Toward Compositional Verification of Interruptible OS Kernels and Device Drivers

Hao Chen Xiongnan (Newman) Wu Zhong Shao Joshua Lockerman Ronghui Gu
Yale University, USA
{hao.chen,xiongnan.wu,zhong.shao,joshua.lockerman,ronghui.gu}@yale.edu

Abstract

An operating system (OS) kernel forms the lowest level of any system software stack. The correctness of the OS kernel is the basis for the correctness of the entire system. Recent efforts have demonstrated the feasibility of building formally verified general-purpose kernels, but it is unclear how to extend their work to verify the functional correctness of device drivers, due to the non-local effects of interrupts. In this paper, we present a novel compositional framework for building certified interruptible OS kernels with device drivers. We provide a general device model that can be instantiated with various hardware devices, and a realistic formal model of interrupts, which can be used to reason about interruptible code. We have realized this framework in the Coq proof assistant. To demonstrate the effectiveness of our new approach, we have successfully extended an existing verified non-interruptible kernel with our framework and turned it into an interruptible kernel with verified device drivers. To the best of our knowledge, this is the first verified interruptible operating system with device drivers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods; D.3.3 [*Programming Languages*]: Languages Constructs and Features; D.4.5 [*Operating Systems*]: Reliability—Verification; D.4.7 [*Operating Systems*]: Organization and Design—Hierarchical design; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Security, Languages, Design

Keywords Program Verification; Certified OS Kernels; Interrupts; Device Drivers; Abstraction Layer; Modularity.

1. Introduction

An operating system (OS) kernel serves as the lowest level of any system software stack. The correctness of the OS kernel is the basis for that of the entire system. In a monolithic kernel, device drivers form the majority of the code base; 70% of the Linux 2.4.1 kernel are device drivers [7]. Furthermore, such drivers are found to be the major source of crashes in the Linux and Windows operating systems [7, 5, 13]. While recent efforts on seL4 [21] and CertiKOS [14] have demonstrated the feasibility of building formally verified OS kernels, it is unclear how to extend their work to verify the functional correctness of device drivers. In CertiKOS [14], drivers are unverified, and it is not obvious how to extend their framework to model devices and interrupts. In a microkernel like seL4 [21], device drivers are implemented in user space, and, though its proofs guarantee driver isolation, it does not eliminate bugs in its user-level drivers.

A major challenge in driver verification is the interrupt: a non-local jump to some driver code, triggered by a device. When device drivers are implemented inside the kernel (for better performance), the kernel should be interruptible; otherwise, it can lead to an unacceptable interrupt processing latency. Reasoning about interruptible code is particularly challenging, since every fine-grained processor step could contain a non-local jump, and, upon return, the machine state could be substantially changed. Even worse, it is not clear how such reasoning should be done at the C level, which is completely interrupt-unaware. Existing work either assumes that interrupts are turned off inside the kernel [14, 27], or polls the interrupts at a few carefully chosen interrupt points [21].

Furthermore, interrupt hardware is not static, but is configured by software. In order to verify any interesting device drivers (serial, disk, *etc*.), we first need to model the interrupt controller devices (e.g., LAPIC [17], IOAPIC [16]), and formally verify their drivers. This is important because, if the interrupt controllers are not initialized properly, it may lead to undesired interrupt behaviors. Device drivers also interact with interrupt controllers to mask/unmask particular interrupt lines. These issues have been overlooked in past work, where interrupt controllers are assumed to be properly initialized and their drivers are correctly implemented [2].

Finally, verifying an interruptible operating system with device drivers also faces the following challenges.

Devices and CPU run in parallel. Thus, the executions of CPU instructions and device transitions can interleave arbitrarily. Code verification on this highly nondeterministic machine can be challenging, since it needs to consider device state transitions, even when the CPU is executing a set of instructions unrelated to external devices. Recent work [1, 2, 3] tries to address this by enforcing a *stability* requirement that device states only change due to CPU operations. This requirement is, however, too strong as devices interacting with external environments are not stable: a serial device constantly receives characters through its port, a network card continuously transfers packets, an interrupt controller (IC) asynchronously receives interrupt requests, *etc*.

Devices may directly interact with each other. Existing work assumes that a device driver monopolizes its underlying device and devices do not influence each other [2]. This assumption does not hold for many devices in practice. For example, most devices directly communicate with an interrupt controller by signaling an interrupt.

Device drivers are written in both assembly and C. Existing device driver verification is either done completely at the assembly level [2, 9] or the verified properties are only guaranteed to hold at the C level [29, 30]. For realistic use-cases, proven properties should be translated down and then formally linked with the assembly-level proofs.

The correctness results of different components should be integrated formally. For example, the correctness proofs of device drivers and the OS kernel need to be formally linked as an integrated system, before one can deliver formal guarantees on the OS as a whole. Not doing so can introduce semantic gaps among different modules, a scenario which introduced actual bugs in previous verification efforts as reported by Yang and Hawblitzel [34]. Unfortunately, this formal linking process was found to be even more challenging than the correctness proofs of individual modules themselves [2]. Even OS's with user-level device drivers can suffer if the correctness proofs of their drivers are not formally linked with those of the kernel. For example, if some device driver code triggers a page fault at the user level, the behavior of the corresponding driver is linked to the behaviors of the page-fault handlers and address translation mechanism of the kernel.

In this paper, we propose a novel compositional approach that tackles all of the above challenges. There are two key contributing ideas. One is to build up a certified "virtual" device hierarchy, and another is a new abstract interrupt model, built upon a realistic hardware interrupt model through contextual refinement. We use these to build an extensible framework that systematically enforces the isolation among different operating system modules, which is important for scalability of any verification effort and critical for reasoning about interruptible code.

Our paper makes the following new contributions:

- We present a new extensible architecture for building certified OS kernels with device drivers. Instead of mixing the device drivers with the rest of the kernel (since they both run on the same physical CPU), we treat the device drivers for each device as if they were running on a "logical" CPU dedicated to that device. This novel idea allows us to build up a certified hierarchy of extended abstract devices over the raw hardware devices, meanwhile, systematically enforcing the isolation among different "devices" and the rest of the kernel.
- We present a novel abstraction-layer-based approach for expressing interrupts, which enables us to build certified *interruptible* OS kernels and device drivers. Our formalization of interrupts includes a realistic hardware interrupt model, and an abstract model of interrupts which is suitable for reasoning about interruptible code. We prove that the two interrupt models are contextually equivalent.
- We present, to the best of our knowledge, the first verified interruptible OS kernel and device drivers that come with machine-checkable proofs. The implementation, modeling, specification, and proofs are all done in a unified framework (realized in the Coq proof assistant [32]), yet the machine-checkable proofs verify the correctness of the assembly code that can run on the actual hardware.

The rest of this paper is organized as follows. Sec. 2 gives an overview on how we build our device hierarchy while enforcing isolation from the rest of the kernel. Sec. 3 defines a formal machine model extended with raw hardware devices. Sec. 4 presents the device objects, hardware interrupt model, and abstract interrupt model, and shows how we prove contextual refinement between the two interrupt models. Sec. 5 presents two case studies of our verified drivers: one for a serial port and another for the interrupt controllers. Sections 6 and 7 give an evaluation of our new techniques and describe the lessons we learned, the limitations, and future work. Finally, we discuss related work and then conclude.

2. Overview of Our Approach

Instead of verifying an operating system from scratch, we start from an existing verified kernel. Gu et al. [14] presented a compositional framework for building certified abstraction layers with deep specifications. There, a *certified layer* is a new language-based module that consists of a triple (L_1, M, L_2) plus a mechanized proof object showing that the layer implementation M, built on top of the interface L_1 (the *underlay*), is a *contextual refinement* of the desirable interface L_2 above (the *overlay*). A *deep specification* L_2 of a module M captures everything *contextually observable* about running the module over its underlay L_1. Once a certified layer M with a deep specification L_2 is built, there is no need to ever look at M again, since any property about M can be proven using L_2 alone. Using this framework, Gu et al. [14] have successfully verified an OS kernel called mCertiKOS.

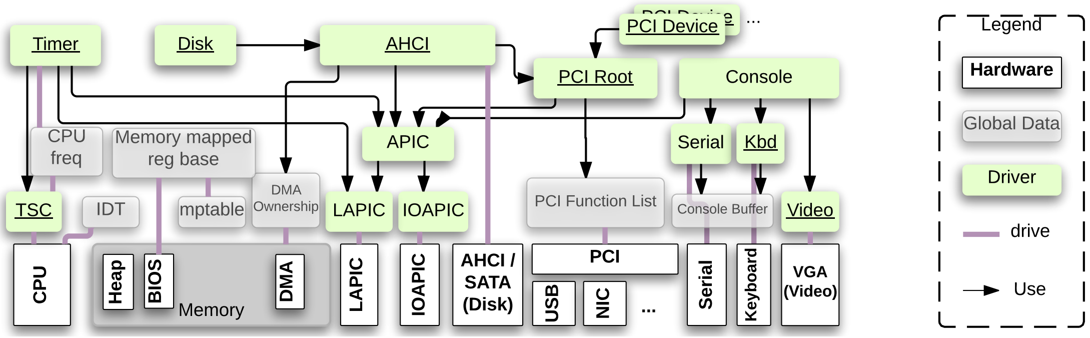

Figure 1. The device driver hierarchy of mCertiKOS

The mCertiKOS kernel currently runs on the 32 bit x86 architecture. It provides a multi-processing environment for user-level applications using separate virtual address spaces. It implements both message passing and shared memory inter-process communication protocols. As a hypervisor, it can also boot recent versions of unmodified Linux operating systems inside a virtual machine. Unlike large commercial operating systems like Linux or Unix, mCertiKOS kernel only implements a small subset of the POSIX-like API, e.g., process creation and control, physical and virtual memory management, and inter-process communication. It does not implement signals, pipes, *etc*. The current file system implementation in mCertiKOS is not certified.

Figure 1 shows the device hierarchy of mCertiKOS. Here the black boxes represent raw hardware devices; the green boxes denote the device drivers, and the gray boxes are the data structures used by the drivers. The purple/black lines show how these device and driver components are related. Note that the drivers in mCertiKOS are not verified; they are implemented in about 1,600 lines of C and assembly code, and would be considered as part of the trusted computing base (if they are kept inside the kernel).

