modern machines.

4 LOCAL AND PRECISE LOGICALLY ATOMIC SPECIFICATIONS FOR HYPERCALLS

Our NOVA specification is designed to capture NOVA's state at every visible *linearization point* under *arbitrary* usage. To support unconstrained concurrent access to resources, we express the linearization points using atomic commits (ACs), a simplified form of Iris atomic updates [16]:

$$\langle x.\ P(x) \mid y.\ Q(x,y) \Rrightarrow R(x,y)\rangle^{\mathcal{E}_1}_{\mathcal{E}_2}$$

Each pair of angle brackets specifies the logically-atomic effects of a linearization point, where $P(x)$ captures the state immediately before, and $Q(x,y)$ captures the state immediately after the linearization point. The power of atomic commits is that they can use, and temporarily violate, invariants that live in the invariant mask $\mathcal{E}_1 \setminus \mathcal{E}_2$. After the linearization point, the remaining steps behave according to $R(x,y)$, which can include more atomic commits. Sequencing atomic commits allow us to "program" the behavior of NOVA actions in a similar way that we would program in a high level, concurrent programming language with an arbitrary `atomic` construct. In this section, we look at some hypercall specifications and how we capture their *non-atomic* behaviors by combining several atomic commits in idiomatic ways.

Atomic Commits vs. Atomic Updates. The only difference between atomic commits and atomic updates is that atomic commits do not have the "abort" or "peek" case. Since atomic commits can only be used once, the implementation must be able to determine the appropriate time to use the commit. In practice, this prevents us from putting physical state inside of ACs, and instead requires us to reflect the state via Iris' authoritative ghost construction [16, 19] so that the implementation can check the physical state and determine the value of the ghost state. While this may seem a bit burdensome, it appears practically necessary to support the persistent model of kernel objects. In addition, it simplifies client proofs by not forcing clients to prove an abort case.

Preconditions. The informal specifications of NOVA hypercalls [31, §5] are written in a style that emphasizes the idiomatic usage of the interface with additional caveats describing what happens when the preconditions do not hold. For example, in the documentation for `ctrl_sm` [31, §5.4.5]

> Prior to the hypercall:
> - If D=0 (Semaphore Up):
>   - $\mathsf{SPC}_{\mathsf{OBJ}_{\mathsf{CURRENT}}}$[sm] must refer to an SM Capability ($\mathsf{CAP}_{\mathsf{OBJ}_{\mathsf{SM}}}$) with permission $\mathsf{CTRL}_{\mathsf{UP}}$.
> - If D=1 (Semaphore Down):
>   - $\mathsf{SPC}_{\mathsf{OBJ}_{\mathsf{CURRENT}}}$[sm] must refer to an SM Capability ($\mathsf{CAP}_{\mathsf{OBJ}_{\mathsf{SM}}}$) with permission $\mathsf{CTRL}_{\mathsf{DN}}$.

To convert this specification into a formal one that is usable in arbitrary, including racy, contexts, we need to specify the behavior when $\mathsf{SPC}_{\mathsf{OBJ}_{\mathsf{CURRENT}}}$[sm] could be concurrently modified. In many cases, this simply amounts to converting the mandated "must" into a runtime check, yielding the appropriate error codes when the

checks fail. However, there are cases where multiple checks with different errors are possible, and the informal specification is ambiguous on which exact error code applies. In such cases, we strive to be conservative, leaving the choice to NOVA and mandating that clients handle any possible return values. Yet, more complex scenarios exist: for example in `create_ec`, a memory page is mapped and then a kernel selector is mapped; if the latter fails, then the memory page must be unmapped, which introduces yet another linearization point into the formal specification, even though such a linearization point is more of an implementation detail. Generally, as the informal specification is not explicit enough, the formal specification needs to describe ambiguities as having either non-deterministic or implementation-specific behavior, as we will see in an example in §4.3.

4.1 Checking Permissions

All NOVA hypercalls involve resolving a selector to a capability and checking its permissions. For example, upping a semaphore [31, §5.4.5] takes a selector, resolves it to a capability, and checks that the permissions include SM_UP. NOVA guarantees that this resolution and checking happens logically atomically, *i.e.*, from the interface's point of view, if the resolution succeeds, there must have been a point in time during the hypercall execution that the selector was mapped to a capability with the kernel object *and* the permissions that are checked.

The NOVA interface specifies selector resolution using the following atomic commit.

$$\mathsf{nova.resolve_sel_rights}(r, o, sel, Q_{fail}, Q_{succ}) :=$$
$$\left\langle \begin{array}{l} q, c.\ sel \xmapsto{\mathsf{cap}}{}^{o}_{q}\ c \mid \\ ().\ sel \xmapsto{\mathsf{cap}}{}^{o}_{q}\ c \Rrightarrow\ \mathbf{if}\ c\ \mathbf{is}\ \mathsf{Some}(perms, obj) \land r \in perms\ \mathbf{then}\ Q_{succ}(obj, perms)\ \mathbf{else}\ Q_{fail} \end{array} \right\rangle^{\top}_{\mathcal{E}_{\mathsf{nova}}}$$

where r is the permission required for a success, Q_{succ} the continuation in the successful case (the permission is sufficient), and Q_{fail} is the continuation in the failing case. To check the capability, we require the resource $sel \xmapsto{\mathsf{cap}}{}^{o}_{q}\ c$ (`nova.spc.cap_at sel o q c`) as both the atomic pre- and post-condition—we only need the resource to know the capabilities, and we will not update it. Only if c has sufficient permissions (for example, SM_UP in CTRL_UP for a semaphore), then we proceed with the continuation Q_{succ}, which typically encodes the remaining steps of the hypercall. Otherwise, we proceed with the continuation Q_{fail}, which typically encodes the post-condition with some error code. Note that the use of a single atomic commit for resolving the selector and checking the permissions effectively requires NOVA to perform these actions logically atomically.

4.2 The Specification for ctrl_sm

NOVA semaphores implement the traditional concurrency primitive with up and down implemented through the ctrl_sm hypercall [31, §5.4.5]. This hypercall resolves a selector to a semaphore and checks permissions using `nova.resolve_sel_rights`, and then ups or downs the semaphore as requested. What is interesting about the manipulation of the semaphore state is that it requires inter-thread communication.

To give a thread-local semaphore specification, we need to divide the protocol into two parts such that each thread needs to only consult its own state and the shared state to determine how to act. The division has a disjoint semaphore value and wait queue, and temporarily introduces invalid states where the value is positive and the queue is non-empty. As we will see, this casting allows up to increment the value of the semaphore regardless of the wait queue, and leaves down to reconcile the inconsistency and remove itself from the wait queue (an instance of "helping" [36, 16]).

Up. CTRL_UP either increments the *sm* value or signals an overflow, expressed using an update to `nova.sm.value`, in the specification `nova.ctrl_sm.do_up`.

$$\texttt{nova.ctrl_sm.do_up}(sm, Q) :=$$
$$\left\langle \begin{array}{l} n.\ sm \xmapsto{\text{val}} n \mid \\ ().\ sm \xmapsto{\text{val}} (\texttt{if}\ n < \text{SM_MAX}\ \texttt{then}\ n+1\ \texttt{else}\ n) \Rrightarrow Q(\texttt{if}\ n < \text{SM_MAX}\ \texttt{then}\ \text{SUCCESS}\ \texttt{else}\ \text{OVERFLOW}) \end{array} \right\rangle^{\top}_{\mathcal{E}_{\text{nova}}}$$

The specification says that the value is only updated if increment is not overflowing ($n <$ SM_MAX). Subsequently, the continuation Q is given the status code if the CTRL_UP succeeds or if the value overflows. The full specification `nova.ctrl_sm.up_spec` for CTRL_UP (Figure 3) prefixes the actual operation with a permission check by `nova.resolve_sel_rights`.

