Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor’s Thesis Nr. 353b

Systems Group, Department of Computer Science, ETH Zurich

Characterization of Interrupt Handling in Board Management Controllers

by

Tobias Oberdörfer

Supervised by

Daniel Schwyn
Dr. Michael Giardino
Prof. Dr. Timothy Roscoe

March 2021 – September 2021

Abstract

Outside the Systems Group at ETH Zurich, there are not many research possibilities in regards to Board Management Controllers (BMC). However, research into them is nevertheless significant as BMCs are responsible for many things, one of them being the emergency response on hardware faults. Even with such a critical responsibility, the response time of the BMC interrupt handler on the Enzian system, developed at ETH, was preliminarily measured in the order of hundreds of milliseconds [1]. Further, this vital system could not be characterized in detail up to this point, as no tools to do so worked on the operating system of the BMC.

This work shows how it is possible to get some tracing to work for the OpenBMC-based kernel in use on Enzian. Additionally, a test suite named the TestBench, was created to trace multiple Linux tools repeatedly and consistently. Inefficiencies in the current implementation were found utilizing these tools and the improvements to them reduce the total response time of the interrupt handling by 30-45%.

Acknowledgements

Throughout the writing of this thesis, I have received a a lot of support and assistance. Without all of that, this work would not have turned out the way it has.

First, I would like to thank my supervisor Prof. Dr. Timothy Roscoe for the possibility to work on such an exciting and cutting edge project like Enzian and for all the helpful comments throughout my time working on it.

It is hard to find words to express my utmost gratitude towards my co-supervisors Dr. Michael Giardino and Daniel Schwyn that invested countless hours helping me understand the complexity of the Enzian system. I am incredibly grateful for the always helpful answers they gave whenever I asked them, no matter how basic, and the fantastic feedback they provided me.

I would also like to thank everyone else in the Enzian Group of ETH Zurich, specifically Dr. David Cock, for answering my questions during the weekly Enzian meetings and so often giving very insightful tips. An additional thank you to Ben Fiedler for the excellent idea on how to improve this work.

Lastly, I wish to express my appreciation for all the help my friends and family gave me while writing this thesis. In particular, I would like to thank my roommates for being my sounding board and all the astute thoughts they had for me. Special thanks also go out to all my friends and family members that proofread this work and gave me so valuable feedback. An enormous thank you goes to my girlfriend who encouraged me along the way to keep working and for all the breaks we had that fully recharged me.

Contents

List of Listings

Chapter 1

Introduction

It is a critical issue if an emergency response system in any state-of-the-art computer system takes hundreds of milliseconds to react. The timeliness of this emergency response can determine whether the hardware is damaged or not, for example, if the cooling system malfunctions. Unfortunately, such a slow response time is the case for the current implementation of the fault interrupt handler custom-built for the Enzian Board Management Controller (BMC). Furthermore, the characterization of this emergency response was not done up to now, as the necessary tracing tools on the Enzian BMC kernel (EnzianBMC) were not yet working.

In this work, we address these critical issues by first showing how to get tracing and profiling tools on EnzianBMC to work in the different stages throughout the interrupt handler. These, now functioning, Linux tools are then combined with the custom-built TestBench to trace the emergency shutdown function repeatedly and consistently. Doing so led to the discovery of inefficiencies that, after improvement, reduced the response time by up to 45%.

1.1 Motivation

While many Board Management Controllers are proprietary and therefore inaccessible for general research, this is not the case for the BMC of the Enzian project in the Systems Group at ETH Zurich. This opens the avenue to work on and improve upon existing BMC software.

In preliminary experiments on the Enzian BMC the emergency shutdown function on an idle system took around 200ms [1]. This emergency shutdown, called a scram, is triggered by the BMC whenever any critical hardware issues are detected. Examples of such issues are overvoltage, overcurrent, or simply the cooling fans not working correctly. Any of these issues could damage hardware components like a CPU or an FPGA. Because of this, it is essential to shut these components down as quickly as possible to protect them from further harm. Hence for such a vital function to the system's health, the preliminary scram response time measured is very significant.

1.2 Challenges

Improving the emergency response time of the EnzianBMC is possible by characterizing the interrupt handler doing this shutdown. This characterization can be done using tracing or profiling, which both give information on where the most significant time is lost within the current implementation.

One challenge to achieving this characterization is the sheer complexity of today's servers, even more so of the Enzian project. After understanding some parts of Enzian and its Board Management Controller, the next challenge is that the kernel running on Enzian's BMC does not have any working tracing or profiling tools to do the characterization. Knowing there are multiple tools available for Linux to characterize different places in the kernel, finding a way of combining the output from them was another. The last challenge known from the beginning was that any Linux distribution, even for an embedded system, is non-deterministic by design. This non-determinism stems from many sources like parallel execution or the possibility of an interrupt occurring at any time. Other causes can be scheduling or simply non-deterministic choices taken in software.

1.3 Solution

Before getting tracing or profiling tools to work on EnzianBMC, knowledge of which tools are even available in Linux is required. For that reason, we discuss the available tools in the background. There we also analyze which tools could be used to characterize the interrupt handler and which can be used on EnzibanBMC.

The Yocto Project, described in subsection 2.2.3, is the build system for the operating system (OS) of the Enzian BMC which is also used to add tracing tools. Unfortunately, adding them was more than using the respective Yocto recipes. Problems encountered ranged from missing patches for the tool in question to the embedded system not having enough flash memory to store the final image with everything added.

We ended up using the Linux tool perf and multiple Python scripts for the characterization. Dmesg was used initially for kernel driver timing but replaced with higher fidelity tracing using perf and kernel driver tracepoints. Python logging with the `time` module was used to profile the biggest part of EnzianBMC, called the power-manager, because of less overhead compared to other available options.

To work around the non-deterministic nature of Linux while characterizing the interrupt handling, all tool usage had to have reproducibility in mind. This way, it is possible to show statistical insights throughout the power-manager and the kernel by repeatedly running the same test. To achieve this reproducibility, the so-called TestBench described in section 4.2 was implemented.

Utilizing the tools we brought into working conditions on EnzianBMC, it is possible to characterize the interrupt handler. The results of our initial characterization, which we again found to be in the order of hundreds of milliseconds, together with the inefficiencies found in the current implementation, can be seen in chapter 5. Some of these inefficiencies were expected and are now validated, while most of them were further inspected and improved. All improvements combined resulted in around 45% reduction in emergency response time.

Chapter 2

Background

Characterization of the interrupt handler on the EnzianBMC needs a breadth of knowledge. This includes tools that can access the necessary parts within the kernel to application layer programs executed using an interpreter. Which hardware triggers interrupts, how the EnzianBMC receives them or what the power-manager issues as a response to these interrupts.

Some of this needed knowledge is discussed in this section. It starts with what the Enzian platform is, followed by what is usually defined as a Board Management Controller. Because BMCs interact closely with the physical world, real-time systems are presented. An introduction to the build system for EnzianBMC called the Yocto Project is followed by one about the main components of the power-manager doing the fault interrupt handling. What interrupts are and a graphical representation of the path a fault interrupt takes on the EnzianBMC can be seen in section 2.3. Lastly, we introduce different tools available on Linux used for tracing in this work, including those that were actually used and those we failed to get working with reasons why.

2.1 Enzian

The Systems Group of ETH Zurich developed a heterogeneous server-class computer named Enzian. This server consists of a 48-core Marvell Cavium ThunderX processor alongside a Xilinx Virtex UltraScale+ XCVU9P FPGA as the main FPGA. The BMC is an Enclustra Mercury ZX5 [2] module sporting a Xilinx XC7Z015 System on Chip (SoC)[3]. The latter has as an embedded system beside a CPU and an FPGA, 1 GiB SDRAM and 576 MiB flash memory.

Much work for the Enzian project went into the BMC software stack as existing software had to be modified to work with the newly created board. This work on the EnzianBMC includes implementing drivers, real-time tasks, and making the BMC work according to its responsibilities as a fail-safe for the Enzian board. As a fail-safe, the BMC has to react to issues and, in particular, faults across the Enzian server and power off components before they are damaged. Powering off these components is a vital goal of the EnzianBMC, but its timeliness was never checked because no tooling to do so worked.

2.2 Baseboard Management Controller

Baseboard Management Controllers (BMC) are usually on the motherboard of another system as a specialized service processor to monitor the state of that system. This other system can be any hardware device needing monitoring, such as a smartphone, personal computer, network device, or server like Enzian. Besides possibly communicating information about the system's state over the network to an administrator, they also allow the latter to remotely manage the device. These management capabilities can range from power cycling, initializing or configuring the primary system, to completely reflashing it. Even though BMCs are widespread, no universal definition of what they exactly should be capable of or contain exists and as such the specifics of them can differ widely.

A BMC in servers like Enzian can commonly access internal physical variables such as temperature, power supply voltage and current, fan speeds, and OS functions. They can not only log all of these variables, but if some stray outside specified limits, they power off the primary system to safeguard it from further damage. This damage to hardware can be from overcurrent breaking components to heat destroying them without appropriate cooling. Powering off or rebooting the primary system is possible in the first place because the BMC is often not connected to the same power rails or power supply as the rest of the device.

BMCs generally are not standardized, often closed platform, and highly vendor-specific. They still do mostly align with the Intelligent Platform Management Interface (IPMI) [4] standard, even though it is now advised by Intel [5] to use a more modern standard like DMTF Redfish[6].

The software running on BMCs is similarly not standardized, and the only well-developed and documented open-source project is OpenBMC [7]. It is very modular, based on Linux and D-Bus, and implements configurable BMC functionality. This functionality contains most of what a typical BMC software stack has to be capable of, including power, cooling, and network management and support for sensors, logging, and simulation. OpenBMC has as founding organizations and supporters Microsoft, Intel, IBM, Google, Facebook, and an extensive and active community. It is configured and built using the Yocto build system described in more detail in subsection 2.2.3.

In general, because of the complexity involved in a BMC, configuring one securely is very hard. This leads to them being excellent attack vectors often neglected and hence part of the top five hardware attack vectors [8].

2.2.1 Linux as a real-time system

In addition to giving remote access to hardware, the control loops running on a BMC are there to shut down the main machine if it is outside of its safe bounds. Hence BMCs and their operating system should operate as close to real-time as possible. The Enzian board has, in addition to the CPU, an FPGA producing an immense amount of heat. For this reason, it is even more relevant to check sensors and to shut the system off within as little time as possible if something should be amiss.

Real-time systems often interact closely with the physical world and hence are very sensitive in respect to their execution time. Such real-time systems are built into cars, planes, or industrial robots. They have to react to changing circumstances within a specific time window as otherwise they run into critical failures. These critical failures can mean permanent damage or even physical harm to people. Real-time systems are usually built up such that each abstraction layer gives guarantees to the layers above. These guarantees do not exist in the Linux kernel by default and consequently also not in EnzianBMC.

Linux PREEMPT_RT patch

Even if the Linux kernel is by default far from a real-time operating system there do exist efforts to reduce the disparity. The most prominent of these efforts is the Linux PREEMPT_RT kernel patch [9].

One of the most pressing issues this patch tries to address is the fact that in the default Linux kernel many systems are by design non-preemptible. This means that once they start running, they will run until they finish with whatever work they have to do. In regular applications, where the Linux kernel is not used as part of a real-time system, this is not a problem as we are talking about sub-millisecond delays.

However, this is not the case for real-time systems in general and is a primary source of problems when using Linux as one. There are different approaches to different subsystems within the kernel to fix these issues. One of the main ideas is replacing or modifying subsystems with equivalently functioning systems that can be preempted.

One such example in the PREEMPT_RT patch is converting the typical Linux interrupt handlers into preemptible kernel threads. Another modified subsystem is the mutex locks within the kernel being made preemptible through reimplementation with rtmutexes. For a more detailed explanation of what the PREEMPT_RT patch changes within the Linux kernel see their official website [9].

2.2.2 EnzianBMC — OpenBMC on Enzian

Enzian is a racked server-class machine, which means users may not always have physical access. For this reason, it also supports maintenance of the board in the form of a Board Management Controller (see section 2.2). As OpenBMC is configurable and compatible with all types of embedded BMCs the kernel running on the Enzian BMC is OpenBMC version 3.2 with modifications and additions. The main parts added are the power-manager described in 2.2.4, which consists primarily of four services, as well as a modified driver from Xilinx for GPIO interrupt handling. This OpenBMC kernel with all adjustments is called EnzianBMC.

As EnzianBMC is still a BMC, its primary goal is to keep the board in a running state. Meaning the EnzianBMC powers the rest of the server on and off, checks the state of the hardware, and importantly keeps the ThunderX CPUs and Xilinx FPGAs from overheating. Most of this functionality goes through the services of the power-manager. Because the power-manager runs on the Board Management Controller, it has access to many internal physical variables. Some of those are voltage and current regulators on different rails of the CPU and FPGA. Others are the fan speeds and temperature sensors throughout the case.

Another essential part of the BMC on Enzian is to give everyone working with Enzian's the possibility of rebooting, shutting down, programming the FPGA or even flashing a new OS remotely. If no physical changes to hardware are necessary, most of these can be achieved through a series of steps regardless of the current running state of the system. Using these features, fixing, for example, a deadlocked kernel without physical access becomes possible.

2.2.3 Yocto Project — Chosen build system for EnzianBMC

The Yocto Project or Yocto is an open-source collaboration project that is described as not being a distribution but creating one for you [10]. As BMC's are embedded devices, and OpenBMC already uses the Yocto Project and OpenEmbedded within it, the former was chosen as the build system for EnzianBMC. Yocto features very high reproducibility as it compiles everything from scratch while using caching of prebuild binaries for speed improvements. It is further very scalable and gives many configuration options using abstractions named meta layers and recipes. The possibility to search for such meta layers and recipes does exist [11] and was used in this work. After appending these to a project, they add the defined features to the system. Because of these abstractions and the possibilities they offer Yocto comes with the cost of higher complexity and a steeper learning curve. The products received from a successful Yocto build are a toolchain for this system, a root file system, a kernel image, and a bootloader image. The latter three are used to flash a tracing-enabled kernel to the Enzian BMC. Detailed steps on how to create these files for EnzianBMC can be seen in subsection A.1.1.

Compared to other build systems like Buildroot[12], based around cross-compilation with makefiles, Yocto is faster because of actual caching and also has more industry support. A similar system that could be used but is not meant for a BMC like EnzianBMC is the OpenWrt project[13]. This is because it is more geared towards network devices and hence less flexible for a general embedded system like the BMC with all of its sensors and responsibilities. One could even think of using a normal distribution like Arch or Debian and adjust it for an embedded system, but this would be challenging as they are not optimized for that purpose.

2.2.4 The power-manager

The power-manager at this time is mainly in the form of four Python services: the GPIO, Power, Fault, and Telemetry service and an interrupt handler receiving GPIO interrupts written in C. All four Python services are accessible and communicate using D-Bus and have an executable wrapper script called <name of service>_service.py. These wrapper scripts are in the main directory of the power-manager repository and used to start the services in a D-Bus main loop. At boot, systemd similarly uses these scripts to start the power-manager. The actual functionality of the Python services is implemented in the respective `src/enzian/dbus/<service name>.py` files within the power-manager repository.

Descriptions of what functionality each part of the power-manager implements, which mechanisms are used, and how they interact with each other follow.

XGPIO driver

The XGPIO driver is a modified Xilinx xgpio kernel driver. The kernel calls this driver when an interrupt from hardware arrives for which it is responsible. Once called, the xgpio driver, using a shadow register, will determine the pins that changed in value. For each of these values the driver does the translation to the GPIO pin mapping within the kernel. It then calls further up the stack ending in the GPIO service sending D-Bus signals for precisely those translated GPIO pins. Any services running on the EnzianBMC can subscribe to these signals to get notified once they change.

GPIO service

The GPIO service handles all GPIO communication from Python by the power-manager with the hardware. All signals from the other services and even the shell script go into this service and are then sent over GPIO to hardware. As this service is the only one receiving GPIO interrupts from the kernel, the primary interrupt handler within the power-manager is added here. This handler method is named `irq_handler` and, after getting the GPIO value, sends this value with the correct pin name out over D-Bus. Sending it results in all other handlers in services of the power-manager connected to this specific pin name being executed.

Power service

The power service is the main interaction point of the power-manager to communicate with different devices or change the whole system state over SMBus and I^{2}C. The interaction with various devices is implemented in separate device classes abstracting each device-specific interface into one common one. Additionally, this communication to hardware is done in a single-threaded fashion, which is crucial as both SMBus and I^{2}C can not handle contention. The power service exposes a lot of its functionality through D-Bus and is the primary way of configuring and querying the hardware.

One example of a method exposed to configure hardware is `set_device_control`, which can be used to set configuration values like fan speed. Setting specific binary values on the power sequencers can be achieved by another method called `device_write`. A more critical method is the `common_power_up`, which powers up and initializes the CPU, FPGA, and the fans in the correct order. This order has to be done strictly according to specifications as otherwise hardware could be damaged. All this exposed functionality is used by the other services, such as the fault service, to set the fan speed or configure the power sequencers to shut off the power during an emergency response. The `common_power_up` can similarly be used by any service or even an external script to power up the system.

Fault service

A relatively small service but one with essential functionality is the fault service. It decides for all signals that it receives what the appropriate reaction is. This decision logic depends on the type of interrupt handler called.

There exists the generic `alert_handler` checking the received signal against which signals are configured to result in an emergency shutdown. If the received signal is one of them, the `scram` method is executed. Then there is the `fault_handler` method directly connected to the GPIO service on the fan fault pin. This interrupt handler method shuts everything off as a security measure when the fan bus reports a fault. Lastly, there is the `unknown_alert_handler` which listens for device alerts from unknown devices and always results in a shutdown. This is done because we can not determine which device has an issue if we do not know where the alert originated from.

This `scram` method called by the above handlers uses internally both the GPIO and power service to shut off the Enzian board as quickly as possible. The method primarily characterized throughout this work is this `scram` because the speed of this method is crucial for the health of the system.

Telemetry service

The telemetry service is a minor service and not (yet) widely used. It is possible to add device monitors to this service, each defining one value to be read from a specific device. When started, this service will read the values of the monitors defined in set intervals that are configurable. Reading of values is done using exposed methods of the power service. The telemetry service additionally implements the functionality of sending the accumulated telemetry data as a signal on D-Bus. This, in turn, can be used to make a dashboard about the most critical system values like voltage and current draw of the Enzian board.

Helper scripts

Helper scripts are, for example, the start_services.py starting the above four services. This start script is also run at the start-up of the BMC and can be used to get the services running again should a `systemctl restart` not do so. Another useful script is the shell.py executed with the argument `bring_up`, which internally makes use of power_sequence_bringup.py. These two scripts combined create an interactive Python shell with all four power-manager services in the environment as objects and many more utility methods already defined. Examples of such methods are `init_sequencers`, which initializes the sequencers of the different hardware rails to get them into a consistent state, `common_power_down` and `common_power_up`, which power down or up the regulators, and `init_bringup_script`, which does more initialization.

