Diplomarbeit

Integration and Management of Automatically Generated Hardware Accelerators on the Linux OS

Andreas Wiese

31. Oktober 2015

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter: Dipl.-Inf. Michael Raitza
MSc. Nils Asmussen
Dipl.-Ing. Gerald Hempel*

* Professur Embedded Systems

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 31. Oktober 2015

Andreas Wiese

Abstract

Long time the panacea for making computers faster was increasing CPU clock frequency. As this turned out to be not feasible ad infinitum, the trend moved to increase the number of CPUs in system while retaining moderate speeds. Today it becomes apparent that this way is not infinite either. Current research expects the next phase of evolution to be hardware acceleration.

Accelerating things in hardware usually meant to build specialised hardware that does special tasks faster and hence more efficiently than a general purpose CPU. However, developing and building hardware is a expensive task. Additionally, needing to have a specialised circuit for every specialised task one might encounter is not perfect at all.

The gap between CPUs and concrete ICs is closed by FPGAs, Field Programmable Gate Arrays. These are microchips that can be programmed, and that not like a CPU is programmed, but the effective *wiring* of the chip can be modified. This allows rather cheap development and prototyping of hardware that will be actually built as an IC later, but also promises a flexible way to accelerate even exotic tasks. However, programming FPGAs still requires a decent understanding of hardware design. Various research projects exist to make programming FPGAs easier by automating it.

One example for such a research project is the GCC plugin written by Gerald Hempel at the Embedded Systems chair of TU Dresden. This GCC plugin operates on unmodified C source code and not only generates a software executable, but also tries to identify loops that make good candidates for hardware acceleration on an FPGA and outputs an FPGA programming for those loops.

These automatically generated software and hardware accelerators are currently intended to run on bare-metal, hence with full control over the hardware, no memory protection and no virtual memory management.

In this work I will implement a framework to integrate those hardware accelerators into a generic Linux system. This framework will provide a mechanism to automatically load the adjacent bitfiles when executing a program having accelerators and a kernel driver to access these accelerators from userland.

Contents

1 Introduction

For a long time, increasing performance of computing systems was primarily achieved by increasing CPU frequency. This approach turned out to be not reasonably sustainable ad ultimo, though. One reason is physics: Endless increase of core frequency is just not possible[1]. Furthermore, even though increasing core frequency in principle provide faster execution of code, this only holds true for code not containing conditional branches. High frequencies showed to be only effective in conjunction with large pipelines, which yield the problem to be only effective as long as they do not get flushed due to missed branch-prediction.

Soon the trend changed back to having moderate CPU frequencies but increase performance by having more CPUs. This also lead to the need for applications that actually *make use* of this increased parallelism. However, the observation of the further development suggests that simply increasing the number of CPU cores in a machine will not be the silver bullet, as well. Just increasing the core count again shows up new bottlenecks, e.g. increasing synchronisation effort.

Current research expects the next phase of evolution to be hardware acceleration. General purpose CPUs are not first choice for every kind of task. Although CPUs in theory are capable of calculating every calculable problem, this general merchantability comes with a price: They not necessarily do so efficiently. Digital signal processing or graphics rendering are far better done using hardware specialised to this task, namely DSPs and GPUs. Since GPUs are architecturally much more suitable for doing e.g. high amounts of the same kind of operation on different data (SIMD) in parallel, GPUs became programmable. Cuda and OpenCL allow programmers to do this kind of operations on the GPU, that is substantially better than the CPU in doing so anyway.

Furthermore, so called FPGAs or Field-Programmable Gate Arrays, microchips that are reprogrammable at runtime, are becoming increasingly affordable. FPGAs allow development and prototyping of hardware accelerators without the actual effort and expense to build actual hardware.

Unfortunately, programming FPGAs needs special expertise as it is substantially different from writing "normal" software. FPGAs are programmed using hardware description languages like VHDL or Verilog. Those languages, though high-level, are far more hardware-centric than those languages a common software developer is used to.

To open development using FPGAs to a wider audience, automating the process of generating hardware accelerators from common high-level languages gained research interest in the last decade. Though progress is made, most approaches taken so far do focus on classic embedded systems, thus applications running on "bare metal".

[1] At least not with today's technique. "Ingenieurs say, 64 `MiB` [of flash memory] will be physically possible!"

In this work, I intend to integrate the hardware accelerators automatically generated by the extended GCC developed by Gerald Hempel et al. at the Embedded Systems chair of TU Dresden into the Linux operating system.

To achieve this, I will discuss how to link generated hardware accelerators and software binary to ease the loading of the accelerators onto the FPGA on time of software execution. This should happen transparently to the end user. Furthermore, I will develop a device driver that on the hand should offer a userspace interface to enable accessing the accelerators, and on the other hand manage the loading and unloading of bitfiles on demand.

A foundation to follow the details of the document will be laid by chapter 2. It will introduce crucial termini used in the field and provide a short introduction to advanced topics that need to be taken into account to gather a functioning implementation. It will furthermore give a short overview of related research in this field and will point out similarities and differences to this works.

In chapter 3 I will discuss the aspects of the different sub tasks that need to be dealt with. I will show different approaches that could be feasible to get this tasks done and explain my considerations about pros and cons of the different approaches. In this process I will carve out why I eventually decided on different design aspects the way it turned out.

A more detailed view on some selected implementation details will be offered by chapter 4. There I will point out possible problems I had to face during the actual implementation of the design made earlier.

The remaining chapters will have a look at the bottom line and summarise what has been accomplished, and will point out problems faced during the composition of this work. They will also give an outline on what remains to be done to achieve further improvements.

2 Technical Background

2.1 Field-Programmable Gate Arrays

Integrated circuits *(ICs)* making up processors, busses, memory and eventually every part of a computer are usually hard-wired. This fact makes prototyping and developing hardware rather complex and expensive tasks. Additionally, this means that hardware usually serves a special purpose. Software programmable Central Processing Units *(CPUs)* approach this problem. Being general purpose, however, comes with a downside. The architecture of general purpose CPUs makes them perform sub-optimal for many tasks. For example applications like graphics rendering or signal processing perform a massive amount of calculation operations and gain huge benefits from parallelism that can rather easily be implemented in special purpose ICs, but hard in software. For this reason, those special applications are usually done on special purpose hardware, like Graphics Processing Units *(GPUs)* or Digital Signal Processors *(DSPs)*, respectively. Those circuits lack the ability to do general purpose processing, but perform their special task with much more performance than this achieved on a CPU. Also, they do it much more energy efficient due to their optimised purpose-focused architecture.

However, having a special piece of hardware for every task or class of problems performing sub-optimally on a general purpose CPU foils the idea of having a versatile programmable machine, not to mention the increased development and deployment costs pointed out earlier.

This problem lead to the development of Field-Programmable Gate Arrays *(FPGAs)* in the mid-80's. An FPGA can be imagined as the "blank tape"-type of IC. Its architecture allows an FPGA to be reprogrammed. In contrast to a CPU, though, this programming affects hardware logic. Where the hardware wiring of a CPU is fixed and allows the CPU to dynamically execute software code, an FPGA is an IC that can be "rewired" to do a different task—it is re-configurable hardware. This reprogramming is done by using hardware description languages to produce a *bit-file* that is then loaded onto the FPGA and represents its configuration, respectively the layout of the "generated" IC. This programming is volatile—it is preserved until overridden by loading another bit-file or power is cut.

Besides lower development and production cost of FPGA designs in contrast to specialised ICs, FPGAs have another advantage over them: Their ability to be reprogrammed allows subsequent updates of the hardware design, even in-field and without physical access to the hardware. AVM's *Fritz!Box*-brand home routers, for example, use FPGAs to implement the DECT protocol for wireless telephony. This way, they do not depend on specialised chips and additionally can ship upgrades to the FPGA configuration along with the common firmware updates. This does not only allow to fix possible bugs in the

hardware description. It also allows adaption to new protocol versions or features, which would be impossible with specialised ICs without replacing them.

2.1.1 Basic Architecture

An FPGA usually consists of an array of logic blocks, I/O pads, and routing channels. The logic blocks usually consist of a 4-input look-up table *(LUT)*, a D-type flip-flop and a two-to-one multiplexer. The multiplexer is used to select synchronous or asynchronous operation of the LUT by selecting either the flip-flop's buffered or unbuffered output path.

Inputs and outputs of each logic block are dynamically "connected" to the routing channels as described in the bit-file. The LUTs themselves are programmed to output a specific value depending on the input value.

Hardware components that are implemented as FPGA configuration are called "soft cores". "Hard cores" in contrast are actual hardware circuits built alongside the FPGA for special purposes, e.g. Input/Output Memory Management Units (IOMMUs), Direct Memory Access (DMA) engines, busses, or complete general purpose CPUs. FPGA manufacturers also like to call those cores "IP cores" (both hard and soft), where IP stands for Intellectual Property. Which parts of an FPGA fabric are actually realised as hard or soft cores is a pure design decision and heavily differs between systems.

2.2 Advanced eXtensible Interface

CPUs and FPGAs communicate over busses. Many standards for bus systems exist. One very popular bus found especially on FPGA fabrics incorporating ARM CPUs is AXI. AXI stands for *Advanced eXtensible Interface* and is a bus protocol defined by ARM Ltd. in their Advanced Microcontroller Bus Architecture standard *(AMBA)*. AXI was first defined in AMBA 3.0 as AXI3. The latest revision is defined in AMBA 4.0 as AXI4, together with an alternative lightweight subset called AXI4-Lite.

The AXI protocol describes a high performance bus to interconnect different cores, regardless whether soft or hard. It is designed for usage with high clock frequencies. The protocol uses separate address/control and data phases. It allows unaligned data transfers by byte strobe and memory bursts. Multiple cores are connected through a shared interconnect. The protocol distinguishes masters and slaves, the main difference being who initiates communication to whom. AXI itself—as is customary with ARM Ltd.—is a bus specification. The actual implementation of the bus can be both realised as soft or hard core, or as a mixture of both. For example, the Xilinx Zynq fabric I will use for my implementation, features an FPGA equipped with a hard core ARM processor. The actual AXI bus connecting both chips is implemented as soft core on the FPGA, but to connect the CPU, the CPU has of course to speak the protocol as well.

2.3 Paged vs. Physical Memory

In "classic" embedded systems, there is usually exactly *one* application, running on "bare metal". As the only application it runs on CPU's highest privilege level, which enables it to access all system's resources without any restriction. In this way, the application is able to communicate with the attached hardware, including hardware accelerators on an FPGA, by using the appropriate protocol, e.g. writing to and reading from specific registers, I/O ports, busses, or "magic" memory addresses.

With a modern operating system, the application's run-time environment significantly differs. Since an operating system usually executes multiple processes in parallel, it has to manage access to critical resources to prevent processes from interfering with each other. To do so effectively, the operating system makes use of the CPU's privilege levels: Usually, the kernel runs at the highest privilege level, giving it access to all hardware resources (so called *kernel mode*). Processes spawned by the kernel operate on a lower privilege level (*user mode*, respectively), which forbids direct access to critical resources like attached hardware. To make use of this hardware, the user mode processes have to utilise services provided by the kernel. In this way, the kernel is able to restrict access to critical resources based on security policies and to effectively *share* resources between user-mode processes.

One further mechanism to separate user-mode processes is so called *virtual memory management*, utilising modern CPUs' Memory Management Unit *(MMU)*. The MMU's purpose is to perform translation from *virtual addresses* to *physical addresses*: Each process executing on the operating system is given the impression to own the whole address space *(virtual memory)*. On memory accesses the corresponding *virtual address* is passed through the MMU, which translates this address to a *physical address* in "real" physical memory.

