Efeu: generating efficient, verified, hybrid hardware/software drivers for I^{2}C devices

Daniel Schwyn[*]
daniel.schwyn@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Zikai Liu[*]
zikai.liu@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Timothy Roscoe
troscoe@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Writing device drivers is notoriously hard, and driver bugs are a major cause of system failures and vulnerabilities. The problem is particularly acute in bus-based protocols like I^{2}C, where driver correctness is only half the story: correct functioning of the complete subsystem depends on *all* components on the bus interoperating correctly. Unfortunately, developers cannot control all aspects of a platform, and must interact with existing devices (peripherals and/or hardware bus controllers) which may misbehave. Failures in a protocol like I^{2}C, often used in critical low-level system management, can result in permanent damage to the hardware, whether a server or a satellite.

Existing techniques for creating high assurance drivers rarely tackle this interoperability issue. We present Efeu, a framework for implementing verifiably interoperable drivers for I^{2}C devices. Using model checking-based verification, Efeu generates driver implementations in software, reconfigurable logic for FPGAs, and, notably, *combinations of both*. The split between software and hardware can be varied at implementation time and the hardware/software interface is generated automatically, enabling efficient exploration of the design space. Using Efeu, we design and evaluate a verified I^{2}C driver stack, and demonstrate that Efeu finds optimal hardware/software tradeoffs to favor either throughput, CPU usage or FPGA footprint. For each objective, Efeu generates drivers with performance comparable with hand-optimized hardware/software drivers.

CCS Concepts: • **Software and its engineering** → **Input / output**; **Model checking**; *Domain specific languages*; *Source code generation*; • **Hardware** → **Buses and high-speed links**; **Model checking**; Reconfigurable logic and FPGAs.

[*]Both authors contributed equally to this research.

Keywords: I2C, interoperability, verified drivers

ACM Reference Format:
Daniel Schwyn, Zikai Liu, and Timothy Roscoe. 2025. Efeu: generating efficient, verified, hybrid hardware/software drivers for I^{2}C devices. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3689031.3696093

1 Introduction

We present Efeu, a system for generating drivers for complete I^{2}C subsystems from formal specifications. The resulting software stacks are suitable for server Baseboard Management Controllers (BMCs), embedded controllers in mobile phone systems-on-chip (SoCs), or resource-constrained Internet-of-Things (IoT) devices. Moreover, the I^{2}C drivers are high-performance and verified to behave correctly using a model checker, even when the system includes devices which *do not correctly follow* the I^{2}C standard. Finally, the generated drivers can be in C, or Verilog for FPGAs, *or a hybrid of the two*, enabling efficient determination of the optimal hardware/software split for a given platform.

The I^{2}C protocol (short for Inter-Integrated Circuit) is at the heart of almost all modern computer systems and critical to their correct behavior. Bugs in I^{2}C can result in inefficiencies in energy usage, hardware lockups, and in some cases permanent hardware damage. A correct I^{2}C network within a phone or server is essential.

At the same time, I^{2}C has features that make creating high-assurance driver software particularly challenging. It is a bus-based protocol but unlike, say, PCIe or USB it does not feature hardware facilities for isolation. This means that a bug in the driver for a single device can disable the entire subsystem at runtime. Unfortunately, an I^{2}C subsystem for a typical machine includes dozens of devices from many vendors, which (like PCIe) often exhibit *quirks*: deviations from the standard which can confuse other devices or controllers.

For this reason, hardware I^{2}C controllers may interoperate with only a limited number of empirically compatible devices, and are frequently replaced with "bit-banging" drivers which directly manipulate bus signals from handwritten software. This impairs protocol performance, increases CPU load, and reduces energy efficiency.

Despite this, the design and implementation of a trustworthy I^{2}C software stack has received relatively little attention from the research community. Efeu addresses this challenge.

Efeu provides a language for specifying I^{2}C devices, which includes the ability to express known deviations from the protocol (quirks) by the devices. A complete hardware platform's I^{2}C network can be expressed by composing such specifications within the language.

The Efeu compiler can then generate a driver for the I^{2}C host controller together with drivers for the all the attached devices. This generated driver suite can be in C, or alternatively in Verilog for synthesis onto an ASIC or FPGA, increasingly used for server board management controllers.

Crucially, the Efeu compiler can place boundaries between generated software and hardware functionality *between any two layers* in the I^{2}C protocol stack. In its simplest form, this enables the optimal split between hardware and software implementation to be empirically determined, without writing any additional code. It can also be used for debugging, for example by replacing an optimized hardware layer with a more instrumented software layer.

The Efeu compiler also generates a specification of the complete I^{2}C subsystem (with quirks) in Promela [23], allowing correctness to be model-checked using SPIN [28].

This is our second attempt at engineering verified I^{2}C stacks, following a somewhat simpler previous approach [30]. Efeu uses the same Promela specifications that the lower 3 layers (`Symbol`, `Byte`, `Transaction`) of the I^{2}C stack are verified against, but is otherwise completely new. In contrast to our previous approach, Efeu is designed to enable the verification of realistic I^{2}C topologies with multiple attached devices, and targets the generation of both realistic software *and hardware* components for implementing efficient, usable I^{2}C stacks. We provide a detailed evaluation of the verification and driver performance in section 4 and section 5 respectively.

In the next section, we elaborate on why I^{2}C matters, what makes it different from driver assurance for interconnects like PCIe and USB, and the canonical structure of an I^{2}C stack. Following this, in section 3, we describe the Efeu language, compiler, and verifier, and how it addresses the challenges we have laid out. In section 4 we explain how the I^{2}C stack is model checked, and show that it can be done in practical time, and in section 5 we show that Efeu allows a range of different trade-offs in hardware/software implementations to be generated from the specification of a real hardware platform, and also that the resulting drivers are comparable with hand-tuned software and hardware in terms of throughput, CPU cycles, and hardware footprint. We survey the broader landscape of high-assurance device drivers in section 6, and conclude in section 7.

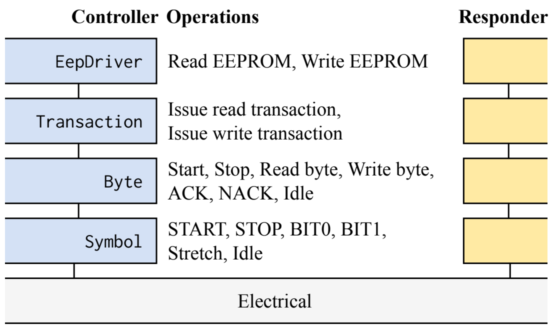

Figure 1. The I^{2}C stack with an EEPROM driver as an example application.

2 Background and problem statement

In this paper, we address how to create high-performance, correct driver stacks for *bus-based* devices, in particular those using I^{2}C. By *high-performance*, we mean competitive with state-of-the-art handwritten drivers in terms of throughput, latency, and CPU usage. By *correct*, we mean that the driver is proven to function as specified and not interfere with other devices sharing the bus.

Driver defects have long been identified as a major cause of system failures and vulnerabilities [11, 26, 56], resulting in much work on improving driver assurance via synthesis of drivers from specifications, post-hoc verification of manually written drivers, and formally-derived hardware/software codesign. We survey this work in section 6, but focus here on what makes the high-assurance I^{2}C case different.

2.1 The importance of I^{2}C and related protocols

Despite receiving much less attention in the literature than devices using, e.g. PCIe or USB, I^{2}C and related protocols SMBus [62] and PMBus [67, 68] are fundamental to the operation of almost all computers today, from small IoT devices through mobile phone SoCs and platforms to large-scale servers and rack-scale systems [78].

Whether controlled by the conventional OS kernel, or by "hidden" parts of the *de facto* OS [25] like monitor code or BMC firmware, I^{2}C is the base protocol used to control almost all the components of a machine: configuring voltage and clock frequency, monitoring temperature and power, etc. I^{2}C bugs lead to board lockups, pathological power inefficiencies, or at worst hardware damage [1, 8, 10]. Hardware confers great privilege on the software stack controlling the I^{2}C bus in a machine; indeed, this is a security concern in its own right, a topic we return to in section 7.

2.2 What makes I^{2}C different?

I^{2}C [53] is a serial bus protocol using two wires, the serial clock line (SCL) and the serial data line (SDA). An I^{2}C device is either a *controller* or a *responder*[1] (Figure 1). Controllers initiate and control data transfers between themselves and responders, identified by 7-bit addresses. An additional bit distinguishes read transfers (a responder transmits data to the controller) and write transfers (the controller transmits data). SDA is driven by the transmitting device while SCL is normally only driven by controllers, but responders can "stretch" the SCL clock if they cannot keep up.

What makes I^{2}C specifically challenging is the need for interoperability. I^{2}C connects many components in an SoC or motherboard, and these are designed and supplied by many different vendors. A correctly I^{2}C functioning subsystem depends not only on the driver for each individual device being correct, but also on *all* the devices and drivers interoperating correctly.

Of course, interoperability is not a challenge restricted to I^{2}C– USB and PCIe devices must also work together, for example. However, both USB and PCIe controllers by design provide isolation in hardware between devices, such that drivers do not need to be aware of the whole protocol stack. Even so, deviations from standards are common: the Linux kernel contains over 6000 lines of code for handling so-called *quirks* [21, 63] in PCI(e) devices alone.

Interoperability is different for I^{2}C devices. The bus-based nature of the protocol means that a misbehaving device or controller can prevent all other I^{2}C devices on the bus from functioning. A single driver bug can render the entire I^{2}C bus unusable. Unfortunately, like PCIe, numerous I^{2}C devices have quirks [2, 52]. For example, the KS0127 video decoder [60] expects the "stop streaming data" command to appear in a non-standard position [20] and block the bus indefinitely if the controller is unaware of this.

This means that a "one-driver-one-device" approach to high-assurance drivers is insufficient. It also means that any formal approach must be able to handle device quirks.

Moreover, in practice I^{2}C controllers are *often implemented in software.* A hardware implementation is usually provided, but generally unused due to a lack of confidence in interoperability. The Raspberry Pi I^{2}C controller, for example, does not correctly handle clock stretching. Controller and responders can thus desynchronize, leading to lockups and data corruption [49]. This issue is *not* a driver bug in the traditional sense: the I^{2}C controller driver can correctly program the controller and, as long as no responder uses clock stretching, the system works correctly.

Consequently, many I^{2}C controllers consist of low-level "bit banging" software directly driving the SCL and SDA signals, even if a hardware controller is available. This allows post-hoc workarounds for quirks, but has a cost. While I^{2}C bandwidth requirements are relatively modest this still results in slowdown (see section 5.2) and the heavy use of CPU cycles (and associated energy) becomes an issue for low-end embedded devices. Recently, some vendors have proposed using reconfigurable logic to help with I^{2}C functionality [24, 65].

The bug in the Raspberry Pi also shows that relying on hardware manufacturers to fix bugs once discovered is not a solution: the bug was originally discovered in the first Raspberry Pi model in 2013. Newer models released in 2020 are however still affected by it [33].