2.3 Interrupts and interrupt handling

Interrupts are events changing the execution flow of a program and can be generated by hardware or the CPU cores. When an interrupt occurs, the current flow of execution is suspended, and an interrupt handler runs. The first thing the called interrupt handler does is figure out who fired this interrupt and respond accordingly. For a Board Management Controller fault interrupts by voltage, current, and temperature sensors are vital because handling them can reduce further damage to the system. These interrupts usually notify the system of bounds that are transgressed and hence probable imminent issues with the hardware. If the interrupt handler did not begin emergency measures, it will jump back to where progress was suspended. This jump back is done to provide the illusion that nothing happened to the running processes.

Generally, there are two properties of interrupts. One is if they are synchronous or asynchronous, the other is if they are maskable or non-maskable. Synchronous interrupts can happen inside the CPU by executing code. Because faults can happen anytime, these interrupts from hardware that the faults trigger and EnzianBMC receive are asynchronous. The difference between maskable and non-maskable interrupts is that maskable interrupts can be ignored by configuring them as such, but a non-maskable one can not. The power-manager on EnzianBMC uses such masking to ignore any faults that occur during powerup, during a normal shutdown, and after an emergency shutdown. The fault interrupts happening during these short periods of time are not indicative of actual defects but rather that some parts of hardware are not yet ready. Examples are under-voltage during powerup of the regulators or overcurrent during start-up of some components.

For more detailed information about interrupt handling in operating systems in general, one can take a look at Interrupt Handling Schemes in Operating Systems [14].

2.3.1 Fault interrupt path on the Enzian BMC

The characterization in this work is done in particular on the custom interrupt handler for hardware faults. The path taken by a fault interrupt through EnzianBMC can be seen as a block diagram in Figure 2.1. Even if this specific path is characterized here, the tools shown can be used for any other interrupt handler as well.

An interrupt notifying the CPU of an event is as usual for interrupts first demultiplexed to be then handed over to the appropriate software stack. In the case of EnzianBMC, the software that will handle the fault events characterized is the modified xgpio driver. After this driver found that pin values changed it hands the interrupt over to the power-manager. The power-manager receiving this interrupt decides in the fault service if the machine should be shut down as a security measure. If this is the case, the fault service using the power service talks through the kernel over the I^{2}C bus back to the relevant hardware to shut off the power.

2.4 Tracing and profiling

Whenever software is too slow for what it is supposed to achieve, there are two options. Either to rewrite all of it from scratch in a more performant way or improve the speed of the existing implementation. The former is often not feasible for multiple purposes like complexity, available features, or simply not having the time a rewrite takes. To improve software, on the other hand,

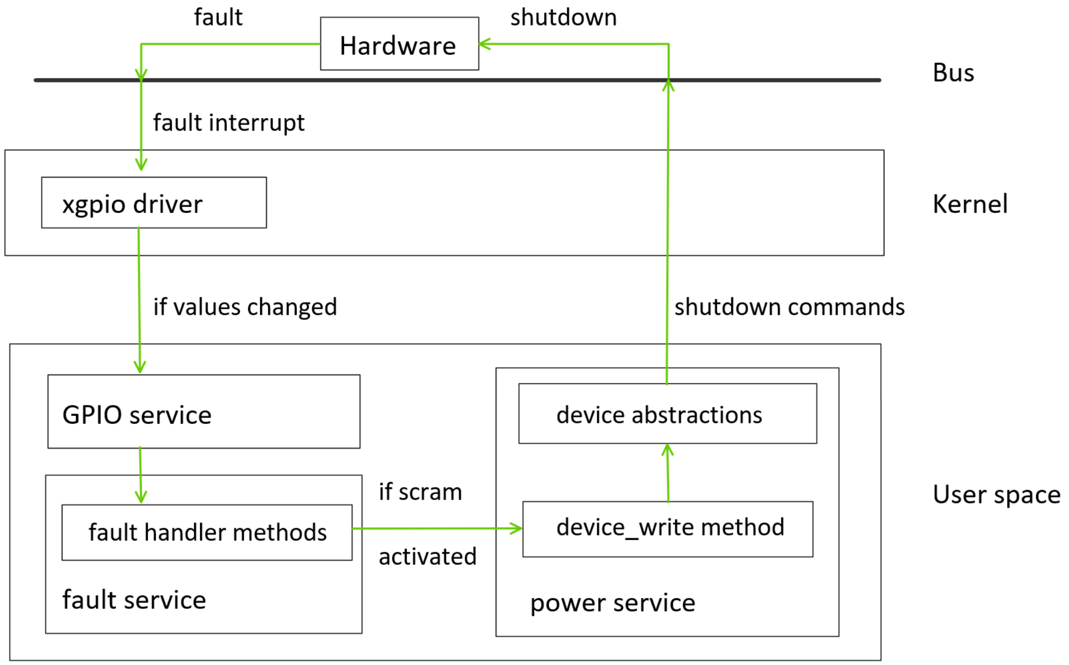

Figure 2.1: Hardware interrupt path through EnzianBMC in green.

knowing where the bottlenecks in the existing implementation are is necessary. Finding these is where tracing and profiling tools can help, even if the distinction between them is often not very well defined. There is generally not much up-to-date research on tracing or profiling Linux, but information can be found in mailing lists and in writeups [15].

We define tracing as recording information about the execution of a program in great detail and high frequency. Because of this high detail tracing is used to understand what goes on beneath the many layers of abstraction in modern software. Tracers look for specific instrumentation points within the executed code to get such a level of detail. The Linux kernel has multiple types of these instrumentation points called kprobes, uprobes, User-level Statically Defined Tracing (USDT) probes, and tracepoints. Their differences are subtle, but knowing that solely USDT probes and uprobes are used in userspace applications is sufficient for this work. Whenever these tracepoints are hit during execution, they are saved as an event to a trace file. Each event contains information about when precisely which tracepoint was hit with which data. This data can be from nothing to all the information accessible at that point in time. The information contained in these trace files is then later used to understand what happened during execution.

Closely related to tracing is logging. It is defined as the high-level analysis of less frequent events and is typically used for a different reason than tracing. A part of the EnzianBMC characterization is using logging regardless to augment trace data.

Profiling is defined as a dynamic program analysis that measures a few program properties like space and time complexity. Profiling tools can often give first insights but do so with coarse granularity. Because of this, they usually can tell where inefficiencies are in source code but not precisely how they are reached. To be able to say the latter, a more fine-grained trace of the program execution is necessary.

2.4.1 Trace and profiling tools available for Linux

As the scope of this work is to characterize the interrupt handling of an embedded Linux system, we have many different frameworks and tools from the Linux ecosystem to choose from. A tool differs from a framework in that users commonly interact with a tool and a framework mainly offers capabilities to tools. Many Linux tools often have widely overlapping functionality in userland, the kernel, and directly on hardware because the same framework in the background is used. Given that no dedicated tracing on EnzianBMC was done up to this point, it was unclear which of the widely used and open source options would be ideal for the task of characterization.

The tools listed below are in order of relevance to doing the characterization of the interrupt handler on EnzianBMC and to get an overview of tools in existence for future reference. This is by no means an exhaustive list of tools for tracing and profiling on Linux, but all the (non-proprietary) tools thought appropriate for the task. Which of those can work on EnzianBMC was mostly figured out by trial and error. They were tried because of their functionality but ultimately not used for different reasons like not having enough storage capacity on EnzianBMC. Given these issues are resolved, for example, with an external drive for more storage, trying at least some of them again is worth it.

More details regarding system performance can be found in the book called “Systems Performance: Enterprise and the Cloud” [16] by Brendan Gregg and on his homepage [15].

Perf

perf, perf tool, perf command or sometimes called perf_events and originally Performance Counters for Linux (PCL) is a performance analyzing tool in Linux [17]. It has been available in the Linux kernel since version 2.6.31 and was initially there to instrument CPU performance counters. A Yocto recipe exists that, after being fixed, is used to get perf successfully onto the EnzianBMC. perf can use any instrumentation points placed at logical locations in code, making it now more potent than it originally was. These are not only performance counters in Linux anymore but also system calls, file system operations, bus interactions, scheduling events, etc.

These perf_events have low overhead when the frequency at which the tracepoints are hit is low[1]. Further, they do not impact the performance of the running system much. This is achieved by disabling any instrumentation points when not in use and dynamically creating tracepoints using the respective frameworks when needed.

Because the perf command-line tool is so feature-rich, it is split into subcommands that have similar features grouped. Some of those subcommands are the following:

- `perf record` records events for later reporting. It will produce a `.data` output file in a binary format containing all the traced events that were configured.
- `perf report` is a simple command breaking down events in a given trace file by different metrics like process or function and many more. This is the subcommand most often used when quickly checking what is happening on a system.
- `perf timechart` is the tool whenever a visualization of system behavior during a workload is needed. The trace file this subcommand is expecting can be created by running `perf timechart record` or an adequately configured `perf record` command.

[1] This is the case because the only low-level tracing we are interested in are the fault interrupts and the bus interactions caused by these interrupts. Both of these are compared to other tracepoints like CPU counters and more a lot less frequent.

- `perf script` is the primary programmatical way of interacting with the created trace file. Executed without any arguments, however, it will print a detailed trace of the recorded workload.

Because perf is so feature-rich and it was possible to fix its Yocto recipe, it is used throughout this work. Further, every kernel module can easily be extended to implement static tracepoints directly accessible by perf.

Extended Berkley Package Filter

The first instantiation of the Extended Berkley Package Filter (eBPF) [18] was added to Linux 3.15 and is an extension to the BSD Package Filter [19]. eBPF allows loading any eBPF program to run in the kernel. Because the kernel has unrestricted access to many more system variables and processes than userspace programs, eBPF is not only interesting for tracing and profiling but also for other applications like security [20].

One nice feature of eBPF is the low overhead compared to userspace tracing tools for high-frequency tracing. This speedup is gained by not sending the tracepoint data to userspace but directly filtering it inside the kernel. In addition to not doing any calculations in userspace this simultaneously means having fewer context switches.

As eBPF is part of the Linux kernel, it can be activated with the *CONFIG_BPF_SYSCALL* menuconfig flag in normal distributions and similarly in Yocto. Because eBPF is only a framework for tracing, profiling, and much more, it is not helpful for most users in itself but only in combination with tools using it. Some tools that use eBPF are BCC, bpftrace, and ply.

BCC and bpftrace

BPF Compiler Collection (BCC) [21] is a toolkit for creating efficient kernel tracing and manipulation programs using the bpf syscall in Linux. It includes many useful tools and examples and crucially enables writing eBPF programs very quickly in Python. As BCC uses eBPF in the background, their capabilities are the same but BCC offers better usability.

bpftrace [22] is a high-level tracing language that uses BCC to interact with eBPF in Linux. The C-like language allows for quick and efficient use of eBPF. Because BCC code is very complex and has a higher skill floor than bpftrace the latter is more accessible.

For eBPF and either BCC or bpftrace, there exist readymade Yocto recipes for installing them on EnzianBMC. The issue that arises during installation is that both of those recipes install a full LLVM compiler [23] with eBPF support built-in. They do so because BCC uses LLVM to compile the written filters into executable eBPF code. As the LLVM compiler is around 200MB in size, the image to flash onto the BMC grows by too much, given that our BMC only has roughly 500MB of flash storage for everything.

Making a custom recipe that tries to reduce the additional size might be possible, but while other tools needing less work are good enough, this overhead is not worth it.

ply

ply [24] is an open-source frontend that uses the bpf syscall to make use of the eBPF virtual machine inside the Linux kernel. Compared to BCC and bpftrace, ply is designed with embedded systems in mind and hence does not need LLVM installed. Its only prerequisites are a libc and the Linux BPF support built into the kernel. ply would be perfect for Enzian as everything needed locally on the BMC is the small ply binary compiled against the EnzianBMC headers. This is a major advantage over BCC and bpftrace with their need for storage while still being very efficient and able to connect to most of the events that the other tools can. ply's scripting language for filtering over the different events is further easy to learn and concise.

There are multiple reasons why ply is not used to characterize the interrupt handler on EnzianBMC. One is that ply was missed in the first search for existing tracing tools on Linux as it is not yet widely known. At present, there does also not exist a Yocto recipe to install ply easily. Additionally, ply is still under heavy development and has neither a stable nor well-documented language except for a few examples. Lastly and most importantly, perf was already fully working by the time ply was found.

OProfile

OProfile [25], an open-source project initially released in 2001, was one of the first tracers that worked very well to do system-wide and single process profiling. It had little overhead from the beginning and even support for call graphs. Besides perf and eBPF, OProfile has the most features out of the box. It provided those features for a long time using its own framework of a Linux kernel driver and a daemon. In Linux kernel version 2.6.31 this framework got dropped in favor of the more general and powerful Performance Counters for Linux. Since the kernel release 5.12 a few months back, OProfile is not part of the kernel anymore, but its very user-friendly userspace tools continue to work using the perf_event subsystems. Because it is not part of the kernel anymore, it now has to be installed as a package. Doing so with Yocto was possible until the Yocto Project 2.5 Release. In the migration information of this particular release is mentioned under point 4.13.2 Removed Recipes [26]; "oprofile: The functionality of this recipe is replaced by perf and keeping compatibility on an ongoing basis with musl is difficult."

The version of Yocto Project currently used by EnzianBMC is 3.2, which means this recipe does not exist anymore. Even though it ought to be possible, no further work was put into researching how to install it anyways. If the developers of the Yocto Project decided to remove OProfile and suggest perf, there is no reason to question their verdict.

LTTng

Linux Trace Toolkit: next generation [27], or LTTng for short, is one of the newer open-source endeavors to cater to high-performance tracing in Linux. It is more of a toolkit comprised of many tools to trace the Linux kernel than a single tool or framework. It does so using its own framework and allows tracing of user applications and the kernel simultaneously. This allows LTTng to do almost everything the other tools can with high performance and without in-kernel programming. LTTng operates using kernel modules to provide the tracing capabilities and is reasonably lightweight.

An exciting feature of LTTng is that it has good support to trace Java, Python, and C/C++ on top of the standard kernel tracepoints. It is very flexible by either streaming the events to a remote machine, tracing them locally, monitoring live events, only keeping a set size snapshot or directly the ring buffer's content on a persistent file. There already exists a Yocto recipe for LTTng, which should make installing it relatively straightforward.

LTTng was not chosen to characterizing the interrupt handler over perf for the simple reason that its existing Yocto recipe did not work immediately either. As far as was checked, the issue is similar to other tools the final size of the image[2].

[2] It does not right away look like exactly the same size issue. Because of this and its features, taking another look at LTTng for future tracing efforts on EnzianBMC is worth it.

SystemTap

SystemTap [28] is one of the tracers that uses its own framework, is very powerful, quite a bit more complex but generally regarded as less safe. It works by building kernel modules from user scripts against the Linux kernel it is running on and then loading them. Besides root access, the kernel's only other prerequisites are a C/C++11 compiler to compile the scripts, elfutils to parse debugging information, and Python for the tools themselves.

SystemTap had issues in the past with kernel panics or freezes. This was often the case because SystemTap was the first to use new kernel tracing capabilities that might not be entirely bug-free. Even though it is now supposed to be safer, it is still suggested only to use SystemTap in development or fault-tolerant environments. Regardless of that, SystemTap has an existing Yocto recipe that should make it very manageable to install on EnzianBMC.

SystemTap was not used because of its expectation that all debug symbols are available at run time. In normal circumstances, when compiling a kernel from scratch, this is not an issue. However, similarly to BCC and bpftrace, this is challenging on the BMC of Enzian because fitting all debug symbols on the available flash memory is not possible. Furthermore, the suggestion to only use it in fault-tolerant environments, with the BMC being a critical component, made the decision to favor other tools relatively easy.

Others tools worth mentioning

There do exist more tracers on Linux; some worth mentioning are ftrace [29], strace [30], sysdig [31], and dtrace4linux [32]. Most of these are either less supported, in development, or have fewer capabilities than the others mentioned.

ftrace is one of the Linux internal tracers to find out what is going on inside the kernel where it can be used for debugging or tracing latencies and performance. Although ftrace is typically considered the Linux tool called the function tracer, it is multiple tracing utilities aggregated. ftrace uses in the background the tracefs file system similar as perf can and is as such not needed[3].

strace only traces on each syscall of a program using an outdated API. Further, it has as a bug "A traced process runs slowly" [30].

sysdig is relatively new and gives a complex set of kernel-level instrumentation and analysis tools to provide deep system visibility, with native support for containers. It is not used because of its primary focus on cloud systems like Kubernetes.

dtrace4linux is an effort to port Sun DTrace to Linux, which would have some exciting capabilities if it were stable and integrated deeper into the Linux kernel.

2.4.2 Other tools used for characterization

Because a big part of the power-manager is written in Python, the tools and modules used to trace or profile it are mentioned here. For each of the options, the argument why it was not used is given.

Other Linux tooling commonly not categorized as tracers but used throughout this work are dmesg and systemd-journald. Even though the purpose of these two is not tracing, they are still helpful if other tools are impossible to set up.

Python tracing and profiling

To achieve the goal of characterizing the interrupt handler, it was imperative to be able to trace or profile the services written in Python. Python has many packages available in regards to tracing

[3] Additionally, the `perf ftrace` command is a wrapper around ftrace.

and profiling. Some that could characterize the power-manager code and are mentioned in chapter 3 are summarized below, grouped by the type how they would. These groups are the profilers like `scalene` [33] and `cProfile` [34] as well as tracing tools like CPython [35] compiled with `--dtrace` and `stapsdt` [36]. The third group with the `timeit` [37] and `time` [38] module is the only one we ended up using.

Currently, no other Python profiling package than `scalene` is worth installing. It can do more than all others, including but not limited to a lower slow down, multiprocess support, memory, and GPU profiling. Another welcome feature for developers who often can not optimize library code is the differentiation of time spent in user code versus libraries. Even as the Python profiling tool with close to the least overhead, it still has an overhead from 26% to 54% [33].

The profiler shipped with Python is `cProfile`. `cProfile` is mentioned here because of the sole fact that it does not have to be installed and hence can be used without any additional work. But as can be seen in comparison with `scalene`, the overhead incurred by `cProfile` is even higher.

Because of this significant execution time overhead incurred by the Python profiling modules they were not chosen for the characterization of the interrupt handler on EnzianBMC.

To trace Python there exist two primary ways. At a lower level with the interaction of the kernel, or using function hooks like `sys.settrace()` from Python itself. The latter is precisely what the `trace` [39] module does. As this is on the same level and with similar techniques to `cProfile` the overhead incurred is again bigger than can be achieved with other options.

Achieving high-frequency tracing with the interaction of the kernel is possible since Python 3.6 by compiling the CPython interpreter with the `--dtrace` option adding USDT probes to it. As these tracepoints are on the layer of the interpreter implementation, they have a high time resolution and simultaneously high code resolution. Running a CPython interpreter with `--dtrace` enabled, tracepoints for every function entry and return, the garbage collector start and end, as well as every line in source code, can be traced. Python with `--dtrace` would allow for tracing with regular Linux tracing tools, but this will also produce a lot more tracing information than is needed requiring a lot of post-processing.

