Towards high-assurance Board Management Controller software

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 317

Systems Group, Department of Computer Science, ETH Zurich

Towards high-assurance Board Management Controller software

by

Cedric Heimhofer

Supervised by

Daniel Schwyn
Dr. David Cock
Prof. Dr. Timothy Roscoe

September 2020 – March 2021

D INFK

Abstract

Traditional software running on (base)board management controllers (BMC) is not high-assurance. It provides no guarantees about security properties or correctness. This is in stark contrast to the high sensitivity and criticality of many tasks it performs.

To this end, we built BMC system software for Enzian, a research platform developed by the Systems Group of ETH Zurich, based on the formally verified microkernel seL4: We designed a statically configured system using CAmkES with which we implemented power management for the main Enzian board. This entailed writing drivers for some devices and replaying bus traffic that was captured ahead of time to configure additional devices. We bring up a selected subset of the power and clock tree and put emergency procedures in place.

We evaluated the emergency procedures under different types of synthetic workloads. Good end-to-end response times and sufficient isolation of real-time tasks were observed. The major contribution to the execution time of the emergency procedures was found to come from bus operations.

With this work we made the first step towards high-assurance BMC software and demonstrated that seL4 is a viable alternative, offering not only strong guarantees, but also good performance.

Contents

List of Figures

List of Tables

1 Introduction

Computers are complex machines. Naturally, bootstrapping and maintaining them in a healthy, operational state is non-trivial: Tasks such as supplying power to the various components, clock management, temperature monitoring, fan control and other functions are crucial for correct operation. These tasks are usually performed by a separate chip called the (base)board management controller (BMC). Often, the software running on the BMC is highly manufacturer specific and proprietary. It is generally invisible to the host operating system (OS).

For correct host operation, correct BMC operation is imperative. Any guarantees by the host OS fundamentally break down if the BMC does not behave as assumed. Because of their closed nature it is difficult to make any statements about the BMC itself. This in spite the fact that the BMC is a sensitive piece of the system: Due to its low-level access to devices, incorrect operation can have devastating effects on hardware, possibly breaking the devices altogether.

To make matters worse, BMCs frequently incorporate remote management functionality. In this case, the BMC needs to be connected to a network interface, which is sometimes shared with the host. This raises additional concerns from a security perspective, as the threat model is now drastically different. Although BMC have increasingly become a target for security researchers, little is known about producing BMC software that is correct, either by design or through verification. To this day, most BMCs have remained a black-box giving no guarantees whatsoever.

1.1 Enzian

The Systems Group of ETH Zurich developed a heterogeneous server class computer named Enzian. It consists of a 48-core Marvell Cavium ThunderX processor supporting the 64-bit ARMv8-A architecture, alongside a Xilinx Virtex UltraScale+ FPGA. Both chips control a relatively large amount of DRAM.

Since the Enzian board was designed from scratch, BMC software also has to be created. A lot of work is being put into the Enzian BMC: Device drivers have to be written, various different kinds of policies have to be put in place and real-time tasks need to be implemented. The functionality of the Enzian BMC is discussed in more detail in 2.1.2.

1.2 Problem statement

A task needs to be high-assurance if failing to complete it successfully can be costly. High-assurance is often put into the context of cyber security, where it refers to resilience against malicious agents. However, the term is more general. It can also refer to protection against unintended behavior or against failure of a component.

BMCs make up a vital part of modern server computers and thus perform many tasks that need to be high-assurance. Unfortunately, traditional BMC software does not provide any assurance guarantees. Early BMC specifications have neglected best security practices [6] and many implementations by manufacturers have historically contained serious flaws [3]. Open-source implementations do exist (as discussed in section 2.1.1), but they are not free from such flaws [4, 5].

Additionally, due to the complexity of modern hardware and the embedded nature of BMCs, it has become increasingly challenging to develop and maintain such a complex piece of software and bugs become more likely.

1.3 Contribution

For increased trustworthiness of the Enzian BMC, we port the formally verified microkernel seL4 [10] to the Enzian BMC hardware. We therefore provide the first step towards high-assurance baseboard management: Not only does seL4 provide high-level of assurance, kernel behavior is also specified on an abstract level. This allows for guarantees by the seL4 kernel to be extended in the future, to cover other aspects of the system.

We implement partial power management functionality to demonstrate that such an approach is a feasible alternative to the current approach featuring a much larger trusted computing base. We present a method to capture I^2C traces on the existing BMC implementation that can be replayed on seL4. With the help of this method, we configure a selected set of devices.[1] We also implement emergency procedures and show that short response times can be achieved over a variety of different workloads – without any notable compromises during system design.

[1] More concretely, we configure the clock-tree, the power supply unit and the following voltage regulators: both ISPPAC, the IR3581, the IR3581 loop devices and all MAX15301 devices.

2 Background

In this section we discuss BMC software in general and present already existing software solutions. This is followed by an overview of the required functionality of the Enzian BMC and a brief refresher about the I^2C bus.

We then turn towards seL4 and explain some of the core concepts and functionality exposed by the kernel. We will apply these concepts in section 3, where we design the BMC system based on seL4 and CAmkES (see below).

2.1 Baseboard Management Controller

There is no universal definition of a BMC and a certain amount of ambiguity about what constitutes to "baseboard management" is always involved. This is exacerbated by the fact that BMCs are often a closed platform and hardware vendor specific. As such, it becomes difficult to determine what constitutes to the BMC software: A hardware designer can eliminate the need for a BMC, by using hard-wired power-up sequences [25], closed-loop fan controllers, factory-programmed clock generators and boot ROMs, and so on. Indeed, BMCs – often running a full-fledged OS – cannot themselves require a BMC.

There are a few standards when it comes to BMCs, the most prominent one being the Intelligent Platform Management Bus (IMPI) [9]. It has heavily influenced what is considered a typical BMC, but board manufacturers usually add additional functionality. In the following, we try to list some of the typical functions of a BMC: [18, 27, 8]

Health monitoring and management Many BMC provide health monitoring services such as temperature sensors, watchdogs and hard disk fault monitors. Quite a few BMC make this data available through a web interface. It can also include recovery actions or other response measures.

Data collection and logging One of the key tasks of a BMC is to collect data about the state of the hardware. It often has alert capabilities or even runs a notification service such as a mail server.

Booting The main CPU does not only need power and clocks signals to boot, but also software. In many cases the BMC can install firmware, provide a virtual boot media or install a host OS directly.

Remote access The BMCs functionality is made available over the network or some other interface. It may also function as a keyboard, video and mouse (KVM) switch such that system administrators can also remotely log into the host OS.

We identified some properties that typically set a BMC apart from other system components:

- It does not run on the main CPU. It is in a different power domain, so the BMC runs on a separate piece of hardware.
- It does not interfere with the execution of the host OS. In particular, it does not audit what the host OS is doing, perform attestation or enforce any security policies. Nothing prevents it from implementing any of this, but generally a BMC does not.

2.1.1 Existing BMC software suites

Almost each server or motherboard manufacturer has its own BMC firmware. In this section we focus on the few publicly available projects. The observations are loosely based on the principles and case-studies for trustworthy BMCs as stated in [1].

OpenBMC is an open-source BMC implementation based on Linux and D-Bus [18]. It is the most widely supported BMC system software. It takes a modular approach, allowing vendors

to add their functionality as D-Bus services. It uses the Yocto Project to configure a static executable and unneeded functionality may be disabled by configuring the BitBake recipes. It has a security working group that meets bi-weekly and aims to improve OpenBMC security.

It has a large feature set, so it can be difficult to configure securely. This is exacerbated by the fact that BMC standards were not designed with security in mind. Additionally, vendors would historically emphasize functionality over security, which means it ultimately boils down to the user to properly configure the software supplied by the vendor. The large code base means it can be difficult to maintain and the probability of software bugs increases. Being a Linux based operating system, it can use its access control model to enable multiple trust domains. However, it does not seem to properly isolate security domains (yet) [19].

u-bmc is an experimental BMC based on u-root, a Go userland for Linux [31]. It is closely related to OpenBMC, but it removes many unsafe features and replaces them with alternatives. There are only few boards supported.

By limiting functionality, u-bmc reduces the chances of programming mistakes. Its limited adoption however may cause more programming errors to go unnoticed. High-assurance code is historically "obtained" by extensive testing and auditing. u-bmc does not seem to try to change that.

OpenPower POWER8 chips contain a special On Chip Controller (OCC), a PowerPC 405 running the real-time open-source operating system SSX [17]. The OCC implements power and temperature monitoring of processor and memory. It throttles these agents if the values exceed a specified threshold. The host still has a traditionally BMC (such as OpenBMC) and the OCC reports the values to the BMC.

By spatially partitioning the baseboard management software into a real-time domain and best-effort domain, a high degree of temporal and spatial isolation is achieved. Additionally, the real-time tasks are now in a different failure-domain. However, a large portion of the baseboard management functionality (such as fan control) is still performed by a traditional BMC.

2.1.2 The Enzian BMC

The main task of the Enzian BMC is to manage power supplies of the main CPU (Marvell Cavium ThunderX) and the main FPGA (Xilinx Virtex UltraScale+ XCVU9P). It has access to the flash device which is used to boot the ThunderX and it is responsible for programming the XCVU9P. It controls all fans (ThunderX, XCVU9P and case fans) and it manages the LEDs and switches.

Physically, the Enzian BMC is an Enclustra Mercury ZX5 module containing a Xilinx XC7Z015 System on Chip (SoC). As a member of the Zynq 7000 series, the SoC contains a (dual-core) ARM Cortex-A9 and a "small" FPGA. The module has 1 GiB SDRAM and 576 MiB flash memory.

The BMC has a dedicated gigabit ethernet network interface. It is connected to the ThunderX via a network controller sideband interface (`B_NCSI`) and to the XCVU9P via a multi-gigabit transceiver link (`B_C2C`). All of Enzian's UARTs[2] are routed to the BMC, which then can forward the signals to the device that exports the UART signals over USB. The BMC also has a JTAG interface.

System bootstrapping The BMC can be used to program the flash storage device used to boot the ThunderX. The flash device itself is a 16 MB NOR flash, accessed by both parties via an SPI multiplexer.

The XCVU9P can be programmed through a slave-serial interface by the BMC. The ThunderX cannot load an initial bitstream to the FPGA.

[2]There are two UARTs from the ThunderX and one from the are XCVU9P. The BMC itself has additional UART controllers.

Power managmenent Enzian contains various clock generators, voltage regulators and other devices to provide power and clock signals its components. Some of them need no configuration or can be configured statically to various degrees, either in the factory or in the field. Others do not contain any non-volatile memory and need to be configured dynamically. Many devices can be enabled (or disabled) manually. More sophisticated devices also collect sensor readings which is exported over a I^2C bus interface.

The BMC dynamically manages just over 20 devices used to provide power and clock signals to the ThunderX, XCVU9P and DRAM. These devices are arranged in a tree-like configuration. They need to be configured and enable in the right order before the ThunderX or XVU9P can be started. This is by itself a complex task [20].

The BMC must also respond to and handle SMBus alerts (refer to section 2.2) raised by the devices it manages. Most devices have protection mechanisms built into them (e.g. a MAX15301 voltage regulator can be configured to turn itself off if the temperature exceeds a certain limit), but some of them require that the BMC manually intervene. To prevent damage to the board, it is important that this latency be kept small.

There are two sequencing controllers on the Enzian board, one for the CPU subsystem and one for the FPGA subsystem. These sequencing controllers manage most of the enable signals and monitor the voltages of the individual rails.

Software The Enzian BMC runs a version of OpenBMC. All of the interfacing with the main Enzian board is done over the BMC's own FPGA. The relevant controllers are instantiated on the programmable logic. We use a modified version of the Linux kernel provided by Xilinx, which provides all relevant drivers.