2.3 The I^{2}C protocol stack and ecosystem

We now describe the I^{2}C protocol from bottom up (paraphrasing the standard [53]), alongside our model of the protocol (Figure 1). We show a complete, end-to-end example in Figure 2. Both controllers and responders have the same layers described below, but differ in their implementations.

At the **`Electrical` layer**, both SCL and SDA have external pull-up resistors, and devices may only drive the lines low. Multiple clock speeds are defined, but we target the commonly used Fast Mode (400 kbit/s, or 400 kHz SCL). In the Efeu model, the `Electrical` layer represents the levels with `0` and `1` and models the pull-down behavior with bit operations. We do not model the precise bus timing, but assume a bus adapter that translates bits into half cycles on the bus, allowing the stack to work with discrete time. Currently, this adapter is written by hand, but could be synthesized using an approach like Chinook [14].

The **`Symbol` layer** converts between I^{2}C symbols (START, STOP, BIT0, and BIT1) and the SCL and SDA electrical levels using the encoding in Figure 2. Two further operations, IDLE and STRETCH, are defined: IDLE is a no-op to the bus, and STRETCH performs clock stretching, pulling SCL low for one cycle. This is the only operation with which a responder can drive SCL. Otherwise, responders passively respond to clock cycles. In our model, controllers handle clock stretching at the `Symbol` layer, waiting for its completion before returning to the upper layers.

I^{2}C is byte-oriented. The **`Byte` layer** encodes and decodes bytes to and from bits, as well as acknowledging each byte or not: ACK is encoded to the BIT0 symbol and NACK to BIT1. `Byte` also detects arbitration loss if multiple controllers collide on the bus, reporting this upwards.

Above this at the **`Transaction` layer**, an I^{2}C transaction starts with a START symbol, the 7-bit target device address, and a read/write bit. Payload bytes follow, supplied by either the controller or the responder depending on the transaction type and are ACKed or NACKed by the receiving device. A transaction is terminated by a STOP symbol, or another START – known as a repeated START – in which case the controller keeps the bus busy without releasing it.

The top layer is specific to a responding device class. In this paper, we use the **`EepDriver` layer** as a running example, modeling a byte-addressable EEPROM, the Microchip 24AA512 [51]. The controller `EepDriver` issues transactions

[1]The standard refers to them as "master" and "slave"

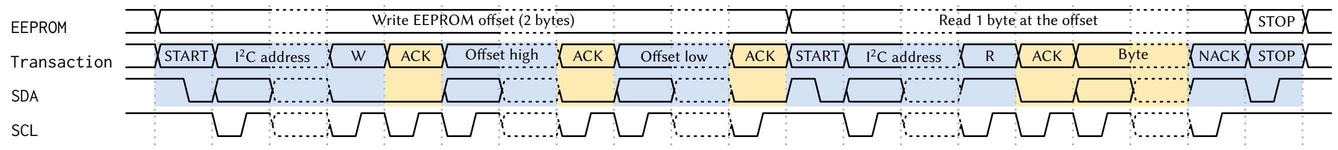

Figure 2. Timing diagram of a 1-byte read at a given EEPROM offset. SDA is driven either by the controller (blue) or the responder (yellow) in a cycle. SCL is always driven by the controller. Dashed levels indicate more than one cycle.

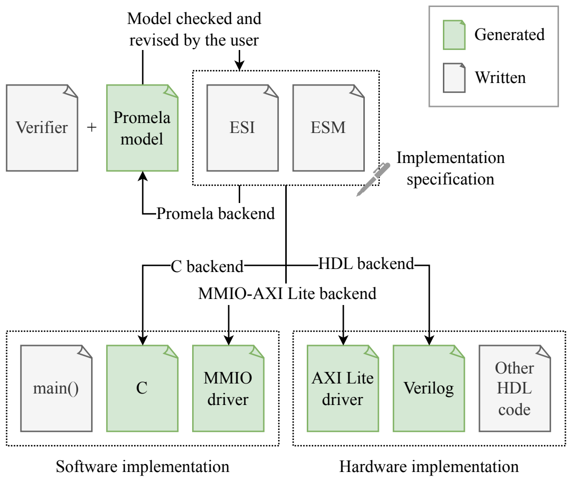

Figure 3. Efeu workflow.

```
layer CTransaction;
layer CEepDriver;

enum CTAction {
  CT_ACT_WRITE,
  CT_ACT_READ,
  CT_ACT_STOP,
  CT_ACT_IDLE,
};

enum CTResult {
  CT_RES_OK,
  CT_RES_FAIL,
  CT_RES_NACK,
};
interface <CTransaction,
           CEepDriver> {
  <= {
    CTAction action;
    u8 addr;
    u8 length;
    u8 data[16];
  },
  => {
    CTResult res;
    u8 length;
    u8 data[16];
  }
};
```

Figure 4. ESI for a controller `Transaction` and `EepDriver` layers and their interface. "<=" shows a channel from the `CEepDriver` to `CTransaction`, and vice versa.

to perform EEPROM *operations*. To write data to the EEPROM, a write transaction is issued with a two-byte data offset followed by the payload bytes to write starting from the offset. To read data from the EEPROM, `EepDriver` first issues a write transaction carrying the data offset, followed immediately by a read request to stream out data starting from the offset. Figure 2 shows the timing diagram of reading 1 byte from an offset.

3 Efeu design and implementation

The workflow of Efeu is shown in Figure 3: a developer writes the implementation specification (devices and topology) of the platform, and Efeu translates it into a Promela model for model checking and iterative refinement. When they are satisfied with the specification, Efeu generates implementations in C and Verilog.

3.1 Specifying the driver stack

The structure of Efeu specifications follows the one we developed in previous work [30]: developers write specifications top-down, first declaring layers and then defining each of them as an indefinitely-running finite state machine (FSM). The layered structure makes components reusable across specifications (see section 4).

Some design decisions in our previous approach were however unsuitable for generating real-world drivers. A major one was to fix the direction of communication between layers at specification time: the layer that `calls` another layer initiates communication and the other layer responds by `yielding`. As we detail in section 3.3, this lacks the flexibility needed in real world settings. Communication primitives in Efeu specifications are therefore symmetric, and the compiler chooses a suitable implementation for the desired scenario. To enable this, the layer declarations also need to include interface declarations. We developed a new lightweight domain specific language (DSL) for this called ESI (Efeu System Information).

Figure 4 shows an ESI example of controller `Transaction` and `EepDriver` layers and the interface connecting them. Interfaces consist of a channel in each direction. In a channel, each data field has a type and a name. Supported types include `bit`/`bool`, unsigned byte (`u8`), 16-bit and 32-bit integers (`i16` and `i32`), enumerations, and 1-dimensional arrays.

Layers are then specified as FSMs in another DSL. In our previous approach [30] we focused on verification and designed the specification language to be easy to translate to Promela. As the new communication model in Efeu required us to change the specification language for layers anyway, we decided to instead make it resemble a subset of C. This improves its usability for writing specifications of real systems by allowing existing tools like syntax highlighting, formatting and static checking to be reused. The new DSL is called ESM (Efeu State Machine) and differs from C as follows:

- The only built-in types are `bit` and `bool` (one-bit), `byte` (or `unsigned char`), `short` and `int`.
- ESI Interface definitions become `struct`s; no other `struct` definition is allowed.
- ESI enumerations become C `enum`s; other `enum`s are allowed, but unlike C, corresponding integer values may not be specified.
- Only the unary operators plus (`+`), negate (`-`), bitwise not (`~`), and boolean not (`!`) are supported.
- The only control flow statements supported are `if`, `while`, and `goto`.
- Each layer is an indefinitely-running function without `return`. No other function definitions are allowed.
- Promela [23] reserved words like `len` and `timeout` are also reserved in ESM.
- Otherwise, ESM supports no pointers, global variables, functions, or variable initialization at declaration time.

Within each layer function, two language primitives, `talk` and `read`, are used to communicate with adjacent layers. Their function stubs are generated by the Efeu compiler. Given two adjacent layers `A` and `B`, `A talk B` is a blocking *round-trip* communication over the interface between `A` and `B` resembling two coroutine switches [39, 48]: values are sent from `A` to `B`, and the operation continues in `A` when `B` issues a corresponding `B talk A` with return values. `A read B` only differs in that no initial values are passed to `B` but `A` waiting for synchronization.

3.2 Efeu compiler overview

The Efeu compiler ESMC compiles ESI and ESM code to various targets: Promela code for model checking, C code for software drivers, and/or Verilog code for hardware drivers. Previous work pioneers translating C-like languages to Promela [29, 32] and Verilog [31, 44, 46, 57]. Efeu adopts the idea but apply it to a single unified specification language.

ESMC is built on Clang/LLVM [42, 69, 73] and leverages existing components. ESMC adds 7823 lines of C++ code (excluding blank lines and comments) to Clang/LLVM, together with 136 system tests to cover this code. ESI files use a custom lexer/parser into an internal representation. ESM code is processed by the Clang frontend, which performs type checking and constructs an abstract syntax tree (AST). Any errors, warnings and/or comments are reported to the user in a readable format through the Clang diagnosis engine [70]. By reusing Clang, ESMC inherits formatted diagnostic messages and C preprocessor support, enabling conditional compilation, compile-time polymorphism, and modular design. The backends operate on the Clang AST.

3.3 C backend

Efeu generates C that can then be compiled into executables or libraries. Recall that in ESM, layers are indefinitely-running FSMs written as functions without return. A straightforward implementation option would transform these functions into threads and the `talk`/`read` operations into inter-thread communication, but this introduces scheduling overhead and dependence on the OS-specific thread implementation.

Instead, we implement layers as stack-based coroutines purely in C with minimal runtime support required, ensuring portability across systems and highly efficient switching between layers.

In principle, to implement two connected layers as stack-based coroutines, either one can be the callee. However, in real applications, as generated drivers are integrated with the rest of the OS, the choice of which layers become callees affects the usability of the generated code. We therefore introduce the concept of a *call graph* in the C backend and allow the developer to specify an *entry point* to this graph at compile time, which the compiler provides a function interface to.

Figure 5 shows three examples. In the leftmost one, when the generated code is intended to be used as a driver library, it is naturally invoked with the entry point as a top-level function. Efeu performs a depth-first search (DFS) on an undirected graph where nodes are the layers and edges are the connections. The `talk`/`read` operations on the forward edges become function calls, and the reversed ones become continuations. Code generated in this way can be directly compiled into a usable library. The second example shows the ideal graph when the generated code is used as a part of a server process in the OS: an event loop ("callee" of the OS scheduler) reads values from the bus driver, invokes the stack from the bottom, reads the next electrical levels to write to the bus, and sends them to the physical bus driver. The third example is a command-line simulator of one controller and one responder. The `Electrical` layer is called by an infinite loop. The top layer of the controller reads inputs from the user, which go through the whole stack, and the results are printed at the top layer of the responder.