If using CPython with tracepoints or compiling it anew is not an option, there exists the `stapsdt` [36] package for x86_64 architectures[4]. There does not exist a recipe by Yocto to install but generally allows tracing of Python code using dynamically created USDT probes generated with the libstapsdt library [41].

Other than adding USDT probes to the Python code or using built-in Python modules there still exists the third option of modifying the source code itself to do something similar to tracing. For this artificial tracing either the `timeit` [37] module or the normal `time` [38] module are worth considering.

The former is neither tracing nor truly profiling but can measure execution time very precisely. `timeit` is very much intended to be used for snippets of code in isolation. This is because the functionality of `timeit` is achieved by running the code snippet thousands of times while measuring each cycle's execution time. As such, it is not used to trace the power-manager as a whole but to figure out the overhead tracing produces in idealized examples.

The other module to consider is `time` which since Python 3.7 and on UNIX-based systems has

[4] This essential point of only x86_64 support is not mentioned clearly in either the module or the libstapsdt library. The only indications for this library to only be available to x86_64 architectures are part of a name in the Makefile, and the single comment `// TODO (mmarchini) add other architectures (this only works for x86\_64)` on line 9 in the *sdtnote.c* source file [40].

the method `clock_gettime_ns(clock_id)`. Here the first argument defines the clock source to be used, which can be, for example, `time.CLOCK_MONOTONIC_RAW`. The returned value by this method is the time in nanoseconds and with that precise enough to use for tracing of the power-manager services.

dmesg

dmesg [42] is short for display message and does nothing more than what its name suggests; printing the messages of the kernel ring buffer to the console. It can do so with a timestamp in the form of `[seconds.microseconds]` similarly to perf, but choosing which clock source it uses is not possible. Reading the kernel ring buffer it can still be helpful to look for rough relative timings of different kernel modules, drivers, or the kernel itself.

If dynamic debug print support is built into the kernel, it is possible to enable and disable debug prints over the debugfs system specially designed for debugging purposes without many rules for developers. This freedom of being able to write to these files whatever is used by dynamic debug to be able to define which functions to enable with which options.

systemd-journald

The primary purpose of systemd-journald [43] or just journald is to collect and store structured logging data from different sources through the system in a central and easily accessible place. Such sources are log messages from the kernel, the libc syslog syscall, and the native Journal API. The systemd service manager additionally invokes all service processes with the standard output and standard error connected to journald by default.

The command-line utility to read out these journal logs in a pleasant manner is `journalctl`. `journalctl` has many additional options, of which the below are the two used in the TestBench. First the *lines* option, which tells `journalctl` how many journal entries we are interested in seeing[5]. The second one used is *follow*, changing the behavior of `journalctl` from just printing already existing journal entries to doing so continuously also for newly added entries.

[5] This is equivalent to reading the whole journal log using `journalctl` and then calling tail on the output with the same number of lines.

Chapter 3

Interrupt characterization design

One of the main goals of a Board Management Controller is to react to changes in hardware. The BMC kernel is notified about such changes over hardware interrupts to the CPU and then has to decide how to handle them, possibly invoking an emergency shutdown. The challenge in characterizing the interrupt handler of EnzianBMC lies in the fact that many components are working together to achieve the goal. As such, it is necessary to understand all components and measure them and their different properties simultaneously. Even more, some parts work together only over a loose message-based connection, which makes it difficult to take measures of single components by themselves still acting as part of the whole.

We first take a look at what needs to be done to characterize such a complex system and which tools can be used to trace the kernel or the Python-based part of the power-manager. The differences between installing instrumentation on an embedded system like EnzianBMC and a normal distribution for a personal computer are mentioned. Following that is an argument why it would be good that EnzianBMC is a real-time system and also why it is not done in this work.

3.1 What needs to be done

The fault interrupt handler on EnzianBMC has different components working together which makes tracing them as one somewhat hard. As such, the better option is tracing each part and then later combining the traces gathered. We divide the system to narrow down the scope of the tools that need to be used to do the respective tracing. The characterization of the modified Xilinx xgpio driver and the kernel is discussed in the first part, the tracing design for the power services in the second. Two ways are examined in subsection 3.1.3 of how it is possible to get tools in a working state onto the Enzian BMC. The last part of this section is a short argument why applying the PREEMPT_RT patch to EnzianBMC is generally a good idea but not done in this work.

3.1.1 Outside Python

Before any part of the power-manager in Python code is notified of changes in hardware, the interrupt event already went through the kernel and the modified Xilinx xgpio driver.

Hence the first place in software that will get called after the hardware interrupt is triggered is the global interrupt handler of Linux. This global handler looks up the registered handlers for this type of interrupt and hands it over to our modified xpgio driver. Tracing this part of the Linux kernel would be possible using tracepoints within the kernel. Finding inefficiencies in the Linux kernel is outside of the scope of this work which is why this part of the interrupt handler is assumed as a minimal static overhead. This assumption is reasonable given that this part of the kernel is widely used and hence already well optimized.

There exist different methods to trace the xgpio driver called by the global interrupt handler. As we are in a development environment in which leaking information from the kernel is acceptable, the already existing kernel-debug prints can be used. Reading these debug prints is possible using dmesg, which prints the kernel's message buffer while adding a timestamp to each message. After initial testing, the timestamp added by dmesg to the output was found to be insufficiently precise. Modifications to the xgpio driver can get higher precision tracing information about the interrupts that are handled. These tracepoints added can be enabled and picked up by most tools that support dynamic tracepoints like perf and eBPF.

If the power-manager sees that the machine is outside its safe bounds, it will shut off the relevant power rails and hence the CPU and FPGA. The decision logic to do so is in the fault service, which uses the power service to send the commands for a shutdown over SMBus and I^{2}C. Hence to completely characterize the interrupt handler tracing the communication on the SMBus and the I^{2}C bus is necessary. Tracing these channels with a self-built kernel like EnzianBMC is relatively easy. The only requirement to do so is activating and enabling the respective Linux kernel tracepoints for those busses. With that many standard Linux tools can access these tracepoints on the I^{2}C and SMBus read, write and result functions.

Besides these hardware communication channels, there are additional communication channels in use throughout the fault interrupt response. The hardware communication to the CPU we assumed constant. The xgpio driver using IO signaling to Python and the power-manager services using inter-process communication over D-Bus. The challenging problem to solve here is measuring the time of all these communication channels. If the interrupt handler ran alone on EnzianBMC, it would be possible to follow a single interrupt from when it triggers in hardware to the changes it entails while measuring each communication step along the way. However, as a Linux system is never wholly idle, this becomes a lot harder. Making this even more challenging is that more than one type of communication channel is in use. As such, all communication channels must be traced simultaneously and critically with the same clock source for timestamps. The latter is required to be able to combine the gathered traces later.

Grouping the different communication channels away as an abstraction is a solution to this problem. Henceforth whenever communication time is mentioned, think of it as the time needed to do the communication over whichever channel is used in that interaction. This abstraction also makes the time measurement of communication easier because we can compare the relative time from the send operation of one component in the interrupt chain to the respective receive operation of the next.

3.1.2 Within Python

In addition to tracing the interrupt response inside the kernel space the Python-based part of the power-manager also needs to be characterized. Theoretically tracing this Python part is not difficult because Python has many modules capable of doing so, as seen in section 2.4.2.

Unfortunately, both the built-in module `cProfile` and the `scalene` package have for our use case relatively high overhead. As such, both of those are not a possible solution to timing in the power-manager. The `trace` module is similarly not used for this reason of an overhead, but more importantly, it does not offer a precise enough time resolution either.

Compiling the CPython interpreter with the `--dtrace` option is likewise not ideal because much more information than needed is gathered. Doing this might have worked and was a fallback option despite the additional post-processing and possible non-negligible overhead. Ideal for tracing the power-manager would be the `stapsdt` python package. But as mentioned in section 2.4.2 it currently works only on x86_64. Adding ARM support would likely be possible, but because libstapsdt has parts even written in assembly, the port of this library to ARM is outside the scope of this work.

The only option left is to measure the time exactly where needed, followed by somehow saving this measurement. Doing so offers a more precise measurement of time with less overhead and the possibility for quick progress of the actual emergency shutdown. Getting time in high precision on Unix based systems is possible with the `time.clock_gettime_ns(clk_id)` method from the Python built-in `time` module. This high-precision timing is offered with an overhead of around 4 microseconds as can be seen in subsection 5.1.1. As it turns out, it is imperative to define this method argument `clk_id` for tracing-related purposes for mainly two reasons.

First, if you do not use a monotonic raw clock source, the time you get might not be the raw hardware-based time but time subject to the Network Time Protocol (NTP) and similar. Secondly, combining timing information from within Python to other Linux tracing tools is only truly possible using the same clock source everywhere. Using the same clock source means getting some guarantees that at least the relative time between all the tools used for different measurements is correct.

Regardless of how precise the time taken is, it still takes roughly 4 microseconds to get this time. Because of that, it is nearly impossible to get to a similar time resolution as kernel tracers that do so in tens of nanoseconds.

Outputting this now relatively precise time measurement has to be done in either a performant way or in such a way as not to slow down the interrupt handler heavily. The latter option was chosen because the power-manager is already using the Python logging framework and as such, saving the timestamps was very easy. This way, the overhead is minimal during the tracing itself as the logging can be done after the interrupt handler is finished. Additionally, it is possible to easily differentiate the tracing outputs between the different power-manager services running, as the gathered data will be in the respective journal log.

3.1.3 Installing instrumentation on EnzianBMC

To improve the interrupt handling response time on EnzianBMC, figuring out where improvement effort is best applied is crucial. Tracing and profiling can help, but both are often only done during optimization and not on production systems, so default configurations usually do not include the relevant tools. The default EnzianBMC built does not have any working tracing or profiling tools installed either. However, they are needed to characterize the interrupt handler outside the Python-based part of the power-manager. On an embedded system like the Enzian BMC, installing new tooling is more challenging then on other distributions that are not optimized for embedded systems. This comes down to not having any internet connection[1] nor installation and build tools. As such, adding new tools to EnzianBMC was done in two ways: copying using SSH and installing at built time.

Copy and run

The easiest way to get instrumentation onto EnzianBMC, which is usable without dependencies and network connectivity, is to use SSH to copy them. This can work well for anything whose dependencies are already met, such as Python using standard modules, simple shell scripts, or any standalone executable binary compiled for EnzianBMC. Cross-compiling such a binary needs the

[1] No internet connectivity helps avoid exposing the powerful system that is the BMC to the outside world, even by mistake.

build root of EnzianBMC to link against and an ARM build toolchain which both could be achieved using Buildroot [12].

Install at built time

Expecting the tracing and profiling tools to need kernel modifications, rebuilding the kernel using Yocto Project was more efficient for anything that was not copied and run. This includes standalone binaries that could be copied after being successfully built because the build root folder for EnzianBMC is already in a fully working state on the build server.

This means adding new tools and libraries to either kernel or userland on the EnzianBMC is done by including recipes for those. Doing so through Yocto might be an overhead compared to cross-compiling them from a build root manually for some tools but can also be significantly less work for others. Adding a new tool to Yocto can be done in three ways needing increasingly more effort: Yocto has a built-in option to add the tool, a recipe for it is already available, or the recipe has to be created.

In the best-case scenario, the recipe for the tool in question is defined already by the Yocto Project itself, meaning it is enough to add an option into the build configuration. Because in Yocto this option exists for many Linux tracing tools[2], the initial idea was to use this to test out the different tracing tools. Unfortunately, as mentioned in subsection 2.4.1 many of these build configurations on EnzianBMC did not immediately work. Section 4.1.2 describes the work done to get the recipe of perf working.

The second scenario is someone else created a recipe, in which case adding the whole recipe and enabling it should be enough to get the tool working.

If the tool to be put on the EnzianBMC is custom-made and needs installation, or no one created a recipe yet, creating one is the last option. Creating a recipe using the Yocto Project requires more knowledge than just adding one to a project and configuring it[3].

3.1.4 EnzianBMC as a real-time system

Because the goal of EnzianBMC is to shut down the main machine as quickly as possible if something is awry, it being a real-time system as described in subsection 2.2.1 makes a lot of sense. That Linux, a multi-purpose open-source operating system runnable on many different architectures and widely different scales, can work well as a real-time system is not unthinkable, but in practice, very hard. There exist people who would like to use Linux as one, which is why there are efforts to improve the Linux kernel's real-time capabilities.

The most noteworthy of these is the Linux PREEMPT_RT patch. This patch tries, as described in section 2.2.1, to fix multiple significant issues of the Linux kernel with respect to a real-time system. In the end, even if the Linux PREEMPT_RT patch presents itself as an excellent first entry point to make EnzianBMC more real-time capable, the possible gain from doing so is currently not worth the overhead. Applying this patch to the current kernel build is not trivial and presently there exist other avenues where speedup can be gained easier.

[2] perf, ftrace, systemtap, oprofile, LTTng do exist and were tried with this option.

[3] In the course of this thesis, a custom recipe was temporarily added for the libstapsdt library, the backend to the Python package `stapsdt` seen in section 2.4.2.

3.2 Created tooling

Most operating systems and computers are generally viewed as being non-deterministic, with Linux being no exception. This is a consequence of the many processes running in parallel, having asynchronous interrupts incoming from the real world, and preempting running processes for more important ones. Because of this, the need for running the same measurements multiple times was clear from the beginning. This poses the challenge to bring this non-deterministic system automatically and reliably into a similar state to run all the necessary measurements.

In addition to the non-determinism, the created tool also needs to solve the aggregation of information from the different tools in use. This arises due to not being able to create dynamic USDT probes within the Python power-manager on the ARM-based EnzianBMC hardware. Otherwise, it would have been possible to let a chosen tool like perf handle all of the needed aggregation through tracepoints.

The decision to write a big part of the tooling in Python was made without much hesitation. This has multiple reasons, one being Python is, for the most part, directly executable on the EnzianBMC as it is already in use for the power-manager. Others are that using Python means the tool can very quickly be adjusted in place without the need of recompilation and that it can be run interactively.

3.2.1 The TestBench

The created tool which runs different tests was named the TestBench. It will repeatedly gather all output from other tools used to trace and profile in a modular way and save it into a single place.

By the time the TestBench started to take shape, it was clear that the modularity was very much needed for adding and removing different tools quickly. This is the case in view of the many ways it is possible to trace the different elements from the interrupt to shut down and the issue of not knowing from the start which ones will work well.

The TestBench is built up of the following three layers. The lowest layer can be thought of as the tool abstractions called commands for the different interaction methods with them. The next layer up are the profiles defining how to enter an emergency response from software repeatedly. The last layer is an abstraction for the different states the EnzianBMC can be in during a measurement with the TestBench.

Diverse interaction methods

The challenging part of creating the TestBench in a modular way was to be able to support the various forms of how to interact with different Linux tools.

Luckily, all Linux tools chosen have a command-line interface. The most general way for the TestBench to start a multitude of tools is to use their respective command-line interface. The issue to solve with this approach is that while some tools might need only to be run once at the end of tracing, others might need to run while the EnzianBMC is handling the interrupt. Another additional point to think of is that these tools can sometimes not be run in sequence, but have to run in parallel.

What follows are the general ideas of what capabilities the abstractions of the Linux tools for the TestBench have to have.

Dmesg (see section 2.4.2) interaction is used to get some basic timing information from the xgpio driver without changing anything about the driver. dmesg has by far the most effortless interaction compared to all the tools in use. This is due to dmesgs only feature being it printing out the content of the kernel message buffer. With a single command run after the EnzianBMC did the scram, it is possible to get all of the debug messages that occurred in the kernel, given the buffer didn't overfill.

Journalctl (see section 2.4.2) is used to get the logging information of the power-manager. According to the general use of journalctl, it should only be a single call after the scram to read the respective logs. In the end, it was easier to call it with `--lines=0` as an argument and let it run during the emergency response. This way, tracing for only the current interrupt response is in the output, removing the need to find the correct trace in the log later.

Perf (see section 2.4.1) currently has the most complex interaction from the tools implemented as modules in the TestBench. This arises from the complexity and possibilities perf has. Perf has to be run while the emergency response happens to do the tracing. This will create an output file containing all the traces. Given this trace file created by `perf record` there exist different subcommands of perf, each giving different insights into the gathered data within it. These different outputs are produced by calling perf again with that specific option and the output file from before.

Repeatedly entering emergency shutdown

One goal of this TestBench is to be able to run the same experiment repeatedly. To do so means the TestBench can start, and after precisely one emergency response, it stops the test run. Waiting after the TestBench start for the hardware to throw a fault interrupt was not an option for obvious reasons, which left as the only option to force the emergency shutdown. Causing a fault interrupt on the Enzian BMC, which results in a scram, can be done in two ways. Either electrically pulling the fan fault line low, which needs access to the BMC to add a physical probe, or doing something in software that looks similar enough to an actual defect in hardware. In the end, the latter option seemed more appropriate because constantly having a physical probe attached to the BMC and syncing this probe to software is more complex than doing both in software.

The two ways that are used by the TestBench were called the *direct scram* and the *fan scram*. Both can repeatedly and reliably be triggered from software and result in much of EnzianBMC behaving like during an actual hardware defect. As a significant part of the custom interrupt handler is written in Python the *direct scram* is directly calling the same method the fault service calls after deciding to scram. *fan scram* uses the power service to reconfigure temperature thresholds in the hardware sensor of the fans. Doing so allows it to set the fault temperature below the current temperature. This results immediately in an actual fan temperature fault interrupt arriving on EnzianBMC.

Measurement environments

Because of the complexity in following the interrupt and its response throughout the kernel, it was not feasible to test the interrupt handler of EnzianBMC on different kernels. The refinement of this idea is to do the same set of modular tests while EnzianBMC is in different states, so-called environments. Environments contain from how the EnzianBMC is configured to what else is executed at the same time as the interrupt handler runs. As the aggregation of tracing tool outputs already necessitated the TestBench, it was comparatively easy to add support for changing but reproducible environments. As initially it was not yet known which environments are interesting to look at and what changes to the environment would be interesting to compare with each other this addition had to be again very modular.

3.2.2 EvaluationScripts

Although the TestBench gathers outputs of tools over multiple runs over diverse environments and creates a lot of data to sift through, it does not do so itself. This is done by the EvaluationScripts, which were created to get insights into the EnzianBMCs interrupt and response chain using this gathered data.

The EvaluationScripts are written in Python and designed similarly in a modular way to how the TestBench is. This decision was taken to easier accommodate a priori unknown evaluations which might, later on, be interesting to do. These scripts roughly fit into three different categories. Doing this grouping allows for them to be maybe reused in the future.

Parsing is done in a way to give relatively easy access to the raw parsed data of the TestBench itself. The challenge here is that the parsing of the output of a tool added to the TestBench will depend on that very output, as the tools do not always have the same output format. This entails that most of the parsing scripts are specific to the output of the tool they are supposed to parse. Hence other than the coarse concept of how such an output can be parsed they do not have much in common with each other.