We take mCertiKOS's lowest level machine model, *LAsm*, and extend it with device models. We model devices as finite state transition systems interacting with the processor and the external environments. Since devices run concurrently with the processor, parts of the device state change without the processor explicitly modifying them. Though these "volatile" device states can change nondeterministically, the processor itself only ever observes a "current" state when it reads the device data via an explicit I/O operation. The processor does not, and in fact *cannot*, care about any states that the device may enter between these observed states. Therefore, instead of designing fine-grained small-step transition systems that model all possible interleaved executions amongst the processor and devices, our devices simply perform an atomic big-step transition whenever they are observed, i.e., when there is a device read/write operation from the CPU.

Next, the machine model needs to be extended with the hardware interrupt model. The processor responds to an interrupt by temporarily suspending the current execution and then jumping to another routine (i.e., an interrupt handler). Interrupts can be triggered by both hardware and software. Software interrupts (e.g., exceptions, system calls) are relatively easy to reason about, since their behaviors are always deterministic. For example, a page fault exception occurs whenever the accessed address belongs to an unmapped page or a page with wrong permission, and a system call is triggered by an explicit instruction. However, hardware interrupts (IRQs) are unpredictable; when we execute some code with interrupts turned on, at every fine-grained processor step, the machine state (e.g., registers and memory) may undergo significant changes. Recent work on verified operating systems (including mCertiKOS) neglects this kind of reasoning, ignoring one of the largest kernel threat-surfaces [14, 20, 3]. Finally, modeling interrupts is important because it also opens the way toward enabling interrupts within the kernel.

On top of this lowest-level machine model, each kernel module can be related to either device drivers (denoted as *DD*) or the rest of the kernel (denoted as *K*, representing non-device-related kernel components). To introduce, verify, and abstract each such kernel module into an abstract object with atomic logical primitive transitions, we need to prove the following isolation properties:

- For each function in *K* or user space, which has interrupts turned on, the interrupt must not affect the behavior of the function. Although the code can be interrupted at any moment, and the control flow transferred to a place outside the function, it will eventually return with states (which the function relies upon) unchanged.
- Devices which directly change the memory through Direct Memory Access (DMA), do not change any memory that the execution of any function in *K* depends on.
- For each interruptible device driver function in *DD*, any interrupt not related to the current device must not change any state related to the current device.
- In case that all interrupts related to a device are masked out, no interrupts can affect the state of the interrupt handler for the device.

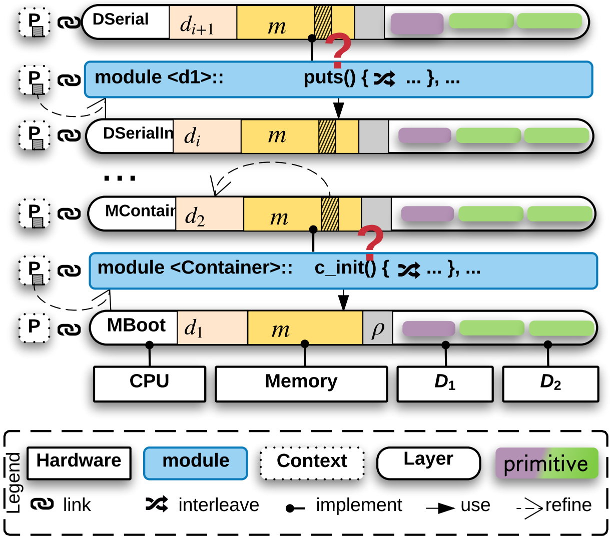

Figure 2. Abstraction layers w. interrupts: a failed attempt

For a particular fixed set of functions, the proof of the above properties may not seem hard. However, they have to be proven repeatedly for all possible combinations of currently introduced sets of functions and devices. This immediately makes the verification of an interruptible operating system with device drivers unscalable.

Furthermore, it is not obvious how to apply techniques of Gu et al. [14] to handle hardware interrupts. Figure 2 shows one such attempt. Here, P denotes the kernel/user-level context code; MBoot, MContainer, DSerialIntro, and DSerial denote several kernel and driver layers. With interrupts turned on in the kernel, it is immediately unclear how to show contextual refinement among different layers. For a kernel function like c_init, it cannot be easily refined into an atomic specification as the code can be interrupted at any point during the execution by a device interrupt, unless all possible interleaving of interrupts are encoded into the specification itself. Similarly, for a device driver function like puts, the code can be interrupted at any moment by interrupts triggered from other devices or the device itself.

In this paper, we propose a systematic way that strictly enforces isolation among different entities by construction. Our approach consists of the following two key ideas.

First, rather than viewing drivers as separate modules that interact with the CPU via in-memory shared-state, we instead view each driver as an extended device. We utilize abstraction layers and contextual refinement to gradually abstract the memory shared between a device and its driver into the internal abstract states of a more general device. Furthermore, we use the same technique to abstract those driver functions that manipulate these data into the abstract primitives of a higher level device. After this, our approach ensures that those abstract states can no longer be accessed by the other entities, through, e.g., memory reads and writes, but, rather, can only be manipulated via explicit calls to the device interface. We repeat these procedures so we can incrementally refine a raw device into more and more abstract devices by wrapping them with the relevant device drivers (see Fig. 3). In the rest of the paper, we call this extended abstract device a *device object*, to distinguish it from the raw hardware device. Note that in our model, device objects are indeed treated similarly to raw devices, and both have quite similar interfaces.

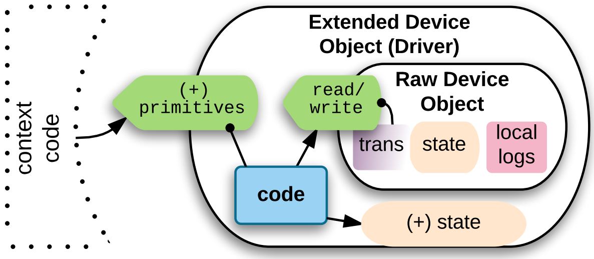

Figure 3. The driver as an extended device

Second, we introduce and verify the interrupt handler for each device at the lowest machine model, which is not yet suitable for reasoning about interruptible code. This is possible because, for each device, we require that either the interrupt be disabled or its corresponding interrupt line be masked inside the interrupt handler of the device. Next, we introduce a new abstract machine with a more abstract interrupt model, that provides strong isolation properties amongst different device objects and the kernel, in which any future (context) code with interrupts turned on can be reasoned about naturally. We prove a strong contextual refinement property between these two abstract machines: any context code running on the machine with the abstract interrupt model (overlay) retains an equivalent behavior when it is running on top of the machine with the concrete hardware interrupt model (underlay).

Figure 4 shows the layer hierarchy of our interruptible kernel with device drivers. We treat the driver code as if it runs on its own device's "logical CPU," and each logical CPU operates on its own separate internal states. Thus, the approach provides a systematic way of assuring isolation among different device objects (running on its own local logical CPUs) and the rest of the kernel.

On the kernel side (the layer hierarchy on the left hand side of Fig. 4), the contextual refinement is achieved in the same way as in Gu et al. [14] since the hardware interrupts (from the other logical CPUs with separate states) no longer affect the execution of any kernel primitive (like c_init), i.e., the kernel is completely interrupt-unaware.

Similarly, the device driver functions are no longer affected by the hardware interrupts triggered from other devices. For each device D running on top of its own logical CPU, we first introduce and verify part of the driver in the *critical area*, i.e., the low-level device functions that should not be

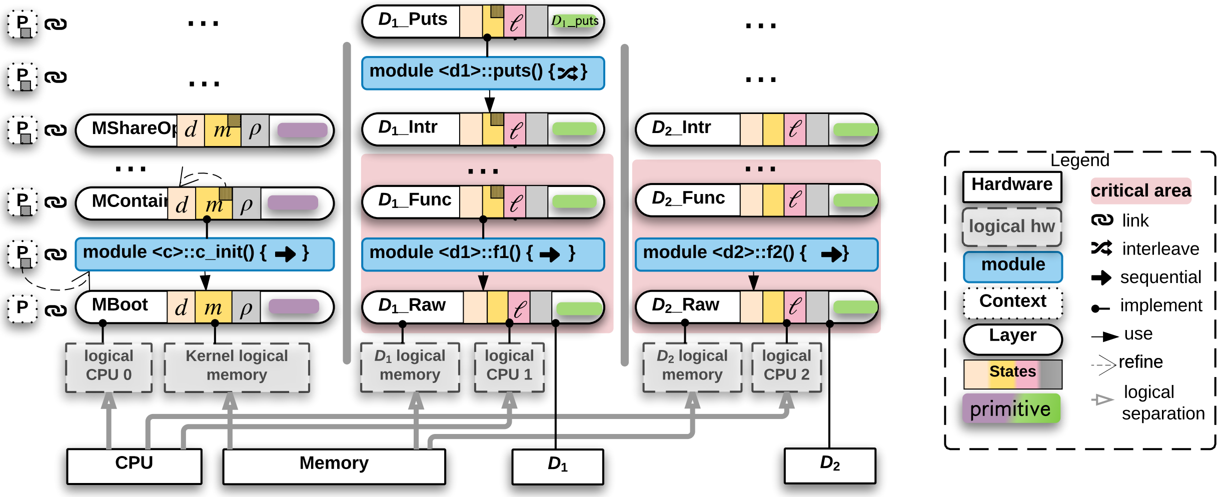

Figure 4. Building certified abstraction layers with hardware interrupts: our new approach

interrupted by the same device, and the interrupt handler of the device. Next, we use contextual refinement to introduce a new layer that has a more abstract interrupt model. On this layer, we can introduce and verify even interruptible driver code (e.g., puts) while still enforcing strong isolation and providing clean interface to the kernel.

3. Machine Model with Devices

In this section, we present our machine model, which is based on the Intel x86 architecture. We start from the *LASM* machine model presented in Gu et al. [14], and extend it to model devices and interrupts.

Our devices are modeled as finite state transition systems interacting with the CPU and the external environments. Each read/write (input/output) operation initiated from the CPU triggers an atomic big-step transition in the corresponding device. Device transitions (i.e., trans in Fig. 3) are affected by two types of interactions, one by the CPU and another by external events.