Given a call graph, `talk` and `read` operations become function calls and continuation calls. Figure 6 shows a `talk` operation. To pass values between layers, function (layer) signatures are also transformed. When a layer is a callee in one connected pair, it passes input values from the other layer by value, and output values by reference (as pointers). The transformations use the Clang Rewriter [71]. Other parts of code remain unchanged as they are already valid C.

3.4 Verilog backend

Efeu also generates Verilog for programmable logic like FPGAs. The backend reuses more of the Clang/LLVM pipeline.

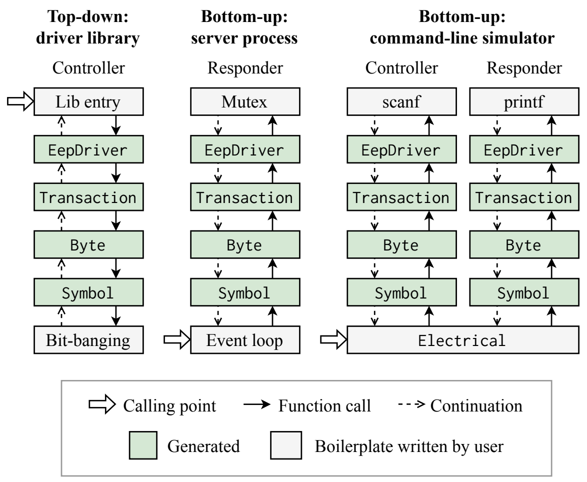

Figure 5. Examples of call graphs.

ESI

```
layer This;
layer Other;
interface <This, Other> {
  => { i32 x;
       i32 y; },
  <= { i32 z;
       i32 w; }
};
```

ESM

```
v = ThisTalkOther(x, y);
```

Transformed Direct Call

```
Other(x, y, &v.z, &v.w);
```

Transformed Continuation

```
*_ThisToOther_x = x;
*_ThisToOther_y = y;
_continuation_pos = <N>;
return;
_continuation_<N>:
v.z = _OtherToThis_z;
v.w = _OtherToThis_w;
```

Figure 6. Transforming a `talk` into a function call or a continuation. <N> is a placeholder for a newly allocated continuation index.

The Clang AST is further lowered to LLVM IR [74], and thence transformed to Verilog. LLVM IR uses static single-assignment (SSA) form, which maps well to combinatorial logic in Verilog. Each function (layer) becomes a Verilog module. Basic blocks are converted to states. IR instructions are translated into blocking assignments in Verilog to preserve data dependencies between instructions. They will be analyzed by the electronic design automation (EDA) tool used to implement the circuit to extract parallelism. Arithmetic instructions are translated to the corresponding ones in Verilog. Branch instructions (conditional, unconditional, switch, the ϕ node [74]) become state transfers. Instructions that involve pointers (such as stack allocation, load and store) are converted to operations on registers. As ESM disallows pointers and global variables, all pointers appearing in IR can be located with static analysis. The detailed translation rules can be found in a separate report [45].

`talk` and `read` require special handling, since they involve communications with other layers and require more than one cycle. Efeu uses ready/valid handshaking, a flexible and lightweight design widely adopted in designs like the AMBA AXI4 protocol [5]. On a unidirectional channel, the sender outputs data signals and a *valid* signal. The receiver outputs *ready* when it can accept more data.

A `talk` results in the following four states (`read` results in states 2 to 4), encoded as additional basic blocks.

1. Output data. Assert valid. Wait until peer asserts ready.
2. De-assert valid. Assert ready. Wait until peer asserts valid.
3. Save data from peer. Assert ready.
4. De-assert ready.

3.5 Generating hybrid hardware/software drivers

Efeu can also generate *hybrid* drivers, with multiple hardware/software layer boundaries defined at compile time. The hardware/software interface is based on AXI Lite [5]: between layers that straddle the boundary, data fields and valid and ready signals from the handshaking protocol are memory-mapped at different offsets. Figure 7 shows the case with `EepDriver` in hardware and `Transaction` in software, corresponding to the ESI interface in Figure 4.

The hardware handshaking protocol assumes sender and receiver being in the same clock domain [5]: after a cycle where both valid and ready are raised, the sender needs to lower the valid signal immediately on the next clock cycle if there is no more data to send. Otherwise, the receiver treats data on the bus as the next valid packet. Similarly, the receiver needs to lower the ready signal unless it can immediately accept more data on the next cycle.

This is not the case when one side is in software. If the valid port uses a simple register, the software side might not be able to reset it in time, resulting in the same data being transmitted multiple times. Similarly, if the software does not reset its ready port in time, the hardware may send multiple packets that overwrite one another, causing data loss.

We solve this by performing automatic resets on the hardware side in the AXI Lite driver. Software writing a non-zero value to its output valid port means the data in the data registers is valid *once*. If the data is consumed, the valid signal is lowered in the hardware on the next cycle. Similarly, writing non-zero to the output ready port means the software side is ready to accept *one* packet. Once a packet is in place, the ready signal is lowered by the hardware on the next cycle.

On the software side, as with any device, waiting for the valid signal can be done by either polling or using interrupts. We implement both: as we show later in section 5, they have different impacts on performance and CPU usage.

The software and hardware stub code are generated based on a minimal OS-specific library, currently implemented for Linux and seL4 [37] in less than 100 lines of code each. For the Linux implementation, a small kernel module (less than

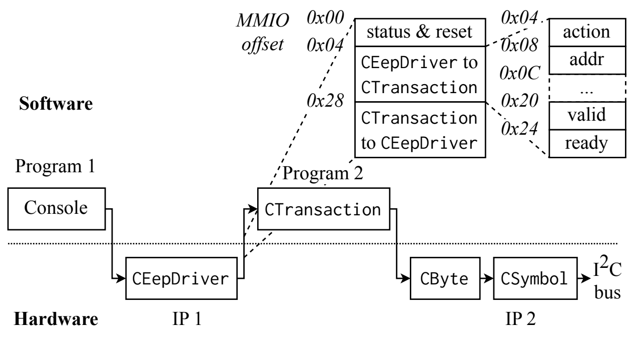

Figure 7. Multiple hardware/software boundaries. MMIO-AXI Lite interface between `CEepDriver` and `CTransaction` (corresponding to the ESI definition in Figure 4).

150 lines of code including blank lines and comments) creates userspace I/O (UIO) [40] device files for Efeu hardware based on device tree entries. The generated drivers then run in userspace. The library only `mmap`s the device file and provides functions for directing reads and writes to the virtual base address obtained from `mmap`, plus a function to wait for an interrupt which uses a blocking read to the UIO device file. On seL4 we rely on the Microkit SDK [27] to map the device registers into the virtual address space of the driver's protection domain. The read and write functions are then implemented the same as in Linux. Waiting for an interrupt is implemented using the blocking `seL4_Recv` syscall.

Efeu can generate drivers with more than one hardware/-software boundary, as in Figure 7. These configurations are not optimal for performance, but can be useful for debugging (e.g. temporarily replacing a hardware layer with software).

3.6 Promela backend

The Promela backend transforms the AST into input for the SPIN model checker, preserving syntactic information like variable names and control flow from the AST and allowing the developer to make easy correspondence between the ESM code and the generated Promela. Most ESM constructs have straightforward analogs in Promela, including variable declarations, operators, and control flows. The notable translation rules are listed as follows.

- `bit` and `byte` are not built-in types in C. Stub code generated by ESMC `typedef`s them as `unsigned char` to make the ESM code syntactically correct but they are translated to exactly `bit` and `byte` in Promela.
- Enumerations translate to `mtype` [23].
- ESM channels translate to rendezvous (synchronous) channels in Promela [23].
- Layer functions translate to Promela processes [23] with channels passed as parameters, allowing users to write parameterized verifiers (section 4.4).
- In Promela, `if` statement encodes non-deterministic choices [23]. If no option is executable, the expression is blocking. However, in ESM when the condition does not hold, the `if` block is skipped. We encode such behavior by generating an `else -> skip` block if there is no `else` branch in ESM.

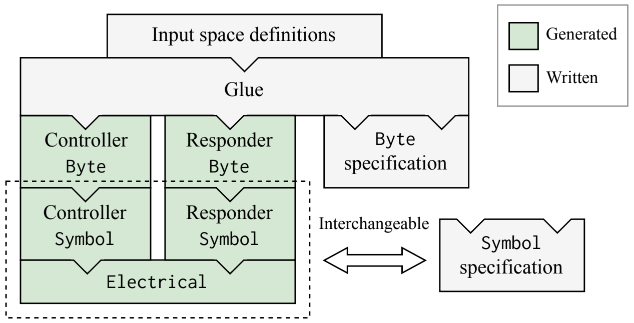

Figure 8. Architecture of the `Byte` verifier.

The generated Promela models the system, but must be combined with verifier code for input to the model checker. Note that Efeu does not provide full formally-verified end-to-end properties: we trust ESMC to generate correct code (with extensive tests) and the downstream toolchains to correctly compile them. Removing this gap in the proof could be attempted using a range of techniques for compilers [43, 64] and EDA tools [41, 44, 47] but is beyond the scope of this paper.

4 Verification

In the following, we describe how Efeu is used to verify a generated I^{2}C stack following the approach from our previous work [30], starting with the simplest case of one controller and one responder. We then discuss how we extend the technique to support multiple devices using parameterized verifiers in section 4.4. We also explain how to model non-standard devices and quirks in section 4.5.

4.1 Approach

For each layer except `Electrical`, we verify that the stack conforms to the behavior specification and there is no live- or deadlock in the system. Functional correctness is checked by assertions and the absence of live- and deadlocks is verified automatically by the model checker, both regarding the specified behavior. Figure 8 shows the architecture of the `Byte` verifier as an example. The unit-under-test is the `Byte` layer, which is connected to the layer below. An input space specification defines the valid input to the system. Inputs are fed to both the stack and the behavior specification and the outputs are compared.

The state search space of the whole stack is non-trivial. To mitigate the state explosion issue, we apply the technique we proposed previously [30], which substitutes lower layers with the corresponding behavior specification. This significantly reduces the model checking runtime (see section 4.3) and consequently allows larger input spaces and/or larger systems (section 4.4). Each different class of layer has a different type of behavior specification, as follows:

The **`Symbol`** behavior specification specifies how symbols are combined on the bus. For example, a START symbol and an IDLE symbol (passively listening) are combined into a START operation received by both devices. BIT0 plus BIT1 results in BIT0 due to the pull-down characteristic of the I^2C bus. The corresponding input space specification specifies valid control sequences from the `Byte` layers above.

The **`Byte`** behavior specification specifies how the controller and the responder should interact at the byte level. For example, when the controller writes a byte, the responder should be listening and the byte should be seen by both devices ultimately.