These parsing scripts make working with their output easier by doing sanity checks and clean the data. Cleaning here means removing the Linux tools' unnecessary output because this information will clutter the next steps. On the other hand, sanity checks are needed for the simple reason that the used tools sometimes do not behave the way we might expect them to. This can result in outputs parsed incorrectly, which can easily ruin the evaluation afterward if not checked for.

Calculation and aggregation of the parsed output is done sequentially on the TestBench data. The calculation step solves the challenge of not being able to measure communication time as easily on the different channels, but still giving this information to the next step. This step is not limited to this specific case but can also be used to do any sort of calculation on the provided data, like resizing all timestamps to the same unit.

The aggregation step after the calculation is designed for later evaluation of several measurements effortlessly. This same step furthermore helps evaluation between different types of environments.

Evaluation is the category containing scripts that return the post-processed data from the TestBench in some human-readable format. This format can either be the data in some concise format or directly visualizations. The latter is not always necessary, but it is challenging to gain insights quickly from the data without visualization.

Chapter 4

Implementation

Characterization of the interrupt handler on the EnzianBMC needs tools that can trace the interrupt throughout the whole system. As no Linux trace tools were in working condition, we describe in section 4.1 what we configured for dmesg, how the perf recipe was fixed, and how tracing the power-manager using the `time` module and logging works. This is followed by section 4.2 describing how the abstraction, repeatability, and consistency of the TestBench is implemented. At the end of this chapter, we explain how to parse the TestBench output and visualize this gathered data.

4.1 Getting some tools to work

Unfortunately, as mentioned in the background, many of the tools tried were not straightforward to get working on EnzianBMC. As such, the steps needed for those used to characterize the interrupt handling are shown here. How to activate timestamps for the otherwise working dmesg is described first, followed by a description of how the Yocto recipe for perf is fixed. Lastly, the implementation details of how tracing Python without the enormous overhead incurred by profiling and without USDT probes works.

4.1.1 dmesg

dmesg is the display message utility. The timestamps for dmesg are activated in the kernel build menu (see subsection A.1.3 for how) along with support for dynamic printk support. For this to work, setting a high enough log level and enabling the respective debug prints of the xgpio driver was necessary.

```
echo 8 > /proc/sys/kernel/printk
echo -n 'func xgpio_handle_channel_irq +pf; func xgpio_irqhandler +pf' >
 ↪  /sys/kernel/debug/dynamic_debug/control
```

Listing 1: Two commands to set maximal kernel log level.

The former is possible using the first commands in Listing 1 , whereas the latter can be achieved using the debugfs subsystem with the last command. The option *+p* tells dynamic debug that we want to enable the available debug call site defined after *func*. The *f* configures it to additionally print the function name of the call site[1].

[1] To disable a call site again or to remove only some options from it use the same command but a minus instead of a plus.

4.1.2 Perf

Perf was decided to be the tool to use for tracing of I^{2}C, SMBus, and more within the kernel. Perf is mighty and, more importantly, included in the Linux kernel. Despite having a premade Yocto recipe and being part of the Linux kernel, perf was tricky to get working on EnzianBMC. If the setup from the Yocto Wikipedia for tracing and profiling [44] would work, then all you would need to do is add `EXTRA_IMAGE_FEATURES += " debug-tweaks tools-profile"` to your Yocto build configuration. A description of what is added using these two additional image features can be found on the Yocto wiki page for tracing and profiling [44].

If you configure an otherwise clean current EnzianBMC kernel to use these extra image features, it fails at build time. The issue here comes from the fact that EnzianBMC is not based on the newest upstream Linux kernel and hence is missing some commits[2].

As perf is defined in the kernel source tree [17] it is missing some patches that were only added in newer kernel versions and are not part of the source for our recipe. This meant to build EnzianBMC with perf these newer patches to perf had to be backported for which a step-by-step description can be found in subsection A.1.2. All of the needed fixes, after making sure they worked reliably, were combined into the single git patch seen in subsection A.2.1. Applying this patch to the current EnzianBMC based on OpenBMC version 3.2 results in compilation with a fully functioning perf.

An otherwise unmodified EnzianBMC with this patch has working perf, but many tracepoints within the Linux kernel will be missing as they have to be enabled in the kernel menuconfig. Additionally, it might be that perf only returns hex addresses meaning the debug symbols are missing. Adding those at build-time is an option that will increase the size of the produced image by a lot. This issue is alleviated by not including debug symbols and then only manually installing the necessary debug symbols afterward using dbg-pkgs[3].

4.1.3 Python logging

Because the `time` module has the least overhead from all tracing and profiling modules for Python seen in section 2.4.2, using it with logging is the chosen option for tracing Python. Luckily, the Python logging framework is already in use by the power-manager to log information about its current state and is known to be working. systemd is used to start the power-manager services and as such journald described in section 2.4.2 will receive all log messages made within the power-manager.

In Listing 2 you can see how to read the journal log of the fault service continuously without previous entries. To read the logs of the other power-manager services replace the *Fault* within it with any of *GPIO*, *Power* or *Telemetry*.

```
journalctl _SYSTEMD_UNIT=systems.enzian.Fault.service --lines=0 --follow
```

Listing 2: Use `journalctl` to get logging output from the fault service.

Even if journalctl adds a timestamp to each log entry, this timestamp is not precise enough to use for tracing. This is reasonable because, under normal circumstances, one would only use the human-readable timestamp in a log to see when an event roughly happened. Adding to the log message a more precise timestamp enables tracing in the precision with which it is possible to measure time.

[2] Work is currently being done to upgrade the EnzianBMC to a newer version of the Linux kernel. After that is done using the tools-profile configuration directly should be tried again.

[3] Most of this is described in some detail also on the Yocto Project tracing and profiling wiki page.

As the built-in `time` module has the least overhead in Listing 3 is an example of how it is used throughout this work to profile the power-manager. This code snippet is again from the fault service, more precisely the `fault_handler` method, which will be called whenever a fault line is activated. Specifically, take note of the added lines 2, 4, and 6. Both lines two and four will get the time and in between of them, the emergency response method is called. Because masking the scram is still part of the response to the fault, but already after the power was cut off from the CPU and the FPGA, the time is taken before the mask and only afterward in line six logged.

```
1  # Scram before logging - time is of the essence.
2  start = time.clock_gettime_ns(time.CLOCK_MONOTONIC_RAW)
3  scram(self.logger, self.gpio, self.power)
4  end = time.clock_gettime_ns(time.CLOCK_MONOTONIC_RAW)
5  self.mask_scram()
6  self.logger.error("fault_handler scram start at %.6f end at %.6f", start, end)
```

Listing 3: Example of how logging is used to profile the scram in fault_service.

After having added the tracing using logging throughout the power-manager an output of the before shown *journalctl* command during a scram will look something like Listing 4. The first line is the default output from *journalctl* followed by the log entries. For brevity, all output except the `fault_handler` method output is removed. Each log entry is printed with a timestamp, the system name, and the service name. The number in square brackets at the end of the service name is its PID during runtime. This generic information is followed by the log level and the system name of the service logged. For tracing, the ERROR log level is misused to see the log entry whenever any logging is activated. The last thing per printed line is the log message from the service. In our case, this is always some way of identifying what in the source code this entry is tracing as well as when it started and ended. The latter three will be later combined with knowledge of where they are to gain useful tracing information about the power-manager.

```
1  -- Logs begin at Fri 2021-07-16 14:46:35 UTC. --
2  ...
3  Sep 02 08:39:28 schibenstoll02-bmc systems.enzian.power-manager.fault[17980]: [ERROR
    ↪  systems.enzian.Fault] fault_handler start at 6138237913074 end at 6138390072720
4  ...
```

Listing 4: Output of the fault service during scram with added tracing.

4.1.4 Adding tracepoints to xgpio driver

The EnzianBMC kernel before this step was able to be used for tracing the I^{2}C and SMBus interactions with decent time resolution. However, as dmesg was not precise enough, tracepoints were added to the modified Xilinx xgpio driver.

Following instructions [45] a new header file for the xgpio driver was created. This header file contains the definition of the two xgpio USDT probes using the macros defined in *<linux/tracepoint.h>*. In Listing 5 you can see how some of the macros are used to create a tracepoint definition that outputs the same information as the respective debug print it will be placed at.

The first argument, here *irqhandler*, is the name of this tracepoint. This is followed by the definition of what type the arguments to this tracepoint should have, as well as what name. *TP_STRUCT* defines the struct used to save the information from this tracepoint in the internal kernel buffer. The *TP_fast_assign* can be seen as a restricted C-style code block, which defines how the arguments are used to fill the information buffer. Lastly, we also define how this tracepoint is then outputted from the kernel struct.

```
...
TRACE_EVENT(irqhandler,
        TP_PROTO(unsigned long ipisr),
        TP_ARGS(ipisr),
        TP_STRUCT__entry(
                __field(        unsigned long,  ipisr   )
        ),
        TP_fast_assign(
                __entry->ipisr = ipisr;
        ),
        TP_printk("xgpio handler called, ipisr=0x%lx", __entry->ipisr)
);
...
```

Listing 5: Partial header file for adding tracepoints to xgpio driver.

After having created the correct header called *gpio-xilinx-trace.h*, we can now include this new header and use its functions to define static tracepoints. An example of how the functions created by the macros inside the trace header can be used is seen in Listing 6. The only lines added are those seen in this example from lines five to seven, where line five is added for efficiency reasons.

```
1  ...
2  ipisr = xgpio_readreg(mm_gc->regs + XGPIO_IPISR_OFFSET);
3  pr_debug("xgpio handler called, ipisr=0x%lx\n", ipisr);
4
5  if(trace_irqhandler_enabled()) {
6    trace_irqhandler(ipisr);
7  }
8  ...
```

Listing 6: Use of the defined tracepoint in the xgpio driver.

After defining those probes and adding them to the EnzianBMC with a kernel patch, they directly worked. How to apply such a patch to EnzianBMC can be seen in subsection A.1.4. Making the output of these added tracepoints the same as the debug prints meant easier switching of dmesg for perf in the TestBench.

4.2 The TestBench

Given a running EnzianBMC kernel with tracing tools, the next step is to use those to characterize the interrupt handling. To do this more efficiently, the amalgamation of scripts called the TestBench was created. It can parse the outputs of any tools over different runs in diverse configurable environments. Utilizing this TestBench, it is possible to compare various configurations statistically along with the possibility for someone to compare them with newer versions of EnzianBMC in the future. The latter is not only restricted to running the same suite of measurements or do the same type of calculations. Instead, one can add new types of parsing using old data or alternatively add new tools to gather more data on each future run. Writing the TestBench mostly in Python allows for very modular and yet quick development, which enables the ease of adding new parts.

The TestBench is conceptually split into two categories. On the one hand, we have the part executed on the EnzianBMC (see subsection 4.2.1). This includes all the modules to use different tracing tools from the EnzianBMC, the defined environments in which to run them and the scripts which glue these parts together. Additionally, some scripts that can be used to test specific things like logging overhead in Python are also part of this category. On the other hand, we have the scripts that run on the machine where the gathered data will be saved described in subsection 4.2.3. These scripts run the part of the TestBench on the EnzianBMC multiple times over SSH and copy the gathered data back after each run.

4.2.1 Part running on the Enzian BMC

Copied to the Enzian BMCs is only part of the TestBench, namely the *onBMC* folder. This folder contains all the scripts written to use tools like perf and the main TestBench script used to execute a run. The top level of the *onBMC* folder has the main script runTestBench.py, which ties all scripts in the subfolders testBench and bmcUtils together. The execution of this script results in the measurements of one interrupt with its response chain for each environment configured. Generally, the TestBench is implemented abstractly in three layers to solve the design challenges.

As the main layer, we have the scripts that serve as abstract modules of the tools they implement to the rest of the TestBench. This part solves the need to combine possibly many different tools and their usage into a package that can be quickly modified but also executed repeatedly and consistently. Each of these modules is implemented as a Python class, named after the commands they implement.

To give the possibility to force the interrupt in modular ways another layer was added. These so-called profiles define, after the environments are set up, and all the tools are tracing, what needs to be invoked to achieve an interrupt being triggered.

Another layer is the abstraction for the different environments that should be tested. These are defined again as classes extending from the same Python base class, where each of them can take a list of commands and profiles. After setting up the environment, these classes are then run separately for each defined profile and command. Between each run of measurement, the environment will be reset to achieve the goal of running them as consistently as possible.

Command class

The Python classes wrapping tools are an important part of the TestBench to solve the challenge of having to use multiple different tools. These classes are called commands and are specific to the tool they abstract for the rest of the TestBench. Each one of them extends from the same base class called the BaseCmd seen in Listing 7. Using the interface defined by the BaseCmd they can have a wide range of capabilities and even be made configurable. To do so, the BaseCmd class has three essential abstract methods. Their definitions are `runRecord(self)`, `stopRecord(self)` and the `runScript(self, out_to_file)`.

The first of the three essential methods is the `runRecord(self)`, which as its name suggests will start the recording. The conceptual idea here is that some commands can use this method to start the tool they abstract before the interrupt occurs. The TestBench makes sure to start all commands before the interrupt will be forced. Abstractions similar to the one for dmesg, which only have to run after the interrupt transpired, can implement this method with a simple `pass`.

```python
class BaseCmd(ABC):
  ...
  def _time_sync(self):
  ...
  @abstractmethod
  def runRecord(self):
  ...
  @abstractmethod
  def stopRecord(self):
  ...
  @abstractmethod
  def runScript(self, out_to_file=None):
  ...
```

Listing 7: Interface defined by the BaseCmd class

`stopRecord(self)` follows the same design philosophy and is used by commands to know when to stop their respective tools. Once this method is called on a command object the interrupt is already over. No promise of how long ago the interrupt took place is made.

`runScript(self, out_to_file)` was added for efficiency reasons to be able to move any labour intensive post-processing away from `stopRecord(self)`. The idea of this method is not to have to wait too long until the stop propagates. This is done not necessarily for the sake of the commands themselves, but because anything running on the BMC slows everything else down. Resetting resource-intensive environments is one of the workloads that should be stopped asap to have again more available CPU cycles for post-processing. The single argument given to this method is a string defining if the output of this tool should go into a file or not if the string is `None`. This means with a `runScript("./out.txt")` call the command should write its output to the text file named `out.txt`.

There do exist more methods in BaseCmd, but not all of them are as crucial as the previous three. One of the less important ones is `_time_sync(self)`. This time sync function is used by the commands that implement it to make a marker in the trace file to synchronize the start of a trace with all other commands. Even if the header of this method is defined in the base class, each command class that wants to make use of time sync needs to call it at the end of the `runRecord(self)`. The evaluation scripts can use these time sync events within the trace to easier synchronize or check that all tools do have the same relative time. The latter is crucial to be able to say anything about the timing between trace information from different tools.

Implementations of commands

Currently, the TestBench has six commands extending from BaseCmd and abstracting amongst other things dmesg, journalctl, and perf. Two of them, `RunAllCmd()` and `PIDCmd()`, are not wrapping a tracing tool in Linux, but rather show another way of using this interface defined by BaseCmd.

The `RunAllCmd()` is a utility command created to combine multiple other commands extending from BaseCmd into a single one. This enables us to create and configure multiple commands and combine them as we see fit. During a single interrupt we only have to start this run all command once to start and later stop all others.

The `PIDCmd()` is similarly a command that is useful for a specific goal to tracing besides calling other tools. Showing the correct name of the different power-manager services instead of only their process ID makes understanding the results a lot easier, which is what this command is for. It implements the functionality to get the mapping from power-manager service PID to its name and does so for each environment reset. The latter is required because resetting the environment includes restarting the power-manager services. Additionally, this command can be configured using its argument `prioritize` to set the Linux scheduling priority of the vital processes of the power-manager.

The `PerfCmd()` is an abstraction for the generic `perf record` and `perf script`. As such, this implementation of the BaseCmd is currently the most complex one. Part of this intricacy originates from the many possible configuration options it offers. Some but not all of them are:

- `command_identification` is a string that will be used to identify this instantiation of the perf abstraction.
- `trace_targets` is an array of strings as defined by `perf record` to tell it which events should be traced. An example is `['i2c:*','smbus:*','xgpio:*']`, where this will instruct perf to record all I^{2}C, SMBus and xgpio tracepoints.
- `record_args` are additional command-line arguments to define for the `perf record` being used for tracing.
- `use_clock_monotonic_raw` is of type boolean. If true, this appends to the record argument array `["--clockid", "CLOCK_MONOTONIC_RAW"]`. This argument `--clockid` will inform perf to use as a clock source for its timestamps monotonic raw.
- `script_args` as the name suggests are additional arguments to `perf script`. These work analogously to the `record_args` for the record subcommand of perf.
- `trace_filters` is similarly only used in the post-processing stage of the `PerfCmd()`. `['pid','time','event','trace']` is an example array, which will then become the comma seperated argument for the `perf script` filter argument.

To give any command object the instruction to start tracing use the `runRecord()` method. During start, the Linux perf command built to run is for example `perf record -e 'i2c:*','smbus:*' -o perf_I2C_smbus.data`. Using this command to start a subprocess of which the reference is saved within the command object solves the problem of needing to run multiple commands in parallel. Here the argument given to the perf output option `-o` is also generated by the abstraction module. Because the `PerfCmd()` uses all the functionality needed to implement most tools, as well as for brevity, it is the only command class described in such detail.

In the same way to how a Linux command is built for `perf record` during start, there will be one built for `perf script` during the script call. This time the `script_args` will be used as well as the output filename of `perf script` as the input filename.

Other such commands implemented to be able to trace the full interrupt chain within EnzianBMC are `DmesgCmd()`, `PerfTimechartCmd()` and `JournalCmd()`. `DmesgCmd()` is the easiest wrapper implemented as dmesg is called once before tracing, to clear the kernel message buffer, and once at the end, to get the messages. `PerfTimechartCmd()` is analogous to the normal perf abstraction. Their only difference is that this command calls the timechart subcommand instead of the script subcommand. `JournalCmd()` is similar to the perf wrapper. All it does is call the journalctl command seen in Listing 2 for different services, the same way the `perf record` subcommand is called.

Profile class

Waiting for an actual fault in hardware to happen to trace it is not an option for the TestBench. Doing so can not be done in a reliable and reproducible way. Another issue would be that we do not want an actual fault in the hardware as this could potentially destroy the board. Because of this, the TestBench must programmatically launch an interrupt that looks to tracing the same as an actual interrupt thrown by the hardware. As this work is primarily interested in the response of the EnzianBMC to a fault interrupt, we only look at reproductions of such interrupts. Throughout the power-manager code, the response to such an interrupt is called a `scram`.

```
...
class ToProfile(ABC):
  ...
  @abstractmethod
  def doAction(self):
      # e.g. fault.direct_scram()
      pass

  @abstractmethod
  def cleanUp(self):
      pass
  ...
```

Listing 8: Most important part of the ToProfile class

The `ToProfile()` base class seen in Listing 8 is used by the TestBench to repeatedly initiate such a scram response. Which form of interrupt is seen by most of the EnzianBMC on a call of `doAction()` depends on the implementation of the respective profile.