Device Transitions caused by the CPU The CPU may trigger a device transition through I/O instructions or memory-mapped I/O operations. These operations can be categorized into the following two actions:

Definition 1 (CPU Operation on a Device)**.**

$\mathcal{O}$ ::= input n — *Read value from the register at address* n
| output n v — *Write value* v *to the register at address* n

For every device, we define an atomic transition function δ^{CPU}, which takes the current device state s and a CPU operation o, and returns the new state s'. Note that δ^{CPU} is not a CPU transition, instead, it is strictly a *device transition* triggered by a CPU I/O operation.

Device Transitions caused by External Events Device transitions can also be caused by events from the external environment, such as the keyboard or network, with specific transitions depending on the kind of event. When modeling these external events, we take a minimalistic approach: though the devices can receive all kinds of different external events, we only model those that change the observable behavior of the device. Thus, the events do not map one to one to the transitions in the device hardware but rather to the CPU observations on the hardware. We model the device interfaces, not the device internals. The device interface contains all the information that a programmer can know about its states. Some example events are:

Definition 2 (Device External Events)**.**

E ::=
(* *UART device* *)
| Recv (s : list char) — *UART receives string s*
| NoSendingCompAck — *Sending is not complete*
| SendingCompAck — *UART completes the sending*
(* *Keyboard device* *)
| KeyPressed (c : $\mathbb{Z}$) — *A specific key is pressed*
| KeyReleased (c : $\mathbb{Z}$) — *A specific key is released*
...

External events are unpredictable, as their causes are not controlled by the OS. We determinize the behavior of each device by parametrizing it with the set of all possible list of events ℓ^{env} that will be processed sequentially when the CPU performs I/O operations on this device. The atomic transition function δ^{env} takes an external event e as input and changes the device states accordingly.

Note that events, even within a single device, can commute. For example, a serial port serves two roles: to receive user input and to send program output. Accordingly, among the events a serial device can receive are one for the reception of a new input string, and one signaling that some past output operation has been completed. Consider a function that first writes to a serial port, then waits until the write operation is completed by repeatedly reading some relevant status register. During one of these reads the user might send new input to the serial port. It would be reasonable for the device to observe the corresponding *Recv* event during one of the register reads, but doing so would make verifying the write function unnecessarily complex; not only would a function need to handle its own logic, but it would also need to handle any other state transition the device could undergo, even if the result were not observable in the current function.

To address this verification challenge, each device keeps a set of local logs $\vec{\ell} = \{\ell_1, ...\ell_k\}$, each of which is a strict prefix of ℓ^{env}. The serial device from the example above could contain two local logs, one for input and one for output. Then when δ^{env} receives an event that does not correspond to the currently processed action, the event can simply be skipped. When a later action observes a part of the device state which is affected by the event, that action will handle the event. In the serial port example, we would defer handling the *Recv* event until some process reads from the serial port.

Every raw device provides two I/O primitives: `read` n and `write` $n\ v$. The `read` primitive first updates the device state based on the environmental device transition δ^{env} with the next relevant external event in ℓ^{env}, then returns a value from the new state, and finally does the transition δ^{CPU} triggered by this read action. The `write` primitive first triggers the transition δ^{env} to update the device state based on the next relevant external event, then performs the transition δ^{CPU} initiated by this write operation.

In the following, the function $\mathsf{next}(\ell^{env}, \ell_i)$ finds the first relevant event e in ℓ^{env} that has not yet been processed with respect to the local log ℓ_i, and returns the event e plus a new local log that is synchronized with ℓ^{env} up to the event e.

Now, we define the operational semantics of the set of device primitives formally. Let κ be the function retrieving the value of device register addressed by n, then we have:

$$\frac{\begin{array}{ll}(e, \ell_i') = \mathsf{next}(\ell^{env}, \ell_i) & s' = \delta^{\mathsf{env}}(s, e) \\ res = \kappa(n, s') & s'' = \delta^{\mathsf{CPU}}(s', \mathsf{input}\ n)\end{array}}{\mathtt{read}(n, s, \ell_i, \ell^{env}) = (res, s'', \ell_i')} \quad (\mathtt{read})$$

$$\frac{\begin{array}{c}(e, \ell_i') = \mathsf{next}(\ell^{env}, \ell_i) \\ s' = \delta^{\mathsf{env}}(s, e) \qquad s'' = \delta^{\mathsf{CPU}}(s', (\mathsf{output}\ n\ v))\end{array}}{\mathtt{write}(n, v, s, \ell_i, \ell^{env}) = (s'', \ell_i')} \quad (\mathtt{write})$$

Thanks to the local logs, this machine model eliminates much of the nondeterminism that complicates reasoning about asynchronous systems. Nonetheless, it accurately models the observable behaviors of real hardware.

4. Driver Framework with Interrupts

The processor inherently runs in parallel with devices. In Sec. 3, we have presented a machine model representing this level of concurrency. On top of this machine model, we build certified abstraction layers introducing more and more driver code. At each abstraction layer, our model enforces systematic isolation among the different device objects and the rest of the kernel, so that interaction with one device object does not affect the states of other device objects nor the rest of the kernel. Thus, isolation properties are satisfied by construction. This dramatically simplifies our reasoning by allowing us, at any given time, to focus on only the device objects that are currently interacted with.

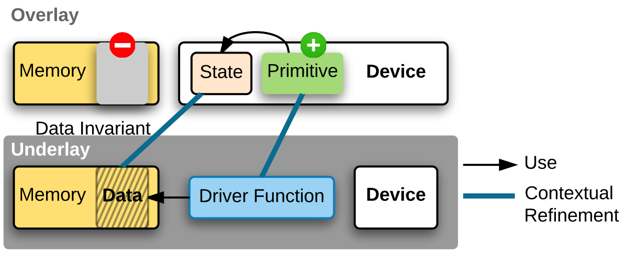

Figure 5. Layer-based contextual refinement

In this section, we define the device object more formally; then we show how to incorporate interrupts into our model while still following our isolation policy.

4.1 Device Objects

A *device object* is a logical abstraction containing a hardware device plus its related drivers. Each device object consists of a set of abstract states, abstracting the private states of the device (e.g., device registers, driver private memory); and a set of primitives, abstracting the module interface. The abstract states are private to the device object, and can only be manipulated by explicit calls to the device object's primitives. This is achieved by establishing a contextual refinement relation from the concrete memory and device function implementation to the abstract state and primitives. As shown in Fig. 5, we follow the layer-based methodology [14] and utilize the CompCert memory permissions [24] to hide the relevant memory at overlay, which prevents the context code from accessing the object's private memory. These logical permissions do not correspond to any physical protection mechanism, but are used to ensure that the abstract machine at overlay gets stuck if any code tries to directly access this portion of memory. The safety proof of our entire operating system (the kernel never gets stuck) guarantees that such a situation never happens. The set of driver functions at underlay, which manipulate the memory that will be abstracted away at overlay, are themselves abstracted into the set of device primitives at the overlay (see Fig. 5).

For example, the console buffer is implemented as a circular buffer in our console driver. The concrete implementations of the buffer operators (*buf_read* and *buf_write*) directly manipulate the concrete circular buffer in memory. At a higher layer, in our abstract console device object, the logical buffer is represented as a list, and the primitives are specified directly over this abstract list, i.e., the *buf_read* simply returns the head element in the list, while *buf_write* adds the new element to the end of the list, discarding a single head element if the size of the list exceeds its limit. The contextual refinement relation between the two layers ensures that any code running on top of the more abstract overlay exhibits behavior equivalent to running on top of the underlay.

The primitives at the underlay can be passed through to the overlay, or hidden if they are no longer needed. For example, once the primitive *ahci_transfer* is introduced at the overlay, the underlay primitives *ahci_read* and *ahci_write*, used to implement *ahci_transfer*, are hidden. This facilitates the invariant proofs as stronger invariants can be introduced at higher layers, which could otherwise be violated by the lower-level primitives.

Combining Device Objects At a certain abstraction layer, some drivers, or more generally, system services, may interact with multiple device objects, by, e.g., transferring data between two devices, or broadcasting messages to multiple devices. At this stage, such devices are no longer totally isolated, but are synchronized through hardware or software mechanisms. This does not fit directly into our model providing systematic isolation among different device objects and the rest of the kernel.

In the above scenario, we introduce at the overlay a single heterogeneous device object, which combines the device objects from the underlay via the newly introduced functions. The abstract machine at overlay thereby provides systematic isolation between the new abstract device object and the rest of the kernel. The internal states and local logs of the combined device object are the disjoint union of the relevant objects at underlay, while the functions that manipulate multiple device objects at underlay become primitives of the new device object, operating on a wider range of internal states, at overlay. As in all device objects, existing primitives can be either passed through to this new device, or hidden.

4.2 Interrupts

We now show how to adapt the interrupts into our setting. We first present our interrupt model at the hardware level, where the interrupt transitions are separately defined for the CPU, the interrupt controllers (IC), and the devices. At this low level we lack the full behaviors of interrupt handlers, so all the primitives verified at this machine level have the precondition that interrupts are disabled or the corresponding interrupt lines are masked. On top of this hardware abstraction layer, we incrementally introduce and verify interrupt handlers for each device through abstraction layers. Above a certain abstraction layer, we have full behaviors of the interrupt handlers, so we introduce a new abstraction layer with an abstract interrupt model, where an interrupt only changes the state of the device object that triggered it. This makes the interrupt completely transparent to the CPU, the IC, and other devices, thus guaranteeing our desired isolation properties. We prove the strong contextual refinement property between these two abstraction layers to ensure that any context program running on top of the overlay retains behavior equivalent to running atop the underlay. Starting from the abstraction layer with the abstract interrupt model, we support verification of any code with interrupts enabled.

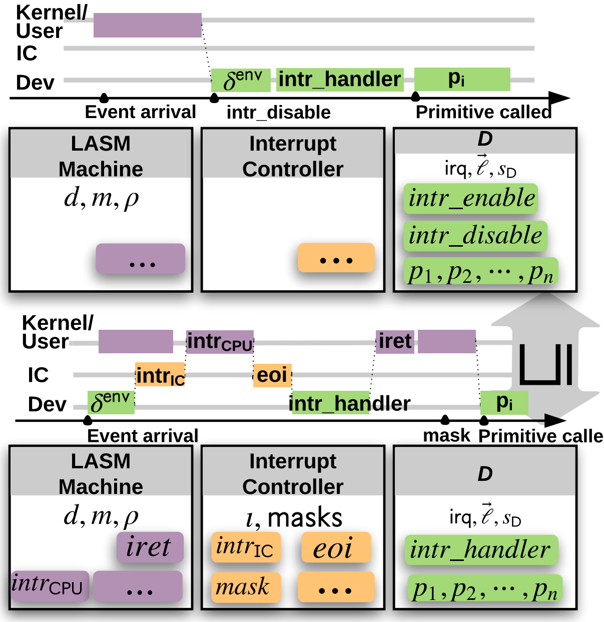

Figure 6. The hardware interrupt model (bottom), the abstract interrupt model (top), and the contextual refinement between these two models.