Power management is done in user-space by a collection of Python scripts communicating over a D-Bus interface. The software consists of the following services:

- The *power service* performs most device interaction and maintains state for the I^2C buses and connected devices. It has exclusive access to the I^2C buses and, by being a single-threaded process, serializes access to them. It contains the device drivers for the I^2C devices. It breaks down the high-level device operations that it exposes over D-Bus, into actual, possibly multiple, bus operations. It contains only limited amount of device configuration information and should be oblivious to the topology of Enzian (in principle).

- The *gpio service* manages individual GPIO pins and serializes access to them. It forwards changes on input pins as signals over D-Bus.

- The *fault service* takes the appropriate measures in case an alert is raised. Once the power service has determined the origin and cause of an alert, it notifies the fault service of the fault.

- The *telemetry service* is used to collect data from the devices. It periodically asks the power service to read monitor data from the devices and logs them to a file.

Device configuration is done by a client program, that knows how to set the device parameters and when to enable the devices.

2.2 I^2C

I^2C (Inter-Integrated Circuit) is a low-speed computer bus [21]. Baseboard management on Enzian heavily relies on communication over I^2C buses: Most devices responsible for providing clock and power to the main CPU and FPGA are configured over I^2C. We therefore decided to provide a short overview of some of the relevant aspects of I^2C.

I^2C is a half-duplex, two wire bus. A device that initiates and terminates transmissions is called a *master device*, while devices addressed by another master device are called *slave devices*. There can be multiple master or slave devices connected to the same bus. Each (slave) device has

a unique address. On Enzian, the BMC is the only I^2C master device and all other devices are slave devices.

To start a transmission, the bus master (in our case the BMC) first issues a special start signal. It then sends the address of the slave device that it wants to communicate with out on the bus. The master then proceeds to read data from the slave device or write data to the slave device. Once it is done reading and writing, it issues a special stop signal, so that the bus becomes free again. To keep slower devices from being overwhelmed by faster ones, any active device is allowed to throttle the bus.

Whenever a byte (or an address for that matter) is successfully received, the receiving side must respond with a single acknowledgment bit. A notable exception is when the bus master is reading from a slave device: In that case the master is expected to not-acknowledge the last byte it is willing to receive, to signal to the slave device that it is going to stop the transmission.

The System Management Bus (SMBus) is a specification that builds on I^2C [28]. Most notably, it defines what are called *SMBus Protocols*. An SMBus protocol simply specifies the number of bytes that are sent and received. For example, a master issuing the "Read Word protocol" must first write 1 byte to the slave device and then read 2 bytes from it. Not all possible I^2C transactions have a corresponding SMBus protocol. For example, there is no SMBus protocol that first writes two bytes to the slave device and then reads two bytes.

Additionally, the SMBus specification (optionally) adds a third wire that functions like a level-triggered interrupt. Since slave devices cannot initiate a I^2C transmission, an out-of-band signal was introduced to allow them to get the attention of the host.

There are 5 I^2C buses on the Enzian board, but only 3 of them are used for power and clock management. The other 2 do not carry any devices and are exposed to external connectors or plugs. Two out of the 3 I^2C buses used for baseboard management feature an SMBus alert line. The SMBus alert signals are available through the GPIO controller.

2.3 Overview of seL4

seL4 is a microkernel. As such, most of the functionality needs to be provided by user space agents. For example, the driver for the GPIO controller that we wrote runs in its own address space in user space. It communicates with other components of the system via inter-process communication (IPC), which explained in more detail in section 2.3.1.

The seL4 kernel provides a capability-based access control model. A capability is a reference to a well-defined object. Examples of such objects include in-kernel data structures (such as a thread control block), a time-slice for a particular core or an interrupt request (IRQ) number. Some types of capabilities also encompass access rights. Whenever a user process requests a service from the kernel it must present one or more capabilities to demonstrate that it has sufficient rights. Capabilities are thus an integral part of the system.

There are currently two versions of seL4 called "branches": The master branch and the mixed-criticality systems (MCS) branch. The MCS version adds real-time scheduling functionality.

This section describes the seL4 API exposed by the kernel and is based on [30].

2.3.1 Inter-process Communication

Conceptually, all of the functionality provided by the seL4 kernel is implemented on top of its message passing mechanism. The messages either go to the kernel (classical system calls) or to other threads (IPC). Messages destined for other threads require an *endpoint object*. An *endpoint capability* is a capability to an endpoint object and it is used to interact with an endpoint (object). Messages can be sent and received over endpoints, both in blocking and non-blocking manner. However, IPC is synchronous, so two threads need to rendez-vous for a message to be sent successfully.

Passing messages revolves around word-sized logical *message registers*. There are a limited number of these, currently 120 (`seL4_MsgMaxLength`). Physically, they consist of CPU registers and part of the IPC buffer, which is special frame referenced in the thread's TCB. When a thread

	IA-32	x86-64	ARM	RISC-V
Fast message registers	1	4	4	4

Table 1: Number of CPU registers available for messages in the fastpath system call.

sends a message to another thread, the message registers get transferred from the sender to the receiver.

Sender authentication Since each thread can block on at most one endpoint, a thread that wants to wait for messages from multiple (client) threads necessarily needs to use the same endpoint for all clients. In case it needs to distinguish between multiple possible senders, endpoint capabilities can be *badged*.[3] When a message is sent through a badged capability, the receiver will obtain the message together with a copy of the badge. In case the sender has multiple badges to the endpoint, it can optionally arrange for multiple badges to be transferred (capability unwrapping).

Fastpath Much of the functionality of monolithic kernels is implemented in user space on microkernels. What is a single system call on monolithic kernels thus becomes two context switches on microkernels. Therefore, IPC on microkernels needs to be fast.

The seL4 fastpath is an alternate kernel code path which is optimized for speed [11], inspired by the L4 microkernel family, in particular L4Ka::Pistachio. In order to use faspath for a given message, certain conditions have to be met. They include:

- The receiving thread must already be waiting on the endpoint at the time the message is sent.
- The receiving thread must have a high enough priority to be scheduled immediately.
- No capabilities may be transferred with the message.
- The message must fit into the CPU registers. The number of available CPU registers (`seL4_FastMessageRegisters`) for message passing is architecture dependent, see Table 1.

Whenever possible, we should try to make use of fastpath.

2.3.2 Synchronization Primitives

The kernel supports event based synchronization, built around notification objects. Although similar to message endpoints in their functionality, the later are not intended to provide synchronization [7]. Notification objects can be signaled and waited upon (and polled). They function a bit like semaphores, as depicted in figure 1.

Like endpoints capabilities, notification capabilities can also have badges: When a thread signals a notification, the badge of the notification capability used for signaling gets returned to the waiting thread. If no thread is waiting, the badge is instead bitwise ORed to an in-kernel word-sized buffer part of the notification object, whose value will be returned upon the next call to `wait`. This allows for different "signals" to be transferred, although they might become combined if no thread is waiting.

2.3.3 Threads and scheduling contexts

The MCS scheduler aims to run time-sensitive tasks (e.g. a NIC-driver) at a higher priority than the critical tasks. The critical tasks can then run in the slack time of the time-sensitive tasks.

[3] Once a badge is applied to a capability, it can neither be removed nor changed. Badged endpoint capabilities can still be freely passed around however. In particular, there can be multiple threads using the same badge on an endpoint. In that sense, badges function a bit like user-managed capabilities.

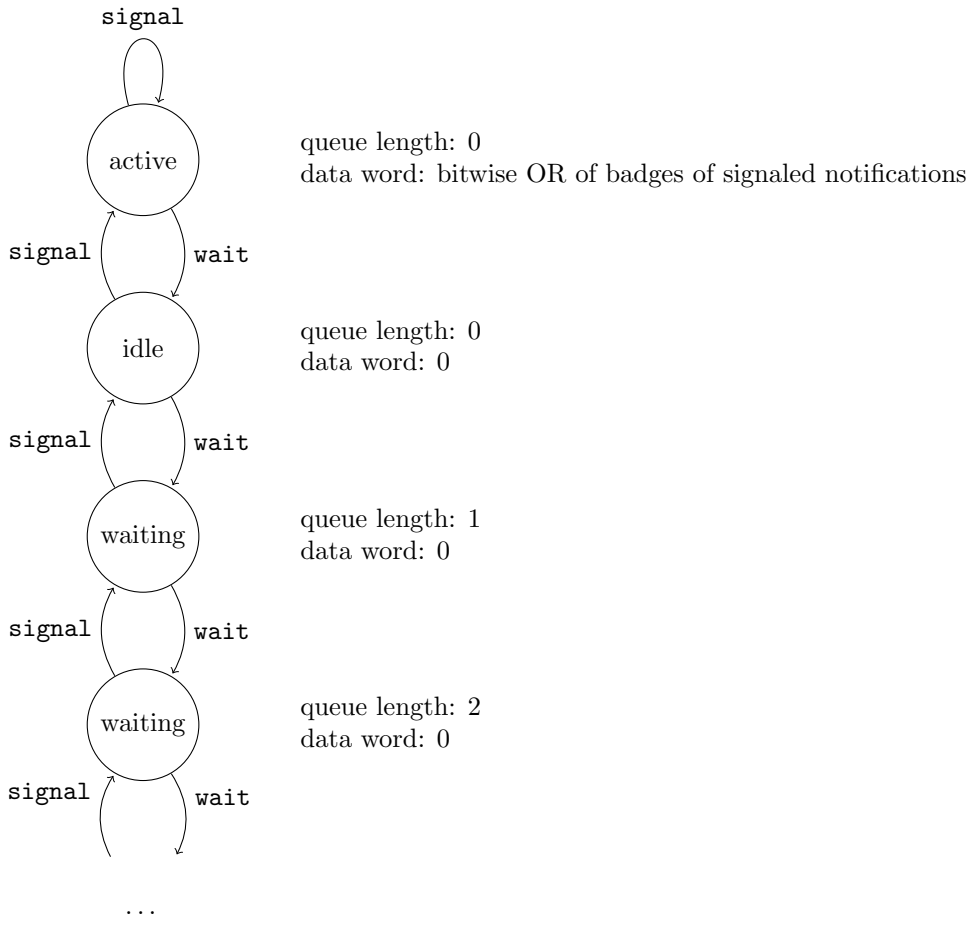

Figure 1: A state diagram describing the three states (*active*, *idle* and *waiting*) a notification object can be in. The size of the queue is not strictly part of the state of the notification object, but the it might affect how the notification object transitions upon `signal`.

Maximum CPU utilization is assigned to the time-sensitive tasks. The kernel ensures that the maximum utilization constraints are enforces so that there is enough slack-time to run the critical tasks [15].

Although we ended up not using the MCS scheduler, we still briefly discuss it here. This is because the classical seL4 scheduler can be considered as a special case of the MCS scheduler (although there are some other differences). Furthermore, we originally intended to use the MCS scheduler (as is recommended).

Threads are represented by *thread control blocks* (TCB) in the kernel. A thread is said to be *runnable* if it could be run as-is (e.g. it is not blocked or suspended, has a valid register set, address space and capability space, ...). A priority between 0 and 255 is assigned to each thread.

A *scheduling context* is a commitment not to run longer than the given time slice. A scheduling context uses two variables to represent a time slice: a period p and a budget $b \leq p$. The kernel ensures, that in any interval of p microseconds, the scheduling context is "used" (including in-kernel execution) for at most b microseconds. A scheduling context is said to be *active*, if it still has budget available. Each scheduling context is associated with a specific core.[4]

Threads and scheduling contexts can be bound together in a 1:1 relation. Threads need a scheduling context to be scheduled. A thread without a scheduling context is called a *passive* thread. A runnable thread with an active scheduling context is said to be *schedulable*. The kernel will schedule in a round-robin fashion all schedulable threads of highest priority.

Budget replenishment A thread bound to scheduling context with period p and budget $b < p$ is called a *sporadic thread*. A thread bound to a scheduling context with period p and budget $b = p$ is called a *round-robin thread*.

For sporadic threads, the scheduler uses the replenishment algorithm as presented in [26]: Each scheduling context maintains a queue of refills. Each refill consists of a portion of the scheduling context's budget and a timestamp when this portion of the budget becomes available. The refills in the queue are ordered according to their arrival time and together they make up the entire budget of the scheduling context. The maximum size of the queue is fixed when the scheduling context is created.