The general idea of the abstract `doAction()` method is that once executed it initiates something looking exactly like a fault interrupt. Doing so is possible in multiple ways, two of which are explored and described in more detail. `cleanUp()` is the method that will be called after all commands stopped recording. As a forced interrupt response was never intended on the BMC this method gives the profile the possibility to clean up afterwards.

Implementations of profiles

At present two different implementations of the profile interface exist called `DirectScramProfile()` and `FanScramProfile()`. The way these two implementations coerce a scram response on the BMC differ significantly. The second profile changes some hardware configuration such that the hardware will throw a fault interrupt even without faulting hardware. `DirectScramProfile()` is called this way because all it does is call the newly added D-Bus method of the fault service. This direct method of forcing a scram response does not go through the xgpio driver as it starts after the decision logic within the power-manager.

This supplementary method called direct scram is only used for tracing and can be seen in Listing 9. Lines two through six define that this method is to be exposed onto the system D-Bus. The `direct_scram(self)` itself uses the power-manager internal function called `scram()` which contains all of the code that defines the scram response of the power-manager. Tracing the response of this forced scram on the BMC is done much the same way as described in subsection 4.1.3.

To use this newly added method, the `DirectScramProfile()` has as a reference to the D-Bus object of the fault service saved. In the `doAction()` call it then uses this reference to initiate the response with `self.fault_service.direct_scram()`. A nice characteristic of doing it this way is that no clean up is required.

```
...
 @dbus.service.method(
   dbus_interface=FAULT_INTERFACE_NAME,
   in_signature="",
   out_signature=""
 )
 def direct_scram(self):
   ...
   scram(self.logger, self.gpio, self.power)
   ...
```

Listing 9: New D-Bus method added to the fault service for easier scram response

```
1  ...
2   def doAction(self):
3     ...
4     read_temp = from_dbus(self.power_service.read_device_monitor("fans",
   ↪   "T_INTERNAL"))
5     self.power_service.set_device_threshold("fans", "T_INTERNAL", "FAULT_HIGH",
   ↪   read_temp-10)
6     time.sleep(1)
7
8   def cleanUp(self):
9     ...
10    self.power_service.set_device_threshold("fans", "T_INTERNAL", "FAULT_HIGH",
   ↪   self.reset_temp)
11    self.power_service.clear_device_faults("fans")
```

Listing 10: Primary functionality of `FanScramProfile()`

`FanScramProfile()` seen in Listing 10 works very differently. It instead uses the power service to reconfigure a faulting threshold on hardware. Specifically, the temperature sensor threshold called `"T_INTERNAL"` of the fans is changed. As can be seen in lines four and five, this profile will first read the current temperature on the `doAction()` call and then set the threshold ten degrees Celcius lower. This results in this device immediately throwing a temperature fault as now the faulting threshold is exceeded. This fault signal initiated was not one on which the power-manager called an emergency response before and hence it was necessary to change the device configuration of the power-manager to do so.

Because this profile changes the hardware configuration of the BMC to force a scram response, we now have to use the `cleanUp()` method of the profile interface. As seen in line ten of Listing 10 to reset the configuration means to set the device threshold back to the actual faulting threshold of the BMC. Additionally, the device is notified that the fault is handled by using the `clear_device_faults` method of the power service.

Environment class

Characterizing the interrupt handling response without controlling the environment in which this interrupt happens is not ideal. For this reason, the TestBench has the option to define the state of the EnzianBMC in which the tracing should be done using a so-called environment. Similar to before, we have a common base class this time named `TestEnvironment`. It implements an interface again for the rest of the TestBench while using profiles and commands to do so.

In Listing 11 the `runEnvironmentTest(self, trace_cmd:BaseCmd, to_profile:ToProfile, output_path="./")` is the only existing abstract method in this base class. The setting in which to run the characterization is set up by this method, but how this is done and what it entails for the EnzianBMC depends on the implementation. Other interesting but not abstract methods in this class are called `_prepare` and `_finalize`. The only method that has to be implemented by each environment class and will be directly called by the TestBench is the `runEnvironmentTest`. Starting the command given in `trace_cmd` with the scram profile `to_profile` at some point during this method call is a must for all implementations. Additionally, every implementation has to call `_prepare` exactly once before running the `trace_cmd` command.

```
1   ...
2   class TestEnvironment(ABC):
3     ...
4     def _prepare(self, trace_cmd:BaseCmd, to_profile:ToProfile, output_path="./"):
5       trace_cmd.setOutputPath(f"{output_path}/{self.identifier}")
6       trace_cmd.setOutputFile(to_profile.getOutputName())
7       self.gpio_service, self.power_service, self.fault_service, self.telemetry_service
    ↪    = resetPowerMng(self.logger, logLevel=to_profile.getGlobalLogLevel(),
    ↪    fixate_to_core=self.fixate_to_core)
8       to_profile.setDbusServices(self.gpio_service, self.power_service,
    ↪    self.fault_service, self.telemetry_service)
9     ...
10    @abstractmethod
11    def runEnvironmentTest(self, trace_cmd:BaseCmd, to_profile:ToProfile,
    ↪    output_path="./"):
12        raise NotImplementedError
13    ...
```

Listing 11: Most important part of the TestEnvironment class

`_ finalize` is used to finish the tracing by calling the `runScript()` method on the command and logs that this environment is done tracing. As such, if the runEnvironmentTest does this finalization step itself calling `_finalize` is not needed.

The call `_prepare` is a must, because this resets the power-manager and reconnects to the services afterward. This can be seen in line seven of Listing 11. Handing the new power-manager service references to the profile in use, which can be seen in line eight. Restarting the power-manager helps to get our system in as consistent a state as possible before each tracing run. Reconfiguring the output path is also done in this method to include information in which environment it runs.

The method `resetPowerMng` called to reset the power-manager does a few things. First it uses the `systemd` command-line interface to restart all of the power-manger services. After having restarted them and if instructed to fixate all services to a single core, it will do so by deactivating all other cores. Lastly, after reconnecting to the services using D-Bus it will then set the python log levels on each of them to the defined value in the profile and return their references.

Implementations of environments

In what state exactly the EnzianBMC is situated in during the characterization of the interrupt can change the results measured. In production, it might not be possible to decide precisely what else is running on the BMC, wherefore taking measurements in different environments is vital to have informative characteristics. The TestBench has the following three environments implemented:

- IdleEnvironment is the EnzianBMC in the idle state. Meaning nothing other than the kernel and the power-manager are running. The state the EnzianBMC is in during this tracing environment is the same as it should be in most of the time.
- TelemetryEnvironment additionally to running the tracing, will use the telemetry service to listen to some monitors. The goal is to emulate extra load on the BMC, similar to an administrator using the BMC for telemetry checking.
- LoadedEnvironment tries to do a similar thing as is done with the telemetry but on the level of Linux. This approximates well what could happen if the BMC is running other software besides the power-manager.

Seen in Listing 12 is the environment implementation of `LoadedEnvironment`. Given that this implementation covers everything the other two do, only this environment is described in greater detail. The comments in this method describe nicely the different conceptual parts of it.

```
1   ...
2     def runEnvironmentTest(self, trace_cmd:BaseCmd, to_profile:ToProfile,
    ↪  output_path="./"):
3       self._prepare(trace_cmd, to_profile, output_path)
4       # Start the dohell script.
5       os.system("rm /tmp/stopDohell")
6       os.system("./testBench/dohell.sh &> /dev/null")
7       # Do actual test
8       trace_cmd.runRecord()
9       to_profile.doAction()
10      trace_cmd.stopRecord()
11      # Stop the dohell script
12      os.system("touch /tmp/stopDohell")
13      ...
```

Listing 12: The LoadedEnvironment class' implementation of `runEnvironmentTest()`

After resetting the power-manager and setting up the command and profile, this environment implementation starts the external script called `dohell.sh`. This script is a modified version of a load generating script used by some in the Linux real-time community [46]. With the preparations of the environment done, lines eight through ten do the actual tracing of the interrupt response. Afterward stopping the load on the system, i.e. the `dohell.sh`, is done by creating the file `/tmp/stopDohell` for which the load generation will check.

Combining them into the TestBench

The combination of the three previously discussed abstractions, namely the tool wrapping commands, the scram response initiating profiles and the environment specifications amounts to the

TestBench. There are many ways of combining objects of these classes, one of which is shown in Listing 13. Any combination that works is then useful to characterize the interrupt response on the EnzianBMC.

```
 1  # A perf command for SMBus and I\textsuperscript{2}C
 2  smbus_i2c_perf_cmd = PerfCmd("i2c_perf_cmd", trace_targets=['i2c:*','smbus:*'],
 3          trace_filters=['pid','time','event','trace'],
 4          do_time_sync=False, use_clock_monotonic_raw=True)
 5  commands = [smbus_i2c_perf_cmd]
 6  # If you want the BMC to run single core (might be beneficial for tracing)
 7  FIX_SINGLE_CORE = True
 8  # Define the profiles to use
 9  profile:list[ToProfile] = [ DirectScramProfile(global_loglevel="INFO") ]
10  # Choose on which environments to run
11  environments:list[TestEnvironment] = [
12    IdleEnvironment(logger, fixate_to_core=FIX_SINGLE_CORE)
13  ]
14  additionalRunComments = ""; currentRunFolder = "./out"
15  # run only specific environments
16  doEnvironments(environments, commands, profiles, currentRunFolder)
17  # Log and shut off the power
18  logger.info(currentRunConfiguration(environments, commands, profiles,
     ↪   additionalRunComments))
19  finishTestBench(FIX_SINGLE_CORE)
```

Listing 13: Combination of parts that amount to the TestBench

This minimal example has all the pieces of the actual TestBench used to obtain the evaluations of this thesis. The only difference is that the actual TestBench script contained in `runTestBench.py` employs all existing commands, profiles and environments in multiple different configurations.

Running this minimal example, the Linux perf command is used to trace a direct scram, while the EnzianBMC is restricted to a single core and idle. Other than creating the respective objects, this is entirely done within the `doEnvironments` call on line 16. This method expects as the first three arguments the environments, profiles and commands. The last argument given to it is the path to where the output of the tracing is saved. `doEnvironments` will sequentially execute every combination of environment, command and profile. As the argument lists in this example only contain one object each, the output will only be from a single tracing run.

The two methods in lines 18 and 19 are part of the utility methods created for the TestBench. The `currentRunConfiguration` will return a string containing informing about all the configurations that are currently set. For reproducibility, this information is added to every output of the TestBench. `finishTestBench` in addition to activating all cores again shuts off the power to the CPU and FPGA on EnzianBMC. The power to those was on in the first place to power them down during the interrupt response.

4.2.2 Test scripts on BMC

In addition to the TestBench there are also some test scripts in the *onBMC* folder. These scripts were used to test various properties of the tools and the system throughout the course of this thesis.

One of these scripts is the `timingTestPerfDmesg.py` script. It is used to test the time difference of how close perf and dmesg timestamps get to the Python ones for the same event. After testing some different events, the one chosen for perf was the file close operation.

Hence the script does for the number of rounds defined a time measurement in Python followed by a file close and another time measurement. Simultaneously we use the Python abstraction provided by the TestBench to trace all `syscalls:sys_enter_close` system events. Part of the information saved by the test script during each round is the file descriptor of the file closed. Using this crucial number after the test is done we can search for the file close syscall of this specific file and read out its timestamp given by perf.

The chosen event for dmesg was directly writing a message to `/dev/kmsg`. This might take longer than a single file close operation but using an unbuffered file is still the fastest way to have a shared event with Python.

4.2.3 Part running locally

The TestBench also has a local part to where the gathered data is saved. This part of the TestBench runs the remote part on the EnzianBMC multiple times and copies back the collected data. The local scripts at the start will first copy the relevant folder containing all needed scripts to the remote EnzianBMC and only then execute them there. These scripts are contained in the *onBMC* folder and copying it first means one can run tests on multiple BMCs without having to copy the TestBench manually. This is usually the better case because of the assurance of having the same TestBench source on the BMC.

The main scripts of this local part of the TestBench are the three bash scripts `runOnRemote`, `syncDown`, and `doMultipleTestBench`. These three scripts combined with the remote part on the BMC will put every interrupt handler test run into a local folder called *syncFolder*. Each timestamped run has all the gathered tracing data for each profile and environment configured. Every single script has a specific purpose where `runOnRemote` and `syncDown` may be used by themselves for theirs.

- **runOnRemote** copies first the *onBMC* folder to the BMC. After that, it executes remotely the Python script given as the first argument. This is most often used to execute the runTestBench Python script on the configured EnzianBMC. Rsync is used to reduce testing time overhead when copying for each run.
- **syncDown** is a tiny script, as all it does is using rsync to copy over the measurements gathered by TestBench executions.
- **doMultipleTestBench** will do multiple TestBench runs on the configured EnzianBMC and copy the measurements back down. It will do the former by first checking the EnzianBMC status. If it is up, then it uses the `runOnRemote` script to run the TestBench remotely. After each time `runOnRemote` with runTestBench as an argument runs successfully, doMultipleTestBench copies back the just created measurements. Additionally, whenever the remote BMC responds strangely, this script will try to restart it over SSH[4].

4.3 Evaluation scripts

After implementing different tests and then running them multiple times, raw measurements are gathered, but nearly no insights are gained from them. These only come from the evaluation scripts,

[4] Restarting the Enzian BMC over SSH or even manually over the console might sometimes fail. The last possible solution for restarting, in that case, is using the JTAGinterface exposed by the BMC.

a group of Python scripts used to parse and evaluate the output of the TestBench. These scripts additionally combine information over many runs to make statistical statements possible. The structure of these is split into roughly four categories, namely the parse, calculation, evaluation scripts, and other utilities.

Calculation scripts are descriptively named after what they compute. The `scramDeltas.py` calculates, for example, relative time deltas for disjoint parts and how much time is approximately lost on communication is calculated by `communicationLatencies.py`. Parse scripts are those scripts prefix with *parse* and a command name as a suffix[5]. They are closely related to the command they parse because the output generated by each abstracted tool can differ widely. The utility scripts implement common functionality needed throughout other scripts like reading in and sanity checking the TestBench output file or converting and parsing timestamps.

The essential part of all of these evaluation scripts is how to parse the TestBench output. With this knowledge, it is easy to add new scripts augmenting the existing ones or gaining completely new insights.

4.3.1 Parsing the TestBench output

The goal was to lose as little tracing information from the abstracted tools as possible. To achieve this, the output of the TestBench is fundamentally unstructured. The only known fact about the output is that it contains results returned by the command and the first line defining the command that generated it. `#Output from <command_identification> follows:` is the form of this first line, where `command_identification` is the identifier defined in the BaseCmd class described in section 4.2.1 and as such in each command. When using the implemented utility command `RunAllCmd()` this line is used as a delimiter between the outputs of each subcommand. All parsing implemented is done using regular expressions in Python. Regular expressions or short regex is a way of defining a search pattern used by string-searching algorithms.

Running multiple commands in parallel is the primary way of tracing done during this work. Doing so allows the combination of the information gained by different Linux tools on the EnzianBMC during a single interrupt response. The output of such a `RunAllCmd()` seen in Listing 14 simply contains the outputs of each command abstraction. This is why understanding the separate parts of this output give an idea of how to parse the complete one.

The first two lines starting at the beginning are made by the PID utility command. Here we see that this command only returns the name of the process and its current process identifier. We have to read these PIDs for every run on the TestBench because restarting the power-manager means spawning new processes for each service. Parsing this output with a regular expression will return the mapping of PID to name.

Lines three through eleven show the reduced output of a journalctl command on the fault service. A full line of this output and how to parse it was already seen in Listing 4. These are the logging messages done throughout the fault service and hence can be exactly matched. Each of them is an interval during the scram response where the power-manager executes a specific set of operations. A description of the grouping and their naming follows[6]:

- *fan set* is the interval of the scram response during which the fans of the BMC are set to max as a security measure.
- *cpu reset* is the time used to reset the CPU. This time interval is the shortest by far and

[5] The suffix may contain information about the configuration of the command it parses.

[6] The order seen here is the same in which the sets of operations are done by the power-manager.

```
1  #Output from pid_cmd follows:
2  testBench: 7968; gpio: 8018; power: 8028; fault: 8059; telemetry: 1488;
3  #Output from fault_journal_cmd follows:
4  -- Logs begin at Tue 2021-07-20 16:14:37 UTC. --
5  [...] fan set start at 11724735.771436 end at 11724773.780939
6  [...] cpu reset start at 11724773.780939 end at 11724777.917078
7  [...] pac_fpga start at 11724777.917078 with end at 11724832.825500
8  [...] pac_cpu start at 11724832.825500 end at 11724887.731378
9  [...] finished all start at 11724735.771436 to 11724893.570735
10 [...] fault_handler start at 11724735.670636 end at 11724896.269073
11 #Output from smbus_perf_cmd follows:
12 8028 11724.739123:        smbus:smbus_write: i2c-4 a=052 f=0000 c=0 BYTE_DATA l=1 [(nil)hD]
13 8028 11724.739662:       smbus:smbus_result: i2c-4 a=052 f=0000 c=0 BYTE_DATA wr res=0
14 8028 11724.739795:         smbus:smbus_read: i2c-4 a=052 f=0000 c=3a BYTE_DATA
15 8028 11724.740501:        smbus:smbus_reply: i2c-4 a=052 f=0000 c=3a BYTE_DATA l=1 [(nil)hD]
16 8028 11724.740506:       smbus:smbus_result: i2c-4 a=052 f=0000 c=3a BYTE_DATA rd res=0
17 ...
18 8063 11724.935225:         xgpio:irqhandler: xgpio handler called, ipisr=0x1
19 8063 11724.935236: xgpio:handle_channel_irq: xgpio handle channel 1 interrupt, val=0x1fff0d5c,
    ↪  changed=0x20, mask=0x1fff0df8, rising=0x1fff0df8, falling=0x1fff0df8
20 ...
21 #Output from perf_timechart_cmd follows:
22         time    cpu  task name          wait time  sch delay   run time
23                     [tid/pid]              (msec)     (msec)     (msec)
24 ...
25 11724.736359 [0001]  python[8059]                                        awakened:
    ↪  dbus-broker[1310]
26 11724.736376 [0000]  <idle>                 0.016      0.000      1.135
27 11724.736415 [0001]  python[8059]           0.000      0.016      1.292
28 11724.736511 [0000]  dbus-broker[1310]                                   awakened:
    ↪  python[8028]
29 11724.736529 [0001]  <idle>                 1.292      0.000      0.113
30 11724.736531 [0000]  dbus-broker[1310]      1.135      0.016      0.155
31 ...
```

Listing 14: A reduced TestBench output using multiple commands.

contains the fault service communicating with the GPIO service and the latter resetting the CPU.