4.2.1 Hardware Interrupt Model

The entities involved in any given interrupts are categorized into three parties (shown in the bottom half of Fig. 6). If a device transition (e.g., from the device D in Fig. 6) triggers an interrupt, it gets sent to the IC. The IC multiplexes several interrupt lines onto the CPU (i.e., the LASM machine in Fig. 6), with the ability to mask and unmask each interrupt line. At each transition, the IC selects the pending unmasked interrupt with the highest priority and forwards it to the CPU. When the CPU receives an interrupt signal, it first checks whether interrupts are enabled on that CPU, and, if so, saves the current context and jumps to the corresponding entry in the interrupt descriptor table (IDT). If interrupts are turned off, the interrupt signal is ignored. Thus, an interrupt involves at most three consecutive transitions: the device, the IC, and the CPU. These three transitions seem inter-related, and isolation among the three entities is non-obvious. In our approach, we first develop a low level hardware interrupt model that separately defines the set of interrupt related operations. Then these three disconnected components are united at some higher level abstract machine model after we have verified all the interrupt handlers.

$$\frac{\begin{array}{c} s_{\mathsf{ic}}.\mathsf{masks}[N_D] = \mathsf{Masked} \\ s_{\mathsf{ic}}.\mathsf{irqs}[N_D] = n \end{array}}{\mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) = (s_{\mathsf{ic}}, \mathsf{IRQ}\ n)} \qquad (\mathtt{intr}^{m}_{\mathsf{IC}})$$

$$\frac{\begin{array}{c} s_{\mathsf{ic}}.\mathsf{masks}[N_D] = \mathsf{Unmasked} \\ s_{\mathsf{ic}}.\mathsf{irqs}[N_D] = n \qquad s_{\mathsf{ic}}.\iota = \varnothing \end{array}}{\mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) = (s_{\mathsf{ic}}[\iota \leftarrow \mathsf{n}], \mathsf{IRQ}\ n)} \qquad (\mathtt{intr}^{u}_{\mathsf{IC}})$$

$$\frac{}{\mathtt{eoi}(s_{\mathsf{ic}}) = s_{\mathsf{ic}}[\iota \leftarrow \varnothing]} \qquad (\mathtt{eoi})$$

Figure 7. Interrupt transition for the IC

$$\frac{\rho[\mathsf{EFLAGS.if}] = \mathsf{Disabled}}{\mathtt{intr}_{\mathsf{CPU}}(d, \rho, \mathsf{IRQ}\ n) = (d, \rho)} \qquad (\mathtt{intr}^{d}_{\mathsf{CPU}})$$

$$\frac{\begin{array}{c} \rho[\mathsf{EFLAGS.if}] = \mathsf{Enabled} \qquad d' = d[\mathsf{isr} \leftarrow \mathsf{true}] \\ \mathit{tfs}' = \mathtt{save\_context}(d'[\mathsf{tfs}], \rho) \\ d'' = d'[\mathsf{tfs} \leftarrow \mathit{tfs}'] \qquad \mathsf{IDT}[n] = p \\ \rho' = \rho[\mathsf{EIP} \leftarrow p][\mathsf{EFLAGS.if} \leftarrow \mathsf{Disabled}] \end{array}}{\mathtt{intr}_{\mathsf{CPU}}(d, \rho, \mathsf{IRQ}\ n) = (d'', \rho')} \qquad (\mathtt{intr}^{e}_{\mathsf{CPU}})$$

$$\frac{\begin{array}{c} (\mathit{tfs}', \rho') = \mathtt{restore\_context}(d[\mathsf{tfs}]) \\ d' = d[\mathsf{isr} \leftarrow \mathsf{false}][\mathsf{tfs} \leftarrow \mathit{tfs}'] \end{array}}{\mathtt{iret}(d, \rho) = (d', \rho')} \qquad (\mathtt{iret})$$

Figure 8. Interrupt transition for the CPU

Interrupt Transition for Devices As described in Sec. 3, every raw device has its own transition function δ^{env} specifying how it reacts to the external events. When a particular transition triggers an interrupt (e.g., see the event arrival and the green box δ^{env} along the `Dev` line in the bottom half of Fig. 6), the device marks an interrupt request bit (*irq*) in its internal state.

Interrupt Transition for the IC When the IC receives an interrupt signal (e.g., see the orange box $\mathtt{intr}_{\mathsf{IC}}$ along the `IC` line in Fig. 6), it first checks whether the particular interrupt line is masked, and if so, it ignores the interrupt; if not, then the IC marks the corresponding interrupt line as pending. The transition rules are defined in Fig. 7. Here, N_D is the corresponding interrupt line number of the device D which triggered the interrupt; it is fixed by the hardware connection, and is mapped to $\mathsf{IRQ}\ n$ by the configuration of the IC; the ι field of s_{IC} indicates which IRQ number is pending; we use $\varnothing$ to indicate that there is no pending interrupt. After the CPU performs its initial interrupt transition, the IC would receive the End Of Interrupt (EOI) signal (e.g., see the orange box `eoi` along the `IC` line in Fig. 6), it clears the pending mark on the interrupt line. The IC also has two primitives *mask* and *unmask*, which set the $s_{\mathsf{ic}}.\mathsf{masks}[N_D]$ of the interrupt line number N_D to Masked and $\mathsf{Unmasked}$ respectively.

Interrupt Transition for the CPU As soon as the IC marks an interrupt line as pending, the CPU will perform its own interrupt transition (e.g., see the purple box $\mathtt{intr}_{\mathsf{CPU}}$ along the `Kernel/User` line in Fig. 6). Let ρ represent the register set, and d be the logical abstract states in the machine model, then the interrupt transition of a CPU is shown in Fig. 8.

We use $\mathsf{EFLAGS.if}$ to represent the interrupt flag bit in the EFLAGS register. If interrupts are disabled inside the CPU, the $\mathtt{intr}_{\mathsf{CPU}}$ primitive is totally transparent. Otherwise, it first changes the logical `isr` state to `true`, saves the current context into the end of the trap frame list ($d'[\mathtt{tfs}]$), and jumps to the corresponding IDT entry. Here `isr` indicates whether the current machine execution is in the interrupt handling mode; the `save_context` function models the hardware behavior of saving the current context into the abstract state ($d'[\mathtt{tfs}]$), which corresponds to the concrete stack frames in the memory (abstracted in layers below).

The primitive `iret` is the counterpart of $\mathtt{intr}_{\mathsf{CPU}}$, and models the behavior of CPU when the interrupt handler returns. It restores EFLAGS (including the old interrupt flag bit) from the context and thus also re-enable interrupts. The `restore_context` function models the hardware behavior of restoring the current context from the abstract state ($d[\mathtt{tfs}]$).

Lemma 1. *The function `restore_context` is a left inverse of the function `save_context`.*
$(\mathit{tfs}, \rho) = \mathtt{restore\_context}(\mathtt{save\_context}(\mathit{tfs}, \rho))$

The CPU also has two primitives *sti* and *cli*, which set the $\mathsf{EFLAGS.if}$ bit to $\mathsf{Enabled}$ and $\mathsf{Disabled}$ respectively.

4.2.2 Abstract Interrupt Model

The low-level machine model, we just described, is not suitable for reasoning about interrupts, since each of the three entities has its own disconnected view. For instance, when the CPU jumps to an IDT entry, it is unaware of the behavior of the corresponding interrupt handler, and when the IC sends an interrupt signal to the CPU, it does not know whether the interrupt will be handled or not. We would like to formally connect these three different views to derive a nice machine model that is suitable for reasoning about the end-to-end behavior of interrupts, i.e., an interrupt triggered by a device only modifies the particular device's internal states, and is transparent to the CPU, the IC, and other devices. To achieve this, we need a model of the full behavior of the interrupt handler for each device.

Starting from the above hardware interrupt model, we incrementally extend a raw device by wrapping it with driver code related to the interrupt handler, until we have fully verified the interrupt handler for the device. Each device has exactly one interrupt handler, which, by our isolation policy, only modifies the internal states of its particular device, and cannot itself be interrupted by the same device.

DisableNoIntr: Disable with no unhandled interrupt

$$\frac{\begin{array}{c}(e,\ell_i') = \mathsf{next}(\ell^{env},\ell_i) \qquad s_{\mathsf{tmp}} = \delta^{\mathsf{env}}(s,e)\\ s_{\mathsf{tmp}}.irq = \mathsf{false} \qquad s' = s[\mathsf{iFlag} \leftarrow 0]\end{array}}{\texttt{intr\_disable}(s,\ell_i,\ell^{env}) = (s',\ell_i)}$$

DisableIntr: Disable with unhandled interrupts

$$\frac{\begin{array}{c}(e,\ell_i') = \mathsf{next}(\ell^{env},\ell_i) \qquad s' = \delta^{\mathsf{env}}(s,e)\\ s'.irq = \mathsf{true} \qquad (s'',\ell_i'') = \texttt{intr\_handler}(s',\ell_i',\ell^{env})\\ (s''',\ell_i''') = \texttt{intr\_disable}(s'',\ell_i'',\ell^{env})\end{array}}{\texttt{intr\_disable}(s,\ell_i,\ell^{env}) = (s''',\ell_i''')}$$

EnableNoIntr: Enable with no pending interrupt

$$\frac{s.irq = \mathsf{false} \qquad s' = s[\mathsf{iFlag} \leftarrow 1]}{\texttt{intr\_enable}(s,\ell_i,\ell^{env}) = (s',\ell_i)}$$

EnableIntr: Enable with pending interrupts

$$\frac{\begin{array}{c}s.irq = \mathsf{true} \qquad (s',\ell_i') = \texttt{intr\_handler}(s,\ell_i,\ell^{env})\\ (s'',\ell_i'') = \texttt{intr\_enable}(s',\ell_i',\ell^{env})\end{array}}{\texttt{intr\_enable}(s,\ell_i,\ell^{env}) = (s'',\ell_i'')}$$

Figure 9. Transition rules for *intr_disable* and *intr_enable*

$$\frac{\begin{array}{c}(s_{\mathsf{ic}}', \mathsf{IRQ}\ n) = \texttt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D)\\ (d',\rho') = \texttt{intr}_{\mathsf{CPU}}(d,\rho,\mathsf{IRQ}\ n)\\ s_{\mathsf{ic}}'' = \texttt{eoi}(s_{\mathsf{ic}}') \qquad (s_D',\ell_i') = \texttt{intr\_handler}_{\mathsf{D}}(s_D,\ell_i,\ell^{env})\\ (d'',\rho'') = \texttt{iret}(d',\rho')\end{array}}{\texttt{intr}(d,m,\rho,s_{\mathsf{ic}},s_D,\ell_i,\ell^{env}) = (d'',m,\rho'',s_{\mathsf{ic}}'',s_D',\ell_i')}$$

Figure 10. Interrupt transition for the whole system, in the case when an interrupt is triggered by the device D on interrupt line number N_D.