The **`Transaction`** behavior specification raises the abstraction level further up. The controller issues read and/or write transactions and the responder observes them. The input space specifies valid control sequences from above as a mixture of read transactions, write transactions, and/or stop operations. It is at this level that model checking scalability becomes an issue. Exploring the whole search space, including the variable payload length and content is infeasible. We therefore currently limit the input space specification for the `Transaction` verifier to a variable payload length of up to 4 bytes and a fixed payload content.

The example **`EepDriver`** behavior specification raises the abstraction level to EEPROM read and write operations. As with the `Transaction` verifier we limit the input space specification to a fixed EEPROM offset for reads and writes and a payload of between 1 to 4 bytes of fixed content.

In our previous work, we chose SPIN as the model checker for its ease of use and maturity. We also used it for this work to reuse the behavior specifications. The verification scalability experiment in section 4.4 shows some limitations of the tool: increasing the number of devices and/or the payload length leads to state space explosion. We expect that more recent model checkers, especially symbolic SAT-based ones [76] would be able to explore the search space more efficiently. Exploring this and alternative verification strategies, like pairwise verification of devices, are however beyond the scope of this paper.

4.2 Verification code size

While Efeu generates the complete stack in Promela from ESI/ESM, the other components of the verifiers (input space definitions, behavior specifications, and glue) require manual effort. The cost of verification will thus vary from platform to platform. As an indication, we report here the code size of both the generated and the handwritten files.

The ESM and C code is formatted with ClangFormat [72] and counted with `cloc` [17]. For Promela code, to our best knowledge, there is no canonical Promela formatter. Therefore, we build an in-house formatter and consistently apply it on both generated and handwritten Promela code. Comments and empty lines are excluded in all code.

Table 1 shows the results; those for generated code are underlined. At the `Symbol` and `Byte` levels, the controller and responder share most of the ESM code using preprocessor macros to include the same files, so we report combined lines. Shared generated Promela code defines common data structures and channels, which cannot be attributed to specific layers. Shared glue code is written once and included in all verifiers.

We see that the generated Promela has roughly the same size as the ESM specification, which is expected due to the close semantics of the languages. The ratio of additional handwritten Promela code to generated code is between 0.96 and 1.49 (excluding shared code). These numbers are only a rough approximation of the cost of verification, but do show how Efeu reduces the verification cost and avoids human errors by automating the translation from the specification to Promela.

4.3 Verification runtime

To demonstrate the practicality of verification and effectiveness of abstraction levels (section 4.1), we measure the runtime of SPIN executing these verifiers on an AMD Ryzen 9 7950X 16-Core machine with 64GB RAM and 128GB swap memory. The SPIN version is 6.5.2. Each verifier is executed 5 times. SPIN can check for either deadlocks or assertion failures (invalid end states) or livelocks (non-progress cycles) in one execution, but not both. Therefore, each verifier is compiled and executed in each configuration, and the runtime is summed up.

Table 2 show average runtime, and also the effect of abstracting lower layers with the corresponding behavior specifications. All verifiers pass. We observe no significant deviation across runs. The maximum coefficient of variation of all measurements is under 2.7%, and so we omit it for brevity.

Verifying `Symbol` is fast. Moving up the stack, the runtime increases dramatically, but limiting the input space allows the I^2C stacks to be verified in reasonable time.

4.4 Scalability

So far we have shown verification for a *single* controller and responder. We now show the scalability of our approach modeling and verifying parameterized systems with multiple EEPROMs. The verifier uses channel arrays in Promela [23]. Multiple responders are instantiated and connected to an `Electrical` layer.

We vary the number of EEPROMs and the maximum length of the reads and writes. The EEPROM offset and payload content are fixed. We also show a "variable payload" with one EEPROM where the first payload byte is chosen from two options non-deterministically. As in section 4.3, we

Table 1. Source code lines of layers

Layer	ESM		Promela				C	Verilog
	Controller	Responder	Generated controller	Generated responder	Behavior specification	Input space and glue	Generated controller	Generated controller
`Symbol`	139		96	95	65	119	159	613
`Byte`	114		143	143	85	203	174	465
`Transaction`	106	177	126	116	114	184	125	571
`EepDriver`	62	85	81	119	55	243	62	374
Shared			111			121		

Table 2. Average verification runtime (s)

Layer	Abstraction level			
	None	`Symbol`	`Byte`	`Transaction`
`Symbol`	0.24			
`Byte`	11.33	4.01		
`Transaction`	104.53	34.79	6.11	
`EepDriver`	584.78	196.31	38.92	9.15

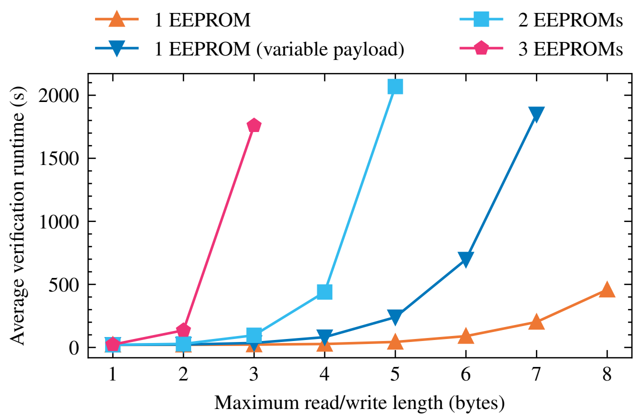

Figure 9. Verification runtime of multiple EEPROMs with different maximum payload lengths.

replace lower levels with behavior specifications to reduce the verification runtime, which is shown in Figure 9. Again, we omit the insignificant standard deviations.

This shows that systems with multiple responders can be verified in reasonable time. However, the state space still explodes as the input space is enlarged or the number of responders increases. We discussed some strategies to further scale the verification at the end of section 4.1.

4.5 Non-standard devices

We now show how to model devices that violate the I^{2}C standard, taking as examples the KS0127 video decoder and the I^{2}C controller on the Raspberry Pi.

The KS0127 video decoder (unlike the successor KS0127B [61]) has a quirk [20, 60]: in an I^{2}C read transaction, it attempts to sample a stop condition at the place where the acknowledgment bit should be. Otherwise, the stop condition is not recognized. Linux introduced a flag (`I2C_M_NO_RD_ACK`) to handle this behavior *solely* for this device [20].

In Efeu, we model this quirk by changing only the `Byte` layer for the KS0127 responder to skip reading the acknowledgment bit in a read transaction. This involves 13 lines of additional ESM code. We also modify the maximum read length in the input space specification to 1 as both the KS0127 datasheet [60] and the Linux driver code [20] only specify reading one byte.

When we combine the modified KS0127 `Byte` and the standard controller `Byte`, SPIN reports the system can enter an invalid end state, showing that the standard controller is *not* interoperable with the KS0127 responder.

Next, we modify the controller `Byte` layer to make it compatible with KS0127, involving 10 lines of additional code. With this modified controller, the verifier passes. Note that above these modified `Byte` layers, the `Transaction` layer can be used unmodified, and the stack fully verified, showing that quirks can be handled within in a single layer.

Similarly, using Efeu, we can efficiently model the hardware I^{2}C controller on the Raspberry Pi that does not handle clock stretching [49], by removing the clock stretching handling code from the controller `Symbol` layer in our specification, requiring 3 additional lines of code (essentially a preprocessor macro). The standard `Symbol` verifier detects problems in the modified stack. If we also remove clock stretching from the input space, essentially modeling a responder that never stretches the clock, the verifier passes.

5 Evaluation on real hardware

We evaluate the hybrid hardware/software I^{2}C controllers generated by Efeu on real hardware, varying the split between hardware and software. In this section, we report the results and show that it is feasible to efficiently explore the trade-offs between achievable bus speed, CPU usage, and FPGA resource utilization. We compare the generated drivers with two baselines: the Linux "bit-banging" kernel driver and the Xilinx I^{2}C IP [3].

The generated drivers run on a Zynq UltraScale+ MPSoC [79]. The MPSoC features a quad-core ARM Cortex-A53 and a Xilinx 16nm FPGA. A modified OpenBMC [22] distribution (kernel version 5.15) runs on the ARM cores. Efeu-generated C code is cross-compiled using GCC 13.2.0 with

`-O3` optimization. Generated hardware parts are placed in an FPGA design, implemented using Xilinx Vivado 2022.1 with the default settings, and loaded on the FPGA. All components in the FPGA design are driven by the 100 MHz clock. Two GPIO pins serve as SCL and SDA, routed to an IO connector and connected to an I^2C bus.

For I^2C responders, we model Microchip 24AA512 EEPROMs [51] which support I^2C Fast Mode (400 kbit/s). However, to demonstrate that the generated drivers work in practice, we use a *real* 24AA512 EEPROM as the responder, connected to the I^2C bus. EEPROMs are slow for write operations: on the 24AA512, page writes can take up to 5ms [51]. During busy periods, the device stops responding to subsequent I^2C requests. We focus on the I^2C performance of the generated drivers rather than that of the EEPROM and so only report read operation performance.

A Keysight InfiniiVision MSO-X 3024T oscilloscope [34] is used to inspect signals on the I^2C bus. The SCL and SDA lines are captured as analog signals. The oscilloscope is capable of finding rising/falling edges and decoding I^2C packets.

To get the best performance from the baseline Linux I^2C "bit-banging" driver, the GPIO delay [18] is set to 1 (0 results in a longer default delay). Internally, the driver polls the GPIO pins using a spinlock to wait between operations [19].

The Xilinx I^2C baseline consists of two parts: hardware IP in the FPGA design with the target frequency set to 400 kHz, and the Linux driver from Xilinx supporting interrupts. The IP offers a similar abstraction level as our `Transaction`, while offering additional functionality like FIFO queues.

We denote software/hardware splits by the *topmost* hardware layer. For example, configuration `Byte` has `Byte` and below in hardware and the rest in software.

5.1 Source code size

To show how effective Efeu is in reducing the effort of writing I^2C stack implementations, we measure the generated source code size. C code is formatted with ClangFormat [72] while Verilog code generated by Efeu is already formatted. Code size is measured using cloc [17].

For layer implementations, sizes of the generated C and Verilog code are in Table 1. The generated C code has roughly the same size as the corresponding ESM specification. The generated Verilog files are a few times larger than the ESM files. Table 3 shows the result for the MMIO-AXI Lite interface across the software/hardware boundary. Vivado uses VHDL for AXI Lite interfaces, so does Efeu. Mixing VHDL and Verilog is not a problem as Vivado (and many other EDA tools) trivially supports it. The interface specification (ESI) is highly compact, while the generated code contains significantly more lines of code.

By specifying the stack in ESI/ESM once, developers save the effort of implementing the same thing in C and Verilog. While the generated code is expected to be less compact than code written by human experts, we conclude that Efeu helps reduce the effort required to materialize the verified stack.