- *pac_fpga* and *pac_ cpu* are the sets of operations that shut down the power for both the FPGA and CPU respectively. The suffix of this name defines which part it is in the scram method as both have near identical power-manager source code.

- *finished all* or sometimes also called *inside_ all* is the time interval containing all above plus the original logging by the power-manager that the emergency response finished.

- *fault_ handler* is the time it takes the power-manager in Python to handle the interrupt until shutdown, similarly to *finished all*. In essence, this gives us the additional logging overhead we cause with all other tracing in Python. This same output is called *direct_ scram* if the scram is called directly over D-Bus and not through a fault.

Example output from a perf command tracing the SMBus and the xgpio driver can be seen in lines eleven through twenty. This output comes from filtering the recorded perf data for PID, timestamp, event, and trace. The first two always have the same form for any event and are followed by the event's name. We get the PID, timestamp, event name, and message on the initial parsing pass. Using the unique event name, it is possible to decide how to parse the message part. Parsing of the SMBus and I^2C messages is done by matching for the address *a=<address>* and command *c=<command>*. For the xgpio debug information, the parsing is done similarly for the *val=<value>* and then converting this hexadecimal value into which actual GPIO pins changed their value.

The last part of this output originates from a perf command which can be used by `perf timechart` in Linux. This subcommand of perf can create a time chart about the processes that were run on each CPU. The evaluation scripts use the same tracing information to calculate how much CPU time is lost on other processes. Using the insights gained, it is possible to say if some other processes should be set to a lower priority or ultimately killed. Given line 22, there is no further need to describe this output.

4.3.2 Gaining insights from parsed data

Having all of this data now allows us to reach our goal of characterizing the interrupt handling on EnzianBMC. Moreover, this data does not have to be only tracing output anymore, but it is augmented with additional calculations as well as aggregated over many different iterations. How to use this information to understand interrupt handling better, now depends on where the interest lies.

Indeed, one way is to look at the data returned by the calculation scripts directly. Even if helpful in some instances by giving the highest amount of detail, often this does not tell a clear enough story. The remainder of the evaluation scripts exist to give more precise insights at a first glance. These insights are then either directly printed out or worked with further. Whenever new questions arise, new parts can be added to these scripts answering them, given enough information is contained in the data. Some of the insights returned for this work are the following:

- Measured bounds on the latency of the scheduling between power-manager services. These include a comparison between different environments and the impact that prioritization of these services has.

- How much time the different grouped operations of the power-manager scram response take.

- What are the physically measured lower bounds on the SMBus for communication with hardware.

- The time from when the modified Xilinx driver receives a fan fault interrupt to when the power-manager code receives this signal.

The visualizations scripts make some of these insights clearer by producing visualizations. To do so `matplotlib.pyplot` [47] and then the `tikzplotlib` [48] package is used.

Chapter 5

Evaluation

Characterizing the response of the interrupt handler on the EnzianBMC was done in steps because of its complexity. Which experiments and how they were done to make statements about the interrupt handling response time in EnzianBMC are described in this chapter.

First, in section 5.1 the general experiments about tracing are described. These gave important information about how precise the tracing is using the newly created TestBench and they also gave the first insights into the interrupt handling response. This is followed by section 5.2 showing the results achieved using the TestBench to find inefficiencies and their improvements.

5.1 Tracing generally

Tracing is per definition on a low level and with high frequency but without a clear boundary to profiling. For the characterization of the actual emergency response, it is important to know how much overhead our tracing generates.

As such, here we take a look at the results of what takes how long to trace and log. At first, we measure the time timing in Python takes, followed by a look at the relative time between the tools and Python, and lastly, the overhead generated by logging in Python.

5.1.1 Timing in Python

Since no Python package was deemed optimal for tracing or profiling the power-manager, other options had to be tested. As discussed in section 2.4.2 this other option is the `time` module by Python.

No exact information is available about the overhead taking time in Python costs, so it was tested with a minimal code snippet. The only metric we can look at is the total execution time it takes to get the time, as the actual measurement of it during the method call is unknown. We approximate the actual measurement by assuming it is taken midway through the method.

The test for the generated overhead of the Python shipped `time` module can be seen in Listing 15. For this test we used Python's `timeit` mentioned in section 2.4.2. As can be seen, the deviation of the timing between runs is negligible compared to their absolute difference. Further, `time.clock_gettime_ns(...)` is undoubtedly very fast with 4 microseconds on average.

This code snippet is called 20,000 times locally on the EnzianBMC while the board is idle. It is crucial to do it on the EnzianBMC because the resources available can make a massive difference in the results. On my personal computer, the same code snippet takes 218 nsec for `clock_gettime_ns()`. This is a relative speed decrease of two between the embedded system that is Enzian and a last-generation gaming laptop.

```
root@enzian-bmc:# python3 -m timeit -n 20000 -s 'import time' 'timed = time.clock_gettime_ns(time.CLOCK_MONOTONIC_RAW)'
20000 loops, best of 5: 4.08 usec per loop
root@enzian-bmc:# python3 -m timeit -n 20000 -s 'import time' 'timed = time.clock_gettime_ns(time.CLOCK_MONOTONIC_RAW)'
20000 loops, best of 5: 4.06 usec per loop
```

Listing 15: Overhead test of the Python `time` module.

5.1.2 Relative time between Python and tools

Knowing the Python `time.clock_gettime_ns()` is the way to go for timing in Python, it is necessary to check whether it is actually the same relative time that other tools use. This is necessitated by our calculations using all timestamps in the traces together.

The script to test this is called `timingTestPerfDmesg.py`. Using this script with perf in the default configuration, we get the outputs seen in lines one through six in Listing 16. Default configuration here is `use_clock_monotonic_raw=False` not defining the clock source of perf in the PerfCmd abstraction of the TestBench. The same test is done for dmesg using the DmesgCmd of the TestBench as well.

```
1  Using i2c_perf_cmd is: perf record -e syscalls:sys_enter_close -o ../testing/i2c_perf_cmd/perf_perf.data
2  ...
3  -5026.0 usec before test and 5072.0 after in iteration 198; 46.0 usec execution of loop
4  -5026.0 usec before test and 5072.0 after in iteration 199; 46.0 usec execution of loop
5  The average delta time from Python time to the perf timestamp is -5026.0 usec
6  All times are in order for 200 iterations of perf.
7  Created dmesg_cmd command to run.
8  Opened the dev/kmsg successfully, starting now
9  ...
10 -5017.0 usec before test and 9585.0 after in iteration 200; 4568.0 usec execution of loop
11 -5020.0 usec before test and 9611.0 after in iteration 201; 4591.0 usec execution of loop
12 The average delta time from Python time to the dmesg timestamp is -5018.0 usec
```

Listing 16: Reduced `timingTestPerfDmesg.py` output without clock source.

Looking at the output of this test script, one can see quickly that neither dmesg nor perf return the same relative time as python with the monotonic raw clock source. We also see that the time difference to Python is more or less the same for the total of 200 test rounds carried out. Similarly, we see that the time difference between perf and dmesg is less than 20 microseconds, which leads to the conclusion that both use the same clock source, but it is not Linux monotonic raw.

```
Using i2c_perf_cmd is: perf record -e syscalls:sys_enter_close -o ../testing/i2c_perf_cmd/perf_perf.data --clockid CLOCK_MONOTONIC_RAW
...
21.0 usec before test and 26.0 after in iteration 198; 47.0 usec execution of loop
22.0 usec before test and 25.0 after in iteration 199; 47.0 usec execution of loop
The average delta time from Python time to the perf timestamp is 22.0 usec
All times are in order for 200 iterations of perf.
```

Listing 17: `timingTestPerfDmesg.py` output for perf with monotonic raw clock source.

Perf and hence also its abstraction PerfCmd give us the option to define the clock source. Using this feature in perf the relative time difference between it and the time we measure near-simultaneously in Python is very close, as can be seen in Listing 17. The about 20 microseconds difference seen between the time measured in Python and the one we read from the perf output can be attributed to mainly two things. First, the roughly four microseconds it takes Python to measure the time can not be neglected. Second, the operation synching perf and Python in this script is a file close operation, which in itself does take some time in Python and the kernel.

Dmesg, on the other hand, does not have any options to set, so the results seen before can not be improved upon. It might be possible to use two perf instances running simultaneously, one with raw monotonic clock source and one without, to somehow readjust the time of dmesg. This was never tried as directly switching to USDT probes and perf for the xgpio driver gave identical possibilities.

5.1.3 Logging overhead of Python

Similar to the question of the fastest way to determine the time in Python, we must know the effort required to store this information. Knowing this effort required the response time measurement can consider the overhead produced of taking the measurement itself. The aforementioned is approximated by measuring the time of the scram twice and incorporating the information gained from subsection 5.1.1.

The first of these two measurements is the time of the whole scram method plus logging. This time can be read out from the TestBench output as the fault_handler or direct_scram time described in subsection 4.3.1. The second measurement is done within the power-manager scram method and can similarly be read from the TestBench output. This second time is called the *inside_all_delta*. After subtracting this time from the previous one, we are left with an approximation of how long our logging took.

To figure out this average logging overhead inside the power-manager the TestBench for all existing profiles and environments is run 50 times. All improvements are already applied as this gives us the most significant impact of logging on our measurements. The only command used for these test runs is the JournalCmd for the fault service of the power-manager as this will return all needed data to be able to calculate the logging overhead.

Profile	Environment	Mean inside_all_delta	Mean total logging	Std	Max
fanScram	idleEnv	199.686	2.639	0.323	3.778
directScram	idleEnv	153.360	2.692	0.444	3.996
fanScram	telemetryEnv	191.927	2.607	0.140	2.939
directScram	telemetryEnv	157.844	2.550	0.501	5.617
fanScram	LoadedEnv	682.308	2.468	0.152	2.693
directScram	LoadedEnv	571.374	2.509	0.128	2.743

Table 5.1: Logging overhead of scram method in the power-manager in ms.

In Table 5.1 we can see the logging overhead inside the scram method of the power-manager in milliseconds across all runs. The data is given per profile in each environment, where inside_all_delta is the total time of the scram method. Additionally, you can see the mean logging overhead; the standard deviation of this logging overhead and the maximal overhead it took in these 50 runs.

Given this data, we see that logging is with about 1% of the total runtime of the scram function less overhead than normal profiling with `scalene` or `cProfile` takes. These modules will return much more information, but that additional information is currently not interesting to us. Another insight is that we do not need to take into account how long it takes to measure the time itself because with at most five microseconds doing so, it is two orders of magnitude faster.

As already logging five times results in about 1% overhead, we noticed that logging throughout Python takes a lot of time and should, whenever possible, be avoided. This insight leads directly to the log level suggestions as improvements to the scram latency in section 5.2.2.

5.2 Interrupt handler latencies

With working tracing tools and the TestBench as implemented in section 4.2 it was finally possible to characterize the interrupt handler on EnzianBMC. The modularity of the TestBench with the different environments allows running a set of tests in different system circumstances. The repeatability of the same test over multiple runs gives us experimental bounds on the expected response time of the interrupt handler.

5.2.1 Using the TestBench

As the TestBench is highly configurable, it was used to get a rough overview of where inefficiencies are during diverse states the EnzianBMC could be in. This gave hints to where improvements should be possible, which then resulted in the speedups seen in section 5.2.2. While using the TestBench, strange differences in run time between the implemented profiles were noticed. Where these differences were coming from and what the implications for the TestBench are is discussed in section 5.2.1.

Inital characterization

Throughout the creation of the TestBench some experiments were done to try out the implemented features. During those, data was measured in single runs and with a variety of different configurations of the EnzianBMC. By and large, the essential data seen was similar to the one shown below in Listing 18 and Listing 19. In both of these tables are the averaged out times over all deltas of the first entire TestBench measured over 25 runs with all useful commands in all configurations. Only Listing 19 has the xgpio to scram time, as on a direct scram over D-Bus we skip this part of the normal emergency response. At this stage, the EnzianBMC was state of the art without any improvements.

Environment	Scram method	Fan set max	CPU reset	pac FPGA	pac CPU
idleEnv	284.127	43.439	7.204	111.971	113.829
telemetryEnv	286.837	54.328	5.894	109.747	109.503
LoadedEnv	940.719	138.254	8.652	379.437	393.014

Listing 18: Initial characterization of interrupt handling in ms using `directScram`.

Multiple insights are already apparent from these experiments. The most crucial but expected one is that majority of the interrupt response time is spent in the Python-based power-manager (scram method). The roughly three to seven milliseconds seen in Listing 19 from the earliest

Environment	xgpio to scram	Scram method	Fan set max	CPU reset	pac FPGA	pac CPU
idleEnv	2.303	330.392	90.209	6.855	112.189	113.487
telemetryEnv	2.507	320.475	88.786	6.276	108.857	109.182
LoadedEnv	7.446	1236.908	333.461	7.640	456.165	415.574

Listing 19: Initial characterization of interrupt handling in ms using `FanScramProfile()`.

measurement point in the xgpio driver to the start of the scram method in the fault service is time that is hard to improve. Most of that comparatively short time is spent outside of custom code or at least in the kernel. Similarly improving the time spent in the driver or even before it will not be necessary as long as there is still an interpreted language like Python on the response path.

Something eye-catching in both profile runs is the amount of time the part of the emergency response titled *pac FPGA* and *pac CPU* take[1]. Both of them correspond to deltas powering down the FPGA and CPU power. Together, these two make up between 68% and 82% of the total scram method time, depending on which profile is used.

The time difference between the `DirectScramProfile()` and the `FanScramProfile()` in all environments is discussed later[2]. It originates from the time delta measured that set the fans to maximal power, called fan set max in in Listing 18 and Listing 19.

Looking generally at the environments, the TestBench shows using the above 3x response time of the `LoadedEnvironment` that the state the EnzianBMC is in has a massive impact on responsiveness. This also shows the importance of having a tool like the TestBench that can run the same experiments under several distinct settings. Easing the addition of new environmental settings or adjusting existing ones makes it easy to test out experiments on a multitude of settings and compare their impact. Additionally, coordinating multiple tools like perf and the power-manager reproducible would be very tedious.

Difference between TestBench profiles

During the initial characterization, we see noticeable differences in execution time between the two implemented profiles. The so-called *fanScram* is always slower than *directScram* despite there being no change other than how the scram method is called. This difference can not be attributed to the method of calling because the measured code is the same.

Through detailed inspection of the traces, additional SMBus interaction for the *fanScram* can be found. These communication messages are parsed and a reduced form is shown in Listing 20. Notice how all of them have `i2c-4` in their message. This number four is information on which bus the message is sent. On the EnzianBMC, this bus is called `PWR_FAN` and has the power and fan alerts on it. There are only four commands used in the additional messages and for the repeated commands the only change is some data bits (not shown). Further, these messages contain only two addresses, namely the addresses 0x00c and 0x052 (read out from a=<address>)[3].

From the datasheet of the fan controller MAX31785 [49] used we learn that the command 0x00 is a page change and 0x80 checks for reasons of faults. The controller needs the 0x00 because it

[1] These two intervals are prefixed with *pac* because the devices written to in the power-manager during them is so. This abbreviation comes from the sequencers type ISPPAC, which stands for In-System Programmable Programmable Analog Circuit.

[2] To figure out where this difference comes from is a deep dive into how the I^{2}C interaction with the fan controller works.

[3] The rest of the I^{2}C information traced by perf is less relevant to figure out what is happening.

```
Read on PWR_FAN command  0 with message: i2c-4 a=00c f=0000 c=0  BYTE
Read on PWR_FAN command 79 with message: i2c-4 a=052 f=0000 c=79 WORD_DATA
Read on PWR_FAN command 7e with message: i2c-4 a=052 f=0000 c=7e BYTE_DATA
Write on PWR_FAN command 0 with message: i2c-4 a=052 f=0000 c=0  BYTE_DATA ...
Read on PWR_FAN command 80 with message: i2c-4 a=052 f=0000 c=80 BYTE_DATA
... (11 times)
```

Listing 20: Parsed SMBus communication difference between *fanScram* and *directScram*

supports up to 11 temperature sensors and six remote voltages on one bus address. Combining this information allowed the conclusion that the eleven times near identically repeated commands are the checks for which of the fans faulted. Hence, as expected, the *fanScram* is generally a better approximation to an actual fault in the hardware, because this checks which sensors are faulty, which is not done during a *directScram*.

5.2.2 Improvements on latency

During the initial characterization of the interrupt handling on the EnzianBMC the TestBench was able to find a few inefficiencies. The most significant ones were further explored and in the end all successfully improved upon. The first improvement described is for inter-process communication between the power-manager services followed by the observed speedups of the different improvements compared to the state-of-the-art EnzianBMC. The last part is improvements for when the EnzianBMC is under high load.

Inefficiencies of inter-process communication

After the initial characterization, the part of the emergency response shutting down the CPU and the FPGA was improved first. As it turned out, the overhead here is caused by the inter-process communication of the fault service issuing device writes to the power service.

To test this overhead, the power service was logged on the `device_write()` method. This method is called by the fault service 20 times during the emergency shut down of the CPU and FPGA each. Because the accumulated time inside the method of the power service did not take as long, the communication was further inspected. The communication cost for this interaction was calculated by subtracting the cumulative working time used within the power service from the total time these method calls took in the fault service.

These calculations show that only about 70% of the total time for shutting down the CPU and FPGA each was spent doing so. In the interest of checking if these calculations are correct, an additional method called `device_write_multiple()` was implemented in the power-manager. This method allows to do the same as `device_write()` but for all signals at once. This meant reducing the 40 inter-process remote method calls to two. As can be seen in Figure 5.1 this consolidation of signals reduced the communication and hence the total time for these deltas by about 34.2% compared to the state-of-the-art. This corresponds to about 69.6% of the total improvements to be gained for this code path in the idle environment.

Scram latency improvements

The primary goal for doing tracing and profiling is to be able to improve the latency of the interrupt handling on EnzianBMC. This is crucial as we want the software running to react as quickly as pos-

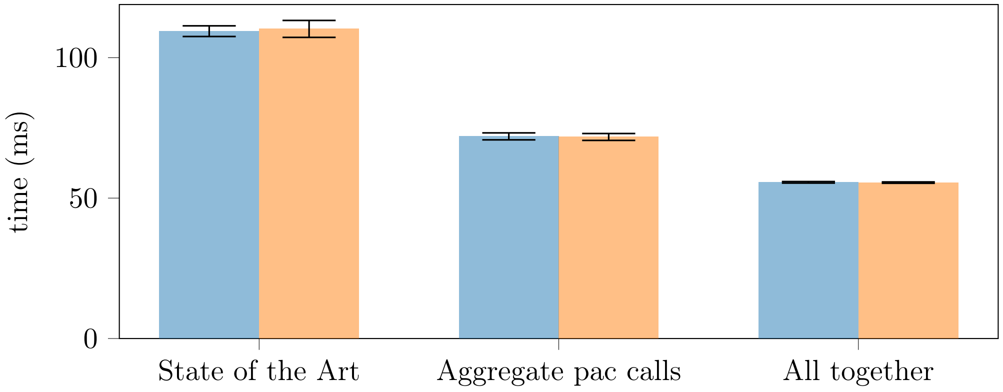

Figure 5.1: Comparision of pac delta consolidation across configurations.
Blue the FPGA pac delta and orange the CPU pac delta.

sible to hardware issues to reduce the probability of actual failures. Using the created TestBench a few inefficiencies were able to be found that can be easily fixed. Following is the list of improvements to the power-manager. Each improvement can be seen in Figure 5.2 with the total runtime of the interrupt response using it. The data for this graph was created using the TestBench over 50 runs for the idleEnv environment and *fanScram*[4].