On the classical seL4 scheduler, all threads are round-robin threads and the length of a time slice is configured at build time.

Migrating scheduling context When a passive (server) thread is blocked on an endpoint and a (client) thread calls said endpoint, the scheduling context from the client thread is transferred to the server thread. If the server thread previously ran on a different CPU core than the client thread is running on, the server thread is migrated to the CPU core of the scheduling context.

Scheduling contexts will not be donated through notifications. They can however, be bound to notifications, such that a passive thread blocked on the notification will receive the scheduling context once it is signaled.

2.3.4 Formal verification

seL4 has a long history of formal verification. The original paper establishing the functional correctness down to the C source code would mark the beginning of an ongoing effort to verify different aspects of a system. The original result would later be extended with binary correctness [22], integrity and confidentiality [23] [16], worst-case execution time analysis [2] and further results.

It is important to note that these claims are usually specific to a instruction set architecture or even platform. The kernel also has multiple configurations options (e.g. SMP, MCS) that are not verified (yet). The seL4 homepage has more information about covered configurations and proof assumptions.

[4] A scheduling context is not a "right" to run on a specific core. In particular, the sum of all values $\frac{b}{p}$ of all scheduling contexts for a specific core can (and often will be) greater than 1.

2.3.5 CapDL and CAmkES

CapDL (Capability Distribution Language) is a domain-specific language to describe the state of kernel objects and capabilities in the system [13]. Beside being useful for debugging purposes, CapDL descriptions are sufficiently specific such that they can be given to the root task to perform system initialization according to the specifications.

CAmkES (component architecture for microkernel-based embedded systems) is a platform independent domain-specific language for the development of system components [12]. It allows high level system configuration and exposes C language bindings to interface with the high-level objects. Behind the scenes, CAmkES generates a CapDL description of the system and relies on CapDL Loader to instantiate it at boot time.

3 Design

In this section we present our thoughts behind the design of the resulting artifact. Naturally, since we are building a system using seL4, much of this section requires some understanding of the operating principles of seL4 (see section 2.3), but a fair portion is independent of the underlying infrastructure. Thus, some of this section also applies to the current Linux implementation of the BMC software. It may, however, be possible that these issues are hidden or otherwise obscured by higher level functionality.

3.1 The BMC programmable logic (PL)

As mentioned in section 2.1.2, the interfacing with the main Enzian board is done over the PL. We therefore briefly introduce the functionality of the bitstream that we load to the BMC's FPGA.

The bitstream is mostly the same as the "regular" one that use for the Linux implementation. The regular one primarily consists of a network controller for the NCSI link to the ThunderX, an SPI controller for accessing the boot memory of the ThunderX, another SPI controller for loading an initial bitstream to the XCVU9P, some UART routing mechanisms and three GPIO controllers for various signals and I^2C buses. Most of these devices we do not actually use.

To this regular bitstream, we added six additional I^2C controllers: One for each I^2C bus of the Enzian board and an additional one for the I^2C bus on the development board. The rationale behind using the I^2C controllers instead of bit banging the GPIO controller (as the Linux implementation) was the following: It seemed simpler to get I^2C controllers working, as then we would not have to worry about obtaining accurate timings to clock the I^2C bus. Whether this ultimately turned out to be true is open for debate (probably so, but see also section 4.2.3).

3.2 Individual software components

There is a non-trivial set of user space software required for a microkernel-based system to do anything useful. At the very least, this set consists of user space bootstrapping functionality and driver software to interact with the physical world. We decided to use CAmkES which takes care of the initialization process.

What we are mostly concerned with is building the individual components that make up the overall system. This entails partitioning the system in multiple isolated address and capability spaces that interact with each other over well-defined connections. In this section, we introduce the individual component and briefly discuss their interfaces as we go along. A high-level overview is given in figure 2.

A Let us first discuss the term *component*. For a thread to be scheduled, it needs to be assigned an address space and a capability space (among other things). Two threads can share address spaces, capability spaces, or both. To obtain the highest level of isolation, one could technically run all threads in their own address space and their own capability space. On the other hand, it makes sense from a performance viewpoint, to share address spaces for highly coupled threads. The following question then naturally arises: Should two threads in the same address space use different capability spaces?

We think there is little to be gained by using different capability spaces for two threads in the same address space. In principle, the address space specifies what a thread *does* and the capability space specifies what a thread *can do*. Note that two threads T and S in the same address space can easily change each other's behavior.[5] Therefore, T can also do anything S can do, by just making S do it instead.

In light of this, a *component* is an address space together with a capability space. Two components are either the same, or both their address space and their capability space differ. A component usually has one or multiple threads.

[5]For example, each thread has write access to the stack of the other thread.

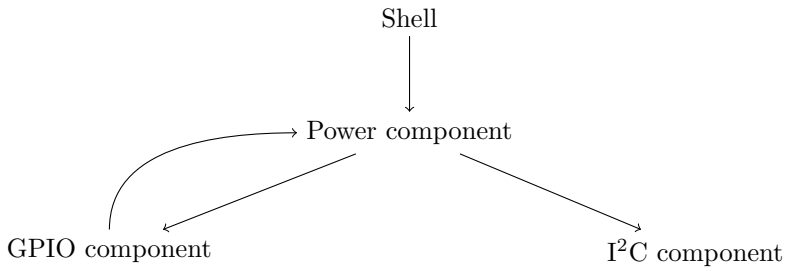

Figure 2: Overview of the system components. The arrows indicate transfer of control flow (return of control flow is not depicted).

This is the default behavior of what CAmkES calls components. However, CAmkES allows components to share address spaces (called "component groups", or "collocated components") but we therefore do not make use of this feature. Hence the term "component" remains as defined.

3.2.1 Device driver software

On microkernel based operating systems, there are generally two ways to write device drivers:

The library-based device driver approach, where the device specific code resides in libraries imported by the components that use the device. In case multiple components want to access the same device, it can become tricky to safely multiplex the device, since the device registers are typically mapped directly to the individual address spaces.

Another possibility would be to create a dedicated component that is responsible for programming the device. This "driver component" then makes the device available to the rest of the system through a device interface. All accesses to the device then need to through this component, which can in principle perform arbitrarily sophisticated multiplexing policies. Usually, this approach still requires a library on the client components.

We generally think the driver component-based approach has the following advantages, although not all of which apply in the current setting:

- The TCB for correct device operation is bounded, no matter the number of components that have access to the device (assuming that proper access-control is implemented). Hence the component-based approach can be said to be more future-proof: Even if more client components are added in the future, the device's TCB will not increase. In the library-based approach, we need all components using the device library to function correctly and to implement the synchronization procedures correctly for proper device operation. Therefore, a component-based approach can be said to also be more foolproof (with respect to programming errors).

- The driver component can perform fine-grained access control. Often, there is multiple ways "use" a device: An UART controller can both send and receive characters, a GPIO controller might have multiple independent pins and a bus controller is effectively nothing but a means to talk to additional devices. The driver component could then perform access-control to the individual device functions independently.

- A component-based approach is more modular than a library-based one. Devices can more easily be replaced, even during run-time: The user might want to hot-swap a device or reprogram the FPGA with a different bitstream. In such a case, one could simply load a different driver component for the new device and reconnect the interfaces possibly without the client component noticing. The driver component might not even be backed by a real device (simulation). All of this is not impossible for library-based implementations, but much harder. This advantage is less prominent in our case, where the system is configured statically using CAmkES.

There are various device drivers and/or device registers we needed to configure. Some of them already had seL4 support (such as the timers of the Cortex-A9) or needed only very little additional programming (e.g. UART). There are two devices that we use extensively and decided to discuss in more detail:

GPIO controller A variety of signals are controlled by the GPIO controllers instantiated on the PL side. Some GPIO pins are only able to sense values, other can only drive the wire. A few pins have an associated tri-state buffer, with the help of which they can do both. It is worth noting that the pins are arranged in multiple *channels*. Pins in a channel are always processed in batch: Their values are read simultaneously and they are written simultaneously. This is a useful property, as it allows multiple pin values to be changed "at the same time". It however also implies that the values in one channel cannot be written independently.

We decided to create a dedicated component for the GPIO controller. That way we can keep the device state at a central location. For example, when writing to a single write-only pin, we can simply look up the value we last wrote to the other pins of the same channel to figure out how we should set the other pin values.

As for access control, the only thing we did was to use separate endpoints for different channels. This way, the native capability-based access-control of seL4 carries over to the GPIO interface at channel granularity. We cannot exploit the capability rights. Recall that endpoint capabilities encompass access rights such as send and receive. However, both GPIO reads and GPIO writes require some kind of action by the GPIO component. It is therefore not possible to easily leverage endpoint access rights to access rights for GPIO operations.

Additionally, one can use badges to track which endpoint capabilities allow for what kind of access. Although we currently do not do any of this, we still briefly discuss such an approach: As explained in section 2.3.1, endpoint capabilities can have badges. Using badged endpoints, it is possible to perform access-control on pin granularity, while still retaining the possibility to set the output of multiple pins at once. There can be at most 3 badges transferred with a single send, but a channel can have up to 32 pins. We would therefore have to compress the pin access-rights into 3 badges, which would be more than enough given that a badge consists of 28 bits on 32-bit systems.[6] Alternatively, one could create predefined access-control lists and identify them by badge values.

I^2C controller An I^2C bus – or any computer bus really – is useless by itself. Similar to GPIO, it is the attached peripheral devices that give it any meaning. We should therefore not consider the I^2C driver code separately, but also how it interacts with the drivers of the individual devices on the bus.

From an isolation point of view, the ideal system would run the driver code for the I^2C controller, as well as each driver for the individual I^2C devices, in a single component, see figure 3. Each device component would use a specifically badged endpoint that restricts what kind of operations it can perform (e.g. to what kind of addresses it can talk to).

In reality however, it is not quite as simple. There are a number of drawbacks to this approach:

- A lot of context switches are performed. To perform a simple device operation from a client component, at least 4 context switches are required. Since the I^2C bus is low-speed bus, this is less of a problem for transactions transferring much data. Additionally, fastpath allows for very fast context switches. However, if the operations are short, the context switch could become a noticeable overhead, which can become problematic when low latency is required.
- There may be bus maintenance operations that need to be performed. Some operations cannot uniquely be assigned to a device (e.g. the SMBus alert signal response) and

[6] For example: If each pin requires 4 access rights (none, read, write, both), we could represent 13 pins in one badge and still have 2 bits remaining to identify the badges.

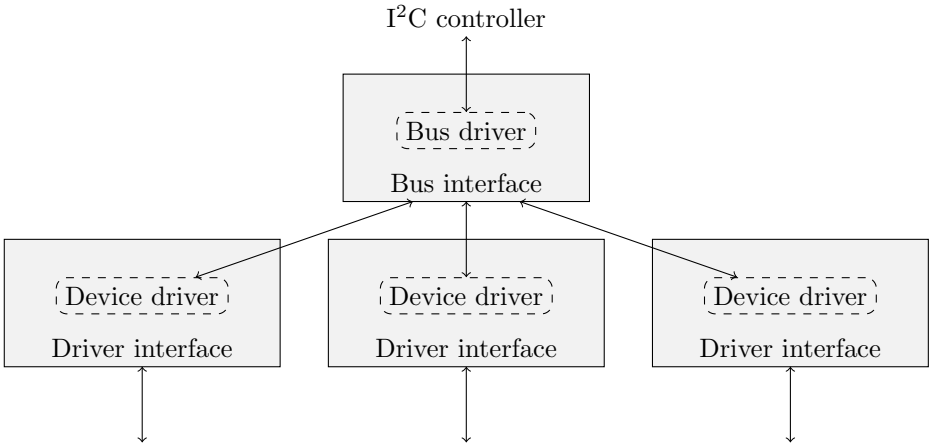

Figure 3: The ideal partition of driver components. The gray boxes each represent an individual component. The driver for the I^2C controller (bus driver) does not contain any device specific code and merely provides a software abstraction of the underlying physical bus. The drivers for the I^2C devices simply put together I^2C operations and ask the bus driver to issue them.

others may require knowledge about which devices are present on the bus (e.g. the SMBus address resolution protocol).