Table 3. Source code lines for MMIO-AXI Lite interfaces

Interface	ESI	Generated	
		C	VHDL
`Electrical-Symbol`	10	71	308
`Symbol-Byte`	16	67	295
`Byte-Transaction`	28	71	308
`Transaction-EepDriver`	24	82	391
`EepDriver-World`	23	80	401

5.2 Achievable bus speeds

The EEPROM supports the I^2C Fast Mode (400 kHz SCL) [53]. However, not all controllers can drive the bus at this speed. We measure the achievable bus speeds of both the baselines and the generated drivers to show how the software/hardware split point and the type of interface (polling versus interrupt-driven) affect the I^2C driver performance.

We measure the bus speed achievable by each controller by issuing 3 EEPROM reads of 14 bytes and inspecting the waveforms captured by the oscilloscope. The oscilloscope is triggered by the first falling edge of SDA, signaling the start condition of an I^2C transaction. It records long enough to cover the whole operation. We use its built-in search function to find all rising edges of SCL. Experiments consistently show 164 rising edges for each EEPROM read operation. We calculate the effective SCL frequency as the inverse of the time between two consecutive rising edges. We show the average frequency and the standard deviation.

In the top half of Figure 10, the average frequencies across the whole operation and the 3 repetitions are shown. The error bars show the standard deviations. The Xilinx I^2C IP reaches bus speeds close to the target frequency with little variation. The achievable bus speed is 386.57 kHz and the standard deviation is 23.75 kHz. The Linux bit-banging driver achieves an average frequency of 162.81 kHz, less than half of the target frequency. The standard deviation is 12.85 kHz.

The polling-based `Electrical` driver achieves a slightly lower frequency of 154.44 kHz with a standard deviation of 12.97 kHz. The interrupt-driven `Electrical` driver does not function correctly due to excessive interrupts being issued.

By moving the `Symbol` layer into hardware, the polling-based `Symbol` driver achieves a higher bus frequency of 263.32 kHz with a standard deviation of 12.77 kHz. Due to the reduced traffic across the software/hardware boundary, interrupts work for the `Symbol` driver. However, the interrupts yet introduce non-negligible overhead—the interrupt-driven `Symbol` driver only achieves 108.76 kHz.

By moving the next layer `Byte` into the hardware, the polling-based `Byte` driver achieves an average frequency closer to the target of 359.98 kHz. However, the standard deviation becomes more significant, reaching 89.82 kHz. The

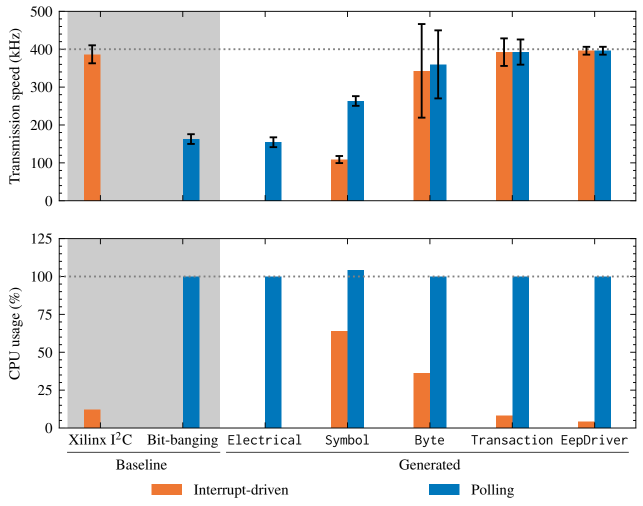

Figure 10. Achievable bandwidth (top) and CPU usage (below). 100% CPU means one core (out of four) is fully utilized. The shaded area are the baselines and the Efeu configurations are labeled with the highest layer implemented in hardware.

interrupt-driven `Byte` driver has less overhead. It shows an average frequency of 342.9 kHz, close to the polling-based driver, and similarly a higher standard deviation of 123.58 kHz.

The `Transaction` drivers move one more layer into the hardware. At this point, the generated drivers have a similar abstraction level as the Xilinx I^{2}C IP. The achievable frequencies of both the polling-based and the interrupt-driven drivers are close to the target frequency, averaging 392.48 and 392.24 kHz respectively. The standard deviations also drop to 33.25 and 36.36 respectively. Compared with the Xilinx IP, `Transaction` drivers have slightly higher bus speeds and standard deviations.

When all layers are in hardware, the `EepDriver` drivers achieve 396.02 kHz (polling) and 396.01 kHz (interrupt-driven). The standard deviations further drop to 10.37 kHz (polling) and 10.34 kHz (interrupt-driven).

To assist interpretation of the differences in achievable bus speeds, Figure 11 shows several waveforms of SCL. When the driver has a large portion in software, such as the Linux bit-banging driver or the `Electrical` driver, SCL is driven slowly and with unstable frequency. The software part takes longer time to process and to communicate data across the software-hardware boundary. In contrast, when most of the stack is in hardware, like the Xilinx I^{2}C IP and the `EepDriver` driver, SCL is driven towards the target frequency stably.

The experiment demonstrates how the splits between software and hardware affect the driver performance. Higher and more stable performance can be achieved by moving layers

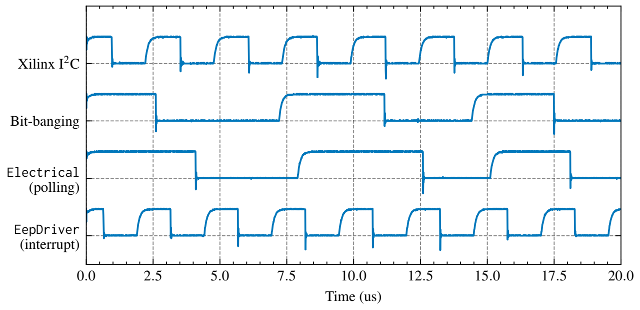

Figure 11. Waveforms of the first few SCL cycles, captured by the oscilloscope.

into hardware. This reduces traffic across the software/hardware boundary, where MMIO operations take time. In addition, when implemented in hardware, layer FSMs transit their states deterministically adhering to the hardware clock.

When the whole stack is implemented in software, the Efeu-generated `Electrical` driver achieves a performance close to the Linux bit-banging driver. However, neither of them reaches the target frequency of 400 kHz. If the platform allows, by implementing parts of the stack in hardware, Efeu-generated drivers can achieve the target frequency, comparable with the Xilinx I^{2}C IP, an optimized hardware implementation.

5.3 CPU usage

The splits of software and hardware affect not only the achievable bus speed but also the CPU usage. If the platform on which the drivers run has limited computing power, understanding the latter correlation helps decide the optimal implementation. In this experiment, we measure the CPU usage of the Efeu-generated drivers and the baselines.

Similar to the experiment discussed in section 5.2, drivers issue EEPROM operations of reading 14 bytes. However, in this experiment, those operations are issued consecutively and indefinitely until manual termination. In this way, we can read out the CPU usage in a stable running state. No other process consumes significant computing power. Behavioral correctness is asserted by placing software assertions and inspecting I^{2}C transactions decoded by the oscilloscope.

The result is shown in the lower half of Figure 10. As expected, all polling-based drivers fully utilize one core. In contrast, using interrupts at the software/hardware boundary reduces the CPU usage. The polling-based `Electrical` driver does not work as explained in section 5.2. The Xilinx I^{2}C IP consumes 12%. The `Symbol` driver consumes 64%. The `Byte` driver consumes 36%. Moving the split point further upwards, the CPU usage drops significantly. The `Transaction` and `EepDriver` drivers takes 8% and 4% respectively.

The experiment demonstrates how the software-hardware split point affects CPU usage of the driver. Naturally, by moving parts of the stack into hardware, the CPU usage decreases

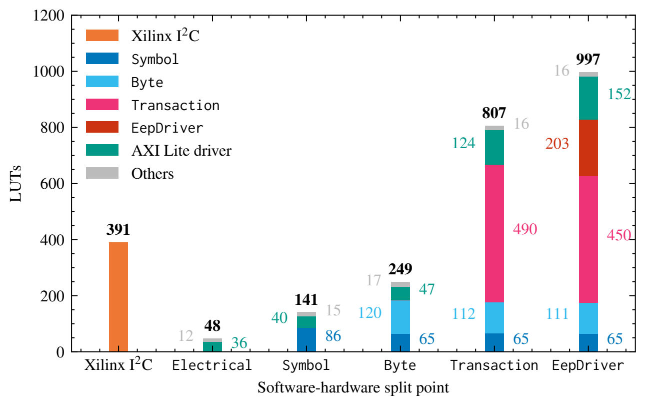

Figure 12. LUT utilization.

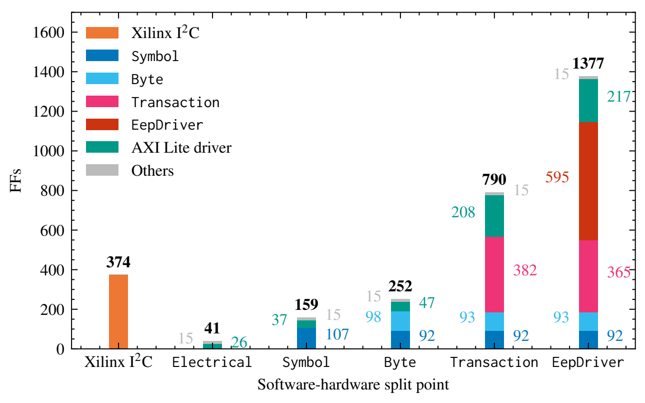

Figure 13. FF utilization.

for the interrupt-driven drivers but not the polling-based drivers. The interrupt-driven `Transaction` and `EepDriver` drivers generated by Efeu have the lowest CPU usage, lower than the Xilinx IP.

5.4 FPGA resource utilizations

In this subsection, we report the FPGA resource utilizations of the hybrid hardware/software drivers. We focus on two main resources on FPGA, Look-Up Tables (LUTs) and Flip-Flops (FFs). The usage data is extracted from the resource utilization report generated by Xilinx Vivado. Since layers are implemented as Verilog modules, Vivado also reports the resource usage of each layer.

LUT and FF utilizations are shown in Figure 12 and Figure 13 respectively. "Others" is calculated as the total LUTs or FFs minus the sum of all other parts, attributed to the bus adaptor and possibly the glue. As the split point moves up along the stack, we see the resource usage of low layers decreases. We believe it is due to Vivado performing cross-boundary optimization among modules.

`Electrical`, `Symbol` and `Byte` use fewer LUTs and FFs than the Xilinx I^{2}C IP. `Transaction`, which has a similar abstraction level as the Xilinx IP, consumes 2.08× of LUTs and 2.11× of FFs. We believe this is a reasonable overhead when comparing a generated driver with an IP crafted by human experts, especially when considering that the Efeu-generated drivers possess the assurance gained from the model checking. Furthermore, compared to the available resources on commercial FPGAs, these utilizations are very small. The FPGA on the MPSoC has 117120 LUTs and 234240 FFs [4]. In terms of the total available resources, the Xilinx IP consumes 0.33% of LUTs and 0.16% of FFs. The `Transaction` driver consumes 0.70% of LUTs and 0.34% of FFs. Even the entire stack on the FPGA (`EepDriver`) requires only 0.85% of LUTs and 0.59% of FFs.