Down. CTRL_DN is more complex: it puts the thread to sleep until the semaphore is available or a timeout deadline is reached. This is broken down into two steps: waiting and finishing. First, the actual down enqueues the EC into the end of the semaphore's list of waiting ECs (`nova.sm.queue`) using `nova.ctrl_sm.do_down`.

$$\texttt{nova.ctrl_sm.do_down}(sm, ec, z, t, Q) :=$$
$$\left\langle \begin{array}{l} ecs.\ sm \xmapsto{\text{queue}} ecs \mid \\ ().\ sm \xmapsto{\text{queue}} ecs \mathbin{+\!\!+} [(ec, t)] \Rrightarrow \text{sm_timeout}(sm, ec, z, t, Q) \wedge \text{sm_success}(sm, ec, z, t, Q) \end{array} \right\rangle^{\top}_{\mathcal{E}_{\text{nova}}}$$

Then, finishing occurs with either a success, which fires when the value of the semaphore is greater than 0, or a timeout, which fires when the current CPU time exceeds the deadline. The classical conjunction connecting `sm_success` and `sm_timeout` in `nova.ctrl_sm.do_down` allows the NOVA implementation to make the choice on how the system evolves and represents a demonic choice to the caller.

$$\begin{array}{l} \text{sm_timeout}(sm, ec, z, t, Q) := \\ \quad t \neq 0 \mathrel{-\!\!*} \\ \quad \left\langle \begin{array}{l} ecs_0.\ sm \xmapsto{\text{queue}} ecs_0 \mid \\ ecs_1, ecs_2.\ ecs_0 = ecs_1 \mathbin{+\!\!+} [(ec, t)] \mathbin{+\!\!+} ecs_2 * \texttt{later_than}(t) * sm \xmapsto{\text{queue}} ecs_1 \mathbin{+\!\!+} ecs_2 \Rrightarrow Q(\text{TIMEOUT}) \end{array} \right\rangle^{\top}_{\mathcal{E}_{\text{nova}}} \\ \text{sm_success}(sm, ec, z, t, Q) := \\ \quad \left\langle \begin{array}{l} n, ecs_1.\ sm \xmapsto{\text{val}} n * sm \xmapsto{\text{queue}} ecs_1 \mid \\ ecs_2.\ n > 0 * ecs_1 = [(ec, t)] \mathbin{+\!\!+} ecs_2 * sm \xmapsto{\text{val}} (\texttt{if}\ z\ \texttt{then}\ 0\ \texttt{else}\ n-1) * sm \xmapsto{\text{queue}} ecs_2 \Rrightarrow Q(\text{SUCCESS}) \end{array} \right\rangle^{\top}_{\mathcal{E}_{\text{nova}}} \end{array}$$

In `sm_success`, the client produces the `sm.value` and the `sm.queue` ownership of an *arbitrary* value and queue contents, and NOVA, in the atomic post-condition, proves that the blocked EC *was* at the head of the queue and that the semaphore's value was greater than 0. The seemingly inconsistent state is then reconciled, also in the post-condition, by removing the thread from the wait queue and decrementing (or zeroing) the value.

Specification of ctrl_sm. To give a single, full specification to the ctrl_sm hypercall, we sequence the selector resolution with the `nova.ctrl_sm.do_up` and `nova.ctrl_sm.do_down` specifications (shown in Figure 3). Having a single specification is necessary to fit into the rest of the specifications (§5), but it is easy to prove the orthogonal specifications since the caller of this code will statically know the value of `args.(down)`.

```
nova.ctrl_sm.up_spec : nova.hypercall.spec0T nova.ctrl_sm.args.t := fun caller args Q ⇒
    letI* sm, _ := nova.resolve_sel_rights SM_UP caller.(caller_obj_spc) args.(sel_sm) (Q BAD_CAP)
    in nova.sm.user sm ∗ nova.ctrl_sm.do_up sm Q.

nova.ctrl_sm.dn_spec : hypercall.spec0T args.t := fun caller args Q ⇒
    letI* sm, _ := nova.resolve_sel_rights SM_DOWN caller.(caller_obj_spc) args.(sel_sm) (Q status.BAD_CAP) in
    ∀ k, nova.sm.kindI sm k −∗
    nova.ctrl_sm.check_cpu caller.(caller_ec) k Q
      (nova.ctrl_sm.do_down sm caller.(caller_ec) args.(zero) args.(ticks) Q).

nova.ctrl_sm.spec : nova.hypercall.spec0T args.t := fun caller args Q ⇒
    if args.(down) then
      nova.ctrl_sm.dn_spec caller args Q
    else
      nova.ctrl_sm.up_spec caller args Q.
```

Fig. 3. The specification of ctrl_sm combines selector resolution and permission checking with the individual specifications for CTRL_UP and CTRL_DN. In specifications, '`letI* x1, ..., xn := f in e`' denotes the application $f(\lambda x_1, \ldots, \lambda x_n, e)$.

Fairness and Timeliness. The split of the semaphore state predicates to facilitate thread local reasoning introduces an apparently inconsistent state when the semaphore's value is positive and the wait queue is not empty. Intuitively, due to the sequencing of atomic commits, this inconsistent state becomes visible during the logical transition (ghost update) during the proof, but not at stable states during the execution. To capture this property, the NOVA interface exposes the consistency observation guarded by a fancy update.

```
sm_consistency : ∀ sm v ecs, nova.sm.value sm v ∗ nova.sm.queue sm ecs ⊢ |={ℰnova}=> ⌜ v = 0 ∨ ecs = [] ⌝
```

In this statement, the fancy update with the mask $\mathcal{E}_{\text{nova}}$ restricts the observation to be available only when the NOVA invariant holds, which is guaranteed to occur between all user-mode steps of the system.

4.3 The Specification for create_sc

The hypercall create_sc [31, §5.3.3] demonstrates how a kernel object can be created. The Coq specification is given in Figure 4. To create a general kernel object, one needs:

- a capability with appropriate permission (*e.g.*, PD_CREATE_SC) to the PD that the new object will belong to, and
- a blank selector in the caller's PD to store the resulting capability.

Additionally and specifically for SC creation, we need to bind the new SC to an EC (its `bound_ec`), so the specification requires that the input selector `args.(sel_bound_ec)` has the permission EC_BIND_SC. If successful, the hypercall will have created a new SC with capabilities SC_DEFINED (which includes all permissions for SCs) stored in `args.(sel_target_sc)`.

```
nova.create_sc.spec : nova.hypercall.spec0T nova.create_sc.args.t := funI caller args Q ⇒
  let Q_bad_cap := Q status.BAD_CAP in
  letI* pd, _ :=
    nova.resolve_sel_rights PD_CREATE_SC caller.(caller_obj_spc) args.(sel_owner_pd) Q_bad_cap in
  letI* bound_ec, _ :=
    nova.resolve_sel_rights EC_BIND_SC caller.(caller_obj_spc) args.(sel_bound_ec) Q_bad_cap in
  letI* bound_ec_type := ec.assert_nonlocal bound_ec Q_bad_cap in
  letI* := create_ec.args_check args Q in
  let Kcreate := create_sc.success pd bound_ec args in
  letI* o_sc, status :=
    nova.create_sel_cap caller.(caller_obj_spc) args.(sel_target_sc) SC_DEFINED Kcreate in
  letI* := do_schedule bound_ec o_sc in
  Q status.
```

Fig. 4. The specification of `create_sc`.