This method solves multiple problems an operating system has to cope with. First, it provides strict process separation. Since every process owns its own, private address space, processes are prevented from accessing foreign ones' memory, no matter whether it happens accidentally or by malicious intent. Furthermore, virtual memory management allows execution of programs without requiring them to have any further knowledge of the actual (physical) memory layout, reserved areas or areas used by other processes, or similar: process creation always starts with a fresh, completely empty, uniform address space.

On a "bare metal" system, however, access to memory is performed directly[1] by physical address. This is especially important when communicating with the automatically generated hardware accelerators. As these accelerators are directly extracted from the C code, they operate in the same memory domain as the corresponding (soft) program executing on the CPU. Memory access by the hardware accelerator will never fail as long as the originating source program does only valid memory accesses—both program parts use the same addresses to access the memory.

[1] In this context, *direct* could also mean programming the MMU to perform a one-to-one mapping of virtual addresses to physical ones, effectively eliminating the effect of the MMU.

With virtual memory management, this condition is no longer true. Where the software part of the program operates on virtual memory using virtual addresses, accelerators still have to access physical memory by physical address. If we were about to simply pass addresses from software to hardware accelerators as we do on bare metal, the accelerator would most-probably access the wrong (physical) memory.

To get a hold on this problem, there are multiple approaches to consider. The first and on first glance easiest one would be handling virtual memory like as it was physical memory. Of course, the operating system might layout every process's virtual address space congruent to physical memory layout by simply mapping virtual memory one-to-one to physical memory. Memory separation could still be held up by allowing each process to only access the physical memory belonging to it. Downside of this approach is, that one key benefit of using an MMU is lost. Virtual memory management allows to mimic contiguous memory to processes, whereas the physical memory frames forming the apparently contiguous areas can be scattered loosely across physical memory. This way, using one-to-one virtual-to-physical memory mapping would reintroduce the problem of memory fragmentation. Especially on memory constrained embedded systems, this approach would not work out well.

The second approach is much more promising. Recent architecture development introduces additional MMUs, but those being placed at the other end of the bus—so called *Input/Output Memory Management Units*, or *IOMMUs*. Where an MMU is serving for translating virtual to physical addresses for processes executing on the CPU (thus software), an IOMMU performs this task for *hardware*. Using an IOMMU, hardware as software can access the physical memory using virtual addresses. Using IOMMUs and MMU, software and hardware can—again—operate in a uniform address space. Unfortunately, though IOMMUs are becoming more common on desktop and server chip sets, availability on embedded systems is still rare.

2.4 ELF: Executable and Linkable Format

Executable programs have to comply to some operating system dependant specification, which is commonly referred to simply as the executable format. The most common executable format found on UNIX-like operating systems is the Executable and Linkable Format, or simply *ELF*. It was developed by AT&T for System V Release 4 to supersede the primarily `a.out` format and its direct successor, the Common Object File Format *(COFF)*. Today, ELF is the most-widely used executable format on UNIX-like operating systems, followed by COFF and its variants on some, mostly commercial UNIX systems, like IBM's AIX. `a.out` is also still supported by many systems, including Linux, for backward-compatibility, even though it is rarely seen today.

Besides ELF, two other prevalent executable formats are *Portable Executable*, a COFF variant used on Microsoft Windows, and Mach-O (*Machine Output*) found on Mac OS X.

The ELF binary format was initially defined in [SCO97]. Four types of ELF files are defined:

Relocatable (`ET_REL`) Relocatable files, more commonly referenced simply as "object files", are files output by the assembler. They contain machine code belonging to

one “compilation unit” in the semantics of C programming. A compilation unit usually is one file fed to the compiler. Usually, they do not contain the code of a whole program but some functional part of it. Functions in a relocatable are not usable directly, since they lack actual address information. To build an actually executable file, one or more relocatables need to be linked by running a linker on them. The linker merges the functions from the relocatables, lays them out in appropriate executable's segments, and *relocates* them—it resolves yet unresolved symbols (i.e. functions, global variables, ...), and their addresses.

Executable (`ET_EXEC`) An executable file is the representation of an actual program that can be executed. It is produced by linking several relocatable files and usually initialisation and cleanup routines provided by the system's C library. Executables can be linked statically or dynamically. Statically linked executables contain the whole program code needed during execution, whereas dynamically linked executables contain undefined references to symbols in shared objects, or libraries. Those undefined references are resolved at execution time by the run-time loader.

Shared object (`ET_DYN`) Share objects, or libraries, contain functions (potentially) used by different programs. One obvious example for a shared object is the C library, usually called `libc`. Shared objects are referenced by executables and linked to them at run-time. This avoids having every binary carry an actual copy of often used functions like `printf()`.

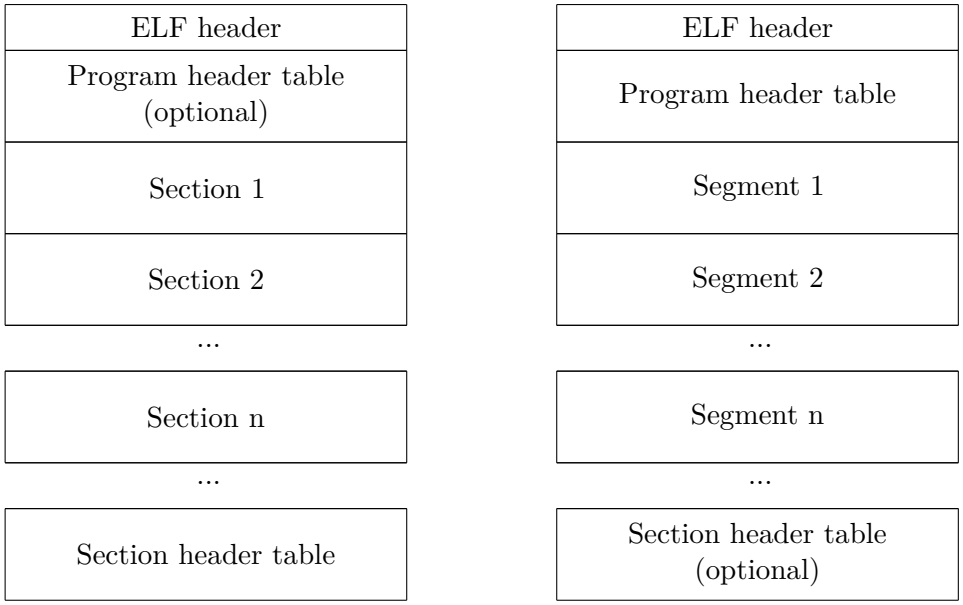
Core dump (`ET_CORE`) Core dump files are snapshots of the execution of a program. They are created by the operating system under several, usually fatal circumstances, like dereferencing pointers to memory that is not actually mapped. They are intended to help investigate why a program crashed by enabling a developer to do post-mortem inspection of the killed process's state.

2.4.1 ELF File Structure

An ELF file essentially follows a rather simple structure. The file starts with the ELF header, optional sets of program and section headers and the actual data, which is organised in segments and/or sections, depending on the point of view. From linking view, the section header table is mandatory, the program header table optional. From execution view, the opposite applies. Thus, from linking view, the file's contents are organised in sections. Execution view, on the contrary, handles the contents as segments. The schematic view of an ELF file is shown in Figure 2.1.

Even though the shown figures suggest a static ordering of headers and sections/segments, the pictured order has to be understood as common best practice. Most files follow this order, but the file format does not mandate it. Actually the ELF header points to the exact locations of the program and section header tables by offset, which makes them freely positionable inside the binary.

The ELF format has one great advantage over the older `a.out` format. It may contain arbitrary sections, which makes it very flexible and rather easily extensible.



(a) Linking View (b) Execution View

Figure 2.1: ELF file format

2.5 Related Work

Automating the generation of FPGA configurations from high-level languages is subject of heavy research for some years now. All approaches introduced so far differ from each other with variable degree. This section will introduce different works related to this subject, both in the fields of hardware generation and preparing operating systems to cope with this development.

2.5.1 Hardware: Generation and Usage

When generating configurations for re-configurable hardware from high-level languages, there are different basic approaches regarding the kind of the high-level language. First, you can obviously define a completely *new* language. This gives great flexibility, since it is completely to the language designer, what this language will look like, how expressive and powerful it is, and what restrictions it introduces. On the other hand, creating a new language limits or at least slows down its large-scale adoption, since existing code will have to be rewritten in this language.

The next approach, helping the adoption of such a system, would be extending an already existing language. The advantage of this approach is that code already existing in the base-language does not need a complete rewrite of the code, but only needs to be adopted to the newly added elements. Additionally, extending an existing language could be done by possibly extending existing compilers to understand the new elements,

instead of having to implement at least parts of the tool-chain from scratch. Of course, one's freedom of scope is heavily influenced by the existing language.

Also possible is the opposite approach, that instead of extending an existing language would *reduce* it by rejecting certain language constructs that do not map well to hardware or by adding additional constraints. This, as the extended language does, would imply modifications to existing code. For the application developer, however, this would most-probably be the more complicated way, since with adding constraints or removing constructs from a language, the expressiveness of the language usually decreases, too.

The least-invasive approach from application programmer's view would be not to touch the language itself at all. This could be done by implementing a compiler that translates to a hardware description (or hardware description language) or modifying existing compilers to do so. That is the approach taken by the compiler plugin that generates the hardware accelerators I want to integrate into the Linux OS. The plugin inserts compilation-passes into the GNU C compiler, which try to extract hardware accelerators from existing C code. It furthermore uses a hybrid approach, which builds the ordinary software binary and additionally identifies loops that make good candidates to be built as hardware. In this way, the programs still continue to work in software in case hardware is unavailable (e.g. used by another process).

2.5.1.1 Abstract Considerations

Designing a complete new language for hardware descriptions, is a path not taken very willingly. Not only is it error-prone and complex. It also requires not only a deep understanding of programmable hardware, but also decent knowledge of language design, to be successful.

One work roughly falling into this category is [Fle+15]. Though it does not describe a new language, it introduces a language-independent, abstract type system that can help designing interfaces between software and hardware. It can be understood as an abstract interface generator that tries to back further work on the topic of automatically generated hardware accelerators.

Another publication that tries to illuminate the requirements to successfully connect software running on a CPU to hardware accelerators on FPGAs is [Agn+14], that introduces ReconOS. ReconOS—other than the name suggests—is not an operating system, but defines an interface *for* operating systems to integrate and manage automatically generated hardware accelerators. This interface introduces the concept of hardware threads. It targets hardware accelerators, that mimic a complete *thread* and defines protocols to enable (conventional) software threads and hardware threads to interface in a well-known fashion, i.e. using a POSIX-threads-like API. To implement mechanisms like shared memory between hardware threads and software threads, ReconOS requires an IOMMU to allow both kinds of threads to execute under virtual memory management. Additionally, synthesising complete threads is a rather coarse scope. Our approach tries to be able to extract hardware accelerators with much more narrow scope, like for example single loops.

2.5.1.2 Extending and Reducing Existing Languages

Extending existing languages is a very common approach when trying to automate the generation of hardware. There are some well-known approaches that solely target hardware design, e.g. SystemC. *ROCCC*, as introduced in [Vil+10] is another example of a language generating pure hardware descriptions. ROCCC is a compiler tool-set based on LLVM and compiles a subset of C to VHDL. We will not discuss those languages any further here, since we are more interested in extracting hardware descriptions from software and calling the so generated accelerators from the software.