This experiment demonstrates the Efeu-generated drivers consume minimal FPGA resource in terms of the total available resources comparable to IP crafted by human experts.

5.5 Discussion

Combining these evaluations, we demonstrate that Efeu helps find the optimal implementations depending on different criteria: performance, CPU usage, and FPGA utilization.

On our evaluation platform, by implementing `Byte` and lower layers in hardware and using the interrupt-driven interface, the driver can achieve about 350 kHz bus speed, consume less than 40% CPU, and use less FPGA resources in both LUTs and FFs compared with the Xilinx IP.

If a bus speed higher than 390 kHz is required, at least `Transaction` and all layers below need to be implemented in hardware. Based on that, implementing the `EepDriver` layer in hardware only provides marginal benefits in the bus speed and the CPU usage. However, if extremely stable bus speed is desirable, `EepDriver` is a good option.

If there is no programmable logic available on the platform, `Electrical` is the only option. In this case, the driver cannot achieve the full bus speed (so does the Linux bit-banging driver). However, the Efeu-generated ones still possess the assurance provided by the model checking.

6 Related work

Driver reliability has been a long-standing issue. There is therefore a large body of work to improve driver quality. There are three main categories of approaches to produce better drivers, and we survey them in turn: hardware/software co-design, synthesis of drivers from specifications, and post-hoc verification of manually written drivers.

6.1 Hardware/software co-design

Early approaches like Chinook [12–14] focused on I/O port allocation, synthesis of multiplexers, arbiters and driver code to share processor interfaces among devices. Chinook also synthesizes low-level hardware interfaces from timing diagrams. The adapter that Efeu uses to ensure the timing on the bus is currently hand-coded in 106 lines of VHDL but could be synthesized with such an approach. CoWare [75] can synthesize more complex hardware to adapt device interfaces to I/O interfaces available on a processor. It also generates code to make these adaptions transparent and maintain the device interface to software. These early systems usually make the assumption that the entire system is synthesized from the specification, and they use ad-hoc protocols for device interaction. In contrast, we demonstrate that we can verify a system built from off-the-shelf components connected with a standard protocol. Ortega *et al.* [55] extend Chinook to be able to instantiate standard bus protocols like I^{2}C or CAN and adapt device drivers to be able to use built-in bus controllers. They consider global system analysis for fulfilling real-time constraints but not for functional interference between devices. They also assume a standard I^{2}C implementation, which leaves the many quirks in I^{2}C hardware out of the picture. Correctness of the co-designed systems is usually tested by full-system simulators that can also be synthesized from the specifications. Efeu gives formal guarantees on the properties it verifies. Later approaches then extend the power of the synthesis: O'Nils and Jantsch [54] present a method to synthesize DMA controllers to proxy off-the shelf devices to offload memory accesses from the CPU. In HINGE [80] a higher level device API can be synthesized and the correct usage of the API is checked at runtime and with BCL [36] the partition between software and hardware can be chosen independent of the specification and partial implementations (only software, hardware or interface) are possible. None of these approaches focus on interoperability of off-the-shelf components like Efeu does.

6.2 Driver synthesis

Earlier work in driver synthesis was mostly concerned with device register specification. Devil [50] and HAIL [66] synthesize low-level register access code from specifications describing the interface to hardware. In NDL [16], one can additionally specify device state transitions and how they relate to the register accesses. The NDL compiler then synthesizes functions for querying and modifying the device state. In Efeu the interfaces are specified with ESI, and we can then not only generate the software to access the registers but also the register interface itself.

While these systems free developers from writing tedious bit manipulation code, this is only a part of modern drivers. Later approaches aim to synthesize full drivers with interface specifications to devices, operating systems and other software. Notably, there is Termite [58, 59] and more recently Ghost Writer [77]. Both systems are limited in the complexity of the device that they can synthesize drivers for. This complexity only gets worse when considering the interactions between devices. With Efeu we therefore chose to go with a verification approach instead. And we will review some existing approaches next.

6.3 Driver verification

Verifying manually written drivers is pragmatic: the quality of existing drivers can be improved without having to reimplement what is a significant part of the OS [11]. SLAM [6] verifies the correct usage of OS APIs by drivers using model-checking. SafeDrive [81] achieves a similar goal, but they synthesize run time checks to catch driver errors and recover from them. Bošnački *et al.* [7] use model-checking and static analysis to verify the correct use of Linux APIs by an I^{2}C driver. The biggest difference to Efeu is that we focus on the interaction between the driver and the devices and between devices and not between the driver and the OS. In that sense these approaches are complementary to ours.

An example of post-hoc verification that focuses on the interaction with the device is Kim *et al.* [35] who verify a flash driver using model checking. There are however limitations to how thoroughly software can be verified if it was not implemented with verification in mind [37]. Many approaches therefore implement the drivers in a highly stylized way. Chen *et al.* [9] target drivers for interruptible kernels that they verify in Coq. Klomp *et al.* [38] target an I^{2}C driver. Pohjola *et al.* propose Pancake [56], a DSL targeted at easily writing verifiable device drivers and keeping the verification cost low. All the above focus on verifying individual drivers that handle a single device and therefore cannot address interoperability issues that arise in shared bus settings such as I^{2}C.

We follow a similar approach with Efeu and specify our drivers in a DSL designed for verifying the properties we are interested in.

7 Conclusion

Amid all challenges of producing correct drivers, bus-based protocols like I^{2}C pose another: interoperability. We have presented Efeu which allows specifying full I^{2}C subsystems in DSLs, model checking it, and generating hybrid hardware/-software drivers. Efeu helps developers to explore trade-off between throughput, CPU usage and FPGA utilization.

While so far we have only applied the Efeu methodology to I^{2}C, we believe that it could be extended to other bus-based protocols like SPI or CAN. These protocols share key features with I^{2}C: multiple agents transmit data by modulating shared wires, and the protocols consist of multiple abstraction levels with potential quirks. The electrical characteristics, like the number of wires that are used, only appear on the lowest layer of an Efeu specification. Furthermore, bus timing is handled by the lowest-level hardware adapter, allowing Efeu to deal only with discrete time steps. This adapter is currently handwritten but could be synthesized using HW/SW co-design techniques for our approach to scale better to other protocols (see section 6.1).

Another item for future work is reducing the trusted compute base (TCB). A large part of it is the Efeu compiler. We outlined some approaches for removing it from the TCB at the end of section 3.6.

I^2C is used in critical systems like BMCs where interoperability issues can lead to catastrophic bugs and security vulnerabilities. Indeed, we intend to use Efeu to generate a trustworthy I^2C driver stack for the BMC on the Enzian research computer [15]. This will require scaling the verification to 10–20 devices on a bus. We outlined some potential strategies to achieve this at the end of section 4.1.

Despite the remaining challenges, we believe that Efeu is an important step towards verifying fundamental components of critical infrastructure: By ruling out interoperability issues, Efeu moves these systems closer to the trustworthiness that they desperately need.

The Efeu compiler and all our I^2C specifications are available as open source[2].

Acknowledgments

We would like to thank our shepherd Yanyan Jiang and the anonymous reviewers for their constructive feedback and helpful comments. Special thanks go to Roman Meier for testing the usage instructions for the artifact. We also thank Google for their generous support of this research.

References

[1] Aldo Aguilar-Nadalini, Kuk H Chung, Cecilia Marsicovetere, Juan F Medrano, Emilio Miranda, Víctor Ayerdi, and Luis Zea. 2023. Design and On-Orbit Performance of the Electrical Power System for the Quetzal-1 CubeSat. *Journal of Small Satelites* 12, 2 (May 2023), 1201–1229.

[2] Amina Albalooshi, Abdul-Halim M. Jallad, and Prashanth R. Marpu. 2023. Fault Analysis and Mitigation Techniques of the I2C Bus for Nanosatellite Missions. *IEEE Access* 11 (2023), 34709–34717. https://doi.org/10.1109/ACCESS.2023.3262410

[3] AMD 2012. *LogiCORE IP AXI IIC bus interface data sheet.* AMD.

[4] AMD 2022. *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891).* AMD. https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf

[5] Arm 2022. *Learn the architecture - An introduction to AMBA AXI.* Arm.

[6] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. 2011. A Decade of Software Model Checking with SLAM. *Commun. ACM* 54, 7 (July 2011), 68–76. https://doi.org/10.1145/1965724.1965743

[7] Dragan Bošnački, Aad Mathijssen, and Yaroslav S. Usenko. 2009. Behavioural Analysis of an I2C Linux Driver. In *Formal Methods for Industrial Critical Systems*, María Alpuente, Byron Cook, and Christophe Joubert (Eds.). Springer, Berlin, Heidelberg, 205–206. https://doi.org/10.1007/978-3-642-04570-7_18

[8] Jasper Bouwmeester, Martin Langer, and Eberhard Gill. 2017. Survey on the Implementation and Reliability of CubeSat Electrical Bus Interfaces. *CEAS Space Journal* 9, 2 (June 2017), 163–173. https://doi.org/10.1007/s12567-016-0138-0

[9] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* Association for Computing Machinery, New York, NY, USA, 431–447. https://doi.org/10.1145/2908080.2908101

[2] https://gitlab.inf.ethz.ch/project-opensockeye/efeu

[10] Zitai Chen and David Oswald. 2023. PMFault: Faulting and Bricking Server CPUs through Management Interfaces: Or: A Modern Example of Halt and Catch Fire. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023 (March 2023), 1–23. Issue 2. https://doi.org/10.46586/tches.v2023.i2.1-23

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. *ACM SIGOPS Operating Systems Review* 35, 5 (Oct. 2001), 73–88. https://doi.org/10.1145/502059.502042

[12] Pai Chou, Ross Ortega, and Gaetano Borriello. 1992. Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems. In *1992 IEEE/ACM International Conference on Computer-Aided Design.* IEEE Computer Society, USA, 488–495. https://doi.org/10.1109/ICCAD.1992.279322

[13] Pai Chou, Ross B. Ortega, and Gaetano Borriello. 1995. Interface Co-Synthesis Techniques for Embedded Systems. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design (ICCAD '95).* IEEE Computer Society, USA, 280–287.