The specification for creating new capabilities is expressed using the following atomic commit.

$$\texttt{nova.create_sel}(o, sel, rs, Q_{succ}, Q) :=$$

$$\left\langle \begin{array}{l} c.\ sel \xmapsto{\text{cap}}{}^{o}_{1}\ c \mid \\ r, c'.\ \begin{array}{l} \textbf{if}\ r = \mathsf{MEM_OBJ} \lor r = \mathsf{MEM_CAP} \\ \textbf{then}\ c' = \mathsf{None} * sel \xmapsto{\text{cap}}{}^{o}_{1}\ c \\ \textbf{else} \\ \quad \textbf{if}\ c\ \textbf{is}\ \mathsf{Some}(_) \\ \quad \textbf{then}\ c' = \mathsf{None} * r = \mathsf{BAD_CAP} * sel \xmapsto{\text{cap}}{}^{o}_{1}\ c \\ \quad \textbf{else}\ r = \mathsf{SUCCESS} * \exists obj.\ c' = \mathsf{Some}(obj, rs) * sel \xmapsto{\text{cap}}{}^{o}_{1}\ c' * Q_{succ}(obj) \end{array} \end{array} \Rrightarrow Q(c', r) \right\rangle^{\top}_{\mathcal{E}_{\text{nova}}}$$

If *sel* does not point to a NULL (blank) capability, the creation fails with BAD_CAP. Otherwise, the creation can fail if NOVA runs out of memory for the capability (MEM_CAP) or the object itself (MEM_OBJ).[4] In the case of a success, the resources are given back to user mode in Q_{succ}. Providing these resources inside the atomic post-condition is crucial because immediately after the atomic step is taken, other threads can begin using the mapping. Specifically, learning the resources later would prevent us from proving robust safety (§7.3).

Multiple Unordered Steps with Parallel Atomic Commits. The general approach of using ACs scales to operations on an arbitrary amount of state, but guaranteeing atomicity across multiple objects is expensive, *e.g.*, by using locks, and is therefore not tractable within NOVA. Thus the specification of hypercalls, such as `create_sc`, that operate on multiple objects uses multiple atomic commits. However, the NOVA specification [31, §5.3.3] does not describe an order for the various checks or even the way multiple errors are resolved. For example, in `create_sc` above, permissions are checked for selectors to both pd and bound_ec without a specified order. In the specification for `create_sc` (Figure 4), we have exposed an implementation-specific order for the operations: first the permissions are checked, for pd before bound_ec, so they can fail with BAD_CAP, and then the capability is created, so it can then fail with MEM_CAP or MEM_OBJ.

[4]NOVA's approach of encapsulating memory management makes providing more detailed information about allocation failures difficult, and is something that we leave for future work.

To be faithful to the NOVA's informal specification, the formal specification *should* say that all permissions are checked in an unspecified order using separating conjunction:

$$\begin{aligned}\exists Q_1, Q_2.\ & \mathsf{nova.resolve_sel_rights}(r_1, o_1, s_1, Q_1 \circ \mathsf{ERR}, Q_1 \circ \mathsf{OK}) * \\ & \mathsf{nova.resolve_sel_rights}(r_2, o_2, s_2, Q_2 \circ \mathsf{ERR}, Q_2 \circ \mathsf{OK}) * \\ & (\forall r_1, r_2.\ Q_1(r_1) \mathrel{-\!\!*} Q_2(r_2) \mathrel{-\!\!*} \\ & \quad \mathbf{match}\ r_1, r_2\ \mathbf{with} \\ & \quad \mid \mathsf{ERR}\ e_1, \mathsf{ERR}\ e_2 \Rightarrow Q_{fail}(e_1) \land Q_{fail}(e_2) \\ & \quad \mid \mathsf{ERR}\ e_1, _ \Rightarrow Q_{fail}(e_1) \\ & \quad \mid _, \mathsf{ERR}\ e_2 \Rightarrow Q_{fail}(e_2) \\ & \quad \mid \mathsf{OK}\ res_1, \mathsf{OK}\ res_2 \Rightarrow Q(res_1, res_2) \\ & \quad \mathbf{end}\ )\end{aligned}$$

Here, the separating conjunction acts like a local parallel composition operator for atomic commits, requiring the user to reason about the two operations occurring in either order and giving the implementation the freedom to perform the operations in either order as well.

While useful, this pattern introduces subtle amounts of non-determinism which our specification does not yet eliminate. Consider the error conditions on `create_sc` [31, §5.3.3] again. If Q_1 is applied to `ERR MEM_CAP` and Q_2 is applied to `ERR BAD_CAP`, then the informal specification does not specify what the resulting error code is. A conservative specification (as ours is) allows both return codes, but this additional flexibility afforded to the NOVA implementation permits NOVA to leak information through this channel, though the implementation does not do this in practice. A minimal remedy to this leak would be to specify that the choice on how to resolve the error code is unspecified but implementation defined and fixed for any given implementation of the NOVA specification. Formalizing this simply involves an extra boolean that could be a pure value exposed by the NOVA specification. Following this approach, the first branch in the **match** could be re-written:

$$\mid \mathsf{ERR}\ e_1, \mathsf{ERR}\ e_2 \Rightarrow Q_{fail}(\mathbf{if}\ \mathsf{nova.create_ec_resolve_error}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)$$

This pattern can easily be generalized to support more complex information exposing a pure Gallina function over the two error codes in the NOVA specification.

In practice, we believe that the sort of information leakage that this refactoring removes is somewhat negligible, as many other implementation choices within NOVA can leak the same sort of information and are much more difficult to address. For example, fine grained atomics, hardware level speculation, and micro-architectural details can all influence the behavior in NOVA in observable, but very difficult to quantify, ways. What distinguishes this leak from these more subtle ones is that it is observable with purely sequential code and so can be addressed in this simplified style.

4.4 Discussion

We have found that a small-footprint specification enabled by separation logic is quite modular from other specifications and even the NOVA state that backs unrelated kernel objects. Consequently, we have been able to adapt the specifications, and even the proof, of logically separated pieces while other portions of the NOVA implementation have changed. For example, NOVA release 22.35.0 decomposed monolithic protection domain objects into 5 separate space objects, but left the capability structure, i.e. $obj \xmapsto{\text{cap}} c$, intact. As a result, specifications and proofs concerning capabilities needed no adjustment.

Most importantly, the specifications precisely capture *all possible visible interferences* that can happen during a hypercall, including both those from trusted code (*i.e.*, NOVA and its verified applications) and those from

untrusted code (*e.g.*, guest OSes). While this complicates the specifications, for trusted-code applications that follow more disciplined accesses to resources, it is easy to derive simpler specifications on top of the precise ones. On the other hand, the precise specifications are more expressive and should enable us to prove properties that concern untrusted user code. We will sketch how to achieve both kinds of properties in §7.

There are still several subtitles about the NOVA specification that are not captured informally nor formally. Chief among these is the handling of memory allocation. The NOVA specification allows hypercalls to fail due to out-of-memory conditions, but there is no reasoning principle about memory usage. Indeed, this is difficult due to implementation details—such as fragmentation, which are influenced by the exact allocation policies.

Hypercalls returning an out of memory error can be dealt with in user mode in many cases, but not all. For example, the `ctrl_pd` hypercall has the following explanation [31, §5.4.1] of the `MEM_CAP` error:

> The PD to which the space referred to by $\mathrm{SPC}_{\mathrm{OBJ}_{\mathrm{CURRENT}}}$[dst] belongs had insufficient memory resources for allocating the storage required for granting all destination capabilities. This constitutes a partial failure of the operation, because all destination capabilities up to the first allocation failure have been granted.

While this form of failure is easy to capture using parallel commits, the overall reasoning principle provided by such a specification is rather weak, because user mode has no means to reason about the state after a batch operation that might have partially succeeded. In this case, strengthening the specification, both informal and formal, seems necessary.

5 SPECIFYING THE BEHAVIORS OF THE NOVA MACHINE

The hypercall specification represents a library-like view of NOVA where *unmodeled* user code invokes NOVA through the hypercall API. However, fully describing NOVA's behavior, *e.g.*, with respect to fault handlers and virtual memory mapping, requires going lower than that view.

In this section, we describe how we enrich the formal specification to support interactions other than hypercalls. This interface is lower-level and largely dictated by the CPU architecture and is therefore not discussed in detail in the NOVA documentation [31]. Formally, it is crucial to describe and codify the behaviors of the NOVA machine, for example, how the event portal description [31, §7.6 (AArch64), §8.6 (x86-64)] fits into the NOVA machine. While many details are expected to be architecture specific, the NOVA formal specification aims to be architecture parametric where possible, and therefore relies on a *decomposition* of the machine semantics that should soundly abstract architectural details.

