Integrating devices in FPGA using an end-to-end hardware/software co-designedmessage-based approach

Thomas Baumela

▶ **To cite this version:**

Thomas Baumela. Integrating devices in FPGA using an end-to-end hardware/software co-designedmessage-based approach. Embedded Systems. Université Grenoble Alpes [2020-..], 2021. English. NNT : 2021GRALM004 . tel-03259401

HAL Id: tel-03259401

https://theses.hal.science/tel-03259401

Submitted on 14 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Thomas Baumela

Thèse dirigée par **Frédéric Pétrot**
et codirigée par **Olivier Gruber**

préparée au sein des laboratoires **TIMA** (UMR5159) et **LIG** (UMR5217) et de l'École Doctorale **Mathématiques, Sciences et Technologies de l'Information, Informatique**

Integrating devices in FPGA using an end-to-end hardware/software co-designed message-based approach

Thèse soutenue publiquement le 24 février 2021,
devant le jury composé de :

M. Tanguy Risset
Professeur, Laboratoire CITI, INSA-Lyon, Rapporteur

M. Gaël Thomas
Professeur, Laboratoire SAMOVAR, Telecom SudParis, Rapporteur

M. Noël de Palma
Professeur, Laboratoire LIG, Univ. Grenoble Alpes, Président

M. Kevin Martin
Maître de conférences, Laboratoire Lab-STICC, Université de Bretagne Sud, Examinateur

M. Frédéric Pétrot
Professeur, Laboratoire TIMA, Grenoble INP, Directeur de thèse

Contents

1 Introduction **1**

2 Context and Problem overview **9**

2.1 Problem illustration 9
2.2 SoC approaches for HW/SW integration 11
2.3 OS approaches to simplify device driver development 12
2.4 Summary 16

3 Message-Based Integration for Embedded Systems **17**

3.1 The Current Integration Challenge 18
3.2 Background 21
3.3 Message-Based Integration Solution 23
3.3.1 The Software Perspective 23
3.3.2 The Hardware Perspective 27
3.3.3 The Overall Lifecycle 29
3.4 Messages and Class-Genericity 30
3.4.1 Using Messages 31
3.4.2 Class-Generic Protocols 34
3.4.3 Heterogeneity 35
3.5 Evaluation 39
3.5.1 Overhead Evaluation 40
3.5.2 Small Embedded Systems 45
3.6 Summary 49

4 Linux Integration **51**

4.1 Background 52
4.1.1 Linux kernel modules 52
4.1.2 User-space interfaces 54
4.1.3 Linux Device Driver Model 56
4.1.3.1 Devices, Drivers, Buses 57
4.1.4 Software-Hardware communication in Linux 61
4.1.5 Some bus implementation examples in Linux 62
4.1.5.1 PCI drivers 62
4.1.5.2 USB drivers 64
4.2 The Extension Proposal 67

4.2.1 A new message bus for Linux 68
4.2.2 Driver API 69
4.2.3 Low Level Driver API 71
4.3 Experiments 75

5 Xen Integration **79**
5.1 The FPGA Adoption in the cloud 79
5.1.1 The architecture of the cloud 79
5.1.2 The state of FPGA adoption in the cloud 83
5.2 The impact of our proposal for the cloud 84
5.2.1 Message-based communication 85
5.2.2 Class genericity 86
5.3 Feasibility 87
5.3.1 Concepts offered by hypervisors 87
5.3.2 Description of our solution 88
5.4 Deeper Analysis 97
5.4.1 Integration in Xen 97
5.4.2 Consequences of our solution for cloud systems 102

6 Conclusion **105**
6.1 Summary 105
6.2 Perspectives 108

Bibliography **111**

Chapter 1

Introduction

The pace of hardware development tends to be continuously increasing. This is the case for instance with MultiProcessor System-On-a-Chip (MPSoC), embedding a lot of specific Intellectual Properties (IPs). Those systems evolve very quickly to match a specific consumer market. For instance, several versions of a Snapdragon platform can be released over a year[1]. Integrating hardware, meaning, making its features available to a high-level software application, is thus a very important issue. Indeed, as hardware is evolving, having new and changing capabilities, and systems become more and more complex thanks to more efficient tools and higher developers expertise, integrating hardware has a growing cost. If this is true for MPSoCs, this is worse for Field Programmable Gate Arrays (FPGA) for which their high programmability makes hardware development even faster and thus the hardware integration an even more critical problem.