[14] Pai H. Chou, Ross B. Ortega, and Gaetano Borriello. 1995. The Chinook Hardware/Software Co-Synthesis System. In *Proceedings of the 8th International Symposium on System Synthesis (ISSS '95).* Association for Computing Machinery, New York, NY, USA, 22–27. https://doi.org/10.1145/224486.224491

[15] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022).* Association for Computing Machinery, New York, NY, USA, 434–451. https://doi.org/10.1145/3503222.3507742

[16] Christopher L. Conway and Stephen A. Edwards. 2004. NDL: A Domain-Specific Language for Device Drivers. *ACM SIGPLAN Notices* 39, 7 (June 2004), 30–36. https://doi.org/10.1145/998300.997169

[17] Albert Danial. 2024. *cloc: v2.00.* GitHub. https://github.com/AlDanial/cloc

[18] The Linux development community. 2024. Device-Tree bindings for I2C GPIO driver. https://www.kernel.org/doc/Documentation/devicetree/bindings/i2c/i2c-gpio.txt Accessed on 2024-05-12.

[19] The Linux development community. 2024. linux/drivers/i2c/busses/i2c-gpio.c at Linux v5.15. https://github.com/torvalds/linux/blob/v5.15/drivers/i2c/busses/i2c-gpio.c Accessed on 2024-05-12.

[20] The Linux development community. 2024. linux/drivers/media/i2c/ks0127.c at Linux v5.15. https://github.com/torvalds/linux/blob/v5.15/drivers/media/i2c/ks0127.c Accessed on 2024-05-08.

[21] The Linux development community. 2024. linux/drivers/pci/quirks.c at Linux v6.9. https://github.com/torvalds/linux/blob/v6.9/drivers/pci/quirks.c Accessed on 2024-05-17.

[22] The OpenBMC development community. 2024. OpenBMC. https://github.com/openbmc/openbmc Accessed: 2022-09-08.

[23] The SPIN development community. 2012. Promela Manual page. http://spinroot.com/spin/Man/promela.html Accessed: 2023-08-02.

[24] Shuying Fan and Supriya Velagapudi. 2023. Implementing Next-Generation Data Center Platform Management Using Agilex 3 and Agilex 5 Devices. https://www.intel.com/content/www/us/en/content-details/787067/implementing-next-generation-data-center-platform-management-using-agilex-5-devices.html Accessed on 2024-05-21.

[25] Ben Fiedler, Daniel Schwyn, Constantin Gierczak-Galle, David Cock, and Timothy Roscoe. 2023. Putting out the Hardware Dumpster Fire. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23).* Association for Computing Machinery, New York, NY, USA, 46–52. https://doi.org/10.1145/3593856.3595903

[26] A. Ganapathi, Viji Ganapathi, and D. Patterson. 2006. Windows XP Kernel Crash Analysis. In *LiSA*. USENIX Association, Berkeley, CA, USA, 149–159. https://www.usenix.org/legacy/events/lisa06/tech/ganapathi.html

[27] Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. 2022. Can We Put the "S" Into IoT?. In *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*. IEEE, New York, NY, USA, 1–6. https://doi.org/10.1109/WF-IoT54382.2022.10152198

[28] G.J. Holzmann. 1997. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (May 1997), 279–295. https://doi.org/10.1109/32.588521

[29] Gerard J. Holzmann. 2000. Logic Verification of ANSI-C Code with SPIN. In *SPIN Model Checking and Software Verification*, Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer, Berlin, Heidelberg, 131–147. https://doi.org/10.1007/10722468_8

[30] Lukas Humbel, Daniel Schwyn, Nora Hossle, Roni Haecki, Melissa Licciardello, Jan Schaer, David Cock, Michael Giardino, and Timothy Roscoe. 2021. A Model-Checked I2C Specification. In *Model Checking Software (Lecture Notes in Computer Science)*, Alfons Laarman and Ana Sokolova (Eds.). Springer International Publishing, Cham, 177–193. https://doi.org/10.1007/978-3-030-84629-9_10

[31] Giang Nguyen Thi Huong. 2011. GCC2Verilog Compiler Toolset for Complete Translation of C Programming Language into Verilog HDL. *ETRI Journal* 33, 5 (Oct. 2011), 731–740. https://doi.org/10.4218/etrij.11.0110.0654

[32] Ke Jiang. 2009. *Model Checking C Programs by Translating C to Promela*. Master's thesis. Uppsala Universitet, Uppsala, Sweden. http://www.diva-portal.org/smash/get/diva2:235718/FULLTEXT01.pdf

[33] kaedros. 2022. Raspberry Pi I2C clock-stretching bug GitHub Issue. https://github.com/raspberrypi/linux/issues/4884 Accessed on 2024-09-05.

[34] Keysight Technologies, Inc. 2020. Keysight InfiniiVision 3000T X-Series Oscilloscopes User's Guide. Accessed on 2023-08-08.

[35] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. 2008. Formal Verification of a Flash Memory Device Driver – An Experience Report. In *Model Checking Software*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer, Berlin, Heidelberg, 144–159. https://doi.org/10.1007/978-3-540-85114-1_12

[36] Myron King, Nirav Dave, and Arvind. 2012. Automatic Generation of Hardware/Software Interfaces. *ACM SIGARCH Computer Architecture News* 40, 1 (March 2012), 325–336. https://doi.org/10.1145/2189750.2151011

[37] 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 (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[38] Arjen Klomp, Herman Roebbers, Ruud Derwig, and Leon Bouwmeester. 2009. Designing a Mathematically Verified I2C Device Driver Using ASD. In *Communicating Process Architectures 2009*. Concurrent Systems Engineering Series, Vol. 67. IOS Press, Amsterdam, Netherlands, 105–116. https://doi.org/10.3233/978-1-60750-065-0-105

[39] Donald E. Knuth. 1997. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[40] Hans-Jürgen Koch. 2006. https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html Accessed on 2024-05-21.

[41] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. 2008. Validating High-Level Synthesis. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer, Berlin, Heidelberg, 459–472. https://doi.org/10.1007/978-3-540-70545-1_44

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, USA, 75.

[43] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[44] Alan Leung, Dimitar Bounov, and Sorin Lerner. 2015. C-to-Verilog Translation Validation. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, New York, NY, USA, 42–47. https://doi.org/10.1109/MEMCOD.2015.7340466

[45] Zikai Liu. 2023. *Generating Trustworthy I^{2}C Stacks Across Software and Hardware*. Master's thesis. ETH Zurich, Zurich, Switzerland. https://doi.org/10.3929/ethz-b-000632755

[46] Jiang Long and Robert Brayton. 2016. A Simple C to Verilog Compilation Procedure for Hardware/Software Verification. In *24th International Workshop on Logic & Synthesis*. IWLS, Mountain View, CA, USA, 99–106.

[47] Andreas Lööw. 2021. Lutsig: A Verified Verilog Compiler for Verified Circuit Development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3437992.3439916

[48] Christopher D. Marlin. 1980. *Coroutines*. Lecture Notes in Computer Science, Vol. 95. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-10256-6

[49] Advamation mechatronic. 2013. Raspberry Pi I2C clock-stretching bug. http://www.advamation.com/knowhow/raspberrypi/rpi-i2c-bug.html Accessed on 2023-11-30.

[50] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. 2000. Devil: An {IDL} for Hardware Programming. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. USENIX Association, San Diego, CA, 14 pages. https://www.usenix.org/conference/osdi-2000/devil-idl-hardware-programming

[51] Microchip 2021. *24AA512/24LC512/24FC512 512K I2C Serial EEPROM*. Microchip.

[52] NIST. 2024. NVD - CVE-2024-26593. https://nvd.nist.gov/vuln/detail/CVE-2024-26593 Accessed on 2024-05-05.

[53] NXP Semiconductors 2021. *I2C-bus specification and user manual*. NXP Semiconductors.

[54] Mattias O'Nils and Axel Jantsch. 2001. Device Driver and DMA Controller Synthesis from HW /SW Communication Protocol Specifications. *Design Automation for Embedded Systems* 6, 2 (April 2001), 177–205. https://doi.org/10.1023/A:1011246731756

[55] Ross B. Ortega and Gaetano Borriello. 1998. Communication Synthesis for Distributed Embedded Systems. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design (ICCAD '98)*. Association for Computing Machinery, New York, NY, USA, 437–444. https://doi.org/10.1145/288548.289067

[56] Johannes Åman Pohjola, Hira Taqdees Syeda, Miki Tanaka, Krishnan Winter, Tsun Wang Sau, Benjamin Nott, Tiana Tsang Ung, Craig McLaughlin, Remy Seassau, Magnus O. Myreen, Michael Norrish, and Gernot Heiser. 2023. Pancake: Verified Systems Programming Made Sweeter. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems (PLOS '23)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3623759.3624544

[57] Nadav Rotem. 2010. C-to-Verilog.Com: High-Level Synthesis Using LLVM.

[58] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic Device Driver Synthesis with Termite. In

Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09. ACM Press, Big Sky, Montana, USA, 73. https://doi.org/10.1145/1629575.1629583

[59] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-Guided Device Driver Synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 661–676. http://dl.acm.org/citation.cfm?id=2685048.2685101

[60] Samsung Electronics 1998. *KS0127 Data Sheet*. Samsung Electronics. https://alltransistors.com/superdatasheets/_pdf/09/ks0127.pdf

[61] Samsung Electronics 2000. *KS0127B Data Sheet*. Samsung Electronics. https://alltransistors.com/superdatasheets/_pdf/09/ks0127b.pdf

[62] SBS Implementers Forum. 2000. *System Management Bus (SMBus) Specification*. Technical Report. http://smbus.org/specs/index.html

[63] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. 2011. A Declarative Language Approach to Device Configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/1950365.1950382

[64] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. *ACM SIGPLAN Notices* 48, 6 (June 2013), 471–482. https://doi.org/10.1145/2499370.2462183

[65] Antmicro Open Source. 2020. ARTIX DC-SCM. https://opensource.antmicro.com/projects/artix-dc-scm/ Accessed on 2024-05-16.

[66] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. 2005. HAIL: A Language for Easy and Correct Device Access. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/1086228.1086230

[67] System Management Interface Forum, Inc. 2015. *PMBus Power System Management Protocol Specification Part I – General Requirements, Transport And Electrical Interface*. Technical Report. https://pmbusprod.wpenginepowered.com/wp-content/uploads/2022/01/PMBus-Specification-Rev-1-3-1-Part-I-20150313.pdf

[68] System Management Interface Forum, Inc. 2015. *PMBus Power System Management Protocol Specification Part II – Command Language*. Technical Report. https://pmbusprod.wpenginepowered.com/wp-content/uploads/2022/01/PMBus-Specification-Rev-1-3-1-Part-II-20150313.pdf

[69] The Clang Team. 2024. Clang: C Language Family Frontend for LLVM. https://clang.llvm.org/ Accessed on 2024-05-21.

[70] The Clang Team. 2024. Clang: Clang::DiagnosticsEngine Class Reference. https://clang.llvm.org/doxygen/classclang_1_1DiagnosticsEngine.html Accessed on 2024-05-21.

[71] The Clang Team. 2024. clang: clang::Rewriter Class Reference. https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html Accessed on 2024-05-21.

[72] The Clang Team. 2024. ClangFormat. https://clang.llvm.org/docs/ClangFormat.html

[73] The LLVM development community. 2024. The LLVM Compiler Infrastructure Project. https://llvm.org/ Accessed on 2024-05-21.