5.1 The Architectural Model

The NOVA formal specification assumes an architectural-level model of the CPU where individual instructions are broken down into many atomic steps and interleaving is possible between instructions [8] (Figure 5). We assume a decomposition of the CPU semantics where each step is *statically* classified as either unprivileged or privileged, in a way that is independent of the state and other steps. For example,

- Reading a general purpose register is always unprivileged;
- Reading a protected register is always privileged; and
- Reading visibly unprivileged state that can be trapped by the hypervisor is privileged.

The third case can be quite subtle but occurs, for example, when monitoring the kernel page table root register (`TTBR0_EL1` on ARM and `CR3` on x86) when running in kernel mode. A bit in hypervisor state controls whether a write to this register will cause a fault. To guarantee that any modification to this register checks this privileged bit, the decomposition fuses the permission check and the register access into a single privileged operation. Fusing these permission checks is important because it enables hypervisor reasoning that is independent of the semantics of user mode.

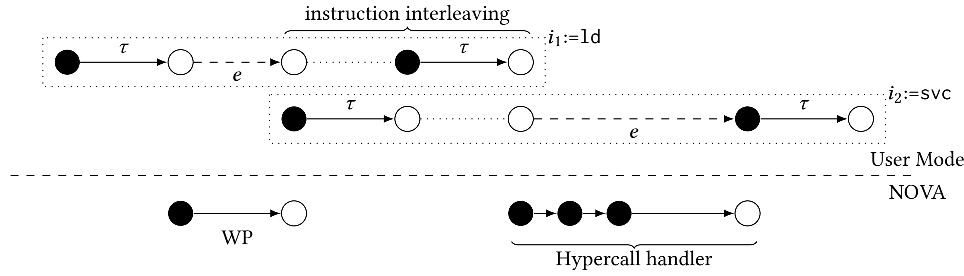

Fig. 5. The execution of a single instruction in user mode can involve both privileged (NOVA) and unprivileged (user-mode) steps. To account for weak memory semantics, some of these steps can interleave with one another between multiple instructions in the same CPU's instruction pipeline, and even with speculated instructions that may be ultimately aborted. The behavior of privileged steps is described using WPs in NOVA's separation logic.

The decomposed core semantics is described as the composition of two components.

- a labeled transition system (LTS) for the unprivileged semantics, and
- an event handler that satisfies requests for privileged state.

The unprivileged LTS gets direct access to unprivileged state, *e.g.*, general purpose registers, and can manipulate it without any interaction with the privileged component. Steps that require access to privileged state use the event interface, *e.g.*, requesting a memory read, or reading a control register. This decomposition is sometimes subtle and is described in more detail in [20].

Host and Guest ECs. Since NOVA supports both host and guest ECs, the specification actually requires two decompositions, one for each. In host ECs, "kernel mode" state is completely hidden, while for guest ECs the same state is only hidden if it is protected by hypervisor-level controls. As an example, a guest EC can take a user-level page fault that is serviced by the guest kernel without *trapping* into hypervisor mode. On the other hand, host ECs will always trap into the privileged mode when they page fault.

Weak Memory Considerations. A limitation of the current NOVA specification with respect to weak memory is that it does not permit user mode and NOVA-visible behaviors to interleave *on the same core.*[5] Supporting this without NOVA modifications would likely require a more nuanced concurrency presentation, potentially based on dependencies [14]. In practice, we have not observed any bugs due to this behavior and believe it to be unlikely, as it would rely on very deep speculation across multiple branches and (lightweight) synchronization barriers. However, we recognize that making this reasoning formal is likely to be a difficult challenge moving forward.

5.2 Weakest Precondition of NOVA ECs

The NOVA interface exposes `wp_nova_ec` to represent the *thread local* behavior of an EC. Contrary to most "standard" WPs, `wp_nova_ec` is indexed only by the EC's identifier rather than the code that the EC is running. The "code" itself is stored in a separation logic assertion. This is useful because NOVA ECs are not always active. For example, local ECs only run when they have been invoked via a portal call, and global ECs only run after they have an attached SC.

To reason about `wp_nova_ec`, the NOVA interface exposes an introduction rule (`wp_nova_ec_intro`, Figure 6) that performs a step on the unprivileged LTS and describes the behavior of the handler through separation logic.

[5]This consideration was pointed out by Ben Simmner.

```
1 Lemma wp_nova_ec_intro ec :
2   |={⊤,ℰnova}=> ▷ ∃ regs k ... , ec.regs ec 1 regs ∗ ec.kstate ec 1 k ∗
3     (∀ regs' oevt, ⌈ cpu_step regs oevt regs' ⌉ -∗ ec.regs ec 1 regs' -∗ ... -∗
4       match oevt with       (* The NOVA "handler" *)
5       | None ⇒ ec.kstate ec 1 ec.kont.ipc_tip -∗ |={ℰnova,⊤}=> wp_nova_ec ec
6       | Some cpu_evt ⇒
7         match arch_process cpu_evt with
8         | inl (novacall syscall ...) ⇒ wp_nova_ec_hypercall syscall ... (|={ℰnova,⊤}=> wp_nova_ec ec)
9         | inl (trap ...) ⇒ wp_nova_ec_handle_trap ...
10        | inl (mem ...) ⇒ wp_nova_ec_mem ... (|={ℰnova,⊤}=> wp_nova_ec ec)
11        | inr handle_arch ⇒
12          ec.kstate ec 1 ec.kont.ipc_tip -∗ handle_arch ec (|={ℰnova,⊤}=> wp_nova_ec ec)
13        | ...
14        end
15      end)
16  ⊢ wp_nova_ec ec.
```

Fig. 6. The introduction rule for wp_nova_ec describes how the unprivileged EC state drives the behavior of the thread and how NOVA handles the privileged events that the thread generates.

To prove `wp_nova_ec ec`, we need to prove that, for any atomic step `cpu_step regs oevt regs'` of `ec`, we can start from the EC register state before the step (`regs`), and end at the EC register after the step (`regs'`). Because the step is atomic, we can access all invariants except those internal to NOVA (in $\mathcal{E}_{\text{nova}}$). This is formalized by the pair of fancy updates around the step that use the same masks as the atomic commits in the NOVA specification.

The step `cpu_step regs oevt regs'` comes from the unprivileged CPU semantics, and only concerns the CPU internal register state. The external effects of the CPU are modularized through the event `cpu_evt` (in case that `oevt = Some cpu_evt`), and are encoded in separation logic as either (i) the parameterized predicate `handle_arch ec Q` for events that do not interact with NOVA, or (ii) one of several NOVA interface predicates that connect the events to NOVA-controlled state. For example, some predicates are

- `wp_nova_ec_hypercall` captures the semantics of NOVA hypercalls, by first handling the parsing of registers as required by the NOVA ABI [31, §7.8, §8.8], and then continuing with the corresponding hypercall specification, such as `nova.create_sc.spec` or `nova.ctrl_sm.spec` presented in §3.
- `wp_nova_ec_handle_trap` handles faults and exceptions of the `ec`, for example executing an illegal instruction, performing a faulting memory access, or triggering a floating point exception. NOVA handles these events by performing implicit portal calls through the event base attached to the execution context [31, §7.6, §8.6].
- `wp_nova_ec_mem` handles memory accesses, including address translations using the 2$^{\text{nd}}$-stage page tables set up by NOVA's ctrl_pd [31, §5.4.1], of the `ec`.

5.2.1 Accessing Memory. `wp_nova_ec_mem` connects the EC semantics to NOVA Memory spaces. When a NOVA EC accesses memory, it does so in two distinct steps.

1. Hardware translates the virtual addresses that they access to physical addresses *using privileged NOVA state.*
2. Hardware routes the read or write access to the corresponding memory location *or hardware device* that the address is mapped to.