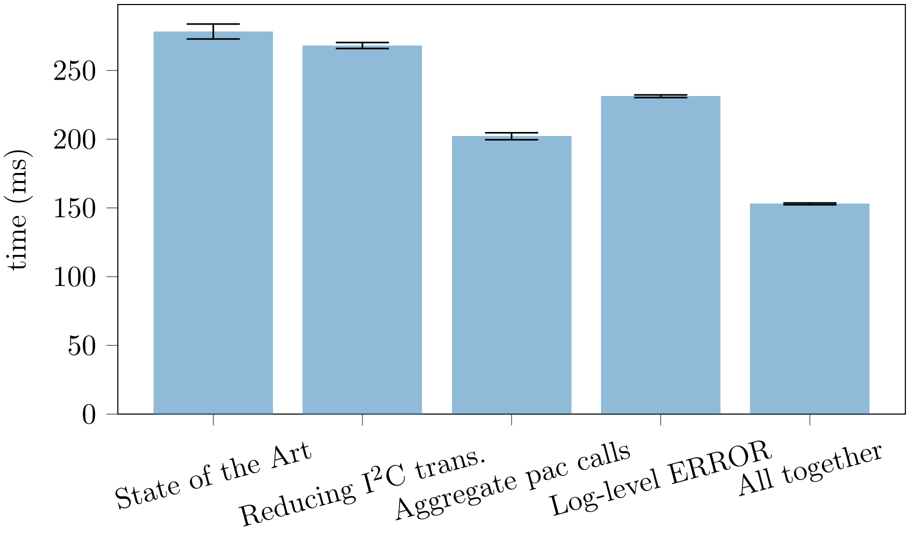

Figure 5.2: Scram method run times in milliseconds with diverse improvements.

One minor improvement has to do with the order of messages sent over I^{2}C while doing the emergency shut down. Setting the fan speeds to maximal power is the first thing the scram method does. During this fan_set, as it is called throughout this work, the power service is used first to initialize the fan controllers and then set the fan speed. Doing both directly after each other instead of two initializations first removes I^{2}C transactions for setting the correct page on the controller.

[4] This configuration of environment and profile is chosen because it is fast to run. Similar or even better performance gains are seen on all other combinations tested.

This improved ordering was found during the deep dive on the issue discussed in section 5.2.1 and its improvement on the global runtime can be seen in Figure 5.2. Even if this change is only a 3.6% latency decrease right now, it can be kept once the power-manager is rewritten in a more performant language.

Compared to the current implementation, the most significant improvement with roughly 27% speedup can be achieved by reducing the 40 inter-process communication of the power-manager services down to two.

The most accessible and still massive improvement to implement is to set the global log level for Python throughout the power-manager to the `ERROR` log level. Doing so will disable almost all logging for the power-manager, but at the same time, a speed improvement compared to state-of-the-art of about 17% can be obtained. This speedup is gained in its entirety by not interacting with journald to output logging messages and is the direct solution to the conclusion in subsection 5.1.3 that python logging is painfully slow.

The last comparison in Figure 5.2 is all of the suggested improvements combined. All of them together results in a total latency speedup of about 45% to the current implementation of the power-manager. Even with this emergency shutdown's boost in response time, the roughly 150 milliseconds are still very significant for such a vital system.

Linux scheduling influence

Process scheduling can currently have a high impact on scram latency. The Enzian BMC being an embedded system, this impact can especially be seen when the restricted resources run low. To characterize this impact, we use the TestBench to trace all scheduler events in Linux in addition to the usual information for a single run. This allows us to examine the wait time incurred by scheduling during the emergency response and how Linux schedules the power-manager services. The TestBench is configured to run only the `LoadedEnvironment`, as there, the impact can be seen the clearest. Further, this experiment is done without any other improvements.

In figure Figure 5.3 a time chart of the scheduling latencies for the critical processes can be seen. These processes are named on the left, with the suffix being the processor core on which it was run. Neither `rest_<core ID>` nor `logging_<core ID>` is a single running process. `rest_<core ID>` is the accumulation of all other processes running on EnzianBMC while the interrupt handler is characterized. The main process taking up all the CPU time seen in Figure 5.3 is the `dohell.sh` script started to simulate load on EnzianBMC. `logging_<core ID>` is the accumulation of everything logging-related like journald and perf[5]. The other five shown are, beside the three important power-manager services, the TestBench and the dbus-broker service used to communicate. The x-axis is the time in milliseconds since the start of the scram response. The red part is the wait-time of each process until it gets to run, which is in blue.

To reduce the impact of low running resources on the BMC it is possible to prioritize the critical processes like the dbus-broker and the power-manager services. Under Linux, this can be achieved by using `renice -n <priority> <PID>` where the priority value can be between -20 and 19. Anything below the base priority zero will mean it will be scheduled faster.

This renice command is used in Figure 5.4 to set the priority of all critical processes to -15 [6]. During the experiments run by the TestBench, this is possible by configuring the PIDCmd to renice them.

[5] In Figure 5.4 it is possible to see the evenly spaced logging messages the power service will do for each device write.

[6] Note that the logging is not niced, because we do not care if the log is late by even a few seconds.

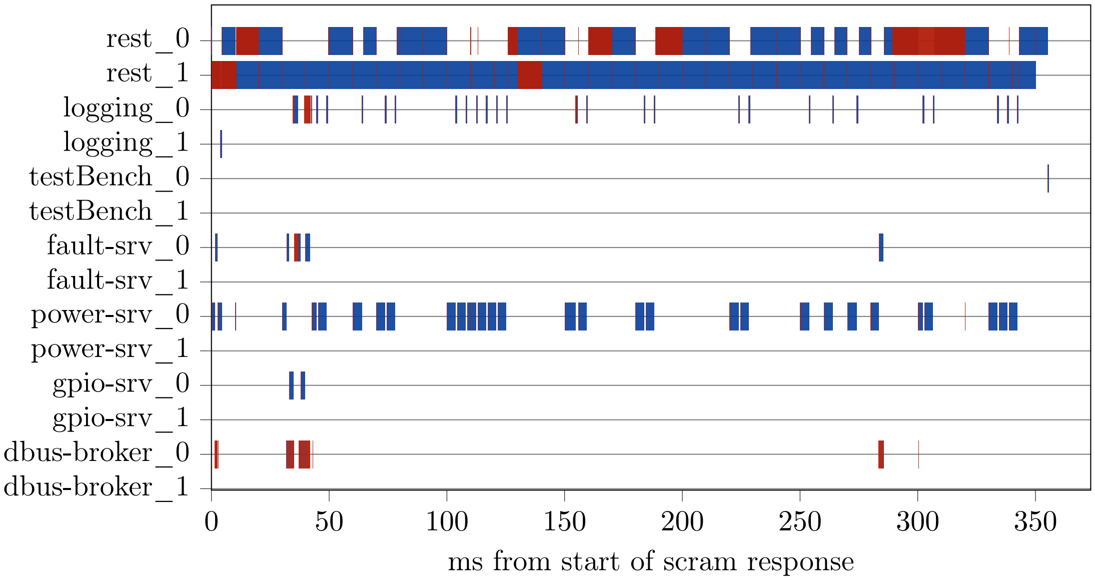

Figure 5.3: Timechart of processes during fan scram in *doHell* environment.

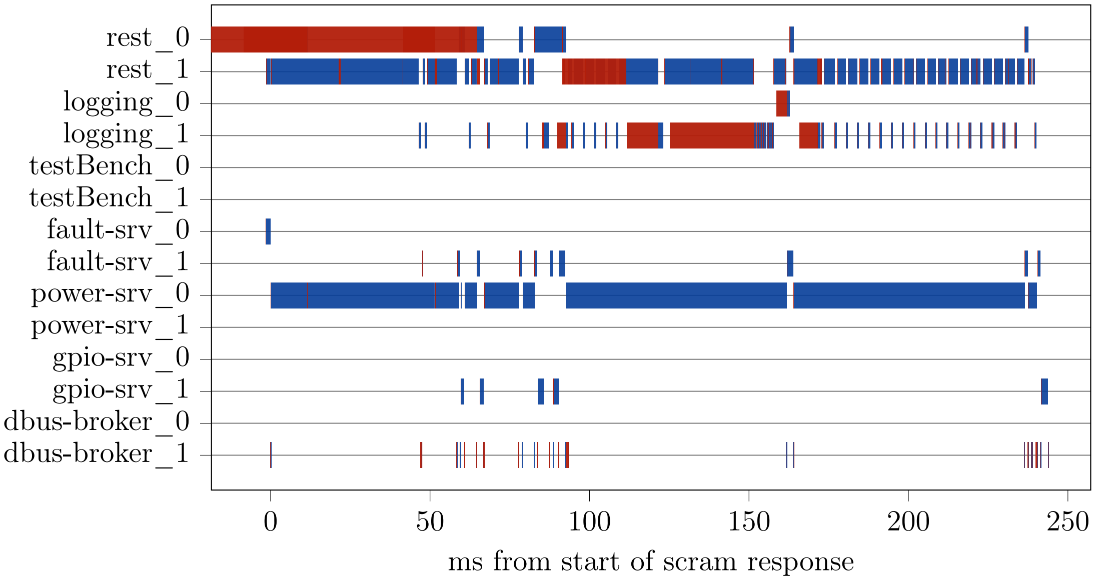

Figure 5.4: Timechart of **niced** processes during fan scram in *doHell* environment.

When looking at these time charts, particular attention should be paid to the power service on core zero and the logging processes. If the power-manager is not niced, they will share core zero, and the power service is repeatedly preempted for both logging and other processes.

Comparing the 355.4 milliseconds run in Figure 5.3 to the one in Figure 5.4 with nicing taking around 236.5 milliseconds, we see a reduction of roughly 33% of the total run time. This speedup can almost entirely be attributed to the fact that the power-manager gets with nicing always CPU time when necessary, even during high load on EnzianBMC. There is significant speedup seen precisely in the idle environment because, in those experiments, there is nothing compute-heavy besides the power-manager to run on the CPUs anyway.

Chapter 6

Conclusion

The Board Management Controller of the Enzian project has a custom interrupt handler broadly called the power-manager. The current implementation handles faults in the hardware with a response latency of hundreds of milliseconds, which could lead to the system components being damaged in case of a hardware failure. Regrettably, this latency could not easily be improved as no trace tooling to find inefficiencies worked on EnzianBMC.

In this work, the available tracing and profiling tools on Linux are first summarized to argue which are ideal for characterization. Many of the tried Yocto recipes did not work immediately. Perfs recipe was no different, but as it was comparatively easy to fix and perf still potent, we fixed it for the OpenBMC version 3.2 and with that also the current EnzianBMC. The Python logging framework was used to trace the power-manager because the other options available did not work on EnzianBMC or had too high an overhead.

Additionally, a new tool named the TestBench was created to gather many traces consistently and repeatedly. This TestBench helps counteract the fact that the Linux system is not a deterministic real-time system. The TestBench and the scripts used for evaluation are developed in a modular way enabling a straightforward addition of new data gathering or visualization methods.

This first characterization of the interrupt handling on the EnzianBMC using the newly available capabilities unearthed various inefficiencies such as logging and inter-process communication. The suggested improvements resulted in a speedup of roughly 45% for the scram response time, these are by no means the only ones possible, but only the first found and quickest to implement. Future work on this using the tooling created could yield further performance benefits.

How the TestBench currently handles gathered data is not ideal, as this results in tedious post-processing in need of revisiting. Moreover, expanding the current TestBench to more environments and profiles could lead to further insights missed by the current implementation.

Future work beside these additions could be fixing other Yocto recipes for EnzianBMC to have more tracing tools available. Similarly, upgrading the kernel EnzianBMC is based on should be considered as this would likely lead to some recipes either directly working or needing fewer fixes.

Bibliography

[1] C. Heimhofer, “Towards high-assurance Board Management Controller software,” Master’s thesis, Swiss Federal Institute of Technology Zurich, 2021.

[2] Enclustra. (2021) Mercury ZX5. Accessed on September 03, 2021. [Online]. Available: https://www.enclustra.com/en/products/system-on-chip-modules/mercury-zx5/#backtotopanchor

[3] Xilinx. (2021) Zynq-7000 SoC. Accessed on September 03, 2021. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[4] Intel. (2013) Intelligent Platform Management Interface Specification Second Generation. Version 2.0 revision 1.1; Accessed on July 02, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/servers/ipmi/ipmi-second-gen-interface-spec-v2-rev1-1.html

[5] A Joint Message from the IPMI Promoters. Accessed on May 02, 2021. [Online]. Available: https://web.archive.org/web/20210502094846/https://www.intel.com/content/www/us/en/products/docs/servers/ipmi/ipmi-home.html

[6] DMTF Redfish Developer Hub. Accessed on May 02, 2021. [Online]. Available: https://redfish.dmtf.org/

[7] (2020) OpenBMC. Accessed on July 02, 2021. [Online]. Available: https://github.com/openbmc/openbmc/tree/dunfell

[8] Eclypsium. (2019) Understanding the top 5 common firmware and hardware attack vectors. Accessed on September 05, 2021. [Online]. Available: https://eclypsium.io/wp-content/uploads/2019/09/Attack-Vector-Paper.pdf

[9] Real-Time Linux Wiki. Accessed on September 01, 2021. [Online]. Available: https://rt.wiki.kernel.org/index.php/Main_Page

[10] Yocto Project. Accessed on July 01, 2021. [Online]. Available: https://www.yoctoproject.org/

[11] OpenEmbedded Layer Index. Accessed on September 03, 2021. [Online]. Available: https://layers.openembedded.org/layerindex/branch/master/recipes/

[12] Buildroot. Accessed on September 03, 2021. [Online]. Available: https://buildroot.org/

[13] OpenWrt documentation. Accessed on September 03, 2021. [Online]. Available: https://openwrt.org/docs/start

[14] P. Mejia-Alvarez, L. E. L. del Foyo, and A. Diaz-Ramirez, *Interrupt Handling Schemes in Operating Systems*. Midtown Manhattan, New York City: Springer, Cham, 2018, https://doi.org/10.1007/978-3-319-94493-7.

[15] B. D. Gregg. Brendan D. Gregg homepage. Accessed on July 04, 2021. [Online]. Available: https://www.brendangregg.com/

[16] ——, *Systems Performance: Enterprise and the Cloud, 2nd Edition (2020)*. 75 Arlington Street Suite 300, Boston: Addison-Wesley, 2020, iSBN: 978-0136820154.

[17] *perf(1) — Linux manual page*, Mar. 2020, accessed on September 07, 2021. [Online]. Available: https://man7.org/linux/man-pages/man1/perf.1.html

[18] *eBPF*, accessed on September 07, 2021. [Online]. Available: https://ebpf.io/

[19] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. USA: USENIX Association, 1993, p. 2.

[20] B. D. Gregg. (2017) Security Monitoring with eBPF. Accessed on September 03, 2021. [Online]. Available: https://www.brendangregg.com/Slides/BSidesSF2017_BPF_security_monitoring.pdf

[21] BPF Compiler Collection (BCC). Accessed on September 03, 2021. [Online]. Available: https://github.com/iovisor/bcc

[22] bpftrace. Accessed on September 03, 2021. [Online]. Available: https://github.com/iovisor/bpftrace

[23] *LLVM Compiler Infrastructure Documentation*, accessed on September 03, 2021. [Online]. Available: https://llvm.org/docs/

[24] T. Waldekranz. (2018) ply a dynamic tracer for Linux. Accessed on September 03, 2021. [Online]. Available: https://wkz.github.io/ply/

[25] J. Levon, *OProfile*, accessed on September 03, 2021. [Online]. Available: https://oprofile.sourceforge.io/about/

[26] (2018) 4.13.2 Removed Recipes. Accessed on September 06, 2021. [Online]. Available: https://docs.yoctoproject.org/current/ref-manual/migration-2.5.html#removed-recipes

[27] *Linux Trace Toolkit: Next Generation (LTTng)*, accessed on September 03, 2021. [Online]. Available: https://lttng.org/

[28] *SystemTap*, accessed on September 03, 2021. [Online]. Available: https://sourceware.org/systemtap/

[29] S. R. et al., *ftrace - Function Tracer*, 2008, accessed on July 04, 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[30] *strace(1) — Linux manual page*, Oct. 2020, accessed on September 03, 2021. [Online]. Available: https://man7.org/linux/man-pages/man1/strace.1.html

[31] D. Inc., *sysdig(8) — Linux manual page*, accessed on September 03, 2021. [Online]. Available: https://man7.org/linl=ux/man-pages/man8/sysdig.8.html

[32] dtrace4linux, *Linux port of DTrace*, accessed on September 03, 2021. [Online]. Available: https://github.com/dtrace4linux/linux

[33] E. D. Berger, "Scalene: Scripting-Language Aware Profiling for Python," *CoRR*, vol. abs/2006.03879, 2020. [Online]. Available: https://arxiv.org/abs/2006.03879

[34] *The Python Profilers - cProfile Module Reference*, version 3.9; Accessed on September 03, 2021. [Online]. Available: https://docs.python.org/3/library/profile.html#module-cProfile

[35] *This is Python version 3.9.7*, accessed on September 03, 2021. [Online]. Available: https://github.com/python/cpython/tree/3.9

[36] M. Marchini. stapsdt 0.1.1. Accessed on September 03, 2021. [Online]. Available: https://pypi.org/project/stapsdt/

[37] *timeit — Measure execution time of small code snippets*, accessed on September 03, 2021. [Online]. Available: https://docs.python.org/3/library/timeit.html

[38] *time — Time access and conversions*, accessed on September 09, 2021. [Online]. Available: https://docs.python.org/3/library/time.html?highlight=time#module-time

[39] *trace — Trace or track Python statement execution*, accessed on September 03, 2021. [Online]. Available: https://docs.python.org/3/library/trace.html

[40] M. Marchini. TODO (mmarchini) add other architectures (this only works for x86_64). Accessed on September 06, 2021. [Online]. Available: https://github.com/sthima/libstapsdt/blob/a982e58b83cba95069a888d85f36ba21ac370453/src/sdtnote.c#L9

[41] ——. libstapsdt. Accessed on September 03, 2021. [Online]. Available: https://github.com/sthima/libstapsdt/

[42] K. Zak and T. Ts'o, *dmesg(1) — Linux manual page*, Jun. 2021, accessed on September 03, 2021. [Online]. Available: https://man7.org/linux/man-pages/man1/dmesg.1.html