At this stage, we have the formal specification of the interrupt handler for a device. Next, through contextual refinement, we encapsulate the behaviors of interrupts into two primitives *intr_enable* and *intr_disable* at overlay for the device, which, as shown in the top half of Fig. 6, render interrupts transparent to the CPU and the IC. The precise transition rules are given in Fig. 9. Here, iFlag is an abstract state indicating whether the particular device interrupt is turned on or off; the next function, as defined at the end of Sec. 3, returns the next relevant event in ℓ^{env} and a new local log synchronized with ℓ^{env} up to the returned event.

The *intr_disable* primitive first synchronizes the device state with the previously unhandled interrupts then sets interrupt as disabled. It performs the synchronization by scanning the log from the last place *intr_enable* was called, until we hit the first event that did not trigger any interrupt. This ensures that subsequent observations on the device (in the abstract model) will be consistent with those performed under the hardware interrupt model. Note that *intr_disable* is defined recursively: it performs the environment transition δ^{env} on each event until we hit an event that does not trigger interrupts (i.e., the DisableNoIntr case); the s_{tmp} state should be discarded since the device transition stops at the point where the last unhandled interrupt is handled.

The *intr_enable* primitive discharges any pending interrupts, then sets interrupt as enabled. This models the physical machine behavior, wherein interrupts (which can occur while interrupts are disabled) get delayed until interrupts are re-enabled. This causes the OS to immediately jump to the interrupt handler after re-enabling interrupts. This repeats until the device no longer attempts to trigger an interrupt within the interrupt handler, and normal execution can continue.

With these two new primitives, the CPU transition in the abstract interrupt model can be completely oblivious of the device transitions. For example, in the top half of Fig. 6, the purple box along the `Kernel/User` line can ignore any event arrival from a device; the CPU for the `Kernel/User` line would only force the device transitions when it wants to make observations about a device (e.g., by calling *intr_disable*, then a high-level device primitive p_i, followed by *intr_enable*).

Contextual Refinement Between Two Interrupt Models To show the contextual refinement between the two abstraction layers in Fig. 6, we prove that the behavior of an IRQ can indeed be made transparent to the CPU and the IC.

Lemma 2. *An IRQ is transparent to the CPU and the IC, i.e., the transitions triggered by the IRQ only change the states of the corresponding device that triggered the interrupt.*

Proof: When the interrupt is disabled on the CPU or the particular interrupt line is masked in the IC, the proof is obvious. When the interrupt is enabled, i.e., the corresponding interrupt line is routed, not masked, and the $\mathsf{EFLAGS.if}$ register bit is set, the state transition of the whole system is shown in Fig. 10. Here, the transition `intr` takes an abstract state d, the memory m, the register set ρ, the state of interrupt controller s_{ic}, the state of the device s_D, a local log of the device ℓ_i, the event list ℓ^{env}, and returns appropriate new system states after the interrupt transition is fully performed. In this case, we need to show that:

$$\frac{(d',m',\rho',s_{\mathsf{ic}}',s_D',\ell_i') = \texttt{intr}(d,m,\rho,s_{\mathsf{ic}},s_D,\ell_i,\ell^{env})}{\begin{array}{c}(s_D',\ell_i') = \texttt{intr\_handler}_{\mathsf{D}}(s_D,\ell_i,\ell^{env}) \land\\ d' = d \land m' = m \land \rho' = \rho \land s_{\mathsf{ic}}' = s_{\mathsf{ic}}\end{array}}$$

This can be proven by composing the interrupt transition rules of the CPU and the IC with Lemma 1. ■

Corollary 1. *IRQs do not affect the kernel, i.e., they do not change any of the kernel's states*[1].

Nested Interrupts Note that the $\texttt{intr}_{\mathsf{CPU}}$ transition in Fig. 8 disables the interrupt. Thus between $\texttt{intr}_{\mathsf{CPU}}$ and `iret` in Fig. 10, the interrupt is turned off, which means that no nested interrupts are allowed. In many cases, supporting

[1] Remember, we consider device drivers a part of the device, not the kernel.

$$\frac{\begin{array}{c}(s'_{\mathsf{ic}}, \mathsf{IRQ}\ n) = \mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) \\ (d', \rho') = \mathtt{intr}_{\mathsf{CPU}}(d, \rho, \mathsf{IRQ}\ n) \\ s''_{\mathsf{ic}} = \mathtt{eoi}(s'_{\mathsf{ic}}) \\ s'''_{\mathsf{ic}} = \mathtt{mask}(s''_{\mathsf{ic}}, N_D) \qquad (d'', \rho'') = \mathtt{sti}(d', \rho') \\ (s'_D, \ell'_i) = \mathtt{intr\_handler}_{\mathsf{D}}(s_D, \ell_i, \ell^{env}) \\ (d''', \rho''') = \mathtt{cli}(d'', \rho'') \\ s''''_{\mathsf{ic}} = \mathtt{unmask}(s'''_{\mathsf{ic}}, N_D) \qquad (d'''', \rho'''') = \mathtt{iret}(d''', \rho''')\end{array}}{\mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env}) = (d'''', m, \rho'''', s''''_{\mathsf{ic}}, s'_D, \ell'_i)}$$

Figure 11. Interrupt transition for the whole system when nested interrupts are allowed.

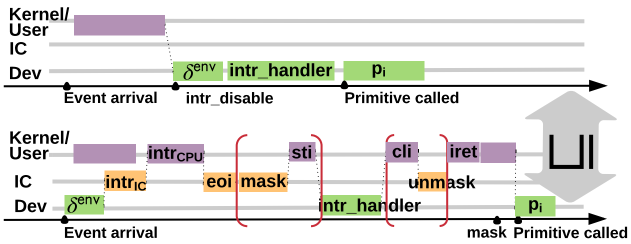

Figure 12. The contextual refinement between interrupt models with nested interrupts.

nested interrupts is critical so that some high priority interrupt processing is not delayed by the low priority ones. The interrupt transition for the whole system with nested interrupts is shown in Fig. 11. Here, before the interrupt handler is called, we mask the interrupt line of the particular device (to make sure there is no nested interrupt from the same device) and then turn on the interrupt on the CPU. Accordingly, after the interrupt handling, we disable the CPU interrupt, then unmask the particular interrupt line before the `iret` transition is performed. We have proved that this model also refines the same abstract interrupt model (see Fig. 12).

5. Case Study

In this section, we present two case studies of our verified drivers. First, we present our device model for a serial port, and show how the relevant drivers are specified and verified. Next, we present our interrupt controller model. We have used a single controller in Sec. 4 to ease the presentation. However, mCertiKOS utilizes two physical interrupt controller devices: the I/O Advanced Programmable Interrupt Controller (IOAPIC) and the Local Advanced Programmable Interrupt Controller (LAPIC). In this section, we only present the IOAPIC device model and the verification of its driver.

5.1 Serial Port

Fig. 13 illustrates a typical serial port with a bounded internal buffer of size 12. It consists of a RS-232 interface and a Universal Asynchronous Receiver/Transmitter (UART) controller. RS-232 delivers electrical signals between the UART controller and the connected cable. The UART controller is responsible for demodulating received data into digital bits and storing them into the internal receiving (Rx) buffer, and also modulating sent data from digital bits and inserting them into the transmission (Tx) buffer.

The hardware UART controller has many features, and the mCertiKOS serial driver only utilizes those parts needed for sending and receiving character strings. When modeling the serial port, we take the minimalistic approach of only modeling the set of features utilized by the existing drivers. The internal state of the serial port device is defined as:

$$\begin{array}{lll} s = (& \mathsf{RxBuf} : \mathsf{list\ char}, & \rhd \textit{Receiving buffer} \\ & \mathsf{TxBuf} : \mathsf{list\ char}, & \rhd \textit{Transmission buffer} \\ & \mathsf{irq} : \mathsf{bool}, & \rhd \textit{Interrupt pending} \\ & \mathsf{Connected} : \mathsf{bool}, & \rhd \textit{Power} \\ & \mathsf{Base} : \mathbb{Z}, & \rhd \textit{Base address} \\ & \multicolumn{2}{l}{\rhd \textit{Line and modem configurations:}} \\ & \multicolumn{2}{l}{\mathsf{RxIntEnable} : \mathsf{bool},\ \mathsf{DLAB} : \mathsf{bool},\ \mathsf{Baudrate} : \mathbb{Z},} \\ & \multicolumn{2}{l}{\mathsf{Databits} : \mathbb{Z},\ \mathsf{Stopbits} : \mathbb{Z},\ \mathsf{Parity} : \mathsf{ParityType},} \\ & \multicolumn{2}{l}{\mathsf{FIFO} : \mathbb{Z},\ \mathsf{Modem} : \mathbb{Z}\ ).} \end{array}$$

There are three external events for the serial device. The serial event $\mathsf{Recv}\ s$ indicates that a string has been received. The $\mathsf{SendingCompAck}$ event implies the device received the acknowledgment that the characters in the transmission buffer have been sent out successfully, while the $\mathsf{NoSendingCompAck}$ events indicates that the sending of characters in the transmission buffer is not yet complete. We have configured the serial device to trigger an interrupt when it receives data (a nonempty string). The device is configured to not to trigger any interrupt when the transmission buffer becomes empty, i.e., when the characters in the transmission buffer are sent out successfully. Thus, before any data is written to the serial port, we have to poll the transmission status until it becomes empty. We have chosen this setup because it covers both interrupt-triggering and polling events.