- The bus controller itself is often an I^2C device. It may have its own address and may be addressed by another master device without previous notice.

Some of these issues can possibly be solved through clever coordination by the driver components, without too many sacrifices. The last thing you would want to do is add a huge component that unifies the isolation and failure domain of all individual driver components.

In our case though, the story is a bit different. Since we will be replaying traces, we do not have drivers for the connected devices. In fact, depending on the granularity of the traces being replayed, we might not even be aware with which devices we are communicating. While we did create a separate component for the driver of the I^2C controller, all device dependent code is located in a central location which is currently the only client of the I^2C component. This component is introduced in section 3.2.2.

The bus interface itself is an SMBus interface, as SMBus is less general than I^2C. The longest possible SMBus protocol is 257 bytes (one direction) and there are 120 message registers available for IPC (see section 2.3.1). Message registers are 4-bytes in size, so we could only rely on message registers when transferring data to and from the I^2C component. Right now, this data – the payload of SMBus transactions – is transmitted over a shared frame that is haphazardly mapped with read and write access into the address space of both components. If proper device drivers were to be added, the interface would need to be substantially redesigned, as the current approach is only a makeshift solution.

3.2.2 Power component

The power component contains the I^2C traces and keeps track of the state of the devices in as much as it can be determined. The component is responsible for (partially) power sequencing the Enzian and exposes a very primitive high-level API that allows for basic power management.

It also runs a special thread that responds to alerts from the devices by taking appropriate measures, which for now simply means unconditionally powering down the board. This functionality will be discussed in more details in 3.3.

3.2.3 Shell

The shell is a simple component that allows a user to interact with the rest of the system and perform simple commands, such as (partially) powering up or down the main board or inspect its current state.

It is controlled via the BMC's UART and runs on the interrupt handler from the UART controller, which it reads and writes characters from/to.

3.3 Alert handler – the real-time element

The Enzian BMC does to not only need to respond to user input, but also to events from the voltage regulators. It goes without saying that the BMC should eventually take the right action. It should ideally do so promptly (worst-case execution time), since an abnormal condition might result in damage to the devices.

In our case, this primarily means responding to the SMBus alert signals: Any device that needs the attention of the host (i.e. of the BMC) can pull the alert line low. The host must then send a special alert response message on the bus. Any device pulling the alert line low will try to respond with its address. Standard I^2C arbitration procedures will allow the device with the lowest address to get its address to the host. Once an alerting device has notified the host of its address, it must release the alert line. The host can then recheck the alert line to see if there are any other devices that need its attention.

Some devices (the MAX15301 devices) will not respond with their address when the host puts the alert response message on the bus. This means we must not run the alert handler when there is no alert active, because doing so would (erroneously) register as an alert from one of the MAX15301 devices.

Another difficulty is presented by the fact that the alert line is non-functional when the devices on the bus are powered down. The BMC is in a different power domain than most of the I^2C devices. This means that when the Enzian is powered off, the BMC can therefore not rely on the alert line being functional.

This puts us in an interesting situation. On one hand we do not want to miss any alerts. On the other hand, we also do not want to run the alert handler "too often". We therefore must be extra careful how we convert this level-triggered interrupt to a message passing architecture as is common in software.

The alert lines are connected to the GPIO controller. Alerts are mainly processed by a special thread that runs in the power component. It is blocked on a notification object that it shares with the GPIO component. The processing of an alert is depicted in figure 4: When a device pulls the alert line low, the GPIO controller generates an interrupt. The kernel then runs the GPIO interrupt handler from the GPIO component which then signals the said notification. This causes the alert handler to become unblocked. Once the alert handler runs, it typically sends an alert response message on the I^2C bus to try to figure out which device caused the alert.

3.3.1 Disabling the alert handler

The first observation is that the alert handler must be disabled when the devices are down: Not only does the alert line not carry any meaningful information, it is also in a state of constant alert. Since we do not want to run the alert handler constantly in a loop when the devices are down, we need some mechanism to tell the alert handler that alerts should be ignored.

The state of the I^2C devices is managed in the power component. The alert handler also runs in the power component, so disabling the alert handler is straightforward: A variable `alert_enable` is shared by the main thread of the power component and the alert handler. It indicates whether all preconditions for proper operation of the alert handler are met. When the alert handler receives an alert notification, it will take action if and only if this variable is set. Whenever the state of any of the I^2C devices is changed (by either thread), this variable is updated accordingly.

Another thing we must be careful of is a "carryover" effect. When an alert is disabled, we have to wait for the alert pipeline to become fully empty before we can enable the alert again.

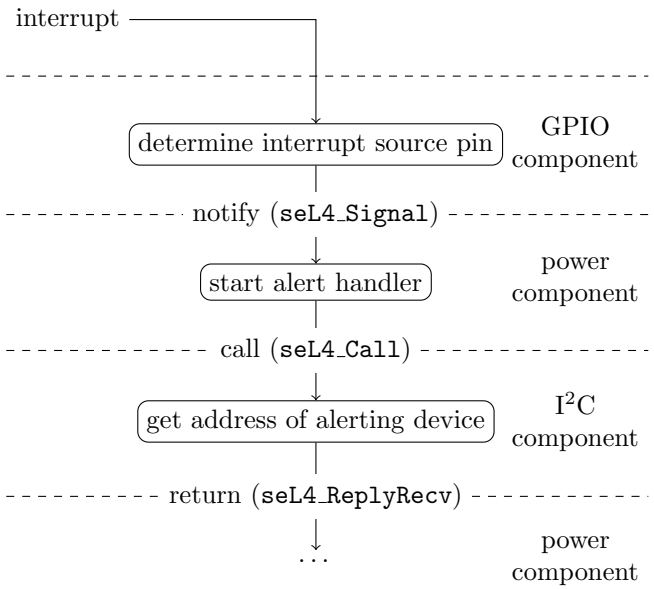

Figure 4: The alert pipeline. Depicted are the first phases of execution of the alert handling mechanism. Various components are involved and their IPC signals and messages are listed. In kernel execution is not shown.

Otherwise, we might get a spurious alert. Consider the following (somewhat) hypothetical scenario of an alert occurring just before the user is rebooting Enzian:

1. An alert occurs and an alert signal is put into the pipeline.
2. Alerts are disabled and the power supply to the main Enzian board is turned off.
3. The I^2C devices go offline and their state is reset.
4. The main Enzain board is turned on again. The I^2C devices become operational and alerts are enabled.
5. Only now, the alert signal from 1. is processed.

This may seem contrived, since at the point where alerts are enabled, it is very likely that the alert signal has passed far enough through the pipeline to be discarded. Nevertheless, it presents a race-condition that we may want to avoid. To do so, we pass a special flush signal through the entire pipeline. This flush signal causes the start of the pipeline (the GPIO component) to stop the signaling of any alerts, so that no alert signals can follow it. Once the signal has reached the end of the pipeline, we mark the pipeline as cleared.

Now that we can disable alert notifications in the GPIO component, we could in principle get rid of the `alert_enable` variable and instead just wait until the pipeline flush completes before we power off the devices. While this is certainly viable, it may be useful to be able to quickly "mask" the alert response so that (almost) immediate changes can be made to the state of the devices.

3.3.2 Enabling the alert handler

When the alert line becomes functional, we have to enable the alert handler. There are mainly two separate tasks that need to be performed:

(A) The GPIO component needs to be configured to notify the alert handler in the power component when an alert on the line becomes active.

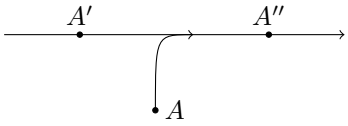

Figure 5: Manually inserting an alert into the pipeline. Suppose an alert (A) is detected when manually sampling the alert line after the alerts are enabled. We then need to determine whether we should also manually insert it into the alert pipeline or not. The alert we detected may already be handled, be in the process of doing so (A'') or it may still be waiting to be processed (A'). Deciding whether to insert an alert requires knowledge about the entire state of the pipeline.

(B) The current state of the alert line has to be sampled, since the alert line could initially be in an active state.

Let T_A and T_B be the time (A) and (B) "occur". Clearly $T_A \leq T_B$ because otherwise, we would miss any alerts during the interval $[T_B, T_A] \neq \emptyset$. If $T_A < T_B$, we are presented with an additional difficulty: Suppose that we did find an alert to be active during (B). Since we do not know if the alert line was pulled low before or after T_A, it is difficult to determine if the alert should be manually processed or not. See figure 5 for an illustration.

To solve this problem, we (atomically) perform this operation in the GPIO component: When alert notifications are enabled at the GPIO component, it also checks to see if there are any active alerts.

3.3.3 Rechecking for alerts

A similar phenomenon can be observed after an alert is handled. There may be multiple devices pulling the alert line low. This means after processing an alert from a single device, we have to again check if there are any active alerts. This is because the GPIO controller uses edge-sensitive interrupts: If the alert line does not go high after an alert has been processed, it means that there are additional devices that want to get the attention of the host. However, no interrupt will be generated for them, because the state of the alert line has not changed.

As before, we solve this problem by pushing the functionality into the GPIO component: Once the handler is done processing an alert from a device, it tells the GPIO component to recheck the alert line and reemit any active alerts. It is important that once the GPIO component emitted an alert notification, it does not emit any further alert notifications until it has been explicitly told by the alert handler to recheck the alert line. Otherwise, we would not solve the problem.

Let us also discuss how rechecking for alerts is different from enabling alerts. Although they both seem to do the same thing, their behavior is different when alerts notifications are disabled. By definition, alerts notifications should only be enabled if they were disabled. Once enabling alert notifications has completed, alert notifications will be emitted. On the contrast, rechecking for alerts may be done at any time. If alerts are disabled, rechecking for alerts should complete successfully, but neither should it enable the alerts, nor cause any notifications to be emitted: When alert notifications are disabled while the alert handler is running, no further alert notifications should be emitted upon rechecking for alerts.

3.3.4 Putting it all together

On first sight, the design we explained above might seem a bit counter-intuitive or even far-fetched. Although we tried to motivate why it is the way it is, it is not clear that it needs to be this complicated. Notice how the solution we come up with closely resembles hardware interrupts. The GPIO component functions as the "device" issuing interrupts (i.e. alert notifications) to the power component. We reinvented much of the functionality of device interrupts: We can enable alert signals (section 3.3.2), acknowledge alert signals (section 3.3.3), disable alert signals (section

Priority	Threads
203	I^2C component
202	GPIO component
201	timer (external component)
200	power component
100	shell

Table 2: Thread priorities. All threads belonging to the same component are given the same priority. (The timer is a component that uses the Triple Timer Counters (TTC) of the Zynq-7000 platform to allow for basic timeout functionality. It is maintained by Data61.) For no particular reason, all of the above threads are statically assigned to the default CPU core 0.

3.3.1) and even sort of mask an alert response (also section 3.3.1, although alerts signals will be dropped). We therefore think the solution we came up with is not all too unnatural.

In hindsight, it turned out to require more effort than expected to protect the alert handler from spurious alert notifications. A more simplistic approach would have been to embrace spurious alert notifications and instead always require the alert handler to check the state of the alert lines before taking any action. At the cost of two extra context switches in the alert pipeline,[7] we could possibly have simplified the overall design. Still, care has to be taken to properly disable the alert handler when the alert line is non-functional, so that the system does not become stuck in an infinite loop trying to handle invalid alerts.

3.4 Scheduling and Synchronization

In this section we describe how we schedule the threads of the individual components. It is important to note how the system is statically partitioned: All devices and other resources used by the power management software, are only available via the power component. If we wanted to make them directly available to other services, we would need much more sophisticated scheduling strategies. The current design would allow the second core to easily seize for itself resources that the power management software depends on, whereby either temporarily or completely preventing any power management.