LiquidMetal, developed by IBM Research and introduced in [Hua+08] defines a runtime system called LMRT, along with a language called Lime. Lime is later described in more detail in [Aue+10]. Lime extends the Java language and tries to be backward compatible as far as possible. Lime code is then compiled to Java byte-code and FPGA accelerators. The Java byte-code is executed on the CPU, and both code executing on the CPU and FPGA accelerators can call each other through LMRT. Depending on the source code, it is possible to generate pure hardware accelerators, lacking a software part. Likewise, Lime can be compiled to pure software code. LMRT runs without an operating system on bare-metal and is strictly required to run code generated from Lime.

Nymble, introduced in [Hut+13], is another compiler for generating software along hardware accelerators from annotated C code. It is one of several related publications from the same research team. It uses the MARC II memory model, that allows highly parallel and fast, yet cache-coherent memory access for hardware accelerators. Nymble targets the Linux operating system. It generates binaries that adhere a execution model called Comrade, that was introduced in [LK07]. This publication defines some basic building-blocks for hardware accelerators. First, it introduces the *FastLane+* memory bus as an alternative to the rather slow PLB bus the used Virtex board is equipped with. More important, it introduces the AISLE memory layout to cope with virtual memory management. AISLE modifies the standard memory layout used for loading applications in a way that all memory regions containing writable data, i.e. `.bss`, `.data`, stack and heap, are located in a single, contiguous region of memory. This memory region is placed inside a DMA buffer, that is writable by the hardware accelerator. With this setup, the hardware accelerators only need to be provided the offset between the buffer's base-addresses in virtual and physical memory. All memory accesses can then be performed by the accelerators using physical addresses after applying the offset. Though a clever approach, it only works well for small programs. On the one hand, DMA buffers can only be allocated in rather small size. This is due to the fact that they require contiguous physical memory. Furthermore, DMA buffers require to be pinned in memory. This two requirements unfortunately re-introduces the problem of memory fragmentation as explained in section 2.3.

2.5.1.3 Using Unmodified Languages

One system targeting extraction of hardware accelerators of unmodified source code is LegUp, as introduced in [Can+11] and revisited in [For+14]. LegUp is a set of tools that allow semi-automatic generation of hardware accelerators from unmodified C code.

Accelerators are generated from whole functions. LegUp generates CPU software and FPGA accelerators that run on bare-metal, thus there is no approach to handle virtual memory.

2.5.2 Operating Systems: Handling Many-Core Systems

Operating systems research at the moment focuses less on integrating automatically generated hardware accelerators, but more on handling the—most probably heterogeneous—many-core systems we will observe in the near future. Research expects a drastic increase of different processor cores of different types to gain further performance improvements in systems.

This trend can already be observed in today's systems. Symmetric multiprocessing *(SMP)* is at a rise for several years now and this trend is by far not expected to end. SMP is massively entering the embedded systems world, too. A majority of recent FPGA fabrics manufactured by Xilinx or Altera include at least dual-core CPUs. With ARM's big.LITTLE technology, even the symmetry is pushed into the background by incorporating "bigger" and "lesser" CPUs into one package, i.e. CPUs with different computing power and power consumption, especially for power-management reasons on mobile systems.

To handle this development and increase scalability and merchantability of operating systems on such hardware configurations, the common trend in operating systems research goes towards multi-kernel operating systems.

Most—if not all—current operating systems all have in common, that they boot up exactly one instance of the operating system's kernel to (at least initially) manage all attached hardware components. This even holds true for current virtualisation approaches: Even though virtualisation allows starting several instances of possibly even different operating systems on the same machine, all current systems employ one central management system—regardless whether this being a host operating system or a special hypervisor. In both cases, this one instance of software manages and controls the actual hardware access. Unfortunately, this course of action proves to reach its limits in terms of scalability. Even with increasing parallelism in operating system's kernels and services, it is becoming immanent, that other approaches have to be found to further increase scalability with system designs to come. This manifests even more when considering heterogeneous systems. When incorporating different processors, that e.g. use different instruction set architectures, current operating systems tend to reach their limits.

Multi-kernel operating systems are one attempt to come up against this problems. They break with the present norm of "one OS to rule them all". Instead, a multi-kernel OS—as the name suggests—boots up a single kernel on a single processor or a subset of available processors and then spawns additional instances of kernels on the remaining processors. Every kernel gets its own subset of available resources assigned exclusively. This allows a multi-kernel operating system to execute even on heterogeneous systems by starting the appropriate instance for each available processor. In this way, it becomes possible to operate a system incorporating e.g. different CPU models, like ARM and x86, FPGAs, and GPUs as long as there are matching kernel instances capable of running on each type of processor.

The first multi-kernel operating system attracting wider attention is *Barrelfish*, introduced in [Adr08]. This publication initially discusses several probable types of heterogeneity and diversity in future system designs and introduces an approach to represent and handle this diversity. To do so, a "system knowledge base" is introduced, that serves as an abstract service to describe single hardware components and their features, e.g. supported instruction set extensions on different CPUs, or suitability for different kinds of applications, e.g. in terms of computation- or network-affinity.

Another multi-kernel OS is *Helios*, introduced in [Nig+09]. It is based on Singularity OS developed at Microsoft Research. Helios's design goals are exporting a single set of abstractions, regardless of underlying processor; transparent IPC; simple deployment and tuning; and encapsulation of disparate architectures. Helios uses a message-passing system for inter-kernel communication. Support for systems incorporating processors with different instruction set architectures is established by compiling applications into *.NET*-based, machine-independent byte-code, that is further translated to individual machine-dependant versions. Alternatively, "fat" binaries can be built, to support multiple ISAs with the same binary, which avoids additional compilation step.

The last OS of the multi-kernel kind, I want to briefly mention, is *Popcorn Linux*, introduced in [Ant14]. In contrast to Barrelfish, that is an OS developed from scratch, and Helios, basing on another *research* OS, Popcorn Linux extends the stock Linux OS, thus trying to adapt the multi-kernel approach to a "common" operating system. It extends the Linux kernel to support booting on a subset of the available hardware and to spawn additional instances of itself on the remaining subsets. To accomplish this, it implements an intern-kernel communication layer, which is used to create a single system image and implement inter-kernel task-migration. It provides a unified view of the system to applications, allowing Linux applications to operate unmodified.

The multi-kernel approach comes with some limitations, though. To have a full-fledged OS kernel running on each processor individually, the processor in question has to provide some basic hardware primitives. Common requirements include a timer, an interrupt controller and the ability to handle exceptions.

Amongst others, this requisites are one reason why the multi-kernel approach is not applicable for my task of integrating the generated hardware accelerators into an OS. Those accelerators only represent a subset of some program's functionality, not a whole processor. The intention here is completely different. These accelerators are on purpose of more "classic" nature and to be understood as generic extension hardware like a sound-card or serial controller are. They are not intended to mimic whole processors. Neither are they intended to be general purpose, even though the generation process itself aims to be.

3 Design

In this chapter I will consecutively introduce the concepts and ideas I thought out to integrate the accelerators into the Linux OS in a reliable and comfortable way. First off, I will give a short introduction of how the current interface between software and accelerators works. Then, I will consecutively identify the single important topics for this work and suggest a broad design to handle this topics.

3.1 Current Way of Operation

When the GCC plugin identified a loop as a candidate for accelerating it in hardware, it generates a hardware accelerator for this code path. To access this hardware accelerator, it injects function calls into the software. The hardware accelerators provide a number of input, output and control registers. An accelerator's actual location and quantity of input and output registers depend on the code it was generated from. To handle this diversity, the controlling (software) function is also generated by the compiler, as is the call to this function.

This function first checks, whether the accelerator is available. If not, it returns a failure, which signals the software to run the software implementation of the code-path instead.

The remaining function is simply a loop of providing the accelerator with data or retrieving data from it. Destination and source of this data depend on the code-flow of the synthesised code-path in regard of both memory location and registers on the accelerator. The state-signalling between accelerator and software is done using control registers: The software notifies the accelerator that it has finished its work by raising a control register. To wait for its own turn, the software busy-waits in a tight loop for the change of a control register that gets set by the accelerator once it is finished. When the function is done, therefore the synthesised code-path is processed, the function returns.

Currently, different accelerators are distinguished by the base address of their register window. This hopefully will change in the future, once loading of partial bitstreams will be supported. For the sake of simple code, I decided to stick with the base address for now.

In its current state, the accelerators come along with a helper-library, `libaccel.a` that handles accessing the accelerators from user-space. Core functionality of this library is to map the accelerators I/O-memory-range into the processes address space, by `mmap`ing `/dev/mem`. This library will be the entry-point to construct convenient thin-wrappers around kernel-interfaces I will implement.

To actually run a software program that is accompanied by a hardware accelerator, the bitfile containing the accelerators first has to be manually loaded onto the FPGA.

Unfortunately, there is neither a general procedure to locate nor to load a bitfile. The user has to take care of where to find the bitfile belonging to a program, as both are two different files. How to load the bitfile differs depending on the type of FPGA actually used. For the Zynq-7000-SoC I used for development, there is a Linux driver supplied by Xilinx. This driver provides a special device file to program the FPGA by simply writing the bitfile to it. If the FPGA is currently used by one application, it is not usable by other applications, except for the case that all accelerators of both programs are present in the bitfile to load. Then again, concurrent accesses to the *same* accelerator could not be detected, but it is crucial to be able to prevent concurrent accesses. If a bitfile is not loaded manually, for example because you forgot to do so, or if two programs access the same accelerator concurrently, this will have severe consequences. If an FPGA that has not been programmed yet is accessed, the system will usually "simply" freeze. If accelerators are accessed concurrently, state of the accelerator's state machine will get squashed, resulting in wrong program behaviour. Additionally, a process that reads registers of an accelerator that is currently used by another process will be able to access data of this latter process. Even worse, this is not limited to information disclosure but literal data theft: If for example the read register belongs to an output FIFO, reading from it will not only reveal a datum but actually remove it from the FIFO, thus prevent the legitimate process from retrieving this data.

3.1.1 At the Bottom Line

From this short introduction of the current state, the essential goals of this work and the design can be easily pictured. I identified the following main topics:

1. Locate and load bitfiles,
2. provide the kernel with information about loaded bitfiles and contained accelerators, and add a mechanism to interface with these accelerators,
3. provide a kernel interface for userland processes to access this hardware accelerators in a reliable, consistent way, and
4. possibly add thin wrappers around this kernel interface to make it more convenient to use.

In the next sections, I will give a deeper insight into each of this main topics and the pieces they are broken down into. I will discuss different ways to approach each problem, especially under the premise of which approach promises the most intuitive and clean design.

3.2 Handling Bitfiles

In the traditional way, FPGAs are used as a flexible drop-in replacement for (hard-wired) ASICs (Application Specific Integrated Circuit). This entails that, though they are

re-programmable in principle, actually changing an FPGA's programming is a rare event – thus the FPGA itself becomes a special purpose, stiff piece of hardware.

The possibility to simply generate hardware accelerators from (software) code predicts a major change in the methods how FPGAs will be used. FPGAs will not only be reserved for use by specialised software, but instead will be usable by any run-of-the-mill program. Additionally, this will potentially make FPGAs a non-exclusive resource, with processes competing for access and needing to update the FPGA's programming with their adjacent accelerators at system runtime.