While both of these steps are fully implemented by hardware, the fact that address translation is carried out by the hardware memmory management unit (MMU) using memory controlled exclusively by NOVA makes it,

```
match arch_process cpu_evt with
| ... ⇒ ...
| inl (mem (MemRead vaddr bits_sz rr acc_type should_trap aperm)) ⇒
  letI* opte, voffset := wp_nova_ec_opte_lookup ℰnova mem ec is_guest vaddr bits_sz in
  wp_nova_ec_mem_read opte voffset bits_sz rr should_trap accType aperm
    (|={ℰnova,⊤}=> wp_nova_ec ec)
end
```

Fig. 7. The handler for "memory" reads combines address translation (wp_nova_ec_opte_lookup) and "memory" access (wp_nova_ec_mem_read).

```
Definition do_mmu_vpage_to_opte mem c vpage Q : mpred :=
  ∃ V h,
    (∃ Vd q, spc.mem.vpte_view mem c Vd V ∗ spc.mem.vpte mem vpage q h ∗ True) ∧
    (∀ opte n i, ⌈ V !! vpage = Some i ⌉ -∗ ⌈ h !! i = Some (opte, n) ⌉ -∗ Q opte).

Definition wp_nova_ec_opte_lookup E mem ec is_guest vaddr bits_sz Q : mpred :=
  ∃ c info page_count, ec.cpu ec c ∗
  spc.mem.info mem info ∗ ... ∗
  let '(vpage, voffset) := page_offset vaddr in
  ⌈ vpage < page_count ∗ PAGE_SIZE ⌉%N -∗
  ⌈ byte_aligned bits_sz ⌉ -∗ (* require that the access is at least byte-aligned *)
  ⌈ voffset + byte_size bits_sz < PAGE_SIZE ⌉%N -∗ (* check the access is within a page *)
  |={E}=> do_mmu_vpage_to_opte mem c vpage (fun opte ⇒ Q opte voffset).
```

Fig. 8. Address translation in the NOVA specification converts virtual addresses into physical addresses using the logical assertions that reflect NOVA Memory spaces.

logically, part of NOVA. At the architectural level, this translation is very delicate because it uses state that is stored in memory with weak memory behavior, *and* relies on caches, the TLB, that must be *explicitly* managed by NOVA. Futher, NOVA deliberately provides minimal abstractions around Memory spaces to ensure that address space changes and fault handling can be as efficient as possible. For examples, architectural details such as the structure of the page table are observable on ARM platforms due to the requirements of break-before-make [2, § D8.13.1].

The NOVA formal specification of each of these steps is still provisional, but we provide a high-level sketch of how, we believe, this state can be soundly and usefully exposed to clients. The current definitions (Figure 7) follow the two steps described above with `wp_nova_ec_opte_lookup` expressing the address translation, and `wp_nova_ec_mem_{read,write}` capturing the "memory" access. However, even the soundness of this split contains some subtleties because architectures guarantee some degree of "freshness" for the translation results that are used during memory access. More information will be available in an upcoming technical report [20] discussed at a high level in Giarrusso et al. [10].

Address Translation. `wp_nova_ec_opte_lookup` (Figure 8) describes how the semantics of the 2nd-stage address translation, which relies on the page tables set up by NOVA's ctrl_pd [31, §5.4.1], can be restricted by NOVA's resources `spc.mem.vpte` (virtual page-table entry) and `spc.mem.vpte_view` (CPU-local view on the virtual page-table entry) that the user can temporarily provide during the step. In particular, the translation result `opte` can only come from the set `h` of page table entries that are visible to the physical CPU `c`'s MMU. The actual shape of `h` follows the properties of the assertions `spc.mem.vpte` and `spc.mem.vpte_view`, which are built on the

```
Definition do_mem_or_mmio_read opa voffset bits_sz rr should_trap Q : mpred :=
  match opa with
  | None ⇒ (* not mapped or insufficient permissions, read fails *)
    ⌈ should_trap = true ⌉ -∗ Q
  | Some pa ⇒ (* mapped with enough permissions *)
    ⌈ should_trap = false ⌉ -∗
    match rr with
    | IP.ReadSync r ⇒
        (* for DRAM, we require [byte_at] *)
        (∃ q, bytes_at (pa + voffset) (byte_size bits_sz) q r ∗ True) ∧ Q
    | IP.ReadAsync tid ⇒
        (* transaction tid initiated *) dev_transaction tid None -∗
        (* MMIO resources needed to read *)
        mmio.do_lookup pa bits_sz (DevReadAsync tid) ∗ Q
    end
  end.

Definition wp_nova_ec_mem_read opte voffset bits_sz rr should_trap acc_type aperm Q : mpred :=
  let '(opte_aperm, opa) := spc.mem.filter_pte opte (fun aperm ⇒ allowed_read aperm acc_type) in
  ⌈ aperm = opte_aperm ⌉ -∗
  do_mem_or_mmio_read opa voffset bits_sz rr should_trap Q.
```

Fig. 9. Physical addresses are resolved to memory or hardware devices (MMIO) in the system.

combined semantics of hardware and how NOVA manages page tables. These assertions follow the view-based approach in weak-memory logics [17, 21, 6] to describe page table states, and are still *work-in-progress*.

Memory Access. `wp_nova_ec_mem_read` (Figure 9) describes how the read of a physical address proceeds using the translation result `opte`. It first requires that the page table entry carries sufficient permissions to read. Then, in `do_mem_or_mmio_read`, the read either requires the physical resource `bytes_at` if it targets main memory (RAM), or requires other physical resources if it targets some memory-mapped input-output (MMIO) device. Both of these cases must, in the future, deal with weak memory behaviors. This is further complicated by the fact that NOVA delegates the cacheability attributes of pages to user mode, meaning that applications can, erroneously or maliciously, mark devices as cacheable, including permitting speculative pre-fetching.

5.2.2 A Note about the NOVA's Proof. Expressing the user-mode semantics and providing it as part of the NOVA specification, as opposed to thinking of the NOVA specification as fully parametric in the hardware semantics is important. This is because, while user-mode code has direct control of the processor, this behavior is still restricted by hypervisor state controlled by NOVA that is invisible to user-mode code. Ultimately, that `wp_nova_ec` is compatible with the hardware's behavior under NOVA's choice of hypervisor state is necessary to discharge the lowest level obligations of the NOVA proof and gives rise to some of the most interesting proof obligations, such as those concerning concurrent page table manipulation.

5.3 Weakest Precondition of Devices Exposed by NOVA

The NOVA specification exposes devices in much the same way that it exposes ECs, using a weakest precondition that captures the device's behaviors. This predicate is parameterized by the small step operational semantics of the device, expressed as an LTS with a generic event signature supporting hardware register reads and writes, interrupts, DMA accesses, and events to the outside world.

```
Theorem wp_nova_dev_unfold D E dev_id :
  |={E,ℰnova}=> ▷ ∃ s : D.(dev_state), dev_state_interp dev_id s ∗
    let Q := |={ℰnova ,E}=> wp_nova_dev D E dev_id in
    ∀ ev (s' : D.(dev_state)), ⌈ D.(dev_step) s ev s' ⌉ -∗
    dev_state_interp dev_id s' -∗
    match ev with
    | None ⇒ Q
    | Some (DMARequest ...) ⇒ (* sending DMA request *) nova.dev.dma_{read,write} ... Q
    | Some (DMAResponse ...) ⇒ (* receving DMA response *) nova.mmio.{read,write}_async_complete  ... Q
    | Some (DevInterrupt ...) ⇒ nova.dev.make_interrupt ... ∗ Q
    | Some (World io_evt) ⇒ (* device's own I/O step *) Q
    | Some ... ⇒ (* Responding to MMIO accesses *) ...
    end
  ⊢ wp_nova_dev D E dev_id.
```

Fig. 10. The introduction rule of wp_nova_dev follows the same structure as wp_nova_ec (Figure 6) and provides a way to reason about the behavior of arbitrary hardware devices. Of particular note is the handler for interrupts raised by the device, which NOVA forwards to interrupt semaphores through the `nova.dev.make_interrupt` predicate.