Note that, the states $s.\mathsf{RxBuf}$ and $s.\mathsf{irq}$ are disjoint from $s.\mathsf{TxBuf}$ under the environment transitions in that the former is for receiving data and the latter is for sending data only. This allows us to use two separate local logs in our device model, ℓ_{tx} (for transmission) and ℓ_{rx} (for receiving), to handle these possibly commutative events.

Next, in Fig. 14, we define the transition functions δ^{env} and δ^{CPU}, where δ^{env} needs to handle all the possible environmental events against the current state, and δ^{CPU} updates the current state based on the input and output addresses and values. We use the notations $[\cdot]$ and ++ to represent a singleton list and list concatenation, respectively. The function `last` takes a length n and a list l, and returns the last n elements of l as a new list if the length of l exceeds n, and returns the original list l otherwise. The function is used to model the action of dropping some elements in the front of the buffer when the length of the new buffer exceeds the hardware buffer size ($\mathsf{BufSize}$).

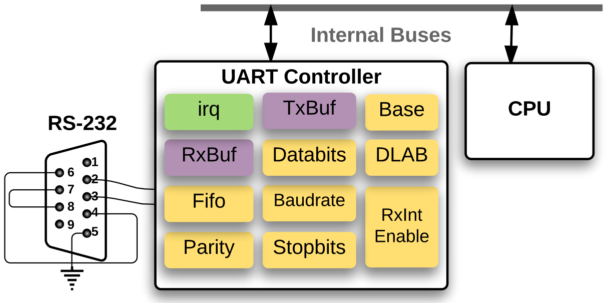

Figure 13. The hardware connections of a serial port

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad s.\mathsf{RxIntEnable} = \mathsf{true} \\ e = \mathsf{Recv}\ w \qquad w \neq nil \\ \mathsf{newBuf} = \mathtt{last}(\mathsf{BufSize}, (s.\mathsf{RxBuf}{+}{+}w))}{\delta^{\mathsf{env}}(s, e) = s[\mathsf{RxBuf} \leftarrow \mathsf{newBuf}][\mathsf{irq} \leftarrow \mathsf{true}]} \quad (\text{recvd})$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad s.\mathsf{RxIntEnable} = \mathsf{true} \\ e = \mathsf{Recv}\ w \qquad w = nil}{\delta^{\mathsf{env}}(s, e) = s} \quad (\text{norcv})$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \quad e = \mathsf{SendingCompAck}}{\delta^{\mathsf{env}}(s, e) = s[\mathsf{TxBuf} \leftarrow \varnothing]} \quad (\text{sent})$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \quad e = \mathsf{NoSendingCompAck}}{\delta^{\mathsf{env}}(s, e) = s} \quad (\text{noack})$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad o = \mathsf{input}\ n \\ n = s.\mathsf{Base} + 0 \quad s.\mathsf{DLAB} = \mathsf{false} \quad s.\mathsf{RxBuf} = w}{\delta^{\mathsf{CPU}}(s, o) = s[\mathsf{RxBuf} \leftarrow \mathtt{tl}\ w][\mathsf{irq} \leftarrow \mathsf{false}]} \quad (\text{read})$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad o = \mathsf{output}\ n\ v \\ n = s.\mathsf{Base} + 0 \quad s.\mathsf{DLAB} = \mathsf{false} \quad s.\mathsf{TxBuf} = w}{\delta^{\mathsf{CPU}}(s, o) = s[\mathsf{TxBuf} \leftarrow \mathtt{last}(\mathsf{BufSize}, (w{+}{+}[v]))]} \quad (\text{write})$$

Figure 14. The environment and CPU transition functions

```
1  void serial_putc (unsigned int c) {
2    unsigned int lsr = 0, i;
3    if ( serial_exists() ){
4      for (i = 0; !lsr && i < 12800; i++) {
5        lsr = serial_read(0x3FD) & 0x20;
6        delay();
7      }
8      serial_write (0x3F8, c);
9      ...
```

Figure 15. The implementation of `serial_putc` in C

```
void serial_puts(char * s, int len) {
  int i = 0;
  while (i < len && s[i] != 0) {
    serial_intr_disable();
    serial_putc(s[i]);
    serial_intr_enable();
    i++;
  }
}
```

Figure 16. The implementation of `serial_puts` in C

By instantiating the device state and transition functions from our general device model in Sec. 3, we create a concrete model of the serial port with the `read` and `write` primitives.

Next, we show how the drivers are specified and verified on top of this model. Fig. 15 shows a code fragment of the function `serial_putc`. There, the `serial_read` and `serial_write` are the two primitives in the serial hardware model, while `serial_exists` is a new primitive (already verified in some underlay) indicating whether the serial device is already initialized. The if statement (line 3) prevents any misuse of `serial_putc()` before initialization. If the s.TxBuf buffer is initially empty, or the device receives a SendingCompAck event during the loop (line 4-6), the program sends the character `c` to the serial port (line 8). The function `serial_putc` is specified as follows:

$$\frac{s.\mathsf{TxBuf} = \varnothing \qquad s.\mathsf{serial\_exists} = \mathsf{true} \\ s' = s[\mathsf{TxBuf} \leftarrow [c]] \\ (e, \ell'_{\mathsf{tx}}) = \mathsf{next}(\ell^{env}, \ell_{\mathsf{tx}}) \qquad (e', \ell''_{\mathsf{tx}}) = \mathsf{next}(\ell^{env}, \ell'_{\mathsf{tx}})}{\mathtt{serial\_putc}(s, c, \ell_{\mathsf{tx}}, \ell^{env}) = (s', \ell''_{\mathsf{tx}})}$$

$$\frac{s.\mathsf{TxBuf} \neq \varnothing \qquad s.\mathsf{serial\_exists} = \mathsf{true} \\ (e, \ell'_{\mathsf{tx}}) = \mathsf{next}(\ell^{env}, \ell_{\mathsf{tx}}) \qquad s' = \delta^{env}(s, e) \\ (s'', \ell''_{\mathsf{tx}}) = \mathtt{serial\_putc}(s', c, \ell'_{\mathsf{tx}}, \ell^{env})}{\mathtt{serial\_putc}(s, c, \ell_{\mathsf{tx}}, \ell^{env}) = (s'', \ell''_{\mathsf{tx}})}$$

The first rule above shows the case when the transmission buffer is originally empty. Here, lsr immediately becomes 1 in the first loop iteration, and the character is written to the transmission buffer in the device right away.

The second rule above shows the case when the initial transmission buffer is not empty. Here, the device performs transition based on the received event e, and repeats the same process until it finally receives the SendingCompAck event. Then, by definition of δ^{env} in Fig. 14, the transmission buffer becomes empty and the next recursive call falls into the first case of the specification.

In Fig. 16, we show the implementation of the driver function serial_puts that writes a string into the serial device by repeatedly calling serial_putc for each character in the input string. Each call to serial_putc is wrapped with calls to serial_intr_disable and serial_intr_enable (both derived from those in Fig. 9) to protect the critical section.

Most of our drivers are implemented in ClightX [14], which is an extension of the CompCert Clight language [23] with abstract states and primitives. For each driver function,

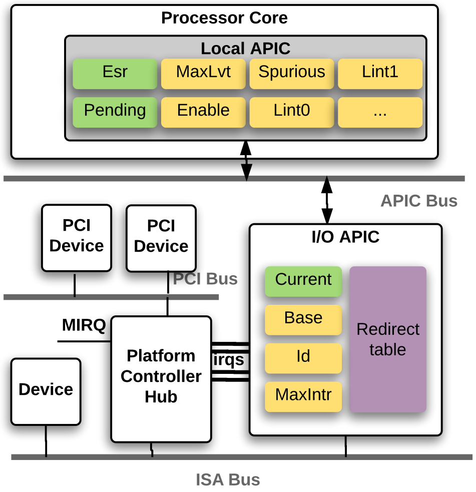

Figure 17. The hardware connections and registers of APIC

$$
\begin{array}{lll}
s = (& \iota : \mathsf{option}(\mathbb{Z} \times \mathbb{Z}), & \rhd \textit{ Current handling IRQ} \\
 & \mathsf{id} : \mathbb{Z}, & \rhd \textit{ IOAPIC ID} \\
 & \mathsf{maxIntr} : \mathbb{Z}, & \rhd \textit{ Max redirection entries} \\
 & \mathsf{irqs} : \mathbb{Z}, & \rhd \textit{ Redirected IRQs} \\
 & \mathsf{masks} : \mathsf{array\ TMask}, & \rhd \textit{ Interrupt line masks} \\
 & \mathsf{dests} : \mathsf{array}\ \mathbb{Z}) & \rhd \textit{ Destinations in LAPIC ID}
\end{array}
$$

Figure 18. Internal states of IOAPIC

$$
\frac{s.\iota = \mathsf{None} \qquad w = \mathsf{IRQ}\ n \qquad s.\mathsf{irqs}[n] = q \\ s.\mathsf{masks}[n] = \mathsf{Unmasked} \qquad s.\mathsf{dests}[n] = d}{\delta^{intr}(s, w) = s[\iota \leftarrow \mathsf{Some}\ (q, d)]} \ \text{(delivered)}
$$

$$
\frac{s.\iota = \mathsf{None} \qquad w = \mathsf{IRQ}\ n \\ s.\mathsf{masks}[n] = \mathsf{Masked}}{\delta^{intr}(s, w) = s} \ \text{(masked)}
$$

$$
\frac{s.\iota = \mathsf{Some}(q, d) \qquad w = \mathsf{EOI} \\ s.\mathsf{irqs}[n] = q \qquad s.\mathsf{dests}[n] = d}{\delta^{intr}(s, w) = s[\iota \leftarrow \mathsf{None}]} \ \text{(EOI)}
$$

Figure 19. IOAPIC transition rules

```c
void ioapic_init(void) {
  int j = 0, maxintr = ioapic_read(1) >> 24;
  while(j <= maxintr) {
    ioapic_write(0x10 + 2 * j, 0x10000 | gsi + j);
    ioapic_write(0x10 + 2 * j + 1, 0);
  }
}
```

Figure 20. The implementation of `ioapic_init` in C

we prove that the concrete implementation satisfies its specification. Our proof is termination-sensitive; we prove total correctness of each function. In the case of `serial_putc`, the maximum iteration counter (12800) is used solely to enforce termination. We maintain an invariant on ℓ^{env} that the serial port receives a SendingCompAck event within 12800 times the `delay()` function is called. This assumption is reasonable because a sending operation that does not complete within this time frame implies an underlying hardware failure.