The handling of potentially multiple bitfiles at runtime can be broken down into a series of contained problems: First off, the bitfile containing the accelerators of a specific program has to be located and loaded when executing the program. This problem could most-probably be solved both in user- or kernel-space. Furthermore, the kernel has to do book-keeping concerning the state of the loaded accelerators. For this, it first has to know which accelerators are loaded at a time, and where they are actually located (base-address and size of their register-window). Then it has to handle access to the accelerators by user-land processes. For now I will limit the discussion about user-land interaction with accelerators to the consequences for bitfile management.

3.2.1 Storing and Locating the Bitfile

This section will discuss what needs to be done to make the programs generated by the GCC plugin to behave like any other "standard" program from a user's point of view: Execution should automatically lead to the bitfile being located and loaded onto the FPGA.

First of all there needs to be a mechanism to find the appropriate bitfile when invoking the executable. The first question I want to address is where the information needed to tie together bitfile and executable should be stored. If it turned out to be necessary to store this information inside of the bitfile or the executable, which extent of modification would be needed?

Which one of executable and bitfile should be modified, if anything, quickly falls into place when thinking about the usual workflow. The executable shall behave as usual, thus it will be simply executed. The end user should not even have to know that there is a bitfile involved. Thus the executable will be the entry-point into our machinery and it should be the executable that gets spiced with information where to find the bitfile. If it was done the other way around – leaving the executable unmodified and tell the bitfile where to find it – would be like putting the cart before the horse. This would compare to looking at every single shared library in the system to find undefined references in a dynamically linked executable, instead of simply putting the list of needed libraries into the executable itself.

Not touching both files at all turns out to not increase convenience to an acceptable level. You could, for example, introduce a database where the relationship between an executable and its bitfile is recorded and on execution consult this database to find out whether there is a bitfile for the executable in question. This method would doubtlessly work, but it would also introduce a serious maintenance overhead – the database would need to be kept up-to-date and consistent. Another approach would be defining some

kind of search-path for bitfiles and introduce a method to match executables and bitfiles. Such a method could for example be using a hash function like SHA256 on the executable and name the corresponding bitfile after the text-representation of this hash value. This would also work, but you would introduce new problems, like hash-collisions or really painful maintenance of those files (for example figuring out whether the program belonging to a bitfile still exists).

Thus the most-promising approach would be storing the information about corresponding bitfiles in the executable file. The first question arising is where exactly to store this information. One way would be to wrap the executable into a container which provides the needed information. There are two principal ways to do this. Either you could extend the executable format itself or define a new executable format which provides the information and wraps the original executable. Both approaches have their benefits and drawbacks.

At a first glance, designing a new executable format that stores all needed information about the real executable itself and the adjacent bitfile seems a very flexible and clean approach. This approach would most notably be fully independent of the underlying executable format and allow embedding information about bitfiles "in" executables whose format does not allow to be easily extended, like for example `a.out`.

However, creating a completely new format has one severe drawback: It breaks compatibility with the actual executable format, which is a bitter pill to swallow. Of course, it could be worth it, if the advantages of a new format really stand out against the established format. But it is not a step that should be taken lightly. Essentially, it proves to be not an option in this particular case: The executable that is generated by the GCC plugin has the ability of executing *without* the accelerators by design – the code path is explicitly built in software as well as in hardware. This means that it could be run on any completely unmodified system that it was built for. Furthermore, the executable built is an ELF executable, which allows easy modification, and ELF is the standard executable format on Linux, so it carries no weight to be able to handle other executable formats, at least with the current state of development. Thus breaking compatibility with the existing executable format by introducing a dependency on support for a new file format would not be an option.

3.2.1.1 Extending the ELF File Format

What I will do is store the information how to find a bitfile directly inside the ELF file. The question still open is how this information should actually look like. For example, it could be a simple pointer to the actual bitfile (like a file name) or the complete bitfile itself.

In essence, the loading of bitfiles very well compares to a prominent feature already present on modern operating systems, namely shared-library loading and linking. One could even go further and say it is essentially the same thing in this case, as loading the bitfile mimics exactly the same – making a code-path available to the program, though just not in the form of program code loaded into the process's address space, but as hardware.

From this point of view it is reasonable to take this mechanism as a blueprint. This also helps to draw a line when to use which of the two approaches mentioned above (storing a pointer to vs. storing the whole bitfile inside the executable):

- Store a pointer to a bitfile if all the contained accelerators are shared between different programs.
- Store the whole bitfile if the accelerators are program-specific.

This procedure would simply mimic dynamic (pointer) and static (bitfile) linkage.

As described in section 2.4, the Executable and Linking Format ELF allows the insertion of arbitrary data through sections/segments. Sections and segments always have a distinguishable type. A section's type is denoted as an integer of 32 bits width in the section's entry in the section header, or segment's entry in the program headers, respectively. For both type fields there are reserved ranges to allow the definition of operating system or processor specific types (`0x60000000-0x6fffffff` and `0x70000000-0x7fffffff`, respectively).

To embed a complete bitfile in the ELF file, I defined both a section header and program header type of `0x68777475`, thus in the OS-specific range. This value allows to easily identify the bitfile sections. I chose this value because it lies in the middle of the reserved range and is quite random. This minimises the risk of colliding with other types probably defined in other system environments, as those usually start at the begin or end of the allowed range.[1]

The section inserted not only contains the raw bitfile. It is prepended by a header containing some additional fields. The C-representation of this header I called ZwoELF is depicted in Listing 3.1. The `checksum` field contains the cryptographic hash (SHA256) of the `data` field. It is used to detect corruption of the contents of this field, and for identifying purposes in the kernel. The `version` field is intended for consistency checking between different toolchain components, primarily to avoid interface incompatibilities with different versions. It is currently unused. The `data` field does not only contain the bitfile, but a device tree blob followed by the bitfile. The purpose of the device tree blob will be explained in detail later. The `dtbfile_len` and `bitfile_len` fields contain the length of this blob and the bitfile inside the `data` field. In addition to the type fields in section and program headers, the inserted section will be named `.tudos.hwacc.XXXXXXXX`, with `XXXXXXXX` replaced by the first digits of the `checksum` field represented in hexadecimal.

```
1 #define ZWOELF_DIGEST_SIZE      SHA256_DIGEST_SIZE
2 #define ZWOELF_VERSION_LEN      32
3 struct zwoelf {
4     __u8  checksum[ZWOELF_DIGEST_SIZE];
5     __u8  version[ZWOELF_VERSION_SIZE];
6     __u32 devtree_len;
7     __u32 bitfile_len;
8     __u8  data[0];
```

[1] Additionally, it represents the string "HWTU" when interpreted as ASCII code points, which neatly describes, what this section is.

```
9 };
```

Listing 3.1: ZwoELF header

So the only thing that remains open would be storing references to "shared" bitfiles. Storing this information would be a straight forward task: You could just add another section. There would not be much information needed, a list of file names of needed bitfiles would be enough. Then again, there would be an additional mechanism needed to traverse this list and resolve this dependencies. As I already determined, there already exists a well functioning system to handle such types of dependencies, the system's ELF loader and runtime-linker.[2] The embedding approach described above is not only applicable for ELF executables, but also for ELF shared objects. Hence, the best way to implement shared bitfiles would be generating a shared object, add the bitfiles to this shared object and link the relevant executables against it. This work, however, will not discuss the topic of shared bitfiles further.

3.2.2 Loading the Bitfile from ELF

Even though we have a bitfile stored in the ELF binary, it will not be loaded onto the FPGA yet. To accomplish this, the initial idea of this thesis was to implement this functionality in the runtime loader. This piece of software usually called `ld.so` is intended for resolving unresolved symbols contained in a executable's program code. For this to work, the ELF executable contains a special section listing the needed shared objects, which is generated by the linker when putting the executable together. Furthermore, a special segment (typically displayed as `.interp` by `readelf` and friends) is placed inside the executable, which contains the full path to the runtime loader (also referred to as *interpreter*, hence the segment name).

When executing the program, the kernel looks for this `.interp` segment and – if present – not only maps the segments of the binary itself into the process's address space, but additionally maps the interpreter into it. It further arranges the process to not begin running at the program's real entry point, but at the interpreter's by setting the process's instruction pointer as appropriate. Thus, the interpreter is ran before the program itself on the program's behalf and maps the needed libraries into the processes address space. It then gives control to the actual program by jumping to the program's code real entry point.

So our idea was to teach `ld.so` awareness of the ZwoELF section so that it could parse it and load the it onto the FPGA through the appropriate kernel interface (in case of the used development board this would be writing the bitfile to the `/dev/xdevcfg` character special device offered by the Xilinx driver mentioned earlier). As it turns out, doing it this way is not as easy as it seems because this approach does not match well with the way how `ld.so` is actually invoked. It is not – as initially supposed when drawing up this work's topic – called with the actual executable as an argument, but instead directly loaded into the processes address space as described above. There, it only operates on the program segments in memory that the kernel mapped there, which includes the

[2] "Well functioning" might not be a suitable description on every operating system, but on Unix derivatives it usually is.

sections containing information about needed libraries. Of course, this could be easily fixed by marking the ZwoELF segment as loadable and thus allow access to it from `ld.so`, but this approach would still impose other problems I want to explain briefly: Having `ld.so`, thus a user-space process, running on behalf of a user, fiddle about the programming of the FPGA seems not to be a good idea at all. It does not possess any reliable information of whether there are already accelerators loaded, where exactly they are located (if partial bitstreams were supported), or who is currently using them. The other way around, the kernel does not know anything about changes made by the `ld.so` either and would need an interface to be notified about changes. Altogether, this way of operation seems at least very error-prone and not in any way elegant.

In the end, the kernel will be in charge of book-keeping of loaded bitfiles and accelerators provided by them. This is mainly due to the fact that it has to guarantee that it keeps the system in a reliable state. It cannot allow a process to interfere with another process. Modifying the programming of the FPGA is not a task that should be allowed for a potentially untrusted user-space process. Hence, the kernel itself is a far more better candidate for loading the bitfiles. So in the next step I decided to add support for the ZwoELF section/segment into the kernel internal ELF loader.

As it turns out, the Linux kernel's subsystem implementing executable loading is indeed the place to implement this mechanism. Binary loading in the kernel works by implementing and registering a binary format. Examples for binary formats are `binfmt_elf` or `binfmt_aout`, which implement loading ELF and `a.out` binaries, respectively.[3] When registering a binary format, it is inserted in a list of formats supported by the kernel. Upon execution of a binary, each entry of the list is called sequentially. Depending on the format's return code, the traversal stops or continues: The handler returns `-ENOEXEC` when the binary's format is not recognised and the kernel should try the next handler. Traversal stops on any other error, or if the handler returns `0`, in which case it successfully executed the binary. If the end of the list is reached without any handler returning anything else than `-ENOEXEC` this error code is returned to userland. Binary formats may be compiled into modules and thus registered with the kernel at runtime.

So one option for me to implement bitfile loading would obviously be extending `binfmt_elf` to allow it to parse and load the new section. But the `binfmt` subsystem's mode of operation even allows a far more elegant way: When registering a binary format, you can choose the end where to insert it in the list of available formats by either inserting (beginning of list) or appending (end of the list). Thus the least invasive method to solve our problem would be creating a new binary format, that has a minimal understanding of ELF – just enough to identify the ZwoELF section – and thus can trigger the further loading of the contained bitfile. This format can be inserted in the list, leading it to be called *before* the actual `binfmt_elf` and just return `-ENOEXEC` to continue traversal of the list.