[74] The LLVM development community. 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html Accessed on 2024-05-21.

[75] D. Verkest, K. Van Rompaey, I. Bolsens, and H. De Man. 1996. CoWare—A Design Environment for Heterogeneous Hardware/Software Systems. *Design Automation for Embedded Systems* 1, 4 (Oct. 1996), 357–386. https://doi.org/10.1007/BF00209910

[76] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. 2015. Boolean Satisfiability Solvers and Their Applications in Model Checking. *Proc. IEEE* 103, 11 (Nov. 2015), 2021–2035. https://doi.org/10.1109/JPROC.2015.2455034

[77] Bingyao Wang, Sepehr Noorafshan, Reto Achermann, and Margo Seltzer. 2023. Synthesizing Device Drivers with Ghost Writer. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems (PLOS '23)*. Association for Computing Machinery, New York, NY, USA, 10–17. https://doi.org/10.1145/3623759.3624545

[78] Robert V. White. 2014. PMBus: A Decade of Growth: An Open-Standards Success. *IEEE Power Electronics Magazine* 1, 3 (Sept. 2014), 33–39. https://doi.org/10.1109/MPEL.2014.2330492

[79] Xilinx 2020. *Zynq UltraScale+ Device Technical Reference Manual*. Xilinx.

[80] Jeong-Han Yun, Gunwoo Kim, Choonho Son, and Taisook Han. 2006. Automatic Generation of Hardware/Software Interface with Product-Specific Debugging Tools. In *Embedded and Ubiquitous Computing*, Edwin Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon-Hae Kim, Laurence T. Yang, and Bin Xiao (Eds.). Springer, Berlin, Heidelberg, 742–753. https://doi.org/10.1007/11802167_75

[81] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, USA, 45–60.

A Artifact Appendix

A.1 Abstract

This artifact includes the Efeu compiler (ESMC) and the ESI/ESM specifications of the systems that we have modeled.

A.2 Description & Requirements

A.2.1 How to access. The artifact is publicly available at https://gitlab.inf.ethz.ch/project-opensockeye/efeu. It consists of two repositories: the compiler is in the `Efeu compiler` repository and the specifications in `Efeu Models`. The version of the artifact used for the paper is on branches called `eurosys-25` and persistently indexed at https://zenodo.org/doi/10.5281/zenodo.13734520.

A.2.2 Hardware dependencies. Nothing requires significant computing resources, except for Experiment 3, where we recommend using a machine similar to our platform (section 4.3) with at least 64GB RAM and 128GB swap memory. We recommend using x86-64 machines.

A.2.3 Software dependencies. We recommend the use of our pre-built Docker image. For installation instructions for Docker, see https://docs.docker.com/engine/install/. Without Docker, you need a UNIX-like OS like Ubuntu 22.04 where all the necessary dependencies can be installed with the following command:

```
$ sudo apt-get update && sudo apt-get install cmake llvm-15\
      llvm-15-dev llvm-15-tools clang-15 libclang-15-dev \
     gcc g++ gcc-aarch64-linux-gnu g++-aarch64-linux-gnu \
     spin python3 python-is-python3 python3-numpy python3-\
     pandas python3-matplotlib less hyperfine cloc clang-\
     format
```

A.2.4 Benchmarks. All benchmarks are in the artifact.

A.3 Set-up

The first step is to get the Efeu source code by cloning the repositories `Efeu compiler` and `Efeu Models` and switching to the `eurosys-25` branch in both. We assume that the repositories are in a directory stored in a shell variable called `${EFEU_ARTIFACT}`.

The `README.md` file in the compiler repository describes how to build the Efeu compiler from source and run its test suite. The `README.md` file in the models repository then describes how to configure a native build environment. We recommend however using the pre-built Docker image we provide that contains the compiler binaries and additional dependencies. To create the artifact container, run the following commands:

```
$ docker create -it --name efeu-artifact --user efeu-\
     builder -v ${EFEU_ARTIFACT}/efeu-models:/home/efeu-\
     builder/efeu-models -w /home/efeu-builder/efeu-models \
     registry.ethz.ch/project-opensockeye/efeu/models-build\
     :eurosys-25
$ docker start efeu-artifact
$ docker exec --user root efeu-artifact groupmod -g $(id -g\
     ) efeu-builder
$ docker exec --user root efeu-artifact usermod -u $(id -u)\
      -g $(id -g) efeu-builder
$ docker stop efeu-artifact
```

You can then start and attach to the container using the following command:

```
    $ docker start efeu-artifact && docker attach efeu-\
     artifact
```

Next, create two build directories: one for a default build that excludes the long-running verifiers and the other one for an extended build with all the verifiers.

```
$ mkdir build build-extended
$ cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++ -S \
     . -B build
$ cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++ -\
     DEXTENDED_BENCHMARKS=ON -DTIMEOUT_COMMAND='timeout 30m\
     ' -DTIMING_COMMAND='hyperfine --runs 5' -S . -B build-\
     extended
```

You can run experiment 1 to test whether you have a working setup.

A.4 Evaluation workflow

A.4.1 Major Claims. Here we list all of our major claims and their corresponding experiments below.

C1. Efeu generates Promela models and hardware/software implementations of the standard I^{2}C stack (section 2.3) and the non-standard devices (section 4.5) from the specification. This is supported by experiment E1.

C2. Efeu reduces the development cost and avoids human errors by automating the translation from the specification to models and implementations. The size of the generated code is reported in Table 1 and Table 3, which can be reproduced by E2.

C3. The (whole or partial) standard I^{2}C stack with one or more EEPROMs can be verified in reasonable runtime and the abstraction levels help improve the scalability, as shown in Table 2 and Figure 9. This can be reproduced by E3.

C4. Non-standard devices described in section 4.5 can be modeled and verified with minimal manual changes. This is demonstrated by E4.

C5. Linux bit-banging I^{2}C driver cannot achieve the target transmission speed, while Efeu-generated hybrid hardware/-software drivers achieves the target performance matching the Xilinx I^{2}C IP crafted by human experts, as described in section 5.2. This corresponds to E5 and E6.

C6. Linux bit-banging I^{2}C driver has 100% CPU usage, while Efeu-generated hybrid hardware/software drivers in the interrupt-driven mode consumes minimal CPU as described in section 5.3. This corresponds to E5.

C7. Efeu-generated hybrid hardware/software drivers consume minimal FPGA LUT and FF resources, as detailed in section 5.4. This corresponds to E7.

A.4.2 Experiments. All the following commands should be executed at the root of the model repository. The time estimates are of the form [human + compute].

Building and evaluating hybrid hardware/software drivers on real hardware as described in section 5 requires access to special equipment (e.g. licensed Xilinx Vivado, the UltraScale+ board, an EEPROM, and an oscilloscope). It is therefore out of the scope of this artifact evaluation. However, we provide raw data we collected and the scripts to reproduce the results for claims 5–7 in experiments 5–7.

E1. [1min + 15min] First invoke the Efeu compiler through the build system (CMake) to generate all code (Promela, C, Verilog, VHDL) of all configurations. Then run the basic set of verifiers. You can do so using the following command:

```
$ cmake --build build/ -t all verify-all
```

The build and verification should report no errors. You can find the generated files in `build/systems`. The verification going through confirms that all the verification code has been correctly generated. To test that the generation of the driver binaries worked correctly, you can run them. As mentioned, most require specialized hardware, but there is one that simulates an EEPROM and writes 14 bytes, then reads 4 of the bytes back. You can run it as follows:

```
$ ./build/systems/eeprom/eeprom/impl/linux-test/top-down-c1\
    -r1/sys-cmp_eeprom_linux-test_top-down-c1-r1
```

The output should contain the following two lines that show the results of the write and read:

```
[CWorld] res: CE_RES_OK
[CWorld] res: CE_RES_OK [2]42 [3]43 [4]44 [5]45
```

E2. [1min + 1min] Count lines of code. We provide a script for formatting and counting. The following commands ensure that all the code is generated and then run the script to print lines of code corresponding to Table 1 and Table 3:

```
$ cmake --build build/
$ ./data/ae/count-loc.py
```

E3. [1h + 15h] Run all verifiers and get runtime. The Verifiers are invoked through CMake. We provide a script to extract the runtime from the CMake log. Running all verifiers in section 4 takes *hours*. We recommend first trying out the data processing script with the provided log files:

```
$ ./data/ae/process-verification-runtime.py
```

The script prints aggregated data as shown in section 4 and generates a figure resembling Figure 9. By default, it uses the *provided* log in the artifact, unless new log files named `./data/ae/ae-*.log` are found. To execute all verifiers and generate those new log files, use the following command:

```
$ cmake --build build-extended -t verify-all -- -j1 -k | \
    tee ./data/ae/ae-$(date +%Y%m%d%H%M%S).log
```

Note that this command invokes CMake in the *extended build* directory. Running all verifiers will take hours (you may want to use `screen` or similar utilities if using a remote server). Some verifiers with higher payload length or larger number of EEPROMs will likely time out or run out of memory. To reduce the necessary user interaction, the command supplies the `-k` option to continue with other verifiers, even if one fails to complete. To only run the set of verifiers used in Figure 9 replace `verify-all` with `verify-eurosys25`.

To process the newly acquired log files, rerun the data processing script. It will list the verifiers for which it cannot find timing data in the logs. To run them, replace the `verify-all` target with one suggested by the script. The script will process all log files it finds to aggregate data from several runs.

E4. [1min + 5sec] Compares the ESM specification of KS0127 and Raspberry Pi I^{2}C with the standard I^{2}C stack.

```
$ diff -u systems/standard/layers/_Byte.inc.esm systems/\
    ks0127/layers/_Byte-KS0127.inc.esm
```

Note that this ESM file is shared by the controller and the responder. Excluding comments, The KS0127 `Byte` ESM adds 13 lines for responder (in line 67 to 93) and 10 lines for controller (in line 113 to 145). Similarly, the following command compares the controller `Symbol` of Raspberry Pi with the standard one.

```
$ diff -u systems/standard/layers/CSymbol.esm systems/\
    raspberry-pi/layers/CSymbol-no-stretching.esm
```

E5. [1min + 5sec] Compute transmission frequencies and variations from data collected from the oscilloscope.

```
$ ./data/ae/timing-and-cpu.py
```

The script processes the CPU usage data and the timing data of rising and falling edges of SCL. A figure closely resembling Figure 10 will be created. Warnings about "invalid values encountered in reduce" can be ignored.

E6. [1min + 5sec] Plot waveforms in Figure 11 from raw data collected from the oscilloscope.

```
$ ./data/ae/waveform.py
```

The script will create a figure closely resembling Figure 11.

E7. [1min + 5sec] Reproduce Figure 12 and Figure 13 from the original reports from Xilinx Vivado.

```
$ ./data/ae/fpga-utilization.py
```