The work addressed by this thesis aims therefore at improving the hardware integration in the context of FPGAs. FPGA technology is incrementally used across a wide variety of domains, from small embedded systems to cloud computing. They are often used to associate programmable hardware logics with processing systems. In such systems, hardware components can be designed and integrated by programming a preexisting chip without having to change the actual hardware. This opens great possibilities regarding programmability, power efficiency and performances. Though, this comes with some drawbacks: Not only programming FPGAs can be complex but integrating the newly created hardware with the software is also a heavy task.

[1] https://en.wikipedia.org/wiki/List_of_Qualcomm_Snapdragon_processors

The advantages of using FPGAs for tasks with fine-grain parallelism increased their adoption in a lot of domains. In particular, their reprogrammability offers great evolution possibilities. It makes system updates easier by simply reprogramming the FPGA without having to change the actual hardware. Reprogramming an FPGA can even be made remotely, making the maintenance of such system easier. Their power efficiency, even though still lower than pure Application Specific Integrated Circuits (ASICs), is far better than running software on a CPU. They also allow faster prototyping thanks to their ability to design and implement hardware without having to build an actual chip. Implementations on an FPGA are made at the transistor level, allowing much better parallelization for applications that need it. This offers overall great performances compared to software implementations, making FPGAs a great solution for embedded systems, medium size systems and cloud computing systems.

In embedded systems, programmable logic is used to deploy hardware logic as devices. These devices can be controllers of external devices such as human interface devices (controllers and displays), storage and network devices, or sensors and actuators. They can also be accelerators for encryption, video and audio processing, and nowadays artificial neural network inference. This reduces the power consumption while offering better performances compared to software implementation running on a CPU.

Figure 1.1 depicts the overall view of how FPGAs are integrated in embedded systems. This allows the FPGA to have access to an I/O interface allowing it to access external peripherals and buses such as USB, Ethernet or I2C. It also interfaces the FPGA with the system bus making possible for the software running on it to access devices on the FPGA. Devices deployed in such FPGAs are integrated through a software driver running on the processing system. Each device has an available driver that integrates the features it offers to upper layers of software.

FPGAs are also used this way for bigger systems, running full fledged operating systems. In those systems the FPGA either sits in the same SoC or board as the processing system or on a separate board usually connected through a PCI link. Even though those systems are bigger than embedded systems, FPGAs are integrated in similar ways. Hardware components are deployed in FPGAs and seen as devices from the software. Those devices must be integrated with their driver within the operating system. A task that can be very challenging regarding the fact that driver updates must follow the rules of each operating system update schedule.

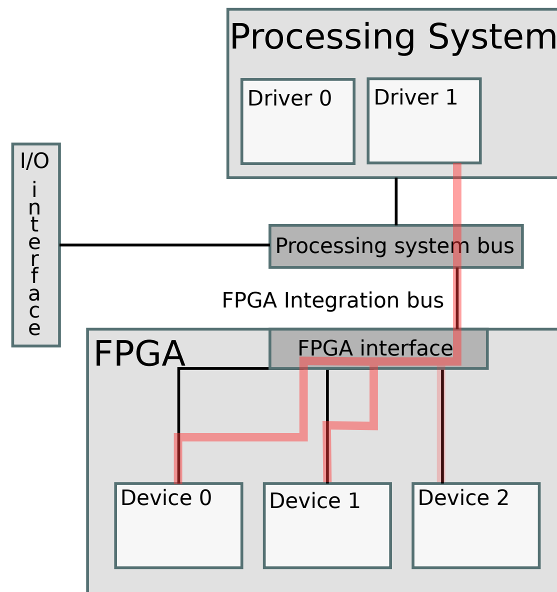


FIGURE 1.1: FPGA Integration for Embedded Systems

FPGAs are also more and more used in cloud computing. Those systems are intrinsically made to share hardware resources to multiple clients. It means that no matter the level at which the client stands (virtual machine, operating system or application level), the FPGA resources must be shared among multiple clients. We decided to focus only on sharing devices implemented on FPGAs. Indeed, sharing the FPGA itself, meaning, allowing multiple clients to reprogram part or all the FPGA is a different topic and is currently addressed by several works. In our case, devices are deployed within FPGAs and shared with the clients that can come and go at any time. As these devices are on a FPGA, they may be updated, removed or new devices may come at any time. This means that integrating such devices requires more dynamicity in all layers.