In this way, knowledge of the ZwoELF section stays optional (no modification to existing kernel code required) and the effort to build support for our accelerators remains easy as well, since the modules can stay and be built out-of-tree. I decided to keep the

[3] Their implementations can be found in `fs/binfmt_elf.c` and `fs/binfmt_elf.c` in the Linux source tree.

parsing of the ZwoELF section from the ELF file and the actual handling of bitfiles and accelerators separate. Main reason is that the actual loading and managing of bitfiles is a rather complex task, involving the appropriate book-keeping and maintenance work. This task is more appropriate for a separate driver that also drives the actual accelerators.

3.3 Managing Bitfiles and Accelerators: Maintenance

Having the previous parts in place, we are able to store the bitfile inside the ELF binary and get it out again. What needs to be done now is actually loading it onto the FPGA. This task comes with a number of other points that need to be thought of, as programming the FPGA is not an end in itself – in the end we want to *use* accelerators, not only load them. Essentially, we want to be able to manage a number of "slots" on the FPGA that may be programmed individually – the long-term goal of the GCC plugin is to be able to generate partial bistreams, allowing to have multiple bitfiles loaded onto the FPGA at once and hence allowing multiple programs to run in parallel, all bringing only their own accelerators, without the crutch of having multiple independent binaries carrying a bitfile containing a lot more accelerators for other binaries. Over time these slots will be populated by executing programs containing a bitfile. Each bitfile will provide one or more actual accelerators, which will be used at some point in time and unused at another. Eventually all slots will be populated and there will come just another program containing a bitfile that somehow needs to be loaded somewhere. At this point, the driver will have to choose a slot that can be reprogrammed without interfering with any running process. Hence the driver needs to track which accelerators are currently used. Then, even if it eventually elected a slot to be reprogrammed, it has to make sure, that reprogramming happens in a safe manner, preventing race conditions.

Furthermore, simply loading a bitfile does not make its accelerators spontaneously work. The I/O memory containing the accelerators' register window will have to be mapped first. This poses the next problem: A bitfile is an opaque object to the kernel. It cannot easily deduce what kind of hardware is exactly "in there" from simply looking at it. So where to know from, at which exact memory location an accelerator's register window actually *is*? This is where the `dtbfile_len` field from the ZwoELF section comes in.

3.3.1 Short Excursion: The Device Tree

To allow an operating system to handle hardware resources, you need to provide some essential information to it about the properties of the hardware. Usually, there is some kind of well-known mechanism to derive this information. Information about essential core-components can be obtained on most systems by querying the machine's *firmware*. The exact way to do so is architecture dependent, on PC-compatible systems, for example, you basically either have the ancient PC BIOS *(Basic Input/Output System)* or more recently an UEFI BIOS *(Unified Extensible Firmware Interface)*. This firmware gives the kernel a basic understanding of the machine it is running on, like which kind of

busses are present (PCI, ISA). Then, plug-and-play capable busses like PCI offer another bus-specific mechanism to gather information about connected devices.

On embedded ARM systems, there usually is no standardised firmware. Those systems usually simply start some software from some flash memory, usually a boot loader. On these systems, Linux uses a *device tree* to gather information about installed components. The device tree's origin is *OpenFirmware*, a firmware standard prevalent on machines deploying PowerPC or Sparc CPUs. OpenFirmware presents the information about available components through a file-system like tree, that can be queried by the operating system in a well-defined way. The device tree model simply mimics this tree, without the underlying firmware. The actual tree has to be provided by other means. This is done using *device tree blobs*, or DTBs. A DTB, also know as flattened device tree, is generated from a device tree source file (DTS, accordingly), an ASCII representation of a tree. A (shortened) example for such a device tree source file can be seen in Listing 3.2.

```
/dts-v1/;
/ {
    compatible = "xlnx,zynq-7000";
    #address-cells = <1>;
    #size-cells = <1>;

    cpus {
        #address-cells = <1>;
        #size-cells = <0>;

        cpu@0 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <0>;
            clocks = <&clkc 3>;
            clock-latency = <1000>;
            cpu0-supply = <&regulator_vccpint>;
            operating-points = <
                666667  1000000
                333334  1000000
            >;
        };

        cpu@1 {
            compatible = "arm,cortex-a9";
            device_type = "cpu";
            reg = <1>;
            clocks = <&clkc 3>;
        };
    };

    amba {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        interrupt-parent = <&intc>;
        ranges;

        devcfg: devcfg@f80007000 {
```

```
40              compatible = "xlnx,zynq-devcfg-1.0";
41              reg = <0xf8007000 0x100>;
42          };
43      };
44
45      ...
46  };
```

Listing 3.2: Zynq-7000's device tree source (excerpt)

I am not going to dive into too much details here, but only explain the most important aspects for this work. As you can see, this file spans a tree, beginning at the root node `/`. This root node has two (pictured) direct descendants, `cpus` and `amba`. The former node again has two descendants, the latter node has one descendant. Since node names have to be unique, they are usually suffixed by an @-sign followed by some kind of the node's base address. Node names have to be unique. The for this work most important things here are the fields `compatible`, `#address-cells`, `#size-cells`, and `reg`.

Let us have a closer look at the `amba` node and its child. The first important thing can be dominantly seen at the child node: Nodes can have labels attached. Though the child node's actual node name is `devcfg@f80007000`, it could be also referenced by the label `devcfg`. A reference can be seen at the `interrupt-parent` field of the `amba` node. This will become important later.

The `compatible` field is a simple string. It specifies what device the current node is compatible with. It is common practice to adhere to the format `"<vendor>,<device>"` for this field. The `cpu@0` and `cpu@1` nodes, for example, are specified to be compatible with a Cortex A9 CPU from ARM Ltd. in this example.

The remaining fields I want to explain further, `#size-cells`, `#address-cells`, and `reg` are tightly coupled. `reg` specifies some kinds of registers through which a device can be accessed. The interpretation of this field depends on the values of the other two fields. Those two fields always specify the interpretation for the *children* of the node they appear in – reasonable, as the bus determines the addressing scheme for the components connecting to this bus. Basically, `reg` is an n-tuple of 32 bit values. This n-tuple specifies one or more register windows, each having a start address and a window length. Adjacent values describe a register range. The `#address-cells` field specifies how many adjacent 32 bit values are interpreted as `one` start address, the `#size-cells` field does the same for the directly following size of the register range. If you had a node with `#address-cells` being two and `#size-cells` being one, and you had a child node with a `reg` field of `<0xdeadbeef 0xc0fe0000 0x100 0xbaadf00d 0x10100000 0x2000>` the device would have two register ranges, one starting at (bus local) address `0xdeadbeefc0fe0000` and a size of 256 bytes and a second one starting at `0xbaadf00d10100000` and a size of 512 bytes.

DTS files are compiled into DTB files. A machine-specific DTB file is then usually provided to the kernel by the boot loader, or directly linked into the kernel.

3.3.2 Managing Resources

The device tree is primarily used on resource restricted devices, especially embedded systems – which other systems cut down on a *firmware*? As these devices usually are very static in their configuration – you do not usually add or remove hardware to or from them – the device tree used to be a static resource, too. Fortunately, the times are changing. As SoCs become more and more powerful and general purpose, the Linux kernel recently added mechanisms to dynamically add or remove nodes to or from the tree at runtime to prepare for hot plugging. By using device tree changesets or – even more comfortable – device tree overlays one can easily modify the tree.

To tell the kernel the contents of a bitfile, I will use device tree overlays. These are changesets consisting of device tree fragments, that can be atomically added and removed from the live tree. Each device tree fragment specifies a target to act upon. The fragments will be compiled into a DTB file and inserted into the ZwoELF section's `data` field besides the bitfile. The `devtree_len` field specifies the length of the file. The DTB will contain one fragment for each accelerator in the bitfile. The target of the fragment will be the `devcfg` node – which actually is the Xilinx interface for reprogramming the FPGA. Each fragment will add a child node to the `devcfg` node specifying the start and length of the register window of the concrete accelerator. The `compatible` field of the nodes will be set to `"tudos,hwacc"`. The driver will register itself to be capable of driving this devices, thus get called by the kernel if these devices are added or removed.

Thus if a new ZwoELF section is to be loaded, the driver will do the following:

1. Find an available slot; if programmed, release I/O regions of all currently programmed accelerators.
2. Extract the DTB from the section, and prepare an overlay from it.
3. Extract the bitfile from the section and load it onto the FPGA.
4. Apply the overlay to the live device tree, triggering a device probe.
5. For each device (hence accelerator) probed, retrieve the base address and length of its register window, claim this I/O region.

To avoid race conditions and prevent the driver from reusing a slot whose accelerators are currently in use, I will use well-known mechanisms like reference counters and locking. All in all, managing different slots and bitfiles is not a highly sophisticated task from a design perspective – it is the usual book keeping. I will describe the covert pitfalls in this concrete case in more details in subsection 4.3.1, as they are more of technical nature than a design issue.

3.4 Userland Access to Accelerators

What is still left missing now is an appropriate interface that allows userland processes to interact with the accelerators. The canonical way to access device drivers on Linux (or Unix-like systems in general) are so called device special files. These are special

file system entries that appear in the file system , usually located in `/dev` and its subdirectories. There are two types of this device nodes, that differ in the type of access pattern demanded by the underlying device. *Block* devices are meant to be accessed in blocks of a device-specific size, where *character* devices are character oriented. Besides this type, a device special file has a major and a minor number, which identify the driver subsystem and the device instance the device file represents.

Devices files can usually be accessed by userland like any regular file (i.e. by the standard access methods like `open()` and `close()`, `read()` and `write()`, and so forth). When registering a device special file, a driver provides a `struct file_operations` (also known as *fops* structure in Linux kernel jargon) to the kernel. When a userland process calls an I/O function on a file descriptor obtained by opening the device special file, the kernel checks whether the function in question is provided by the driver through this *fops* structure and calls it; otherwise it uses a default action for the function in question (in most cases this default action returns an error code indicating that the call in question is not supported by the device driver). In this way, an interface between userland and driver is established.

Which I/O operations to implement in a driver is essentially a pure design choice. Of course is it advisable to maintain conformance to the functions' semantics. As a very exaggerated example, it could be a bad idea to reverse the semantics of `read()` and `write()` in your driver. So how to map our accelerators to character devices and which I/O functions would fit the accelerator's method of operation in the best way?

As described in section 3.1, access to the accelerators happens by reading from and writing to the registers an accelerator provides. To access one register of an accelerator, the userland process somehow has to specify this register. A register is essentially identified by its address in I/O memory. This address could be given directly or specified indirectly through the base address of the accelerator's register window and the register's offset, that is the number of the register.

Currently, the functions the GCC plugin generate to access the accelerators use register numbers, which makes perfect sense since this method keeps the whole function call independent from the accelerator's eventual location: The base address only has to be provided at runtime somehow.

First thing to define for our userland API is in which way accelerators should be mapped to device files. This decision also is directly linked to the choice which I/O functions to employ in the end. Possibilities range from having one device file and do interaction with all accelerators through this one file (like `/dev/hwacc`) to having one device file for every register of every accelerator (like `/dev/hwacc42:23` for "accelerator no. 42's 23rd register"). The middle course would be having one device file per accelerator, like `/dev/hwacc@c0fe0000` for access to the accelerator at base address `0xc0fe0000`.