[43] *systemd-journald.service(8) — Linux manual page*, accessed on September 03, 2021. [Online]. Available: https://man7.org/linux/man-pages/man8/systemd-journald-audit.socket.8.html

[44] Yocto Project - Tracing and Profiling. Accessed on August 28, 2021. [Online]. Available: https://wiki.yoctoproject.org/wiki/Tracing_and_Profiling#General_Setup

[45] S. Rostedt. (2010) Using the TRACE_EVENT() macro (part 3). Accessed on September 06, 2021. [Online]. Available: https://lwn.net/Articles/383362/

[46] I. Molnar. (2005) my 'dohell' script is embarrasingly simple. Accessed on September 03, 2021. [Online]. Available: https://groups.google.com/g/linux.kernel/c/Gc4uncIrVo4/m/Kr18NJfDiAwJ

[47] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[48] N. Schlömer. tikzplotlib - The artist formerly known as matplotlib2tikz. DOI:10.5281/zenodo.5180111; accessed on September 03, 2021. [Online]. Available: https://github.com/nschloe/tikzplotlib

[49] M. Integrated, *Maxim 6-Channel Intelligent Fan Controller MAX31785*, Aug. 2012, revision 3; accessed on September 05, 2021. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/MAX31785.pdf

[50] Yocto Project Quick Build. Accessed on July 04, 2021. [Online]. Available: https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html

[51] G. Kroah-Hartman. (2020) [patch 5.4 90/90] perf bench: Share some global variables to fix build with gcc 10. Accessed on September 05, 2021. [Online]. Available: https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg2257273.html

[52] A. C. de Melo. (2020) libtraceevent: Fix build with binutils 2.35. Accessed on September 05, 2021. [Online]. Available: https://patchwork.kernel.org/project/linux-trace-devel/patch/20200725010623.GA194964@decadent.org.uk/#23521869

[53] C. Du. (2020) perf: Make perf able to build with latest libbfd. Accessed on September 05, 2021. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/acme/linux.git/commit/?id=0ada120c883d

[54] J. Fitzhardinge. (2013) bug.h File Reference. Accessed on September 05, 2021. [Online]. Available: https://docs.huihoo.com/doxygen/linuUnknownx/kernel/3.7/include_2linux_2bug_8h.html

Appendix A

Appendix

A.1 Step-by-step guides

A.1.1 Building a fresh EnzianBMC kernel

Building a new EnzianBMC kernel means first cloning into the EnzianBMC repository on the Enzian build server. After that, following the steps outlined in the quick start guide by Yocto [50] from the building images chapter onwards is sufficient. For minor modifications and more so while developing, use the command-line `devtool`. To build the kernel, use the `bitbake enianbmc-image` command as usual. Once the devicetree_image, kernel_image, and the ubifs_image, the root file system, are ready, they can be flashed. After flashing them, it is possible to boot the new EnzianBMC.

The quickest way of doing so at the time of this thesis depends if it is only a development boot or not. If so, then using the netboot function in the EnzianBMC uboot to boot from tftpboot directly is ideal. This takes the least amount of time and allows for a quick turnaround time on kernel changes. To use this netboot function, use the command `setenv bootscript_image bmc/<username or place>/uboot.scr`. After this all that is left to do is run `dhcp` to get an IP address in uboot and `run netboot` to actually boot.

If permanent flashing to the EnzianBMC is desired use the following steps from within uboot:

- `setenv bootimage_root bmc/<username or place>/` to set the correct root directory for the rest of the commands.
- `setenv bootimage_version enzianbmc-<myversion>` sets the folder name within the root directory defined where the three files generated by Yocto need to be.
- `dhcp` will again get an IP address while in uboot.
- `run update_dtb` updates the device tree of the EnzianBMC.
- `run update_kernel` updates the kernel image.
- `run update_rootfs` puts the new root file system in the flash memory of the EnzianBMC.
- `reboot` the EnzianBMC with the newly flashed kernel.

A.1.2 Backporting perf patches to work on EnzianBMC

The first built error had less to do with perf itself but more with how newer versions of GCC (like version 10 used by us) handle globals [51]. Prefixing some global variables in the source code of perf with *bench_* and putting them in the header file fixed this sharing issue.

As Yocto recipes only call systematically different other build systems in a reproducible way the next problem had to be fixed in the Makefile used to compile perf. This is the case because some dependencies of perfs Makefile [1] changed behavior in a non-backward compatible way. The applied patch was found in a Linux kernel patch [52].

After finally being able to start compiling perf during the built time, the first actual error within perf's source came up. This time a library that perf uses changed some macros to inline functions, which meant that perf would fail to find them[2]. The patch [53] that was backported can handle both versions of the library, even if this would not be necessary for our use case of compiling perf for exactly this version of EnzianBMC.

The last error to fix before perf built for the first time successfully into an EnzianBMC had again to do with the Makefile for perf. The issue this time was that the Makefile tries to use Python 2 to silently install something, while Python 2 is not available any more on this system. We did not even search for a patch for this, as it was fixed by just deleting the offending part of the Makefile. While using perf it was not once noted that there was an issue with this step of the Makefile missing.

Even though EnzianBMC did now compile with perf added to it, perf was not very usable in that first build. This was because the kernel did not yet have tracepoint support throughout the kernel built into it. Surprisingly activating the correct flags in menuconfig to get tracepoint support broke the build again. This built error was the least informative of all and took a while to track down. In the end, it was a BUILD_BUG_ON macro[3], which breaks compilation the same way as during an actual compilation error, but only if its condition is true [54]. The condition that was true and hence created this compilation error was a $MAX\_FILTER\_STR\_VAL \leq 256$ in the include path of SMBus [4]. A fix that did result in a successful built of an EnzianBMC kernel with usable perf was just to set this filter value bound from 256 to 258.

These fixes were combined into the single patch seen in subsection A.2.1 that then can be applied during a built for EnzianBMC. After applying this patch to the EnzianBMC kernel and enabling perf if will successfully compile and be in a working state.

A.1.3 Kernel build configuration

Most kernel tracing and profiling are, as expected, not enabled by default for any Linux kernel build. It was a prerequisite to correctly configure the kernel using the Yocto kernel configuration menu before building the EnzianBMC kernel with support for tracing and profiling. Using Yocto, one can interactively do this configuration through menuconfig using `bitbake -c menuconfig virtual/kernel`, given that the Yocto built environment is activated in this shell.

Initially, while working with dmesg instead of custom tracepoints, activating the following two options at build time was necessary. The first was *Show timing information on printks* found in the printk and dmesg options of the main menuconfig. This will result in timestamps being printed for each printk message. Secondly also *Enable dynamic printk() support* is activated to compile all pr_debug messages into the kernel, which will result in a roughly 2% size increase. Alternatively, it would have been possible to add the DEBUG flag to the xgpio driver, but as the original idea with dmesg was not to modify anything about that driver, the previous option was chosen.

[1] In binutils 2.35, 'nm -D' changed to show symbol versions along with symbol names, which then messed up the Makefile of perf using nm.

[2] The exact macros changed were the bfd_section_* within libbfd.

[3] The BUILD_BUG macros should be used for code that relies on certain constants being set and to use compile-time-evaluated conditions to detect if someone changes it.

[4] The exact relative path was *include/trace/events/smbus.h*

The most critical group of flags to enable tracing can all be set within the tracers category, which itself is again a subcategory of kernel hacking.

- *Kernel Function Tracer* is needed to be able to trace every kernel function. This is possible by inserting a 5-byte No-Operation to be later patched dynamically into a tracer call.
- *Enable uprobe-based dynamic events* is activated to ensure all perf-probe subcommands can be used if desired. Generally, this option allows users to add tracing on top of userspace dynamic events.
- *enable/disable function tracing dynamically* is chosen to impact the typical performance of the system as little as possible. This is achieved by setting No-Operations at every location that ftrace can trace until enabled dynamically later on.
- *Trace gpio events* activate as the names suggest tracing events for the GPIO subsystem. This is useful to be able to trace the GPIO pins the power-manager is using.

After having set these configurations, the kernel can be built and if everything went well should have all kernel tracing functionality.

A.1.4 Applying a kernel patch to EnzianBMC using Yocto

Whenever some changes to the kernel of EnzianBMC are needed, they are added in Yocto as incremental git patches. Doing so is very easy as adding the patch file to the *meta-enzianbmc/recipes-kernel/linux/linux-enzianbmc* folder within the build directory is everything. It does need to have a unique name in that folder because the next part is configuring Yocto Project to use it. This configuration is nothing more than adding this unique file name as an SRC_URI to *meta-enzianbmc/recipes-kernel/linux/linux-enzianbmc.bbappend*. This file for this work with added tracing is shown in Listing 21.

```
SRC_URI += "file://enzianbmc.dts;subdir=git/arch/arm/boot/dts"
SRC_URI += "file://kernel.patch;patchdir=${S}"
SRC_URI += "file://xilinx-gpio-irq.patch;patchdir=${S}"
SRC_URI += "file://backport-patches-perf.patch"
SRC_URI += "file://xilinx-gpio-tracing.patch;patchdir=${S}"
...
```

Listing 21: Beginning of *linux-enzianbmc.bbappend* to configure Yocto to apply kernel patches.

A.2 Git patches

A.2.1 Git patch backporting perf changes for OpenEmbedded version 3.2

```
From 5c6396d09adefa170db65e1ace1eb524f4f9ad88 Mon Sep 17 00:00:00 2001
From: OpenEmbedded <oe.patch@oe>
Date: Wed, 14 Jul 2021 11:55:22 +0200
Subject: [PATCH] Patch for backported patches of perf and some hacking to get
 tracing to work

---
 include/linux/trace_events.h     |  2 +-
 tools/lib/traceevent/Makefile    |  2 +-
 tools/perf/Makefile.perf         |  3 ---
 tools/perf/bench/bench.h         |  5 +++++
 tools/perf/bench/futex-hash.c    | 13 +++++++------
 tools/perf/bench/futex-lock-pi.c | 12 ++++++------
 tools/perf/util/srcline.c        | 17 ++++++++++++++++-
 7 files changed, 36 insertions(+), 18 deletions(-)

diff --git a/include/linux/trace_events.h b/include/linux/trace_events.h
index 2bcb4dc6df1a..d23e8555e22a 100644
--- a/include/linux/trace_events.h
+++ b/include/linux/trace_events.h
@@ -388,7 +388,7 @@ struct trace_event_file {

 #define PERF_MAX_TRACE_SIZE        2048

-#define MAX_FILTER_STR_VAL        256        /* Should handle KSYM_SYMBOL_LEN */
+#define MAX_FILTER_STR_VAL        258        /* Should handle KSYM_SYMBOL_LEN */

 enum event_trigger_type {
         ETT_NONE                = (0),
diff --git a/tools/lib/traceevent/Makefile b/tools/lib/traceevent/Makefile
index 46cd5f871ad7..6cc2c357559e 100644
--- a/tools/lib/traceevent/Makefile
+++ b/tools/lib/traceevent/Makefile
@@ -262,7 +262,7 @@ define do_generate_dynamic_list_file
         xargs echo "U W w" | tr ’ ’ ’\n’ | sort -u | xargs echo‘;\
         if [ "$$symbol_type" = "U W w" ];then                                \
                 (echo ’{’;                                                \
-                $(NM) -u -D $1 | awk ’NF>1 {print "\t"$$2";"}’ | sort -u;\
+                $(NM) -u -D $1 | awk ’NF>1 {sub("@.*", "", $$2); print "\t"$$2";"}’
 ↪   | sort -u;\
                 echo ’};’;                                                \
                 ) > $2;                                                        \
         else                                                                \
diff --git a/tools/perf/Makefile.perf b/tools/perf/Makefile.perf
index 91ef44bfaf3e..04e947fe7a69 100644
```

```
--- a/tools/perf/Makefile.perf
+++ b/tools/perf/Makefile.perf
@@ -785,9 +785,6 @@ install-bin: install-tools install-tests
 ↪  install-traceevent-plugins

 install: install-bin try-install-man

-install-python_ext:
-        $(PYTHON_WORD) util/setup.py --quiet install --root=’/$(DESTDIR_SQ)’
-
 # ’make install-doc’ should call ’make -C Documentation install’
 $(INSTALL_DOC_TARGETS):
         $(QUIET_SUBDIR0)Documentation $(QUIET_SUBDIR1) $(@:-doc=)
diff --git a/tools/perf/bench/bench.h b/tools/perf/bench/bench.h
index 6c9fcd757f31..1cc3074d2af0 100644
--- a/tools/perf/bench/bench.h
+++ b/tools/perf/bench/bench.h
@@ -10,6 +10,11 @@
  * PA-RISC uses different madvise values from other architectures and
  * needs to be special-cased.
  */
+
+#include <sys/time.h>
+
+extern struct timeval bench__start, bench__end, bench__runtime;
+
 #ifdef __hppa__
 # ifndef MADV_HUGEPAGE
 #  define MADV_HUGEPAGE                67
diff --git a/tools/perf/bench/futex-hash.c b/tools/perf/bench/futex-hash.c
index 58ae6ed8f38b..e0af7aca2fee 100644
--- a/tools/perf/bench/futex-hash.c
+++ b/tools/perf/bench/futex-hash.c
@@ -35,7 +35,8 @@ static unsigned int nfutexes = 1024;
 static bool fshared = false, done = false, silent = false;
 static int futex_flag = 0;

-struct timeval start, end, runtime;
+//struct timeval start, end, runtime;
+struct timeval bench__start, bench__end, bench__runtime;
 static pthread_mutex_t thread_lock;
 static unsigned int threads_starting;
 static struct stats throughput_stats;
@@ -101,8 +102,8 @@ static void toggle_done(int sig __maybe_unused,
 {
         /* inform all threads that we’re done for the day */
         done = true;
-        gettimeofday(&end, NULL);
-        timersub(&end, &start, &runtime);
```

```
+        gettimeofday(&bench__end, NULL);
+        timersub(&bench__end, &bench__start, &bench__runtime);
 }

 static void print_summary(void)
@@ -112,7 +113,7 @@ static void print_summary(void)

         printf("%sAveraged %ld operations/sec (+- %.2f%%), total secs = %d\n",
                !silent ? "\n" : "", avg, rel_stddev_stats(stddev, avg),
-               (int) runtime.tv_sec);
+               (int) bench__runtime.tv_sec);
 }

 int bench_futex_hash(int argc, const char **argv)
@@ -156,7 +157,7 @@ int bench_futex_hash(int argc, const char **argv)

         threads_starting = nthreads;
         pthread_attr_init(&thread_attr);
-        gettimeofday(&start, NULL);
+        gettimeofday(&bench__start, NULL);
         for (i = 0; i < nthreads; i++) {
                 worker[i].tid = i;
                 worker[i].futex = calloc(nfutexes, sizeof(*worker[i].futex));
@@ -199,7 +200,7 @@ int bench_futex_hash(int argc, const char **argv)
         pthread_mutex_destroy(&thread_lock);

         for (i = 0; i < nthreads; i++) {
-                unsigned long t = worker[i].ops/runtime.tv_sec;
+                unsigned long t = worker[i].ops/bench__runtime.tv_sec;
                 update_stats(&throughput_stats, t);
                 if (!silent) {
                         if (nfutexes == 1)
diff --git a/tools/perf/bench/futex-lock-pi.c b/tools/perf/bench/futex-lock-pi.c
index 08653ae8a8c4..8cb33f6e9086 100644
--- a/tools/perf/bench/futex-lock-pi.c
+++ b/tools/perf/bench/futex-lock-pi.c
@@ -34,7 +34,7 @@ static bool silent = false, multi = false;
 static bool done = false, fshared = false;
 static unsigned int ncpus, nthreads = 0;
 static int futex_flag = 0;
-struct timeval start, end, runtime;
+//struct timeval start, end, runtime;
 static pthread_mutex_t thread_lock;
 static unsigned int threads_starting;
 static struct stats throughput_stats;
@@ -61,7 +61,7 @@ static void print_summary(void)

         printf("%sAveraged %ld operations/sec (+- %.2f%%), total secs = %d\n",
                !silent ? "\n" : "", avg, rel_stddev_stats(stddev, avg),
-               (int) runtime.tv_sec);
+               (int) bench__runtime.tv_sec);
 }

 static void toggle_done(int sig __maybe_unused,
@@ -70,8 +70,8 @@ static void toggle_done(int sig __maybe_unused,
 {
         /* inform all threads that we're done for the day */
         done = true;
-        gettimeofday(&end, NULL);
-        timersub(&end, &start, &runtime);
+        gettimeofday(&bench__end, NULL);
+        timersub(&bench__end, &bench__start, &bench__runtime);
 }

 static void *workerfn(void *arg)
@@ -178,7 +178,7 @@ int bench_futex_lock_pi(int argc, const char **argv)

         threads_starting = nthreads;
         pthread_attr_init(&thread_attr);
-        gettimeofday(&start, NULL);
+        gettimeofday(&bench__start, NULL);

         create_threads(worker, thread_attr);
         pthread_attr_destroy(&thread_attr);
@@ -204,7 +204,7 @@ int bench_futex_lock_pi(int argc, const char **argv)
         pthread_mutex_destroy(&thread_lock);

         for (i = 0; i < nthreads; i++) {
-                unsigned long t = worker[i].ops/runtime.tv_sec;
+                unsigned long t = worker[i].ops/bench__runtime.tv_sec;

                 update_stats(&throughput_stats, t);
                 if (!silent)
diff --git a/tools/perf/util/srcline.c b/tools/perf/util/srcline.c
index 4105682afc7a..5b1796e98866 100644
--- a/tools/perf/util/srcline.c
+++ b/tools/perf/util/srcline.c
@@ -139,16 +139,31 @@ static void find_address_in_section(bfd *abfd, asection
 ↪   *section, void *data)
         bfd_vma pc, vma;
         bfd_size_type size;
         struct a2l_data *a2l = data;
+        flagword flags;

         if (a2l->found)
                 return;

-        if ((bfd_get_section_flags(abfd, section) & SEC_ALLOC) == 0)
```

```
+        //if ((bfd_get_section_flags(abfd, section) & SEC_ALLOC) == 0)
+#ifdef bfd_get_section_flags
+        flags = bfd_get_section_flags(abfd, section);
+#else
+        flags = bfd_section_flags(section);
+#endif
+        if ((flags & SEC_ALLOC) == 0)
                 return;

         pc = a2l->addr;
+#ifdef bfd_get_section_vma
         vma = bfd_get_section_vma(abfd, section);
+#else
+        vma = bfd_section_vma(section);
+#endif
+#ifdef bfd_get_section_size
         size = bfd_get_section_size(section);
+#else
+        size = bfd_section_size(section);
+#endif

         if (pc < vma || pc >= vma + size)
                 return;
```

ETH

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):

Characterization of Interrupt Handling in Board Management Controllers

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Oberdörfer	Tobias

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

Zurich, 10.09.2021

Signature(s)

Tobias Oberdorfer

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