The proof is achieved semi-automatically using the big-step semantics of the CompCert Clight language. The automation is achieved through Coq tactic libraries including the verification condition generator, arithmetic solvers, various theory solvers (partial map, list, *etc*), and a number of domain specific libraries which handle items such as device transitions and logs. The entire automation libraries are implemented in Coq's tactical language Ltac.

5.2 Interrupt Controller

An IOAPIC device collects interrupts from externally connected devices and distributes them to the corresponding LAPIC. It can be programmed to mask one or more of these interrupt lines, if the OS does not wish to receive interrupts from some device(s).

Fig. 17 illustrates the registers and connections of an IOAPIC device. Following our minimalistic approach, we omit logical destination, remote-IRR configuration, and other features that are not used in our kernel. The internal state of the IOAPIC is defined in Fig. 18, where ι represents the interrupt request currently being processed and its corresponding destination LAPIC ID.

As an interrupt controller, the IOAPIC is treated as a special device. It does not observe any event from the external environment, and thus has neither a local log nor an environmental transition δ^{env}, but instead, it receives interrupt requests from the devices and EOI signals from the LAPIC. We have introduced a special transition function δ^{intr} to specify these interrupt-related behaviors. Accordingly, δ^{intr} takes two kinds of events: IRQ n indicates that an IRQ with number n is triggered by a device; and EOI states that the latest interrupt request has been handled by the OS. The interesting parts of the transition rules for δ^{intr} are shown in Fig. 19.

The interrupt related behaviors of the LAPIC are also modeled, but are omitted here. Note that the models of IOAPIC and LAPIC can be merged into a heterogeneous interrupt controller with the simplified transition rules that are presented in Sec. 4.

In addition to δ^{intr}, the IOAPIC also contains the CPU transition function δ^{CPU} used to specify the read/write primitives of IOAPIC, discussed in Sec. 3.

In order to coordinate the IRQs assigned by the kernel with the external interrupt vector, a kernel usually utilizes the Global System Interrupt (GSI) number. Thus, the IC is first extended into a device object with this extra data as part of its internal state. Then this IC object is further extended into more abstract objects by introducing additional driver layers.

At the top level, the IOAPIC device object provides two primitives, which are used to setup the IRQ mappings in the IOAPIC. Specifically, `ioapic_init` initializes the device when the kernel boots, and `ioapic_enable` links a given interrupt line to an LAPIC when a new device is plugged in or some device changes its working mode.

The code in Fig. 20 shows the initialization of the IOAPIC. It first reads the size of the interrupt redirection table (line 2), and for each entry, marks the corresponding interrupt to be edge-triggered, active high, and masked (i.e. not routed to any LAPIC). The behavior of this function can be described using the following rule:

$$\frac{l = s.\mathsf{maxIntr} \qquad s' = s[\mathsf{masks}[1..l] \leftarrow \mathsf{Masked}] \\ s'' = s'[\mathsf{dest}[1..l] \leftarrow 0][\iota \leftarrow (\mathsf{None}, \mathsf{None})]}{\mathtt{ioapic\_init}(s) = s''}$$

6. Evaluation and Lessons Learned

What We Have Proved The final theorem we proved for our kernel is the contextual refinement relation between our lowest level hardware machine model `x86` (which defines the x86 instructions, the serial device, and the ioapic and lapic devices, etc.), and the top level machine `mCertiKOS` (which defines the abstract system call interface). Let $[\![\cdot]\!]_{\mathrm{x86}}$ and $[\![\cdot]\!]_{\mathrm{mCertiKOS}}$ denote the whole-machine semantics of each machine model, and K denote the (assembly) source code of mCertiKOS, then the theorem is formalized as:

Theorem 1. $\forall P,\ [\![K \bowtie P]\!]_{\mathrm{x86}} \sqsubseteq [\![P]\!]_{\mathrm{mCertiKOS}}$.

The theorem states that for any kernel/user/guest/host context program P, there is a simulation between program P running on top of the top level abstract machine `mCertiKOS`, and the program P linked with the mCertiKOS source code K, running atop the bottom-most machine `x86`.

The abstraction layers also define the data invariants that are proved to hold at any moment of the whole program execution. Some example invariants are: the console's circular buffer is always wellformed, and the interrupt controller states are always consistent, *etc*.

Besides this, our framework automatically derives that all the system calls always run safely and terminate; there are no code injection attacks, no buffer overflows, no null pointer access, no integer overflows, *etc*.

Isolation We take the existing implementation of the CertiKOS infrastructure [14], and extend it with our device and interrupt models. On top of the extended machine model, we have verified a subset of the device drivers in mCertiKOS with 10 abstraction layers. Some layers are introduced to verify concrete driver implementation, while others are introduced purely for logical abstraction (e.g., from a circular console buffer implementation in memory to an abstract list, from the hardware interrupt model to the abstract interrupt model enforcing isolation, *etc*). These abstraction layers are inserted into the existing layers of mCertiKOS as a certified plugin. Thanks to our isolation policy, this does not invalidate most of the existing proofs of mCertiKOS, and the integration only required minimal effort, despite the existing mCertiKOS proofs being unaware of interrupts.

Execution Model and Completeness The majority of our device drivers are specified and verified at C level, then compiled by a modified version of the CompCert verified compiler [14]. The entire kernel (both C and assembly) source code, together with the source code for the verified compiler, are extracted into an OCaml program through Coq's extraction mechanism. When this program gets executed, it compiles the extracted C source code into the assembly, and merges it with the existing assembly kernel source code, to produce a piece of assembly code corresponding to our verified kernel. Thus, our deliverable comes with a piece of assembly code for the entire verified kernel, a high level deep specification of various kernel behaviors, and a machine checkable proof object stating the assembly code running on the actual hardware satisfies the high level specification.

The verified assembly code is then linked with the rest of kernel code (the boot loader and remaining unverified drivers) to produce the actual binary image of the OS. The resulting kernel is practical: it runs on stock x86 hardware and can successfully boot a guest version of Linux.

Verification Effort Using our general device interface, we have modeled a serial device and two interrupt controller devices. On top of these device models, we have verified the related drivers and interrupt handlers. The entire verification effort consists of roughly 20k lines of Coq code added to the existing mCertiKOS verification code base. Regarding the specification, there are 510 lines of code used to specify the machine model including the device hardware, and 126 lines of code for specification of the additional system call interfaces. There are additional 9,829 lines of Coq code that were used to define auxiliary definitions, lemmas, theorems, invariants, *etc*. Note that these 9,829 lines of definitions are outside our TCB, thus does not need to be trusted. In terms of proof size, there are 3,671 lines of Coq code for the layer refinement proofs, 3,589 lines for code verification, 1,802 lines for proving invariants, and 307 lines for linking different modules together.

The entire verification effort took roughly 7 person months, the majority of which went into the design and development of the framework itself, including the extended machine model, general device framework, the interrupt refinement, and the tactic libraries for automating most of the non-intellectual parts of verification task. We anticipate the cost of verification for future drivers would be dramatically reduced.

Bugs Found An extended version of the mCertiKOS kernel has been deployed in a practical system that is used in the context of a large DARPA-funded research project [14]. Yet, through the verification of the console driver, we found a critical bug which may lead to the loss of many characters received from the serial device. The bug was in the implementation of the circular console buffer, where, in some rare cases, the read and write positions to the buffer array overlap, causing the entire contents in the buffer to be lost. The bug was caught when we tried to establish the contextual refinement between the concrete implementation of the circular buffer and its abstract list representation.

Another bug was found in the code for initializing the serial device, where the interrupt was not configured correctly by accidentally setting the Interrupt Enable Register (IER) before the DLAB was unset. This was caught when we tried to prove the initialization code against its specification.

7. Limitations and Future Work

Our verified kernel assumes correctness of the hardware. In our device model, we enforce a set of invariants on the list of external events, which specifies correct hardware behaviors, e.g., all the 8 bit characters are 8 bits, serial port eventually transmits its contents, *etc*. Every function that tries to write to the serial device first busy-waits reading the device's transmission buffer status until it becomes empty. We rely on the above assumption to prove that the loop eventually terminates and when it does terminate, the transmission buffer is empty so we can write to the device again. In the future, we plan to extend our device drivers to handle the hardware errors, e.g., when the serial device does not acknowledge the previous output was successful in the time period specified in the hardware documentation. In this case, we can add states to the device state machine to represent those erroneous cases, and add appropriate error handling code. The process is the same as a non-faulty device. For example, when the serial port does not transmit its contents in a certain amount of time, we can reset the serial port and try again.

Furthermore, as with any verified system, the specification of hardware devices and the top level system call primitives have to be trusted. For the hardware specification, we only model the set of features utilized by the kernel, instead of modeling the entire hardware manual. Our system calls are specified at the top abstraction layer, where all implementation details are hidden. These lead to specifications of a fairly small size (636 lines of Coq code), limiting the possible room for errors, and easing the review process.

Sometimes, the compiler may unsoundly optimize away some memory accesses to the memory mapped registers, e.g., a dead read of a memory mapped device register. In this case, we can use the CompCert built-in calls like `volatile_load`, which are not supposed to be optimized away by CompCert. On the other hand, those operations can also be directly implemented in assembly in our framework.

Some parts of the TCB from the original mCertiKOS still remain, including the bootloader, the Coq proof checker, and the pretty-printing phase of the CompCert compiler.

Verification of Other Drivers Some device drivers (i.e., those with underlined names in Fig. 1) in mCertiKOS still remain unverified. With the new compositional framework and automation libraries we have developed, we anticipate that the rest of the drivers can be verified with a reasonable amount of proof engineering effort.