3.4.1 The kernel scheduler

We decided to use the classical seL4 scheduler (non-MCS version). The main goal of the MCS scheduler is to reduce latency for time-sensitive low-criticality tasks. As of right now, we only do (partial) power management on the seL4 BMC, so there are no tasks which would benefit from preempting the power manager. One could possibly add synthetic low criticality tasks. But if they do not have any slack time in which the high criticality task (i.e. power manager) could run, the total latency induced in the synthetic tasks will just be the the running time of the power manager no matter which scheduler is used.

On the other hand, the power manger itself is probably time-sensitive: It is unclear whether there is a realistic "safety interval" during which no damage to the board is possible.

We could not easily get CapDL Loader to work with both multi-core support and MCS enabled. This is why we did not use the MCS scheduler in a classical seL4 scheduler configuration.

As for priorities, we thus only have to make sure no long-running tasks can prevent the power component from running. To do so, we set the priority of all threads that the power manager depends on high enough, see table 2.

[7] Requiring that the state of the alert lines always be (re)checked would result in the following odd situation during normal operation: After the GPIO component has identified an alert and notified the power component, the power component would immediately switch back to the GPIO component to check if the alert was really present. All this extra latency would be added just because did not bootstrap the alert handler properly.

3.4.2 Devices

The device drivers we use are not asynchronous: They constantly have to talk to the devices for a device operation to make any progress. It makes therefore little sense to reserve a device resource for a task that is not running. If multiple running tasks were to request a device operation, the device driver will simply service the requests in the order the requesting threads are queued on the endpoint (which is first come first serve).

3.4.3 Locks

There were few places where explicit synchronization was required. A notable case is the power component: The interface to the I^2C component is over a shared frame which is not thread safe. Each shared frame therefore gets a lock. We do not have locks for the individual devices: If we need to wait for a device to complete a certain task before it can accept further I^2C commands, we need to block the entire bus.[8]

There is an additional lock `alert` that the alert handler takes out whenever it runs. Whenever a thread of the power component wants to change state that affects the alert handler (e.g. mask alert responses, disable devices that the alert handler depends on, ...), it may only do while it is holding the `alert` lock.

[8]There is only one case where we knowingly have to do this. In that case, the device completes its task quickly (relative to I^2C speeds) and its status is polled anyway. The additional complexity of adding per-device locks is therefore hardly justified.

4 Implementation

4.1 Booting

seL4 already has support for the Zynq7000 SoC, so getting it to boot on the Enclustra Mercury ZX5 module was straightforward. We only to enable the System Level Control Registers (SLCR) register so that the second core could be started. Note that the serial console – in particular the MIO-EMIO signal routing of the serial controller – is already configured by U-Boot, so we were therefore immediately ready to go. Booting seL4 on the ARM platform is done in multiple stages. The boot sequence is the following:

1. The second-stage bootloader, U-Boot, fetches a bitstream from a remote TFTP repository and loads it to the BMC's FPGA. It also fetches the image file generated by the seL4 build system and loads it into main memory.

   The image file is an ELF file containing an intermediate program called "elfloader". A special section (`._archive_cpio`) of the ELF image contains a CPIO archive consisting of the kernel, the init process and (possibly) a device tree blob.[9]

2. The elfloader sets up initial page tables and enables the MMU. It starts the second core. It also loads the files from the CPIO archive: The kernel, the init process and the device tree blob (if present). The kernel and init process are again ELF binaries, hence the name.

3. The kernel configures the remaining page tables and enables the cache, interrupt controller and timer. It sets up the frame containing boot information for the init process and runs the init process.

The init process is CapDL loader which sets up the system as specified by CAmkES.

4.2 Configuring the I^{2}C devices

Since the Cortex A9 does not support the virtualization extension for ARMv7-A, running a hardware virtualized Linux is out of the question. Paravirtualization on seL4 is no longer supported.

The drivers for the I^{2}C devices that are managed by the BMC are written in Python. The power management software currently features drivers for about 7 different devices. It seemed unrealistic to write equivalent drivers on seL4 in the given time frame. Additionally, there is the costs of maintaining two up-to-date versions of the drivers.

One approach would have been to somehow compile the Python code to C code that could be executed on seL4. Unfortunately, the runtime environment on seL4 is very limited. One would have to severely mutilate any existing Python implementation to run on seL4. Even if we could produce equivalent C code, we would still have to inject our own I^{2}C backend: The Linux system calls, on which the I^{2}C Python module that we use relies on, will not work on seL4.

Instead, we chose a different, more simplistic approach: We capture all I^{2}C traffic on the Linux implementation and replay the right portion at the right time in the seL4 implementation. That way, we can – if the devices behave sufficiently deterministic – put them in the same state as the Linux drivers would put them. As the devices are fairly low-level, such an approach might turn out to "just work". There are a few places where we have to be careful however:

Accessing sensor data There might be subtle differences in analog signals. Voltage readings typically vary slightly across reboots of a device (or even without rebooting). Additionally, sensors may provide more precision than their rated accuracy.[10] In such a case, even if the hardware state remains unchanged, multiple sensor readings might return different results.

[9] The main purpose of the device tree is to pass platform information to the init process.

[10] This is common if the values are reported on a linear scale, but the accuracy is relative. If the measured value is large, the uncertainty interval may easily span multiple units.

```
1608258735.563687 read_byte_data  0x52 TX 0x99 RX 0x4d
1608258735.56487  read_byte_data  0x52 TX 0x9a RX 0x53
1608258735.565965 read_word_data  0x52 TX 0x9b RX 0x30 0x30
1608258735.586392 read_word_data  0x52 TX 0xd1 RX 0x0 0x0
1608258735.591242 write_word_data 0x52 TX 0xd1 0x0 0x20
[...]
1608258739.532292 read_block_data 0x60 TX 0x99 RX 0x2 0x49 0x52
1608258739.533628 read_block_data 0x60 TX 0x9a RX 0x1 0x48
1608258739.562065 write_byte      0x60 TX 0x3
1608258739.563755 write_byte_data 0x60 TX 0x47 0xc0
```

Figure 6: An example I^2C trace log (the fields were aligned for readability). Each line starts with a timestamp and the log must be sorted in increasing order. The second field carries little information and only specifies whether a transaction is a block protocol or not. The third field is the address of the slave device. Any remaining fields describe the data that is exchanged (in the given order).

Performing time sensitive tasks Sometimes an action on a device does not complete sufficiently quickly. In that case, there is a race-condition against other actions that depend on the result. For example, we might have to wait until a voltage stays at a certain threshold for some period of time before we can communicate with regulators that depend on it.

Non-determinism on the processor side There may be multiple agents who want to communicate with the devices on the bus. If they are not synchronized properly, the resulting state may depend on the scheduling of the agents.

There is one type of device that we decided to write a basic driver for: The two ISPPAC sequencing controllers. We needed a way to tell whether the devices functioned as desired. Since the sequencing controllers monitor the voltages of a large subset of the rails, obtaining voltage readings from the sequencing controllers seemed like the most straightforward solution.

4.2.1 Capturing traces

The Python code uses `py-smbus` form the `i2c-tools` to access the I^2C bus in user space. We wrote a module that provides the same class as the `py-smbus` module, but logs any I^2C operations to a file. By not retyping the provided class but instead manually forwarding any calls, we can ensure that we did not miss any I^2C interactions.

We keep one log file per bus. The log files are very simple and an example is given in figure 6. Note that the signature (i.e. number of bytes sent and received) almost uniquely identifies the SMBus protocol of a transaction. There is some ambiguity because block protocols have flexible length.[11] For example, the third and seventh transaction in figure 6 have the same signature, but the former uses a non-block protocol while the later uses a block-protocol.

We need a way to tell them apart, the reason being the following: For non-block protocols, the signature tells the driver when to stop accepting any further bytes. This is different for block protocols: The first byte of a block specifies the length of the block and thus the length of the transaction. Therefore, we need to know whether the signature or the first byte received should determine how many bytes we are to accept before generating a stop condition (in case they do not match).

[11] The signature is also not able to distinguish between the two types of quick-commands. Quick commands do not read or write any data. They only set the direction of the transmission (i.e. read or write) and then immediately generate a stop signal. In the trace, they are represented by just `RX` or just `TX` in the data fields so they do not require any special treatment.

4.2.2 Obtaining high-level context

The I^2C traces are by themselves little helpful, because we need some way to determine what they do on a higher level. Recall that we only want to configure a specific subset of devices. While replaying all transactions for a given device address might be produce the desired effect, we decided to take a different approach:

We partition each bus trace into different segments that can be replayed individually. This means that extra lines which are not part of any segment will not affect what is being replayed. The main advantage is that the granularity of the sections can be arbitrarily reduced until there is enough context information to identify what a particular transaction does. This should make pinpoint problematic transactions easier.

The way we do this is by maintaining a separate file containing timestamps when a section starts and ends. All timestamps (segment and transaction) do not have to be accurate, as long as it is clear which transaction belongs to which segment. To enforce this, we just wait long enough at segment boundaries.

4.2.3 Replaying traces

We generate the C code directly from the trace log and the segment file. For each segment s, we create a function `int s(void);` that replays s. These functions also check if the received data match what is recorded in the trace log. In case the received data does not match or an error occurs during transmission, they return non-zero and we abort the power up procedure.

For now, the transactions are simply hard-coded into the source code. There is no data structure that is parsed at run time, so even transaction payload data is directly embedded into the `.text` section of the executable of the power component. We do not incorporate the delays between the individual I^2C transactions, because we did not need any delay for the most part.

There were were some difficulties with the I^2C controller. It does not seem to support all types of SMBus protocols. More specifically, it does not support:

- The quick-command in read mode.
- Reading blocks of sizes 0 or 1 (not including the size byte). The I^2C controller is a bus master. Therefore, to terminate the transmission when it is reading form a slave, it should not-acknowledge the last byte and then generate a stop signal. However, it seems that the controller needs to be configured to not-acknowledge a byte *before* it is received. Additionally, reading a byte from the bus will immediately start the transfer of another byte. Hence, if the block length turns out to be 0 or 1, we cannot not-acknowledge in time and thus not generate the stop signal in time.[12]

While we do not use quick-command, we frequently read blocks of size 1 (such as the seventh transaction in figure 6). To circumvent this problem, we use an SMBus protocol with the same signature instead and pretend that it was never a block transaction. This has the following drawbacks:

- There might not be an non-block SMBus protocol with the given signature. In our case, it turns out that such an equivalent non-block protocol always exists.
- If the device does not reply with the expected block length, we violate the SMBus specifications. This should not be a problem: We will anyway power down the main Enzian board (which resets all affected devices) if we get a non-matching response.

[12] One could possibly try to generate a stop signal without not-acknowledging the last byte. However, as seen in the example in section 4.3.2, the controller must first not-acknowledge a received byte before it can generate a stop signal. Failing to not-acknowledge the last byte would also violate the SMBus specifications.

4.3 Discussion

In this section we present our experiences while building the BMC software. We also give some observations that were made during this process.

In terms of ease of programming, CAmkES is a great tool for configuring the BMC software. It hides the entire user space bootstrapping complexity and provides a programming model tailored for seL4. For better or worse, it also hides much of the seL4 API. Although this functionality is available to the user by what is referred to as "custom CAmkES connectors", it requires deeper understanding of the inner workings of CAmkES. Unfortunately, the process of writing such custom connectors is not documented, which is why we only made very limited use of them. Looking back, we should have used custom connectors more frequently. Generally speaking, the documentation on CAmkES was otherwise sufficient and the plethora of examples (some of which even involving custom connectors) were helpful.

We never had any problems with the static architecture assumption in our very constraint setting. Given the announcement of the seL4 core platform this assumption will likely be the way forward for some more time longer.

The seL4 kernel and its user land API were well documented. Its core concepts are simple and easily understood. This might be due to the more top-down design approach, but possibly also due to personal bias. The kernel build system is complicated[13] and the configurations options are not well documented.

Some code is auto-generated at build time. While it can make it more difficult to locate a piece of code, it typically also means that the generated code is more specific. For example, the file `kernel_all.c`, containing most of the kernel code, is very useful as a reference.