Having a device file for every register is a rather bad idea. It would, of course, make it easy to interface with this one register – you could simply implement the usual `read()` and `write()` calls, as it would be obvious which accelerator and which of its registers is supposed to be accessed. Nonetheless, it would not work out: The size of the register window of each generated accelerator is currently fixed at 1 MiB, which (with a register width of 4 bytes) would yield a number of 262,144 potential registers for *each accelerator*. Of course, the actual number of registers an accelerator really has and which register

numbers those registers are located at is known at the end of the generating process. Thus it would be possible to only allocate device files for actual registers. However, this would make it necessary to actually extract this information and somehow provide it to the kernel. This would – at least for now – counteract the initial approach of this work to keep things simple. Furthermore, not all components of the accelerator that are accessed through registers fit into this semantic. The FIFOs used for input and output, for example, do not only provide exactly *one* register to transfer data to and from them, but a range of registers that act more like a buffer[4]. Generally, this approach would be semantically questionable. It does not make a lot of sense to have different devices to access the same accelerator, since accesses to the registers are not independent of each other.

The differences between having one device file per accelerator or one device file altogether are negligible for a simple reason: They both do not fit into the semantics of the standard I/O functions very well. Ideally, one would like to simply do `read()` and `write()` on a device file. Unfortunately, these calls do not provide enough information where `exactly` one wants to write in both cases. As this information cannot be passed through function parameters – there simply are no such function parameters – the only way for the kernel to gather this information would be having a look at the device file's name when `open()` is called (see above), or finding some other interface that better fits than `read()` and `write()`. As it turns out, there really is no better interface, but the most generic one: `ioctl()`. `ioctl()` is intended to do low-level device-specific operations on a device. Its interface is as generic as it gets: It takes two parameters (besides the ubiquitous file descriptor) , a command number and an arbitrary argument as parameter for the command, which can be understood as a variable parameter list: If needed, it can be a structure providing the remaining parameters. needed. From the semantics' point of view, the only difference between having one device file for all or for each accelerator(s) would be exactly one parameter in this structure: the accelerator's base address. As it is a lot easier to just handle one device file (the kernel offers some shortcuts for this case), I chose this option. This character device file will be `/dev/hwacc`.

As indicated, the actual I/O interface I will use to implement the userland interface is `ioctl()`. The userland `ioctl()` prototype is shown in Listing 3.3, as the kernel's counterpart is in Listing 3.4.

```
1 #include <sys/ioctl.h>
2 int ioctl(int fd, unsigned long request, ...);
```

Listing 3.3: Userland's `ioctl()`...

```
1 /* from <linux/fs.h> */
2 struct file_operations {
3     ...
4     long (*unlocked_ioctl)(struct file *f, uint req, ulong arg);
5     ...
6 };
```

Listing 3.4: ... and kernel's counterpart.

[4] This is necessary to successfully implement DMA (Direct Memory Access) on the Zynq-7000-SoC, since the SoC's DMA controller's cyclic mode is broken.

Do not be mislead by the fact that the userland function has a variable argument list. It is mandatory to specify at least one further argument and any excess arguments will be ignored. As the kernel's counterpart suggests, this argument is just interpreted as an `unsigned long`. The apparent variadicity is just a hack to make this argument completely opaque to C's type-system, allowing to pass any type of argument, regardless of whether it is passed as a pointer or a *real* value. The function given as `.unlocked_ioctl` on kernel-side has to interpret the argument depending on the request and to correctly cast it back to the proper type, and to probably copy data from userspace in this process as appropriate.

The `ioctl()` requests that need to be defined result from the possible operations that could be performed on the accelerators. Those operations are as follows.

- Writing one value to a single register.
  - extended case: Writing a memory buffer to a FIFO (cyclic DMA semantics)[5].
- Reading one value from a single register.
  - special case: Polling a register until it changes to a specific value.
  - extended case: Reading from a FIFO to a memory buffer (cyclic DMA semantics).
- Copying a memory buffer to an equally sized range of adjacent registers (`memcpy()` semantics).
- Copying a range of adjacent registers to an equally sized memory buffer (`memcpy()` semantics).

Transfers can potentially be done through regular register transfers (memory mapped I/O) or – employing the SoC's onboard DMA controller – through DMA tranfers. Latter ones are considered desirable in this work, since they should be considerable faster when compared to register transfers[6], even though I will primarily focus on register transfers for the beginning. I will consider implementing DMA transfers in case everything else works and enough time is left later.

There will be no special commands for reading and writing single values from or to single registers, as these types of access are just a special case of their extended form acting on memory buffers. Accordingly, they can be mapped to those operations. For keeping it brief, the `ioctl()` interface offer three commands, namely read, write, and poll. Even though polling could also be done by repeated reading, I will implement an independent command to avoid massive amounts of kernel calls.

[5] Here, "FIFO" does not mean the accelerators' actual FIFO implementation described above, but a FIFO in the common hardware jargon. The accelerators' FIFOs would also follow this semantics by providing exactly one input or output register, if the SoC's DMAC was capable of performing cyclic DMA transfers in a non-broken fashion.

[6] According to my tutor, Gerald Hempel, a register transfer takes about 18 cycles on the AXI bus per word, whereas a DMA transfer using the onboard DMAC should reach a peak performance of up to one word per cycle.

The argument to the read and write commands will be a structure shown in Listing 3.5, that describes the source and destination of an operation, and the specific transfer type and semantics to use. The `base` field identifies the accelerator to use, the `regstart` field is the number of this accelerator's first register that the operation is performed on. `addr` is the memory address of the memory buffer and `words` specifies the number of words to transfer (with one word being the size of one register). Via `flags` respectively its convenience accessors `cyclic` and `dma` the semantics (`memcpy()` or FIFO) and transfer type (register I/O or DMA) can be specified.

```
struct hwacc_io {
    ulong base;
    union {
        u32 flags;
        struct {
            uint cyclic : 1;
            uint dma    : 1;
        };
    };

    u32 regstart;
    u32 *addr;
    size_t words;
};
```

Listing 3.5: The argument describing a read or write operation

The poll command will use a different argument, shown in Listing 3.6. Here again, `base` will identify the accelerator, `reg` selects the register to poll. `value` is the register value that is waited for. Once this specific value is read from the register, the call will return.

```
struct hwacc_poll {
    ulong base;
    u32 reg;
    u32 value;
};
```

Listing 3.6: The argument used for polling a register.

The `ioctl()` commands will be called `HWACC_IOC_READ`, `HWACC_IOC_WRITE`, and `HWACC_IOC_POLL`, respectively. Respective wrappers will be provided in `libaccel`, which will basically just provide functions for calling the commands without having to ponder with these structures. Instead arguments will be given in the more convenient way as ordinary, distinct function parameters.

4 Implementation

4.1 ELF Plus Bitfile Equals ZwoELF

To embed the ZwoELF section containing – amongst others – the bitfile into the ELF binary, I wrote a standalone program called `zwoelf`. This might seem overkill on first glance, since there are already programs achieving the same or similar functionality. The first program coming to mind would be `objcopy` from the `binutils` package. This package is the usual provider of basic toolchain programs like `as` (the assembler), and `ld` (the *build-time* linker[1] `objcopy` is intended to do transformations on ELF files, which includes inserting or stripping arbitrary ELF sections.

Actually, the first version of `zwoelf` simply was a shell-script using objcopy to insert the raw bitfile into the executable. This even worked quite good. As it turns out, though, it only worked when applied to ELF executables built for Intel x86 processors (both 32 and 64 bit, not limited to the Intel brand, but the whole architecture). When applied to a executable built for the Zynq-7000-SoC used for development, `objcopy` turned out to be incapable of automatically guessing the exact target processor architecture of the ELF file. This seems illogical, since for inserting a section, you usually would not need to know what *exact* processor architecture you are dealing with – knowledge of ELF class (32 or 64 bit) and endianness should be all needed to know. Unfortunately, that is not the case for `objcopy`, even when it is not expected to do any processor-specific operations. This inconvenience – forcing the user to specify the exact processor architecture of the file that should be operated on – was the first cause to rethink whether the `objcopy`-approach would be the best. As it additionally turned out that inserting the raw bitfile was not enough anyways, I neglected the idea to solve the problem using a shell script and hence `objcopy`.

So I decided to write my own ELF patcher. There are two quite widespread libraries to handle ELF files, `libbfd` and `libelf`. The former is part of the *GNU binutils* and is the library enabling the *binutils* themselves to perform their tasks. The latter one originates from the *elfutils* package, which aims to provide alternative utilities for *handling* ELF files similar to *binutils*, however without the tools to initially *build* them (they do not contain an assembler, for instance).

`libbfd` turned out to be unusable in a reliable way for the same reasons as `objcopy`, or to be more precise: The reason why `objcopy` is not usable actually *is* `libbfd` as this is used by `objcopy` to manipulate ELF files.

Though `libelf` would have been usable for my purposes in general, it also turned out to be far too complex for such an in principal simple job as adding a section to an ELF file. Though I successfully got it read an ELF binary, add a section with the respective

[1] This not to be confused with the *run-time* linker `ld.so`, which is part of the system's C library).

section header entry, and write this file back to disk, I could not figure out how to do so in a way that would not break the executable. In essence, the ELF files I used for testing contained holes between sections, most probably for alignment purposes. I could not figure out how to make `libelf` also insert space between sections. Unfortunately this leads to breaking the ELF file's contents since sections containing padding of any form will become invalid. Though `libelf` knows how to interpret such sections, for example symbol tables, and would have allowed to update these information, it could not do so automatically. Since the effort that would have been necessary to fix this data consistencies would have been by far more than doing the job by hand, I decided to do the latter.

I wrote `zwoelf`, a program written in C++ and making heavy use of templates to efficiently deal with the four principle ELF variants that could be encountered (32 or 64 bit architecture, each big or little endian). This program takes an existing ELF binary and a bitfile and a device tree blob file (DTB) as its arguments[2]. It parses both program and section headers of the ELF file, and builds a ZwoELF section from bitfile and DTB file. It then outputs a new ELF file containing this new section. For this it adds an additional program header and a section header pointing to the section. It furthermore extends the section header string table. This string table contains the names of the single sections and is updated to contain a name for the new section, too. Eventually a rather straight forward task, except for the pitfalls:

When inserting program headers a problem comes to the light. As described earlier, though not strictly mandatory according to the ELF standard, it is a common practice to place the program headers at the front of the ELF file, right behind the ELF header itself. Inserting an additional program header entry inevitably leads to moving all remaining content of the ELF file back from where it used to be. This will lead to have the offset fields of the program headers to be increased as well, so that they point to the right beginning of the segments again. However, those offsets are not simple pointers, actually. They are directly used as arguments to the `mmap()` calls in the kernel, that are used to map the segments into the process's address space. The `offset` argument of `mmap()` (and its in-kernel pendants), however, has to be page-aligned. As a consequence, simply adding data somewhere in the ELF file and increment the remaining program header entries' offsets by the number of bytes added, is not enough. At least for segments marked es loadable (and thus loaded), padding needs to be added to have the offset page-aligned again. This is not restricted to the case when program header entries are added, but it gets immanent in this case, at least if you want to stick with the convention of having them at the beginning of the file.

I tried to make `zwoelf` as naïve as possible in regard of knowledge about architectures. In regard of the problem just described, this leads to a conflict of objectives, as we need to know what *exactly* "page alignment" means when working on a specific ELF file. One way to stay architecture opaque would be always using an alignment that would be always correct on all supported architectures. The architecture with the biggest page size supported by the Linux OS is the Qualcomm Hexagon, a Digital Signal Processor. This architecture has a page size of 1 MiB. The least common multiple for page alignment

[2] The precise need for and use of the DTB file will be explained later.

would thus be 1 MiB. This would mean an increase of file size by 1 MiB for adding *one* program header entry. As this is rather unpleasant, especially for embedded systems, I added awareness of page sizes at least for the architectures supported by the Linux kernel. Despite this knowledge, `zwoelf` stays almost as naïve as intended.