Figure 1.2 shows the overall picture of how FPGAs are integrated in cloud systems. Hardware side, the architecture is the same. Devices are deployed on the FPGA, integrated with the processing system bus to let the software access those devices. Software side the situation is different as multiple operating systems share the same hardware resources. A first layer of software called the hypervisor is in charge of sharing the hardware and thus the devices on the FPGA. There are multiple ways of sharing devices but the general idea is that each operating system has its own set of driver for the devices it wants to handle. Drivers communicate with the hypervisor to handle the device. The hypervisor is then in charge of sharing device operations coming from the drivers of all operating systems. Additionally, some devices may also be completely allocated to one specific guest, making the situation exactly the same as for previous systems.

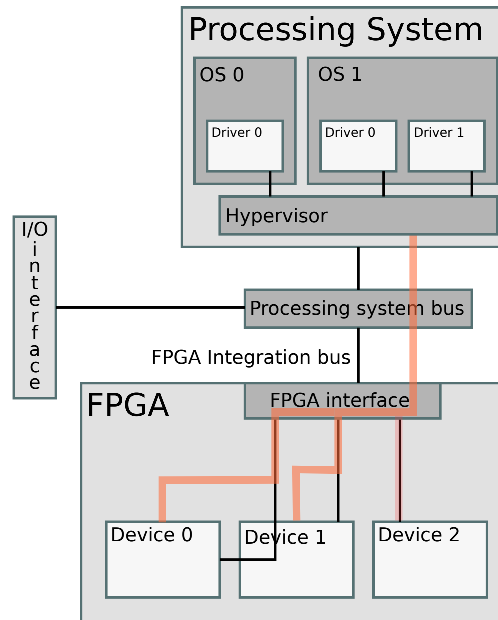


FIGURE 1.2: FPGA Integration for Embedded Systems

As FPGAs come with many power and performance advantages for a lot of different applicative domains, one may ask why the adoption of such a great technology, even though it is moving forward, is so slow. The big part of the answer is that FPGAs come with a major drawback: Integrating devices deployed within FPGAs in an actual software intensive system is a real challenge. This challenge impacts both the hardware and the software sides, making it a co-design issue.

Device makers must adapt their device interfaces to the interrupts controller, Direct Memory Access (DMA) engine, and integration bus specifications of every system. Hardware interfaces are hard to standardize. Even though some efforts have been made no real standard has emerged. The two major actors of the industry recognized this challenge and started to address it, though only partially. They have drastically improved their tools over the years, offering great support for developing hardware with both Hardware Description Languages (HDLs) and High Level Synthesis (HLS). They have also grown their libraries of hardware components making it easier to reuse hardware commodities such as buffers, bus controllers or DMA engines. They also make a great effort into pushing the AXI standard offering great tools and components allowing to more easily interface with the standard. But those tools and resources are essentially focused on improving the design, integration and deployment of hardware on the programmable logic. Even though this is a major and necessary step to help integrate devices in FPGAs, it is only half of the challenge we address.

The integration challenge we face is across software and hardware. Of course, hardware features must be designed and deployed on the programmable logic, but software must be written and installed on the processing system to be able to exploit these hardware features. This means writing software drivers to drive hardware devices. Writing drivers requires both hardware and software skills that not everybody has, making their development time-consuming and costly. Developers must deal with the current hardware-software frontier based on registers and interrupts, exposing a lot of hardware implementation details. They have to work with ever growing documentations often being hundreds of pages long and describing every bit of every hardware register, as they must dive deeply into details to allow driver developers to understand how to drive the device. Those documentation are always evolving as devices are revised every few months, and are usually not free of ambiguity. Also, new devices are appearing all the time, making writing drivers a never-ending struggle. This leads to drivers being hard to maintain safe as they must be updated often, and as such are an important potential source of bug. Considering the diversity of software stacks for which drivers are needed, from bare-metal to various popular versions of the Linux kernel, the situation may become critical if not addressed.

In this thesis, we propose a new simple and effective approach to tackle this issue. Our approach does not require expensive or new technologies. It can be adopted incrementally without having to throw every existing solution out. It is compatible with existing FPGA and processing system technologies without having to change the existing tools. Our solution requires to follow an end-to-end design, taking into consideration a comprehensive view of the integration process across both the software and the hardware. This design can be followed using existing tools and technologies, meaning that a lot of existing improvements can be integrated in our design.