4.3.1 Hardware and FPGA

There is another side to a system beside software. Hardware is sometimes trusted axiomatically. Although hardware components are usually tested at the manufacturer for manufacturing defects, there can also be design errors: For example, the errata of the Cortex-A9 r0 that we are using for the BMC lists just over 60 errata, although some of which may not apply [14]. Six are rated critical, ranging from data coherency failure to complete core deadlocks.

FPGAs can possibly alleviate this issue: The underlying hardware is much simpler since a lot more functionality is now performed in software. On the flip side, this introduces a lot more complexity. Not only can it be difficult to properly configure such highly customizable hardware, it can also be difficult to debug and test. While we did not have major issues with buggy hardware, we configured the BMCs PL incorrectly multiple times.

4.3.2 Current driver programming practices are error prone

Correct programming of devices code plays a vital role in the correct behavior of any system interacting with the physical world. Device programming usually involves processing the device documentation and synthesizing code. Often manufacturers provide driver code, but this code may not be flexible enough, in which case the provided code cannot be used. We identify the following causes of inflexibility:

- Device driver interfaces are not unique. In fact, in current practices of device programming, the operating systems dictates the driver interface. However, different operating systems may have different driver interfaces. As a result, an operating system becomes compatible with numerous device drivers, but a device driver is only ever compatible with one type of operating system. Porting the device driver to a different driver interface is often non-trivial.
- It is difficult to export the low-level hardware interface of the device on a higher-level driver interface without making any additional assumptions. If these assumptions do not apply,

[13] CAmkES exposes its own build macros, which simplifies the configuration process considerably.

the provided driver cannot be used or needs to be modified. For example, a driver may not expose certain kinds of interrupts, even though the device fully supports them.

In light of this, it is not surprising that the device programming model is on a very low level. It is usually addressed at a human reader, who is expected to understand the top-down requirements from the operating system and make ends meet with the hardware interface provided by the device. Depending on the quality of the documentation and the complexity of the device, producing a driver that largely works, may already be an achievement by itself.

On the other hand, access to the low-level hardware interface no longer requires trust in software created by the hardware vendor. In theory, it would allow for the verification boundary to be pushed through the device driver all the way down to the hardware-software boundary. One could possibly aim to prove that software programs the device correctly.

However, such an endeavor requires a formal model of the device. It is difficult to extract a rigorous model from informal documentation targeted at human readers. Such documentation will almost always be incomplete and there is no guarantee that my understanding of the documentation is correct or that the informal documentation even correctly reflects what the hardware is doing. Alternatively, the hardware vendor could release such a formal model. In either case, there does not seem to easy way to verify correctness of the obtained model – and hence of any driver code.

An example To better illustrate the problem of incomplete (or even contradicting) documentation, we give the example of the generally well documented Xilinx intellectual property (IP) core 'AXI IIC Bus Interface'. The documentation is freely available on the Xilinx website [34].

The IP core is a I^2C bus master (and slave). On the processing system, it can be controlled by simple memory-mapped IO. The IP core maintains two FIFO queues for reading and writing data to the bus. It also has a read FIFO limit register, which causes the controller to throttle the bus when the occupancy of read FIFO equals the limit register.

Consider a following scenario: The IP core has been reading from a slave device. After receiving the last byte, it will set the not-acknowledge bit to signal to the slave device that it should stop sending data. Then it will stop the transaction and free the bus. The programming sequence instructs us to do the following:

1. Set the FIFO limit such that a throttle condition will occur *before* the last byte is received.
2. Wait for the throttle to occur.
3. Set a flag in the control register that causes the IP core to set the not-acknowledge bit on the next received byte.
4. Set the limit the FIFO to only one byte and read the FIFO until it becomes empty.
5. Wait for the last byte to cause throttle.
6. Read the last byte and set a flag in the control register to terminate the transmission.

However, it is not specified what happens if we set the FIFO limit to a value lower than its occupancy. The manual clearly states:

> Receive throttling is done when the `RX_FIFO_OCY` matches the value set in the `RX_FIFO_PIRQ`. The throttle condition is momentarily removed when a byte from the receive FIFO is read, thus allowing the transmitter to send the next byte.

Here, `RX_FIFO_OCY` is the read FIFO occupancy and `RX_FIFO_PIRQ` is the read FIFO limit. Nowhere in the document the case `RX_FIFO_PIRQ` $<$ `RX_FIFO_OCY` is discussed. The receive throttle condition is consistently referred to as the state when the two values are equal. So technically, the above programming sequence produces undefined behavior. Moreover, if we read the quote again, we see that the second sentence *exactly* describes what we want to do after step (3): read a single byte.

Incidentally, this way of reading a single byte does not require us to set the FIFO limit to a value lower than its occupancy. So, the behavior when `RX_FIFO_PIRQ` < `RX_FIFO_OCY` might very well be undefined.

We considered three different approaches to solve this ambiguity:

- Guess and apply common sense. As humans with limited processing power, we do this all the time, often subconsciously.
- Experimenting with actual hardware and observing its behavior. Unfortunately, we have no way of knowing whether the behavior is implementation specific and possibly different in other/future versions. This is especially true for devices which themselves are a piece of software running on an FPGA.
- Consult driver code provided by the manufacturer. At this point we are back where we started: relying on the correctness of the manufacturer's device driver.

4.3.3 Asynchrony is difficult

It has become apparent that asynchronous behavior can be difficult to process correctly. It is no secret that multi-threaded programming is not trivial. Likewise, it should not be surprising that asynchronous behavior can be a common source of bugs.

At multiple occasions we encountered (potential) bugs that only occurred when there was a certain interleaving of (asynchronous) events. Such bugs are generally hard to reproduce and debug. If such bugs are present in device drivers, they can be hard to detect. Additionally, the design of a device may promote unsafe programming practices or even make the problem unsolvable. In many cases the solution may not be straightforward: We struggled quite a bit to get the alert handler to the state as described in section 3.3. It is also not clear if we can safely end the struggle there.

An example To better illustrate our point, we give an example of a device that (potentially) tempts the programmer to write code that results in a software bug due to a race condition. The example in question is the Linux device driver for GPIO controller that we use. The device driver is provided by Xilinx and the documentation of the IP core is freely available on the Xilinx website [33]. Here, we focus on the second-level interrupt handler.

The GPIO controller has a global interrupt enable register. It also has a channel interrupt enable register and channel interrupt status register, which allows the individual channels to raise interrupts separately. It does however, not have a status register for the individual pins. If interrupts are enabled, a channel will issue an interrupt when one of its (input) pins change.

There are two difficulties with this kind of hardware. The first one being the obvious requirement to keep shadow copies of the values of the input pins, so that the interrupt handler can figure out which pins values have changed.[14] However, by the time the second-level interrupt handler runs, the value of the pin that caused the interrupt, might already have changed back to its original state, making it impossible to detect that there was any change at all. The hardware designers must have assumed that the second-level interrupt handler will always run before the value of a pin can change again.[15]

The second difficulty is more subtle. You would normally clear an interrupt status only once the corresponding condition has been processed. Indeed, in interrupt handler of the driver code provided by Xilinx, we find the following:

[14] The original device driver from Xilinx does not do this. It treats all pins of a channel as cause for the interrupt. This results in many spuriously triggered edge events whenever the interrupt handler is called. These have to be filtered out manually.

[15] This is true for our purposes. Currently, interrupts are only used for the SMBus alert lines. We need to explicitly act on the alerts raised by the devices before they let go of the alert line.

```
1  val = xgpio_readreg(mm_gc->regs + chip->offset);
2  /* Only rising edge is supported */
3  val &= chip->irq_enable;
4
5  for_each_set_bit(offset, &val, chip->mmchip.gc.ngpio) {
6          generic_handle_irq(chip->irq_base + offset);
7  }
8
9  xgpio_writereg(mm_gc->regs + XGPIO_IPISR_OFFSET,
10                 chip->offset / XGPIO_CHANNEL_OFFSET + 1);
```

We see that first, the current pin values are read into `val` (line 1). Then, for each pin with value 1 and `irq_enable` set to 1, we simulate an IRQ (line 6). Finally, we clear the interrupt status of the current channel (line 9-10).

Consider the following sequence of events:

1. The value of GPIO pin A changes, causing the GPIO controller to issue an interrupt for the current channel.
2. The interrupt handler runs and reads all input values of the current channel (line 1).
3. The value of another GPIO pin B of the current channel changes, but the GPIO controller cannot issue another interrupt at this time.
4. The interrupt handler finishes execution and clears the channel interrupt status (line 9-10) and acknowledges the device interrupt.

Note how the value change for B essentially goes undetected: The GPIO controller does not maintain any per-pin interrupt status, so it cannot tell that we wanted to acknowledge the interrupt from A but not the one from B. It will thus not re-raise an interrupt once 4 is complete until a further transition on the input values occurs. This is clearly not what we want.

5 Evaluation

We compare the resulting artifact to the existing Linux implementation by comparing the performance of the two alert handlers: We measure both response time as well as total run time under different kinds of synthetic load. We then briefly discuss our experience with the I^2C trace replaying mechanisms.

5.1 Scope

Alerts from the I^2C devices generally mean that the main Enzian board is in an abnormal state. It could be something serious, such a short-circuit triggering an overcurrent protection. In such a case, we want to respond quickly. Some devices implement a closed feedback loop allowing them to recover themselves. Still, devices that implement a closed feedback loop but perform an externally visible response to faults (such as shutting down on over-temperature) also need to notify our software "quickly", as they might end up violating the power sequencing requirements.

We therefore decided to measure the amount of time the BMC software requires to "process" an alert from a I^2C device and bring the board back to a safe state. We also experimentally determine the robustness of the alert handler against interference from less critical tasks. Since the BMC has two CPU cores, there is going to be some contention on the shared resources, even if one CPU core is completely dedicated for the alert handler.

5.2 Method

The alert handler on both systems is relatively simple: When an alert is raised, they put an alert response on the corresponding bus to obtain the device address of an alerting device. If a device replies with its address (which might not be the case), the handler reads certain device registers from the device to determine the type of fault raised by the device. If the fault is deemed critical or if no device replies with its address, the alert handler turns off the devices one by one and finally disables the power supply unit (PSU).[16]

By *response interval*, we mean the time interval from when an alert is triggered until a SMBus alert response is issued on the bus (more specifically, until the data line is pulled low, marking the I^2C start condition). The *response time* is the length of said interval. We refer to the amount of time elapsed between the alert line going low and the PSU being turned off, as the *run time*. The response time and run time act as lower and upper bounds for the amount of time the main Enzian board spends in an abnormal state (assuming that the devices trigger the alert immediately).

To measure the response time and run time, we need to access the alert line, the I^2C data line, and the PSU enable signal. The later has a test point on the Enzian board, whereas the regular I^2C data line and alert line do not. We therefore decided to perform the tests on another I^2C bus that carries no devices but is accessible over a connector on the board. This bus also features an alert line. We set the oscilloscope to manually signal an alert and measure the time until we see a change in the data line (response time) or in the PSU enable signal (run time).

5.2.1 Synthetic workloads

We measure the response time and run time under the following synthetic loads:

Idle The operating system is mostly idling and no additional user space load is started.

Stream Each task repeatedly copies 1 MiB of data from one buffer to another. The copy is done in word-sized chunks and we start two such tasks for each core, so that the system becomes overcommitted.

[16] In the Linux implementation there were some additional actions on devices that we did not initialize seL4. We disabled these actions to match the behavior on seL4.

Pingpong A pair of tasks repeatedly pass each other the control flow: Each task first unblocks the other task and then immediately blocks until it becomes unblocked by the other task. We again start two pairs of tasks for each core.

The way we do this on both systems differ. On seL4 we do this by signaling and waiting on notifications. There does not seem to be a direct equivalent of notifications for Linux, so we decided to send one byte of data back and forth using Unix' native IPC mechanisms: (named) pipes.

While we run the same number of workloads, the partition to the two cores is static on seL4 but not so on Linux.