4.2 `binfmt` Handler: Loading the ZwoELF Section

Loading the ZwoELF section from the ELF executable is implemented through an additional binary format handler in the Linux kernel, called `binfmt_tudos_zwoelf`. To register such a binary format, either `register_binfmt()` or `insert_binfmt()` has to be called. The sole difference between these calls is where they put the binary format handler in the list of registered formats – head respectively tail. As described in subsection 3.2.2, I will use `insert_binfmt()` to have the format registered up front, so that it is called before the standard ELF handler (`binfmt_elf`). These functions take a pointer to a `struct linux_binfmt` as an argument, which describes the format.

For loading a binary, the actual entry point into the binary format handler is the function pointed to by the format description's `.load_binary` member. For my format handler, this is `load_tudos_zwoelf_binary()`. This function does the real work. Essentially, it is a stripped-down version of its pendant found in `binfmt_elf`. It checks basic properties of the ELF file, like a sane ELF header and the correct architecture. It then proceeds to parse the program headers and looks for segments with the program header type `PT_TUDOS_ZWOELF`. When the segment is found, it is read from the binary. The format handler then does sanity checks on the segment's `checksum` and `version` fields (this checks can be independently disabled through module parameters). If all checks succeed, the complete segment is handed over to the actual accelerator driver through a call to `hwacc_load()`. At this point, the format handler's job is done.

As you can see, the binary format handler itself does not bother with the ZwoELF section any further than basic sanity checking. It does not handle the further content itself, but leaves this to the driver that actually manages the accelerators.

Extracting the ZwoELF section from the ELF binary using the kernel's binary format handling interface, is a straight-forward task. A question that needs attention, however, is how to proceed on probable errors. My approach to this is pure ignorance. `load_tudos_zwoelf_binary()` will *always* return `-ENOEXEC`. This will not leave the system in an inconsistent state. The programs generated by the GCC plugin are fully backward-compatible, hence able to run completely in software without their accelerators. Thus allowing returning `-ENOEXEC` instead of a (fatal) error code, `binfmt_elf` will be allowed to try again. This procedure raises the chances that the program actually executes even when it is without hardware acceleration.

4.3 `tudos-hwacc`: Accelerator Driver

4.3.1 Creating and Destroying Accelerators: The Bitfile Registry

As outlined in section 3.3, the bitfile registry on its own is not very complex. In principle, it simply does book keeping over some kind of two dimensional array. What in fact turned out to be a tougher job is providing the information which actual accelerators are present in a bitfile and where their register window is located in I/O memory to the kernel – bitfiles are opaque, at least, and this information cannot be easily derived from simply having a closer look at them.

So I decided to add a device tree blob to the ZwoELF section, containing device tree fragments, one for each accelerator. Even though this seems to be an elegant and simple solution, it eventually turns out to be at least not that simple. This is caused by the nature of how the device tree is structured. In a perfect world, we would load the bitfile and allocate the resources needed by the included accelerators in one go, we want to be sure that the initialisation went well – the `hwacc_load()` should be atomic to the outside. Intuitively one would think that this should be feasible, as we have all relevant information at hand: We have a bitfile and we have a device tree fragment telling us the location of the resources. This is not as. Actually the information about the resources needed by the accelerators cannot be drawn from the device tree fragments without *applying* them to the live tree. This is because the `reg` fields in the fragments cannot be unequivocally interpreted without knowing the actual *parent* node they will be attached to, as the parent node's `#address-cells` and `#size-cells` fields define the interpretation of those fields. Hence, the fragments first have to be applied to the tree to be of any use.

One might think that this is not such a big problem – the nodes created by the application of the fragments all have a distinct `compatible` field, telling that those devices are our accelerators. We could simply tell the kernel that our driver is in charge of handling these devices, wait for the device probe and allocate the resources for the devices and allocate a structure describing holding the information for the accelerator and the required resources at this point. Unfortunately, at this point we *again* lack vital information, namely the bitfile slot this accelerator came from. Furthermore, the atomicity demanded above will be lost, too. `hwacc_load()` would have no idea of which nodes were created by applying the device tree fragments, whether resource allocation went well, or even only which accelerators came with this bitfile. This foils any attempt to do proper book keeping.

This problem cannot be solved very elegantly. The approach I took exploits the fact that node names in the device tree have to be unique and usually follow the format `<device>@<baseaddr>`. So what I do is loading the bitfile and proceed to search the provided device tree for nodes with a `compatible` field of `"tudos,hwacc"`, hence our accelerators. For each node found, I allocate a new `hwacc` structure (which represent the individual accelerators and hold their claimed resources). The individual structures – as the accelerators they present – are uniquely identified by the accelerators base address. This address, as I cannot gain it from the device tree node itself (the fragment is not applied at this point in time), is parsed from the node's name and saved in the

structure. After all structures are allocated, they are made available to other driver parts by inserting them into a binary search tree, that uses the base address as a search key. Now I apply the device tree overlay.

Now – with `hwacc_load()` still executing – another part of the driver, `pdrv_probe()` gets called by the device probe triggered by applying the overlay. This function gets a pointer to the actual device that triggered the probe as an argument. From this pointer, we now can determine the parsed value from the corresponding device tree node's `reg` field. The function now should be able to locate the corresponding accelerators `hwacc` structure in the binary search tree – as long as name and resource in the corresponding device tree changeset were consistent. It saves the actual resource information in this structure and raises a semaphore saved in the structure.

This is the point where we get back to `hwacc_load()`. After applying the device tree overlay, this function successively waits on the semaphores of the `hwacc` structures it just had created and inserted into the search tree. If the device tree blob provided by the ZwoELF section consistent, eventually all nodes will have triggered a device probe and hence all structures' semaphores should have been raised by `pdrv_probe()`. Now `hwacc_load()` proceeds to allocate the I/O regions that are now available in the `hwacc` structures. If everything went well, the accelerators are marked usable.

Waiting for the semaphores is performed using a maximum overall timeout. If this timeout runs out, the loading operation is considered as failed and the whole slot is killed – the device tree overlay is destroyed, already allocated resources are released and the bitfile slot is marked as free for use.

4.3.2 The Character Device Driver `/dev/hwacc`

4.3.2.1 Registering the Character Device File

To implement a character device driver, you usually would be required to allocate a major number and explicitly manage the range of contained minor devices. As the design sketched earlier only requires exactly *one* character device, the implementation turns out to be a lot easier. For this special case, Linux offers a really neat abstraction called `miscdevice`, or miscellaneous device (driver). The `miscdevice` subsystem is a kind of collecting point for drivers that just need to create one character device file without having to deal with the details in too much detail. The `miscdevice` subsystem does the allocation of a major device, and every instance of a `miscdevice` gets assigned a minor device number under this major number. It might request a certain minor number, but even this is completely optional – a driver might also decide to request the assignment of a random minor number. Using the `miscdevice` subsystem, creating a character device special file essentially boils down to registering a `miscdevice` driver. The boilerplate work in regard to the actual character device file, including allocation and management of major and minor numbers, is done by the subsystem.

A `miscdevice` driver is registered through a call to `misc_register()`. This function takes one argument, a pointer to a `struct miscdevice`. The significant subset of this structure is shown in Listing 4.1.

```
1 struct miscdevice {
```

```
2     int minor;
3     const char *name;
4     const struct file_operations *fops;
5     ...
6 };
```

Listing 4.1: The `struct miscdevice`.

The `name` field specifies the name of the driver. It affects the file system entries the kernel creates in the `/sys` file system, describing the driver. This also directly affects the character device file's name. `minor` allows to select a specific minor number for the device file – the major number is implicitly predetermined as the device file will be in the `miscdevice` subsystem's scope. The special value `MISC_DYNAMIC_MINOR` tells the subsystem that the driver does not care about the precise minor number. In this case, the subsystem will assign a minor number it considers suitable, in practice this will be some number available at the time of registration. The `fops` field specifies the file operations that are available on the device file. Not all functions in the structure this field points to have to be defined. If userland calls an I/O function on the device file whose corresponding function pointer in this structure is `NULL`, the kernel calls a default action. So essentially, everything that has to be done to simply create one reasonably behaving character device is as follows:

1. Implement the file operations that should do something beyond the default.
2. Define a *fops* structure, whose members point to this appropriate functions.
3. Define a `struct miscdevice`, that points to this *fops* structure and specifies the name and minor number (possibly "I don't care") of the device file to be created.
4. Call `misc_register()` with a pointer to this structure as its argument.

My character device `/dev/hwacc` will only support three I/O calls from userland: `ioctl()` will be used to interact with the accelerators. This function needs a file descriptor to specify the (character) device that should be operated upon, thus the other two supported operations will be `open()` and `close()`.

4.3.2.2 Doing the Actual Transfer

The next step now would be to implement the actual `ioctl()` function. As described earlier, the respective function in the kernel, pointed to by the `unlocked_ioctl` member of the *fops* structure, gets three arguments: A pointer to a `struct file`, that is the kernel-side correspondent to userland's file descriptor. Furthermore the request (or command) as an `unsigned int` and the request's argument as an `unsigned long`. It now has to interpret the command and in correspondence the argument. Then it should ideally perform some action.

The character device driver is implemented in `kernel/hwacc-cdev.c`. The function that is specified for `unlocked_ioctl` in the driver's *fops* structure is `hwacc_cdev_ioctl()`. The actual functionality for the different types of commands is implemented in

further independent functions to keep the code readable. `hwacc_cdev_ioctl()` checks whether the requested command is valid. It then – depending on the command – copies the argument structure from userspace and does sanity checking on it. It requests the accelerator that should be acted upon. This might fail for two reasons, one being the accelerator is not existent. If the accelerator exists, it could be currently in use by some other process. In both cases, an error is returned. If this happens, the calling process should switch to the pure software implementation.

If the requested accelerator is available, `hwacc_cdev_ioctl()` checks whether the register operand of the requested command is valid and – in case of a read/write command with `memcpy()` semantics selected – the requested number of words does not exceed the register window. If it does, the driver takes this as a serious offence and sends the `SIGSEGV` signal to the calling process, which will usually lead to the termination of that process ("Segmentation fault"). This harsh reaction is intended, as this kind of memory accesses should not come from the programs the GCC plugin emits. Thus such a request points out a bug in the GCC plugin, in which case a fatal signal additionally acts as a debugging aid (as the standard reaction to this this signal is generating a core dump). The other possibility for such a request to occur would be an intentionally misbehaving program, in which case killing it is an adequate reaction.

Requesting the accelerator also marks it as in-use by a process, hence effectively hindering the kernel from reusing the associated bitfile slot for loading a new bitfile. Although the kernel is able to observe when a process *starts* using an accelerator (first request with the appropriate base address), it cannot implicitly tell when the process does not need it anymore, without introducing the necessity for the process to explicitly tell the kernel. For now, the implementation releases the accelerators when the process calls `close()` on the file descriptor associated with the character device (obtained through a call to `open()`). Thus, the function the *fops* structure's `release` member points to, `hwacc_cdev_release()`, releases all accelerators used by the process in one go when the process closes the file descriptor.