Our approach was inspired by the principles behind the Universal Serial Bus (USB) and adapted to devices deployed on FPGAs. The two core principles we retain from USB are: 1) provide a message conduit between devices and drivers, completely hiding the old frontier made of interrupts and registers, 2) define device classes allowing to have one driver able to drive multiple devices of the same class. Using a message conduit decouples the driver from its device, bringing a much needed separation of concerns. No more awkward documentations that are several hundreds of pages long, no more dozens of memory-mapped hardware registers and interrupt needed. The only requirement is a high-level message protocol. This will make driver smaller, easier to write and safer to operate. Additionally, it opens the possibility to standardize message

protocols for classes of devices rather than have one specific protocol for each specific device. With class-generic protocols, only one driver is necessary to drive an entire class of device.

The two principles, a message conduit and class-generic protocols, are the two key reasons why any USB key can be plugged into almost any host and work out of the box. It works first and foremost because the physical key itself can be plugged in, which requires standard plugs and cables. Then it works because the USB key is seen as a mass storage device, a class of device that support a standard mass storage protocol. Finally, it works because the processing system runs a generic driver for that specific class of device, integrating it with the rest of the software stack. The overall message-based approach provides the right separation of concerns between hardware design and software coding. Both sides only see the other side as a message sender and receiver ensuring a smooth and cost-effective integration. Unfortunately, the USB specification cannot be leveraged for this. USB is not open, not free and not abstract enough. Its specification is a closed world that locks adopters in and forces for certain cables and plugs, making it hard to customize for certain needs.

Our proposal is to define an open and abstract conduit for sending and receiving messages across the frontier between the programmable logic and the processing system. Fortunately, we do not need to build new cables and plugs. FPGAs already have integration solutions with processing systems that work well either being on the same SoC, board or through existing buses such as PCI. On the FPGA itself, the standardization efforts of the industry, in particular related to the AXI standard, are completely reusable in our solution. In particular, we built our solution using the AXI and AXI-stream standards, using existing tools from FPGA vendors without feeling the need to change them or completely reinvent the wheel. Thus, the challenge is not here, all the building blocks we need exist and will be reused. The challenge is to design the message conduit in terms of its two interfaces:

1. The software interface that all drivers will be using to send and receive messages,
2. The hardware interface that all devices will be using to receive and send messages.

The first objective of this thesis is to define a fully abstracted interface that is simple and safe to use. Having an abstract interface is key because all the systems we will target are very heterogeneous. We thus cannot impose one specific existing standard

as some users may need low-power low-performances interface while other may need high-performances. With an abstract interface, board providers will have the implementation freedom to adapt our proposal to the specifics of their boards, from small embedded systems to larger ones. Indeed, our proposal can accommodate with very different technologies in terms of processing systems, programmable logics and the interconnect buses between them. Adding a layer of abstraction means that we must be mindful of important requirements. Using messages must not introduce any undue overheads in terms of latency or throughput. We must also preserve the footprint and power consumption on the programmable logic side.

We designed our solution first for small embedded systems. This design addresses the integration challenge we described by defining both the software and the hardware interface of our new message conduit. Software side, we designed an asynchronous interface inspired from the Linux driver model, which is familiar to driver developers. Hardware side, we designed an abstract interface based on streaming interfaces used to send and receive messages with the software. The implementation of these streaming interfaces is not imposed by our solution, only the message protocol traveling in them is. This means that one may build our solution with a different implementation than ours and still benefits from its advantages. The prototype we built demonstrates that our solution fits for small systems with low latency and low throughput devices without growing much the size of the implementation.

We also integrated our solution within a Linux kernel, demonstrating its feasibility for bigger systems. We demonstrated that our solution can fit for high-performance devices with negligible performance overheads. We showed that our design integrates well within a Linux kernel without disturbing the existing solutions. It shows that our solution is flexible and can be incrementally adopted by having some devices integrated in our design and some other in existing solutions.

In the context of cloud computing, we analysed that our solution can offer great benefits to improve FPGA integration but also hardware support. Indeed, messages are good solution to share device among multiple guests. They also fit well in famous hypervisors, in particular Xen, which have all the features our solution requires without the need of modifying it. In addition, cloud computing benefits from class genericity enabled by our message-based solution. It allows guest operating systems to require less drivers to support hardware. Our solution also allows to reuse message-based drivers from a non-supervised operating system to a supervised one.