Among those drivers shown in Fig. 1, the keyboard and text-mode VGA drivers can be verified easily since they are not much more complex than the serial driver. The timer and TSC drivers can also be verified, but mCertiKOS's assembly machine must first be parametrized with a good cost model for x86 instructions.

The disk driver (including the PCI and AHCI drivers) is the largest driver in our kernel. The mCertiKOS kernel communicates with the hard disk through the AHCI controllers using memory mapped registers (mCertiKOS also communicates with the APIC using memory mapped registers through the verified drivers). We believe that our device model is general enough to model required features for these devices used by the disk driver. We have already started applying our approach to verify the mCertiKOS disk driver which will also serve as a basis for building a certified file system.

Concurrency Reasoning Our current certified kernel assumes a runtime environment consisting of a single processor, and user processes do not preempt each other. Therefore, our work so far does not support preemptive nor multicore concurrency. With general concurrency, different user/kernel threads may share memory and use a wide variety of synchronization mechanisms that must also be verified. The techniques presented in this paper does not provide such support (since the logical CPUs for devices and the main kernel/user CPU do not share any state).

We are working on adding general concurrency support and believe that it is still a good idea to multiplex each CPU core into multiple logical CPUs. With concurrency support, we hope that each such logical CPU will have its own (logical) scheduler, (logical) memory, and collection of kernel or user threads that may share memory.

Device drivers often do need to modify kernel memory, as in Linux bottom halves (implemented as low-priority threads) or deferred procedure calls in Windows. With support of these "concurrency-aware" logical CPUs, we believe that our technique can be extended to support low-priority kernel threads dedicated to serve Linux bottom-halves. The idea is to treat these device-serving kernel threads (and memory) as part of the logical CPU dedicated for each device. Since we are already treating device driver code as if it runs on its "device" CPU, it is quite natural to place those device-serving kernel threads on the logical (device) CPUs as well.

8. Related Work

Gu et al. [14] pioneered the compositional proof machinery that builds certified OS kernels using deep specifications and certified abstraction layers. We built our certified interruptible

OS kernel and device drivers using the same methodology. Our new compositional proof framework, however, adds two novelties. First, we show how to handle device objects, which are different from regular mCertiKOS kernel objects. The states in these new device objects can be updated either by the kernel (via device methods) or by an external environment, whereas regular mCertiKOS objects can only be mutated synchronously by the CPU; device objects can also be asynchronously mutated by the environment; we introduce a new abstraction of per-device event logs to handle this asynchronicity. Second, we support formal reasoning about kernel code and device drivers running on multiple logical CPUs (see Fig. 4) while under Gu et al. [14], all verified code at each layer must run on a single CPU (see Fig. 2); we treat the driver stack for each device as if it were running on the logical CPU dedicated to that device.

Klein et al. [20] were the first to verify the correctness and security properties of a high-performance L4-family microkernel in a modern mechanized proof assistant [28]. To make verification easier, they introduced an intermediate executable specification to hide C specifics. Gu et al. [14] built their certified mCertiKOS kernel (in Coq) by decomposing it into many abstraction layers; such fine-grained layer decomposition led to significantly lower proof and development effort and also better extensibility. Both kernels, however, lack a realistic interrupt model, so reasoning about interruptible code is not supported. The device drivers are not verified in either kernel.

Hawblitzel *et al* [15] has recently developed a set of new tools based on the Dafny verifier [22] and Z3 SMT solver [8], and applied them to build their Ironclad system which includes a verified kernel (based on Verve [34]), verified drivers, verified system and crypto libraries, and several applications. This is another impressive effort that advances the frontier of system software verification. However, the abstract device model in Ironclad is too high level to model many hardware details. The Verisoft team [27] has done a large body of work aiming to verify an OS kernel with device drivers in a proof assistant [1, 2, 3], but their stability requirement is too restrictive and does not fit well into many hardware devices.

Feng et al. [11, 12] developed a formal Hoare-logic-like framework for certifying low-level system programs involving both hardware interrupts and preemptive threads. Using ideas from concurrent separation logic [26], they showed how to use ownership-transfer semantics to model enabling and disabling interrupts and reason about the interaction among interrupt handlers, context switching, and synchronization libraries. They successfully certified a preemptive thread implementation (as libraries) and a set of common synchronization primitives in the Coq proof assistant. Their work, however, did not model any hardware device or interrupt controller, and their interrupt model is much simpler than ours. They also only proved the partial correctness property (for their certified library functions), not the strong contextual refinement property which we proved for our kernel. Of course, since our current certified kernel does not support preemptive concurrency, we believe there are good opportunities for combining their techniques (for reasoning about preemptive concurrency) with our refinement-based approach.

Ryzhyk *et al* [29, 30] have done much work on the synthesis of device drivers from the specifications. In their approach, both the device and the interface of the corresponding driver are modeled as state machines, which communicate via messages. The generated driver code requires some unverified run-time support. Furthermore, the correctness of the drivers is limited to the synthesized C programs, not the compiled assembly code running on the actual hardware.

In the work of Duan and Regehr [10], a UART driver in the ARM architecture with interrupt is verified. They have created an abstract device model which gets plugged into the instruction set of the ARM6 architecture. In their model, the device state is mixed into the machine state. Thus, they have to carefully consider the interleavings between the execution of the device and the CPU. Albeit a realistic UART model, the driver only consists of 20 lines of the assembly code. The framework is later ported to the Cambridge model of the ARMv7 architecture [9]. Schwarz *et al* [31] proposed a device model where all the devices are executed nondeterministically in parallel with a single core processor. Based on the model, they have proved several noninterference properties among the processor and devices which potentially use DMA or interrupts. Monniaux *et al* [25] have verified a driver with a USB OHCI controller model written in C with a static analyzer. They have showed the verified driver exhibits no undefined behavior.

There are many lines of work in verifying device drivers based on model checking. Amani *et al* [4] proposed an approach to automatically verify the protocols between drivers and the operating system. Thomas Witkowski [33] and Alexey Khoroshilov [18] have verified specific protocols of some Linux drivers using the model checker SATABS and DDVERIFY. Kim *et al* [19] have verified a driver for a flash memory in NuSMV, Spin, and CBMC. Ball *et al* [6] have developed the static analysis tool SLAM, which is included in the Microsoft Windows Driver Developer Kit.

9. Conclusions

We have presented a novel compositional framework for reasoning about the end-to-end functional correctness of device drivers in a certified interruptible kernel. Our formalization of interrupts follows the abstraction-layer-based approach and includes a realistic hardware interrupt model and an abstract model of interrupts (which is suitable for reasoning about interruptible code). We have proved that the two interrupt models are contextually equivalent. We have successfully extended an existing verified non-interruptible kernel with our framework and turned it into an interruptible kernel with verified device drivers. The implementation, specification, and proofs are all done in a unified framework (realized in the Coq proof assistant), yet the mechanized proofs verify the correctness of the assembly code that can run on the actual hardware. To the best of our knowledge, this is the first verified interruptible operating system with device drivers.

Acknowledgments

We thank Quentin Carbonneaux, Hernán Vanzetto, Mengqi Liu, Jérémie Koenig, other members of the CertiKOS team at Yale, our shepherd Jean Yang, and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by NSF grants 1065451, 1319671, and 1521523 and DARPA grants FA8750-12-2-0293 and FA8750-15-C-0082. Hao Chen's work is also supported in part by China Scholarship Council. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] E. Alkassar. *OS Verication Extended - On the Formal Verication of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, Saarland University, Computer Science Department, 2009.

[2] E. Alkassar and M. A. Hillebrand. Formal functional verification of device drivers. In *Verified Software: Theories, Tools, Experiments Second International Conference (VSTTE), Proceedings*, pages 225–239, Toronto, Canada, Oct. 2008.

[3] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, pages 71–85, Edinburgh, UK, Aug. 2010.

[4] S. Amani, P. Chubb, A. Donaldson, A. Legg, L. Ryzhyk, and Y. Zhu. Automatic verification of message-based device drivers. In *Systems Software Verification*, pages 1–14, Sydney, Australia, Nov 2012.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 73–85, New York, NY, USA, 2006. ACM.

[6] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 35–42, Austin, TX, 2010. FMCAD Inc.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[8] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340, 2008.

[9] J. Duan. *Formal verification of device drivers in embedded systems*. PhD thesis, University of Utah, 2013.

[10] J. Duan and J. Regehr. Correctness proofs for device drivers in embedded systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[11] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM Conference on Programming Language Design and Implementation*, pages 170–182, 2008.

[12] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reasoning*, 42(2-4):301–347, 2009.

[13] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th Conference on Large Installation System Administration*, LISA '06, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[14] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 595–608, 2015.

[15] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[16] Intel. 82093AA I/O advanced programmable interrupt controller (I/O APIC) datasheet. Specification, May 1996.

[17] Intel. Multiprocessor specification, version 1.4. Specification, May 1997.

[18] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin Heidelberg, 2010.

[19] M. Kim, Y. Choi, Y. Kim, and H. Kim. Formal verification of a flash memory device driver – an experience report. In K. Havelund, R. Majumdar, and J. Palsberg, edi-

tors, *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2008.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, US, Oct 2009.

[21] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), Feb. 2014.

[22] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010)*, pages 348–370, 2010.

[23] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[24] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformation. *Journal of Automated Reasoning*, 2008.

[25] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In C. Kirsch and R. Wilhelm, editors, *EMSOFT 2007, 7th ACM International Conference On Embedded Software, Proceedings*, pages 30–36. ACM & IEEE, 2007.

[26] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, pages 49–67, 2004.

[27] W. Paul, M. Broy, and T. In der Rieden. The Verisoft XT Project. `http://www.verisoft.de`, 2007.

[28] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[29] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, pages 73–86, Big Sky, MT, US, Oct 2009.

[30] L. Ryzhyk, A. C. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 661–676, Broomfield, CO, USA, Oct 2014.

[31] O. Schwarz and M. Dam. Formal verification of secure user mode device execution with DMA. In E. Yahav, editor, *Hardware and Software: Verification and Testing*, volume 8855 of *Lecture Notes in Computer Science*, pages 236–251. Springer International Publishing, 2014.

[32] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2016.

[33] T. Witkowski. Formal verification of Linux device drivers. Master's thesis, Dresden University of Technology, May 2007.

[34] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. 2010 ACM Conference on Programming Language Design and Implementation*, pages 99–110, 2010.