It is important to note that the scheduling strategies on the two systems differ. Recall that seL4 uses a strict priority-based scheduling algorithm. This means that as soon as the alert handler becomes ready to run, the low-priority workloads on core 0 will be interrupted and only continue once the alert handler has completed.

On Linux however, the situation is different: We use the standard CFS scheduler, which will cause the alert handler to be preempted if the scheduler deems other tasks worthy of the CPU's execution time.[17] This means we have to be careful when interpreting any results, especially when the system is overcommitted.

5.2.2 Estimating the clock speeds

As I^2C is relatively low-speed bus, the time the bus is active during the alert handler is a non-negligible fraction. To allow for a fair comparison, it is therefore important that the clock speeds of both systems is kept as similar as possible.

The seL4 implementation uses a I^2C IP core, whose clock speeds can be adjusted during compile time. Additional timing parameters may be set at run time.

The Linux implementation uses the i2c-gpio driver, which is a bit-banging driver. The clock speed is set in the device-tree via the variable `i2c-gpio,delay-us`, which is an integer. We use a default value of 5. Unfortunately, it is not clear what kind of delay this variable specifies. In the documentation, this setting is described as "delay between GPIO operations (may depend on each platform)". In the provided example, the delay is set to 2 with a comment describing it as "$\sim$ 100 kHz". A quick glance at the code then however, reveals the following (where `pdata->udelay` is the delay value from the device tree):

```
if (pdata->udelay)
        bit_data->udelay = pdata->udelay;
else if (pdata->scl_is_output_only)
        bit_data->udelay = 50;                  /* 10 kHz */
else
        bit_data->udelay = 5;                   /* 100 kHz */
```

It seems that this value is the number of μs to wait in half of a cycle. (Further investigation confirms this fact: During the transmission of each bit, there are delay statements of $\lfloor \frac{\texttt{bit\_data->udelay}}{2} \rfloor$, $\lfloor \frac{\texttt{bit\_data->udelay}+1}{2} \rfloor$ and `bit_data->udelay`, summing up to a total of two times `bit_data->udelay`)

Note that also code execution takes some time, so the delay statements are not the only factor contributing to the length of a clock cycle. Due to the large variety of observed clock speeds on the different platforms (as hinted by the comments), the processing overhead might contribute to a non-negligible amount. We therefore decided to use the oscilloscope and measure the clock frequency manually.

The average clock frequency was close to 67 kHz, but there seems to some variation in the clock speeds (which might be due to the coarse time resolution in the CSV exported by the oscilloscope,

[17] The reason for sticking to the default scheduler is the following: The Linux kernel has a large variety of scheduler implementations and configuration options. There is likely not just one "ideal" configuration, so it is not clear against what we should compare to. Additionally, optimizing the Linux scheduler for our use case was not the principal focus.

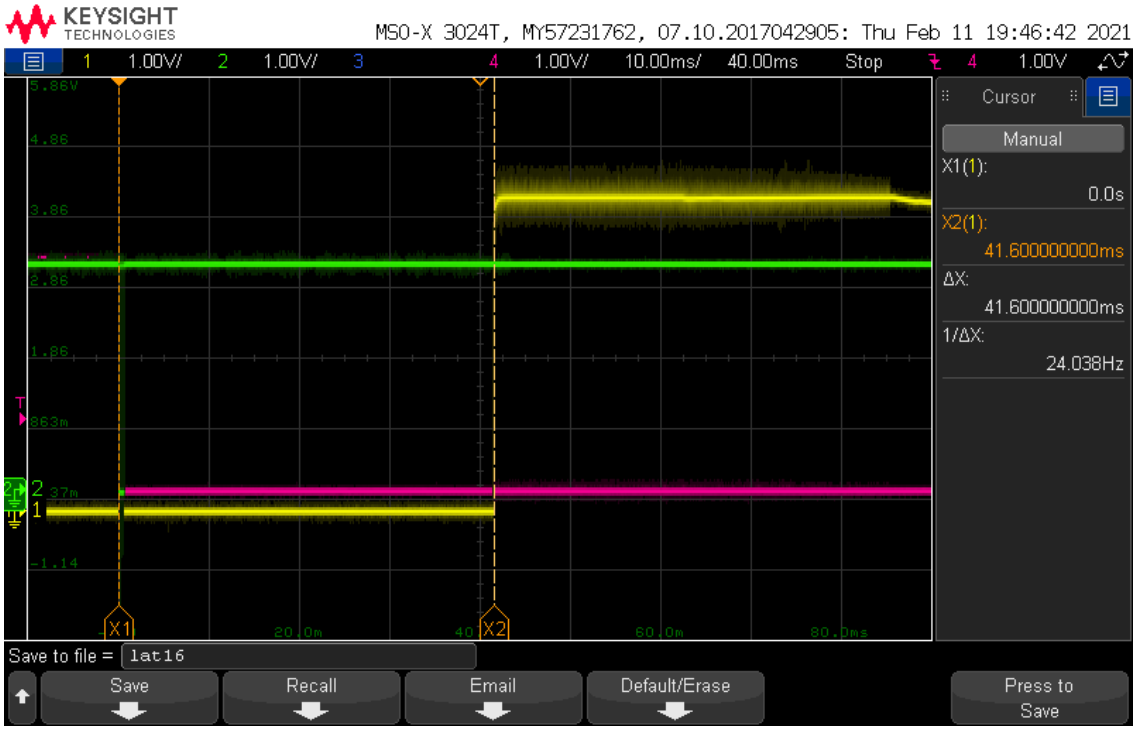

Figure 7: Oscilloscope reading for the run time on seL4 when idle. The pink line represents the voltage on the alert line which is pulled low by the oscilloscope at `X1` = 0 ms. The green line indicates the voltage of I^2C data line (the alert response is barely visible just after `X1`). The yellow line represents the PSU enable signal. The PSU is disabled at `X2` $\approx$ 41.6 ms.

when measuring over a relatively long time interval). The clock speeds of the I^2C IP core where thus set to 67 kHz.

5.3 Results

We manually read off the time intervals from the oscilloscope display. An example of this display is shown in figure 7. Due to the orders of magnitude difference in response time and run time, we measured each of them in separate runs. We generally tried to take 24 measurements. If after 12 measurements, the variation in the data points deemed to be smaller than the precision of the readings on the oscilloscope, we decided to not show any error bars and did not complete the remaining 12 measurements. For example, in figure 7, placing the marker `X2` one pixel to the left would result in a time interval of 41.4 ms. The first 12 readings were all either 41.4 ms or 41.6 ms, so any error estimates would be largely influenced by the exact placing of the marker.

The results are shown in figure 8. The alert handler implementation on Linux is always slower than the seL4 implementation: The response time is about 100x longer on Linux. On the other hand, the run time is "only" 5x longer on Linux than on seL4 (when idle).

The effect of the workload on the response time is similar for Linux and seL4 (although the orders of magnitude of the observed values differ): Running the pingpong workload increases the response time by a factor of about 1.35x. The stream workload more heavily affects the response times on seL4 with a slowdown of 2.75x, whereas the slowdown on Linux is about 2.3x.

On seL4, we observed almost no variation in run time across the different test cases.

The measurements were generally stable, with standard deviations around 5% or less. A notable exception is the response time on seL4 under the stream workload: Although response times were usually around 30 μs, some outliers as large as 40 μs or more were observed.

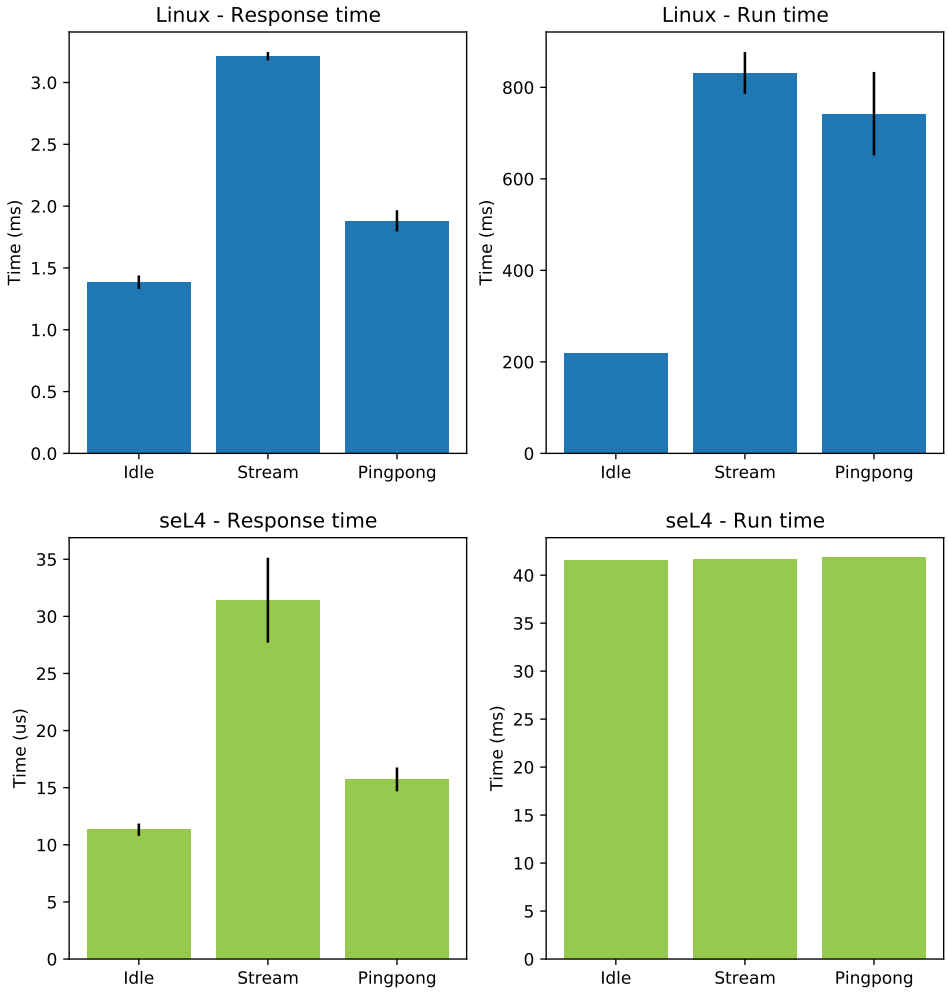

Figure 8: Alert handler response and run time. The ranges on the y-axis differ among the individual plots. Most notably, the response time on seL4 is given in units of microseconds. The mean value of the samples is shown. The error bars represent the (uncorrected) sample standard deviation (where applicable).

5.4 Discussion

We suspect that there multiple causes for the slowdown on Linux compared to seL4, although the exact contribution could not be determined:

Python Being a high-level interpreted language, only a limited amount of context information is available when a script is first invoked, leading to overhead at runtime. The power manager makes extensive use of Python.

D-Bus D-Bus is a message passing infrastructure for Linux. Multiple processes connect to a dbus daemon over which they exchange messages [32]. D-Bus is known to be slow, both in throughput and latency [29].

Additional reasons for the slowdown experienced on Linux possibly include less compile-time optimization and increased functionality (resulting in more complex branching and larger memory footprint and less locality).

5.4.1 Response time

Because the stream workload (which is very memory intensive) incurred the most slowdown, it is safe to say that the leading cause for the variance across workloads is contention either in the L2 cache or memory bandwidth. Even though seL4 uses a big kernel lock, we cannot rule out that the cause of slowdown for the pingpong workload is also due these effects.

The similar effect of the workload on the response times on Linux and seL4 may hint at the following observation: The response time on Linux was likely short enough so that the scheduler always decided to follow the control flow of the alert handler, even if there were other tasks waiting to be scheduled. This is interesting, because the control flow does 2 context switches and executes over a relatively long period of time (up to 3 ms or more). It may also imply that if the alert handler is kept short enough, it may not be affected by background tasks more than seL4, which uses a strict priority-based scheduler. The same cannot be said about the total run time.