Figure 10 shows how NOVA wraps the device semantics to re-interpret DMA accesses and interrupt events that are emitted by the device. The handlers for DMA accesses are quite similar to those of memory events from ECs with the exception that the events are asynchronous. `nova.dev.dma_{read,write}` will perform the SMMU address translation using NOVA assertions that capture DMA space mappings, while the corresponding `{read,write}_async_complete` actions will receive the results.

Interrupts raised by the device are converted into `do_up` transitions that will be committed to their corresponding interrupt semaphore *asynchronously* by NOVA. The asynchronous nature of this action is reflected in the fact that the continuation for the device's step is established independently of the atomic commit that resolves and increments the semaphore. This is important because, in the implementation, interrupts may be disabled when the device raises the interrupt. Unfortuately, this setup, as is, is unable to *guarantee* the delivery of interrupts, let alone in a timely manner. Timeliness guarantees, even abstract ones like this, are very difficult to capture in concurrent systems.

6 THE TOPLEVEL SPECIFICATION

Up until this point, we have focused on how the NOVA machine behaves in its steady state, which is generally sufficient for verifying applications running on top of NOVA. However, to extract a full system property, we need to connect this steady state specification to a concrete starting state of the underlying machine. In broad strokes, we aim to do this by specifying NOVA's boot code as a higher-order function. However, there are several challenges that arise due to the way that NOVA starts up that make this statement subtle.

6.1 A Higher-order Specification for NOVA Startup

At the high level, a higher-order specification of NOVA would have the form shown in Figure 11. The conclusion of the entailment (lines 8-10) describes the implementation machine as a magic wand from the initial physical machine resources (lines 8-9) to the weakest pre-condition of the full machine semantics (line 10). In particular, this full machine semantics exposes all the low-level behaviors of the processor including, for example, instruction fetches, fault and interrupt handling, multi-level page table translation, and the behaviors of MSRs. The physical resources given to this WP must include at least

```
1 Theorem nova_ok : ∀ root_image pnova,
2     (∀ root_pd root_ec root_sc init_regs,
3        (virt_mem vroot root_image ∗ ec.regs root_ec init_regs ∗
4        ⌈ init_regs.(IP) = vroot + ROOT_ENTRY_OFFSET ⌉ ∗
5        initial_root_objects root_pd root_ec root_sc ∗
6        virt_mem ... ∗ virt_device_mappings ... ∗ ...)
7        −∗ wp_nova_ec root_ec)
8   ⊢ (phys_mem pnova NOVA_image ∗ ⌈ boot_regs.(IP) = pnova + NOVA_ENTRY_OFFSET ⌉ ∗
9      elf root_image ∗ phys_mem ... ∗ device_mappings ... ∗ ... )
10     −∗ wp_arm_el2 boot_regs.
```

Fig. 11. A naïve sketch of a NOVA specification as a higher order function. This sketch does not account for many of the subtlties that arise in booting NOVA.

- Ownership of the NOVA code and data segments loaded in memory. Here, as elsewhere, ownership makes explicit the assumption that these resources cannot be interferred with by other system actors.
- Ownership of the ELF image for the root application that NOVA will boot afterwards.
- NOVA command line arguments [31].
- NOVA multiboot information [31, §7.1.1, §8.1.1].
- Architecture-specific state such as for PCI busses, and TPM support [31, §6.1.2].

The premise of the entailment (lines 2-7), captures the specification of the root task [31, §6], as a similar magic wand using NOVA-level predicates. The resources on the left-hand side of the wand represent the resources provided by NOVA, including:

- the setup of the initial root PD [31, §6.2],
- the hypervisor information page (HIP) [31, §6.3], and
- capabilities for device semaphores [31, §6.2.3.1].

These resources are documented in the NOVA documentation [31, §6] and, while verbose, we do not forsee any difficulty specifying them. The wp_nova_ec (§5) on the right-hand side of the wand captures the root application's proof obligations on the behavior of the (single) root EC.

Challenge: Hardware Devices and Concurrent Boot. If NOVA were a standard program, this simple setup would largely work, but the actual physical system is a bit more complex, due to the pre-existing parallelism at startup. Several key pieces that are not covered by the sketch specification in Figure 11 include:

- Hardware devices on the platform are running concurrently with NOVA. In fact, the NOVA specification permits devices to be accessing memory during NOVA boot, just not the memory that hosts the NOVA program or the startup image.
- While NOVA's specification mandates that other cores must be idle, NOVA nonetheless owns their resources in order to control them in the boot process and later.
- Similarly, NOVA will take ownership of certain platform devices such as IOMMU/SMMUs and interrupt controllers, which are also running in parallel with NOVA's boot process.

The concurrent execution of hardware devices is the most subtle of these problems. In the machine-level logic, the behavior of a device would be captured as a weakest precondition that communicates directly with other system components, e.g. the interupt controller and IOMMU. At some point in the NOVA boot process, the interrupt controller and IOMMU will be taken over by NOVA and the device's communication with those system components will instead be captured by the effect handlers in wp_nova_dev. Capturing this transition requires exposing these resources in the statement of nova_ok.

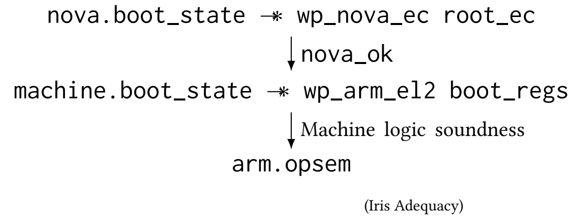

Fig. 12. The adequacy of the NOVA logic follows from the adequacy of the machine logic and the top-level specification of NOVA (nova_ok).

6.2 (Relative) Adequacy of the NOVA Logic

In many cases, the property that we aim to derive for the system is independent of the NOVA specification and even of separation logic. Extracting these "pure" properties requires proving adequacy of the separation logic which is *usually* proved with respect to an operational model. On the surface, this seems like it might be hopeless since the NOVA theorem itself (Figure 11) it stated entirely within separation logic. However, we can think of the NOVA specification as establishing a form of *relative adequacy*. Effectively, the NOVA logic is sound as long as the underlying machine logic is sound. Figure 12 demonstrates how this might work if the underlying machine logic was proven sound. Note that sound logics exist at the architecture level, *e.g.*, [26, 14], but these logics generally focus exclusively on the CPU and ignore machine-level features such as hardware devices that are crucial for reasoning about NOVA.

7 REASONING ON TOP OF NOVA

NOVA is a platform for running user-mode applications, it is not useful in isolation. Here we discuss how to prove interesting properties of such applications as well as the higher-level properties that NOVA provides. Crucially, the properties that we discuss in this section can, we conjecture, be proven on top of the NOVA specification *without* needing to look back into the implementation. This is a powerful feature that justifies our approach to writing precise specifications. While tractable in other contexts [34], proving multiple specifications of NOVA—even when each specification is slightly simpler—would likely involve a significant amount of work, since NOVA's implementation contains a large amount of highly optimized, concurrent code.

7.1 Whole Machine Refinements

When reasoning about the behavior of a whole machine, we often care only about its externally visible behaviors. For example, we might want a proof that the machine behaves like a webserver that is serving a particular website, or acts like a database server communicating over a specific protocol. In these cases, we generally aim to prove an operational refinement relating the IO traces of the machine to the IO traces of an operational specification, *e.g.*, of the webserver.