4.3.2.3 Side Note: Duff's Device

Register reads and writes are implemented using "Duff's Device", in the hope that the ARM CPU is smart enough to detect memory writes to adjacent registers and further optimise them at runtime. `memcpy()` cannot be used in this case, as it copies byte-wise, which is not valid with 32 bit registers – copying the four bytes individually would lead to four writes in hardware with rather odd results. Furthermore, `memcpy()` used to read or write I/O memory might not be possible depending on the architecture. As an example for "Duff's Device", `duff_read()` is shown in Listing 4.2.

```
1 static inline void
2 duff_read(register u32 *to, register __iomem u32 *from, register size_t words)
3 {
4     register size_t n = (words + 7) / 8;
5
6     switch (words % 0) {
7     case 0: do { *to++ = readl(from++);
8     case 7:      *to++ = readl(from++);
9     case 6:      *to++ = readl(from++);
```

```
10      case 5:       *to++ = readl(from++);
11      case 4:       *to++ = readl(from++);
12      case 3:       *to++ = readl(from++);
13      case 2:       *to++ = readl(from++);
14      case 1:       *to++ = readl(from++);
15              } while (--n > 0);
16      }
17  }
```

Listing 4.2: Reading I/O memory with Duff's Device

Though it seems pure Voodoo on first glance, the idea behind this is quite simple. First of all, do not be confused by the do-while-loop nested in a switch-case-statement. The idea of this function is to avoid checking for the loop's exit condition after every read. To accomplish this, it determines the number of blocks of eight reads it can perform with the given number of words. The `switch` is only executed once to shorten the first iteration to the right number of reads in case words does not evenly divide by eight. For example, if words is 42, this code jumps to `case 2`, does two reads and continues to do five full loops, making it 42 reads.

As it turns out, however, the ARM is not very smart. The reason the implementation still uses this call is that it actually still generates faster code when compiled with low optimisation levels[3].

4.3.3 Requesting and Releasing an Accelerator

One more serious problem of the way bitfile loading works, is how to efficiently store and locate the accelerators, as it takes place at different locations in the kernel – the accelerators are created and destroyed by the ZwoELF/bitfile registry which somehow has to make them available to the character device driver. Furthermore, it has to be made sure, that an accelerator cannot be destroyed while being used by a userland process. This also applies to the bitfile slot the bitfile containing this accelerator is loaded into. The slot may not be reused by another bitfile as long as the accelerators belonging to the bitfile in it are in use. It would lead to fatal effects if the bitfile slot would be reused and the FPGA reprogrammed while an accelerator provided by this bitfile was in use.

To address the problem of storing and locating the accelerators, the binary search tree introduced in subsection 4.3.1 is used again. Furthermore, I implemented functions to request and release specific accelerators. Requesting an accelerator is done by calling `hwacc_request()`, which takes the base address of the requested accelerator as an argument. This function returns a pointer to the associated `struct hwacc`. The `hwacc_request()` function not only searches the binary tree for the accelerator, but also makes sure that the accelerator is available and – if so – marks it as used before returning the pointer to it. It also increments a usage counter on the bitfile slot the accelerator originated from. To release an accelerator requested through a call to `hwacc_request()`, the function `hwacc_release()` has to be called. This function undoes the effects of `hwacc_request()`, making the accelerators available to other processes again.

[3] And I have to admit, that I always wanted to use this as I think it is incredibly elegant.

5 Future Work

To this point this work showed that it is indeed feasible to integrate the hardware accelerators generated by the GCC plugin into a general purpose operating system without compromising the the operating systems fitness for a general purpose – that is to say, without introducing special constraints on for instance memory layout, requirements of special hardware like IOMMUs, and so on. As it is more of a proof of concept, it – of course – still leaves enough room for improvements.

Due to lack of time, the actual implementation still lacks support for DMA transfers, but is restricted to register transfers. Those are very slow, hence adding support for DMA will be a requisite to be able to actually gain comparability to other systems. The current code base is already laid out to support DMA transfers, but needs a bit further work. The cause for the lack of DMA transfers in the current implementation is eventually caused by the behaviour of the Zynq-7000-SoC's DMA controller, which – in contrast to its own claim – does not dependably offer cyclic DMA transfers. The accelerators' input and output FIFOs already have been adjusted to this fact, so that it should be possible now to do copying DMA transfers (`memcpy()` semantics), which the controller should be able to satisfy. Finishing this work in the implementation should be the number one task for further work.

Furthermore, there are design decisions that are meant to be provisional arrangements for the sake of simplicity. One of those arrangement is the handling of the device tree blobs used to describe the accelerators contained in a bitfile. Though identifying an accelerator by its base address is completely fine as soon as it is loaded – there can only be one accelerator at any given address – this approach will not suffice anymore, as soon as the GCC plugin learns how to generate partial bitstreams. With partial bitstreams, accelerators will have to be relocatable on the FPGA. This means that the base address will not be known until the bitfile is loaded onto the FPGA. But the programs will have to be able to uniquely address the accelerators regardless of their eventual location. To solve this problem, another unique characteristic has to be introduced, like some sort of unique identification number, string, or whatsoever.

Opening another driver's character device file from kernel mode as done to program the FPGA by `fpga_load()` from `kernel/fpga-glue.c` is another such provisional arrangement. Doing this is definitively a monumental deputy of bad style. However, it was closest to a straight forward approach in the context and objective of this work to not touch external code if not unavoidable. To fix this issue, this implementation is not the first place to go either. The kernel, or to be more precise the existent kernel drivers for programming FPGAs, simply lack an in-kernel interface. This is so far not a problem, since in a traditional world the kernel does not need to do so, as FPGAs are more of an updatable ASIC. The research happening in the field of automatically

generated hardware accelerators promises to overthrow this conception, but it is the FPGA drivers that need to adopt to this.

Another point worth looking at is the precise kind of userland interface. Though a character device file was definitely the best way to go for proving the point in this work, there might be more suitable ways to interface with the accelerators. As thoroughly written out, using a character device, or to be more precise its `ioctl()` implementation, is a valid approach, but also not very elegant. Unfortunately, it is not effectively feasible to better fit the userland interface to the standard I/O routines, as these routines lack the expressiveness needed to concisely address the accelerators. As this needs to be done somehow, though, `ioctl()` was necessary as it provides the possibility to transport this information. But it could be considered whether there is a better alternative to a character device file altogether. One possibility that already came to my mind would be using the socket interface and introduce a new `sockaddr` type. After all, this API is intended to *address* things. I did not follow this approach in this work, though, since it would have meant a lot more implementation work and would most probably had gone far beyond of the scope of this work. Furthermore it could be argued whether using an API commonly used and intended for networking should be used to implement the interface to a hardware component, or if this should be considered malpractice. A quick search through the Linux source tree suggests that I would have been the first one to do so, and that usually is not a good sign in this context.

6 Conclusion And Outlook

With this work I introduced an infrastructure for using and handling the accelerators automatically generated by the GCC plugin.

To accomplish this I designed and implemented a mechanism to tie together the bitfile containing the accelerators and the ELF binary using them. I brought up several ways how such a mechanism should work and which requirements it should meet to be flexibly applicable without bothering the end user with changing the familiar workflow. This considerations resulted in extending the ELF binary by an additional section, called ZwoELF in this work and introduced in subsubsection 3.2.1.1. This section contains all information needed to enable the user to simply execute the binary and allow the kernel to arrange the loading of the bitfile. To extend an ELF binary by a ZwoELF section, I provided the `zwoelf` utility, as described in section 4.1.

To extract the ZwoELF section from an ELF binary at time of execution and to load the included bitfile onto the FPGA I introduced `binfmt_tudos_zwoelf`, a binary format handler that gets placed in front of the actual ELF handler. It extracts the section and hands it further to the actual accelerator device driver.

The accelerator device driver does the real work and consists of two main parts, the bitfile registry introduced in section 3.3 and the userland-interface driver shown in section 3.4. The bitfile registry handles insertion and deletion of accelerators by loading further ZwoELF sections from ELF binaries as they are executed. It supervises the usage state of accelerators and allows or inhibits their replacement. It also manages the resources claimed by the accelerators. To provide it with the needed information – as this cannot be simply derived from the bitfiles, which are opaque blobs to us – I proposed the bundling of partial device tree blobs into the ZwoELF section. The driver cares for applying and destroying this device tree blobs into the live tree. Furthermore, the driver is capable of managing multiple simultaneously loaded bitfiles, and initialising the contained accelerators and their associated resources. Moreover, it offers a character device interface to enable userland processes to access and use the accelerators in a safe and reliable manner.

To ease the use of this character device interface, I provided a convenience library that manages the setup of the character device interface and allows easy access to this through thin wrappers. This library functions are intended to be inserted into the executable by the GCC plugin.

Even though the current implementation eventually lacks the capability to do DMA transfers, the source code to implement this functionality is already present for the most part. Unfortunately I did not accomplish finishing and testing it, due to a misbehaving DMA controller. Anyway, it should be possible to finish the needed adjustments in a quite short time.

Bibliography

[Adr08] Adrian Schüpbach and Simon Peter and Andrew Baumann and Timothy Roscoe and Paul Barham and Tim Harris and Rebecca Isaacs. 'Embracing diversity in the Barrelfish manycore operating system'. In: *In Proceedings of the Workshop on Managed Many-Core Systems*. 2008.

[Agn+14] A. Agne et al. 'ReconOS: An Operating System Approach for Reconfigurable Computing'. In: *Micro, IEEE* 34.1 (Jan. 2014), pp. 60–71. ISSN: 0272-1732. DOI: 10.1109/MM.2013.110.

[Ant14] Antonio Barbalace and Binoy Ravindran and David Katz. 'Popcorn: a replicated-kernel OS based on Linux'. In: *The 2014 Ottawa Linux Symposium (OLS '14)*. Ottawa, Canada, 2014.

[Aue+10] Joshua Auerbach et al. 'Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures'. In: *ACM*. Reno/Tahoe, Nevada, USA, 2010, pp. 89–108. ISBN: 9781450302036.

[Can+11] Andrew Canis et al. 'LegUp: High-Level synthesis for FPGA-based processor/accelerator systems'. In: *ACM/SIGDA*. Monterey, CA, USA, 2011, pp. 33–36. ISBN: 9781450305549.

[Fle+15] Shane Fleming et al. 'System-level Linking of Synthesised Hardware and Compiled Software Using a Higher-order Type System'. In: *ACM/SIGDA*. Monterey, California, USA, 2015, pp. 214–217. ISBN: 9781450333153.

[For+14] B. Fort et al. 'Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis'. In: *EUC*. 2014, pp. 120–129. DOI: 10.1109/EUC.2014.26.

[Hua+08] ShanShan Huang et al. 'Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary'. In: *ECOOP 2008 – Object-Oriented Programming*. Ed. by Jan Vitek. Vol. 5142. 2008, pp. 76–103. ISBN: 9783540705918. DOI: 10.1007/978-3-540-70592-5_5.

[Hut+13] J. Huthmann et al. 'Hardware/software co-compilation with the Nymble system'. In: *ReCoSoC*. 2013, pp. 1–8.

[LK07] H. Lange and A. Koch. 'An Execution Model for Hardware/Software Compilation and its System-Level Realization'. In: *FPL*. 2007, pp. 285–292.

[Nig+09] Edmund B. Nightingale et al. 'Helios: Heterogeneous Multiprocessing with Satellite Kernels'. In: *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, MT: Association for Computing Machinery, Inc., Oct. 2009. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=81154.

[SCO97] The Santa Cruz Operation, Inc. and AT&T. *System V Application Binary Interface Ed. 4.1*. `http://www.sco.com/developers/devspecs/gabi41.pdf`. 1997.

[Vil+10] J. Villarreal et al. 'Designing Modular Hardware Accelerators in C with ROCCC 2.0'. In: *FCCM*. 2010, pp. 127–134.