The Cortex-A9 runs at 666 MHz, so the mean response time on seL4 when idle is 7550 cycles. We think this is acceptable, considering that it does 2 regular context switches and 1 fastpath context switch.

It is difficult to determine the cause for the large variance in the stream workload for seL4. It could be due to cache effects, row conflicts in DRAM, a spurious interrupt or a measurement error.

5.4.2 Run time

The two principal contributors to the run time are the processing overhead and the I^2C bus operations. To put them into context, we will first roughly estimate the amount of time the alert handler spends on the I^2C bus. To do that, we recall that the bus is clocked at around 67 kHz. Each transferred byte consists of 9 bits, when including the acknowledgement bit. For each transaction, there is at least a start and a stop signal, which we consider as one additional cycle each. In our case, the alert handler transfers in total 284 bytes using 84 transactions. This results in 2724 cycles of I^2C operation – approximately 40 ms. According to these calculations, more than 95% of the the run time is spent on the I^2C bus on seL4. For Linux, the I^2C bus therefore contributes about 15% of the run time and the amount of time spent executing on the CPU is about 180 ms.

On seL4, we see almost no variation in run time across the different test cases. This means that the alert handler running on core 0 is sufficiently isolated from whatever is running on the second core. This should not be surprising, since only a small fraction of the run time is dependent on the CPU performance of core 0. The amount of slowdown observed in the CPU processing for the different workloads is difficult to quantify: Because the uncertainties in the measurement and in the above approximation are significant compared to the CPU processing time, any slowdown estimates would not lead to meaningful values.

As for Linux, we only learn that we must be careful to properly configure the scheduler when running background tasks.

5.4.3 Speeding up the alert handler

Reducing the processing overhead is one way to improve the alert handler run time. However, this only makes sense until the duration of the bus operations become the dominating factor. The bus operation by themselves currently take about 40 ms, as estimated above. We would ideally like the fault condition to persist for at most 1 ms. Although the run time merely represents an upper bound, it is currently still far from ideal.

An obvious way to speed up the alert handler is to clock the bus at a higher frequency. 67 kHz is rather slow compared to the standard frequency of 100 kHz or the widely supported fast-mode of 400 kHz. In fact, most I^2C interaction of the alert handler is over a bus solely dedicated to the two ISPPAC sequencing controllers. Both support up to 400 kHz operation. This would reduce the duration of bus operations by a factor of 4x to 6x. However, due to processing overhead, the GPIO bit-banging driver on Linux might not be able to produce clock speed larger than 200 kHz, even if the delay value in the device tree is set to a very small value, such as 1 or even 0.

Another approach is to reduce the amount of data that is transmitted over of the I^2C bus. 80 of the 84 I^2C transaction are used to bring output signals of the sequencing controllers low. Like the GPIO IP core, these output signals are arranged into multiple batches that are processed at once. In particular, all values of a single batch have to be set at once.

By keeping shadow registers of the values of the output signals, we would not have to read them from the device. This would reduce the contribution from the bus interaction by about 55%, as the number of I^2C operation as well as the amount of data transmitted is approximately reduced by a factor of 2. It may further be possible to set multiple output values at once, provided that the power sequencing requirements are met.

Additionally, one could take different actions depending on the state of the board. In that case, we would not try to disable devices that are already off, possibly reducing both bus activity and processing time. However, it is unclear if this does not add more processing overhead if all devices are running.

Finally, one could take very drastic measures and not engage in any I^2C operations beyond obtaining alert status from the devices. Instead, we could immediately turn off the PSU. Although this will violate power sequencing requirement and may not be the safest thing to do, it is similar to what happens on an external power loss.

5.5 I^2C traces

The subtree of the power-tree that we focused on was carefully chosen, so that we would not run into the difficulties explained in section 4.2: Since we wrote very basic drivers for the ISPPAC sequencing controllers, we could obtain sensor readings and perform time dependent tasks on these devices.

It is therefore little surprising that all devices behaved exactly as the traces expected them to. Occasionally, some devices did not acknowledge the data that we sent them. One case was due to invalid configuration, which we had not realized by then. All other cases were related to setting the output voltages of the MAX15301 devices. They were timing issues as we would not observe the not-acknowledgments in the Linux implementation. They were easily fixed by adding two delays of 10 ms each.

It took 878 lines of Python code to capture relevant I^2C traces and convert them to C. 2135 I^2C bus transactions are replayed, transferring a total of 6543 bytes (including the address bytes).[18] This resulted in 19793 lines of almost completely linear C code.

[18] A large portion consists of the configuration for the clock generators which was generated by a tool of the device manufacturer. The amount of "interactive" communication this therefore only a small fraction of the trace.

6 Conclusion

Traditional baseboard management controller software does not provide sufficient assurance, despite their high degree of power over hardware devices. To address this problem, we ported seL4 to the Enzian BMC and implemented partial power management on top of CAmkES. This additionally allowed us to approach BMC software from another angle which offered insights on how to build reliable BMC software. We observed good system performance on the real-time subset under different synthetic workloads. Although the resulting temporal isolation is not ideal it was sufficient for our purposes.

Because we are not using a verified configuration for seL4, we do not get the refinement from the abstract model to the code (or binary executable). Moreover, at the time of writing, the Zynq 7000 platform has not been verified, so a natural continuation of this work could include the extension of the proofs to the Enzian BMC hardware.

Although the properties of the abstract model provide a high-degree of assurance the verification history of seL4 only covers the kernel. While it makes up the core part of the OS, it is only part of the software that runs on the BMC. Verification of user space components is feasible because the kernel is formally specified. Currently, our power management follows a best-effort approach which does not provide very high levels of guarantees. Future work might aim at verification of user space components for a more complete assurance coverage.

We argued that driver code can be difficult to reason about and therefore needs extensive testing to provide any kind of assurance. Same is not necessarily true for FPGAs: Formal verification has been applied to digital circuits (and in particular to FPGAs) [24]. It might be possible to connect these two ends for a fully verified BMC software stack.

If other BMC features – such as data collection and logging – are implemented, we should switch to the MCS scheduler. It is likely non-trivial to configure and might require sophisticated allocation mechanisms to prevent external fragmentation under light to heavy load.

References

[1] *A Case for a Trustworthy BMC*. Cloud Security Industry Summit (CSIS). URL: https://cloudsecurityindustrysummit.s3.us-east-2.amazonaws.com/a-case-for-a-trustworthy-bmc.pdf (visited on 02/22/2021).

[2] Bernard Blackham et al. "Timing analysis of a protected operating system kernel". In: *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE. 2011, pp. 339–348.

[3] Anthony Bonkoski, Russ Bielawski, and J Alex Halderman. "Illuminating the security issues surrounding lights-out server management". In: *7th {USENIX} Workshop on Offensive Technologies ({WOOT} 13)*. 2013.

[4] *CVE-2019-6260*. Available from MITRE, CVE-ID CVE-2019-6260. 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6260 (visited on 03/05/2021).

[5] *CVE-2020-14156*. Available from MITRE, CVE-ID CVE-2020-14156. 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14156 (visited on 03/05/2021).

[6] Dan Farmer. "IPMI: Freight train to hell or Linda Wu & The night of the leeches". In: (2013).

[7] Gernot Heiser. *How to (and how not to) use seL4 IPC*. 2019. URL: https://microkerneldude.wordpress.com/2019/03/07/how-to-and-how-not-to-use-sel4-ipc/ (visited on 09/25/2020).

[8] *HPE Integrated Lights-Out 4 (iLO 4) - Overview*. Hewlett Packard Enterprise. URL: https://support.hpe.com/hpesc/public/docDisplay?docLocale=en_US&docId=c03255140 (visited on 02/21/2021).

[9] Intel et al. *IPMI. Intelligent Platform Management Interface Specification Second Generation*. Version 2.0. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf.

[10] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[11] Gerwin Klein et al. "Comprehensive formal verification of an OS microkernel". In: *ACM Transactions on Computer Systems (TOCS)* 32.1 (2014), pp. 1–70.

[12] Ihor Kuz et al. "CAmkES: A component model for secure microkernel-based embedded systems". In: *Journal of Systems and Software* 80.5 (2007), pp. 687–699.

[13] Ihor Kuz et al. "capDL: A language for describing capability-based systems". In: *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. 2010, pp. 31–36.

[14] ARM Limited. *ARM Cortex -A9 processors. Software Developers Errata Notice*. Version r0 releases. Oct. 5, 2016. URL: https://developer.arm.com/documentation/uan0005/d.

[15] Anna Lyons et al. "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time". In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–16.

[16] Toby Murray et al. "seL4: from general purpose to a proof of information flow enforcement". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 415–429.

[17] *OCC Overview*. URL: https://github.com/open-power/docs/blob/master/occ/OCC_overview.md (visited on 02/22/2021).

[18] *OpenBMC*. URL: https://github.com/openbmc/openbmc (visited on 02/21/2021).

[19] Joseph Reynolds. *daemons should not run as root #3383*. URL: https://github.com/openbmc/openbmc/issues/3383 (visited on 02/22/2021).

[20] Jasmin Schult. "A model-based approach to platform-level power and clock management". Bachelor's Thesis. ETH Zurich, 2020.

[21] NXP Semiconductors. *I²C-bus specification and user manual*. Version 6. Apr. 4, 2014. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[22] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. "Translation validation for a verified OS kernel". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 471–482.

[23] Thomas Sewell et al. "seL4 enforces integrity". In: *International Conference on Interactive Theorem Proving*. Springer. 2011, pp. 325–340.

[24] Satnam Singh and Carl Johan Lillieroth. "Formal verification of reconfigurable cores". In: *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00375)*. IEEE. 1999, pp. 25–32.

[25] Sami Sirhan and Sureena Gupta. "Power-supply sequencing for FPGAs". In: *Analog Applications Journal* (2014).

[26] Mark Stanovich et al. "Defects of the POSIX sporadic server and how to correct them". In: *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2010, pp. 35–45.

[27] *Supermicro Intelligent Management*. Super Micro Computer, Inc. URL: https://www.supermicro.com/en/solutions/management-software/bmc-resources (visited on 02/21/2021).

[28] Inc. System Management Interface Forum. *System Management Bus (SMBus) Specification*. Version 3.1. Mar. 19, 2018. URL: http://smbus.org/specs/SMBus_3_1_20180319.pdf.

[29] Will Thompson. *The Slothful Ways of D-Bus*. Aug. 7, 2011. URL: https://desktopsummit.org/sites/www.desktopsummit.org/files/will-thompson-dbus-performance.pdf (visited on 03/03/2020).

[30] Data61 Trustworthy Systems Team. *seL4 Reference Manual*. Version 11.0.0. Nov. 20, 2019. URL: https://docs.sel4.systems/releases/sel4/11.0.0.html.

[31] *u-bmc*. URL: https://github.com/u-root/u-bmc (visited on 02/22/2021).

[32] *What is D-Bus?* URL: https://www.freedesktop.org/wiki/Software/dbus/ (visited on 03/03/2020).

[33] Xilinx. *AXI GPIO. LogiCORE IP Product Guide*. Version 2.0. Oct. 5, 2016. URL: https://www.xilinx.com/products/intellectual-property/axi_gpio.html.

[34] Xilinx. *AXI IIC Bus Interface. LogiCORE IP Product Guide*. Version 2.0. Oct. 5, 2016. URL: https://www.xilinx.com/products/intellectual-property/axi_iic.html.

Acknowledgments

I would like to thank the Systems Group of ETH Zurich for the opportunity to write my thesis on such an interesting project. The Enzian Project has offered me much insight into modern computer systems and will likely remain as a reference example in my memory for many years to come.

I would like to thank the Enzian team in particular, for their help and suggestions, making this thesis possible in spite of the difficult working environment at the moment. Special thanks also go to my supervisors for their valuable feedback and for always pointing me into the right direction whenever I got stuck.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Towards high-assurance Board Management Controller software

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Heimhofer	Cedric

With my signature I confirm that

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

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

Place, date	**Signature(s)**
Hermetschwil-Staffeln, 2021-03-07	Cédric Heimhofer

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