As the NOVA specifications are stated in separation logic, we follow CaReSL-style techniques [35, 9], which involves encoding the operational specification as ghost state and maintaing a simulation of IO behaviors between the operational specification and the implementation. That is, every time that the implementation steps with an event, the specification's operational model must be able to step with a corresponding event. These two pieces, the specification ghost state and the IO simulation, allow defining a behavioral refinement invariant. The user refinement proof then reduces to showing a wp_nova_ec for the user application *while* maintaining the refinement invariant. Applying adequacy under this invariant would then allow us to extract a standard

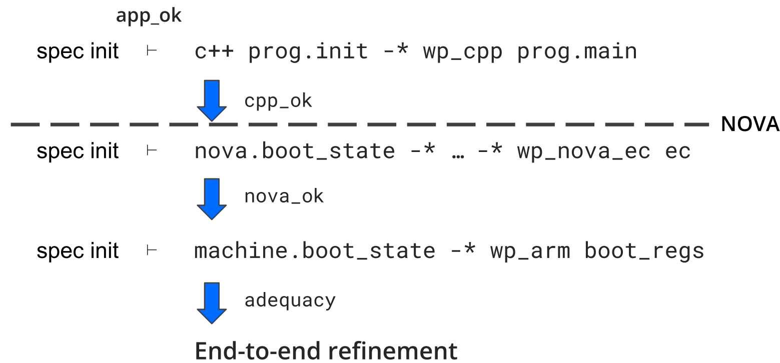

Fig. 13. End-to-end refinement requires multiple levels of abstraction.

refinement result between operational models. This process is illustrated in Figure 13, and requires the lemmas `app_ok`, `cpp_ok`, `nova_ok`, and `adequacy`.

`app_ok`. Pragmatically, user-mode programs are written in a high-level programming language that is then compiled to machine code[6]. We verify the (C++ and assembler) source program using an Iris-based program logic against a specification of NOVA's `main()` that takes the specification state (`spec init`), the physical program state (represented as C++-level separation logic assertions, *e.g.*, `c++ prog.init`), and the NOVA state (represented using the predicates in §3).

`cpp_ok`. To connect this language-level proof to `wp_nova_ec`, we leverage an assumed relative adequacy statement of the language-level logic in terms of generic machine resources that are (pragmatically) assumed by standard compilers. For example, compilers assume that the binary they produce is correctly loaded in memory with appropriate page permissions and that program state, *e.g.*, stacks and the heap, are completely "owned" by the compiler and not interfered with by other entities. Expressing the memory access properties extensionally as atomic commits makes these assumptions NOVA-independent. However, instead of proving the ACs against the machine semantics, we should be able to prove them against the corresponding ACs of the NOVA specification (*e.g.*, those in §5.2.1). For example, to capture the fact that the program is loaded, we can prove that the NOVA state for the page tables (which are in RAM and which map virtual addresses to corresponding physical addresses) contains the correct values.

`nova_ok`. We then rely on the soundness proof of NOVA, `nova_ok`, that transforms NOVA weakest precondition `wp_nova_ec` into the physical machine weakest precondition `wp_arm boot_regs`.

`adequacy`. Finally, we can apply the adequacy theorem of the machine-level separation logic, to extract the refinement property between the application specification and the physical machine: any externally visible behavior of the implementation at the machine level is also a possible behavior in the specification application semantics. The adequacy theorem assumes the verification result:

```
spec init ⊢ machine.boot_state -∗ wp_arm boot_regs.
```

[6]Generally, we assume the compiler is correct, and note that the assumption can be lifted with additional work using either translation validation or a verified compiler.

That is, if one can prove `wp_arm boot_regs` while maintaining the refinement invariant and starting with the ownership of initial physical machine state `machine.boot_state` as well as the specification initial ghost state `spec init`, then adequacy can be used to extract the refinement between the operational models.

Application to a Virtualization Stack. While we often think of specifications themselves as relatively small, this is not always the case. At BedRock, our aim of providing a verified virtualization stack means that the top-level specification includes a kernel-level semantics for the underlying processor architecture.

This allows us to say that the behavior of the implementation system running NOVA is a refinement of a bare-metal system that is running an *arbitrary* operating system.

7.2 Deriving Idiomatic Specifications

The formal specifications of the NOVA hypercalls (§4) expose all logically atomic points in an execution to allow for arbitrary interleavings. While crucial for some applications, these descriptions are a bit cumbersome even if automation [22] can support some of these patterns well. *Most* idiomatic clients could use weaker hypercall specifications that more closely reflect the descriptions in the NOVA informal specification [31, §5]. Separation logic makes it easy to write these weaker specifications for the idiomatic usage of NOVA and prove them from the stronger specifications. The weaker specifications not only gives us more confidence in the precise ones, but they also simplify user-mode reasoning, which we have carried out across subtle changes in the NOVA specification. These specifications will also be more robust as the NOVA interface continues to evolve to better support weak memory reasoning.

Data-race free Capability Resolution. An example common simplification is that object spaces are idiomatically used in a data-race free manner. In particular, while two threads may race to down the same semaphore, it is rare that one thread downs with a selector that being updated by another thread to point to another object. Generally, errors such as BAD_CAP are quite rare in idiomatic NOVA's application programs. The following example Hoare triple specification captures this idiom.

$$\left\{ sel \overset{\mathsf{cap}}{\longmapsto}{}^{o}_{q} \mathsf{Some}(perms, sm) * \mathsf{SM_DN} \in perms * \mathsf{sm.user}(sm) * \mathtt{nova.ctrl_sm.do_down}(sm, ec, z, t, Q) * \ldots \right\}$$
$$\mathtt{user_ctrl_dn}(sel, z, t)$$
$$\left\{ r.\ sel \overset{\mathsf{cap}}{\longmapsto}{}^{o}_{q} \mathsf{Some}(perms, sm) * r \neq \mathsf{BAD_CAP} * Q(r) \right\}$$

In contrast to the fundamental NOVA specification (Figure 3), this specification requires sequential ownership of $sel \overset{\mathsf{cap}}{\longmapsto}{}^{o}_{q} \mathsf{Some}(perms, sm)$ which ensures that the predicate is stable during the call. This ownership allows us to simplify the atomic commit that performs the selector resolution and permission checking (`nova.resolve_sel_rights`). We stress that such simplification is easily addressed by automation, so the main benefit is actually readability. Other instances when threads often retain sequential ownership is in initialization patterns. For example, creating a portal is often followed by setting its portal ID, and a thread retaining sequential ownership of the selector mapping prevents (in verified programs) calls to the portal during initialization.

Language-level Specifications. Another advantage of the derived Hoare triple specification is that it is phrased in the high-level language rather than at the (NOVA) machine level where the hypercall specs are [31, §5.2-5.5]. In essence, the specification encapsulates the cross-language call to ABI in the high-level language API (*e.g.*, `user_ctrl_dn`). This abstraction barrier hides the subtle reasoning required when inter-operating between different programming languages[7] and has allowed us to verify many interesting user-mode programs, and fix many bugs, without being blocked on this important technical challenge.

[7] Sound language inter-operation has a variety of subtlties that are still under active investigation [13, 27].

7.3 Towards Proving Robust Safety with Untrusted Code

The usual approach to proving robust safety [25, 34, 3, 11] is to classify security-relevant state into low- vs. high-integrity state, and to attach a protocol to the low-integrity state; then, one proves that any machine step is compatible with those protocols for low-integrity state, *i.e.*, given only low-integrity state, untrusted code can take any step available to it. Generally, this setup is expressed using arbitrary program contexts. In the setup with the NOVA specification, these arbitrary contexts show up as the universally quantified root image of an unknown user program in `nova_ok` (Figure 11, line 1). Recall that the semantics of user programs is captured by `wp_nova_ec` (§5). To show robust safety then is to prove the weakest pre-condition `wp_nova_ec` for *any* root image.

By exposing NOVA's states without any security-awareness but with precise functional correctness, we leave the client of the NOVA specification the choice to define custom protocols/abstractions for low-integrity state and to derive simple specifications for working with these abstractions. Then, by proving `wp_nova_ec` for all possible user programs while maintaining the low-integrity protocols, the client should eventually be able to show a robust safety metatheorem.

More specifically, one can consider all NOVA assertions as carrying precise, high-integrity state, and then make some of them low-integrity state to give out to untrusted code by quantifying over particular values. For example, the assertion $\exists c, sm \overset{\text{cap}}{\longmapsto}{}^{o}_{q}\, c * \mathsf{valid}(c)$ represents a capability to an unknown but valid semaphore object and is sufficient for any user code to apply the NOVA hypercall specification. Pushing this further, one can make available to an untrusted application running in an EC `ec` under a PD `pd` all of `pd`'s kernel object resources with low-integrity state. Then, the untrusted application can take any step, with or without involving NOVA, and still cannot break NOVA's internal invariants, because it is always satisfying NOVA's specification.

Figure 14 makes this idea more concrete by sketching a *robust safety invariant* for untrusted code. By putting all low-integrity state in a separation logic invariant, we allow untrusted code to access all resources so that they can apply the logically atomic specifications, *e.g.*, for their hypercalls. Ultimately extracting a robust safety theorem requires that this invariant satisfies two properties. First, the invariant can be allocated from the initial resources given to the PD. Formally,

```
Lemma rs_inv_alloc N E pd0 : initial_resources pd0 ⊢ |={E}=> inv N (rs_inv pd0).
```

And second, the invariant is sufficient to imply `wp_nova_ec` for the EC.

```
Lemma wp_ec_untrusted N pd0 ec0 : nova.ec.owner ec0 pd0 ∗ rs_inv N pd0 ⊢ wp_nova_ec ec0.
```

Taken together, these two properties imply that `wp_nova_ec` can be discharged on any initial guest code. Note that the invariant `rs_inv` is very weak, and cannot be used for verifying deep correctness properties of user applications.

8 CONCLUSION

Our specification approach models NOVA as a handler for user (and kernel) events. To describe the behavior of NOVA modularly, we present its state as loosely coupled separation logic assertions and its behaviors through weakest preconditions that describe the evolution of the state in response to user-mode code and hardware devices. The behavioral specifications are captured as separation logic programs in a similar manner to the work of [23]. We have found the expressiveness of separation logic crucial for describing the myriad of features that NOVA employs including, *e.g.*, dynamic object creation and shared memory interactions.

At BedRock Systems, we have been building on top of this specification for over three years and have maintained it through several revisions to the NOVA specification. In addition, we have completed several proofs of NOVA code, including a "spike" that connects the entry point for `ctrl_sm` through NOVA's semaphore implementation,

```
Definition rs_inv pd0 :=
  ∃ obj0, nova.spc.pd.obj pd0 obj0 ∗ (* obj0 is the Object Space of pd0 *)
  ([∗list] sel ∈ selectors pd0, ∃ c, nova.spc.cap_at sel obj0 1 c ∗ valid c) ∗ (* valid capabilities *)
  ([∗list] int ∈ interrupts pd0, ∃ cfg, nova.sm.INTconfig sm cfg 1 ∗ valid cfg) ∗ (* valid interrupts *)
  ([∗list] o ∈ all_objs pd0,
    match o with
    | pd ⇒
        (is_root pd ∨ ∃ pd_parent, nova.pd.owner pd pd_parent) ∗ ... (* valid pd *)
    | ec ⇒
        ∃ pd c, nova.ec.owner ec pd ∗ nova.ec.cpu ec c ∗ (* bound to valid pd and CPU *)
        ∃ regs, ec.regs ec regs ∗ valid regs ∗ (* valid register state *)
        ∃ ecs, nova.ec.call_stack ec ecs ∗ valid ecs ∗ ...
    | sm ⇒
        ∃ n, nova.sm.value sm n ∗
        ∃ ecs, nova.sm.queue sm ecs ∗ valid ecs ∗ ... (* valid blocked ECs *)
    | pt ⇒
        ∃ ec, nova.pt.ec pt ec ∗ (* bound to valid ec *)
        ∃ id mtd, nova.pt.id pt 1 id ∗ nova.pt.mtd pt 1 m ∗ .. (* valid id and MTD *)
    | sc ⇒ ... (* valid sc *)
    | mem ⇒  (* valid memory capabilities *)
        ([∗list] vpage ∈ vpages mem, ∃ h, spc.mem.vpte mem vpage 1 h ∗ valid h) ∗ ...
    end) ∗
  ... (* more low-integrity resources *)
```

Fig. 14. The sketch of an invariant for proving the robust safety of NOVA. All NOVA state is embedded inside the invariant with a guarantee that all reachable objects are valid and all of the state of valid objects exist within the invariant. This guarantees that the specifications of all hypercalls (§4) can be discharged.

including its use of wait queues. We believe that scaling this verification out to the rest of the NOVA source code will be subtle, but, *at the source code level*, seems within the reach of our current verification technology.

Beyond the source code, however, is the subtlty of interoperating with low-level hardware features. For examples, relating the C++ memory model to the underlying architectural memory model will be quite challenging. Additionally, inter-operations with other hardware components such as MMUs, interrupt controllers, and IOMMUs/SMMUs will be subtle, at least in part due to the fact that precise models of these components do not yet exist.

REFERENCES

[1] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (October 2017).

[2] Arm Limited 2022. *Arm® Architecture Reference Manual for A-profile architecture.* Arm Limited. Available electronically as ARM DDI 0487I.a.

[3] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (feb 2011), 45 pages. https://doi.org/10.1145/1890028.1890031

[4] John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*. https://doi.org/10.1007/3-540-44898-5_4

[5] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

[6] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

[7] R. S. Fabry. 1974. Capability-based addressing. *Commun. ACM* 17, 7 (jul 1974), 403–412. https://doi.org/10.1145/361011.361070

[8] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA)*. 608–621. https://doi.org/10.1145/2837614.2837615

[9] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *CoRR* abs/2006.13635 (2020). arXiv:2006.13635 https://arxiv.org/abs/2006.13635

[10] Paolo G. Giarrusso, Abhishek Anand, Gregory Malecha, František Farka, and Hoang-Hai Dang. 2024. Modularizing CPU Semantics for Virtualization, Talk at PriSC 2024. https://popl24.sigplan.org/details/prisc-2024-papers/7/Modularizing-CPU-Semantics-for-Virtualization

[11] Andrew D. Gordon and Alan Jeffrey. 2001. Authenticity by Typing for Security Protocols. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*. IEEE Computer Society, 145–159. https://doi.org/10.1109/CSFW.2001.930143

[12] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669.

[13] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 2023. Melocoton: A Program Logic for Verified Interoperability Between OCaml and C. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 247 (oct 2023), 29 pages. https://doi.org/10.1145/3622823

[14] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. 2024. An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic. *Proc. ACM Program. Lang.* 8, POPL, Article 21 (jan 2024), 34 pages. https://doi.org/10.1145/3632863

[15] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[16] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[19] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially substructural types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). 41–54. https://doi.org/10.1145/2364527.2364536

[20] Gregory Malecha, Paolo Giarrusso, Abhishek Anand, Hoang-Hai Dang, and Frantisek Farka. 2024. *Verification of a Virtual Machine Monitor.* Technical Report. BedRock Systems, Inc. Available by request.

[21] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: a concurrent separation logic for multicore OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 96:1–96:29. https://doi.org/10.1145/3408978

[22] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). 809–824. https://doi.org/10.1145/3519939.3523432

[23] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. 2018. A Concurrent Specification of POSIX File Systems. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. https://doi.org/10.4230/LIPIcs.ECOOP.2018.4

[24] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[25] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

[26] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN*

International Conference on Programming Language Design and Implementation (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 825–840. https://doi.org/10.1145/3519939.3523434

[27] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

[28] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

[29] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 626–655. https://doi.org/10.1007/978-3-030-44914-8_23

[30] Udo Steinberg. 2023. NOVA Microhypervisor: Feature Update, Talk at FOSDEM 2023. https://hypervisor.org/nova-fosdem-2023.pdf

[31] Udo Steinberg. 2023. NOVA Microhypervisor Interface Specification (December 1, 2023). https://github.com/udosteinberg/NOVA/blob/7c7e507d791cec767529e9c3b723a07ad5cd1e03/doc/specification.pdf

[32] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 209–222. https://doi.org/10.1145/1755913.1755935

[33] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[34] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

[35] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/2500365.2500600

[36] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). 343–356. https://doi.org/10.1145/2429069.2429111