The manuscript is organized as follows. Chapter 2 gives an overview of the context in which our work takes place, and discusses the related works that we believe are relevant to our proposal. Chapter 3 presents our first contribution, the design and implementation of a message-based approach that targets the context of embedded systems. Chapter 4 presents our second contribution, that details how our solution can be integrated in bigger systems, in particular those running the Linux kernel, and presents how it can be realized in Linux. Chapter 5 is a less mature contribution, more specifically the implementation is yet to be demoed, that shows how our solution can be useful for the cloud and how it can be integrated to a cloud system, through the Xen hypervisor. Finally, Chapter 6 summarizes the problem we addressed and the solutions we propose, and draws some perspectives.

Chapter 2

Context and Problem overview

Chapter contents

2.1 Problem illustration 9
2.2 SoC approaches for HW/SW integration 11
2.3 OS approaches to simplify device driver development 12
2.4 Summary 16

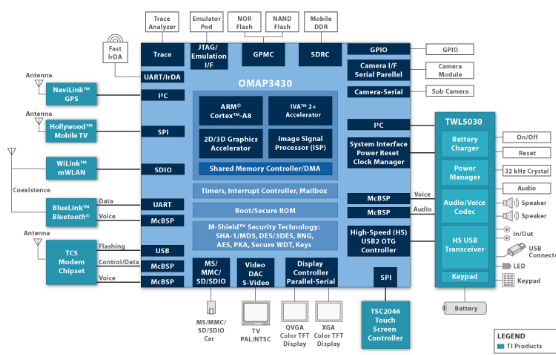
The work we address in this thesis is at the crossroad of several initiatives, some coming from System-on-a-Chip research, others from Operating System research. SOC and OS research have very different objectives, while at the end of the day the integration of both is mandatory to building a useful silicon-based system. We start this chapter by an illustration of the rapid pace of hardware development and the software difficulties it induces. Then, we analyse it by first focusing on the approaches that target integration of IPs into Systems-on-a-Chip from a bottom-up perspective. We then take the opposite standpoint in which Operating System is the core of the work and hardware is a necessary evil. Finally, we draw some conclusions about some missing pieces that would ease the integration of ad-hoc hardware in legacy OS software.

2.1 Problem illustration

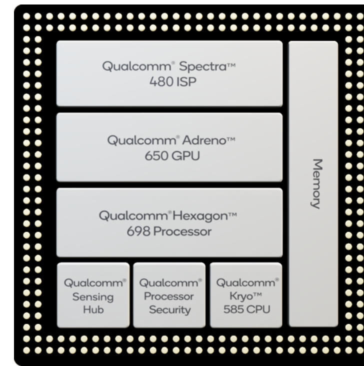
Multiprocessor System-on-a-Chip are today what Application Specific Integrated Circuits were yesterday: pieces of silicon optimized towards an application or a class of applications. Even though they tend to be reusable and are able to handle more and more applications efficiently, the power and price budgets makes it necessary to

embed specific hardware Intellectual Properties (IPs) depending on the target usage and market. Platform-based design [1] has been a keyword driving the industry, as it maximizes hardware reuse and thus shortens its development time.

Looking at mobile platforms is quite instructive: Qualcomm has released more than 150 different Snapdragon platform instances to its customers between 2007 and 2020, Texas Instruments advertised 25 until it left this market and other players in the market have similar numbers. These platforms have become so complex overtime that the industry does not even try to give rough diagrams of what the SoC contains, as illustrated Figure 2.1.



(a) OMAP 3430 block diagram



(b) Snapdragon 865 block diagram

FIGURE 2.1: Typical marketing representation of a SoC in (a) 2010, (b) 2020

Even though fast and power efficient hardware design for SoC is still a real challenge, the huge number of platform versions only exacerbates the already difficult problem of hardware/software integration. Indeed, each platform contains a variable number of IPs and in most cases even the ones which provide the same features are slightly different. Thus, the hardware/software integration process has to deal with ten to a hundred IPs, each having themselves tens to hundreds registers. Morevover, each register has potentially a bit field with up to many functions. Even for the somewhat modest SoC in the Raspberry Pi 4 (Broadcom BCM2711), we counted already more than 2500 registers. Engineers at Texas-Instrument report that the sole bring-up of the clocks and power modes in modern SoCs involves reading and writing more than 1000 different registers [2]. In this context, using (and reusing) an IPs is a twofold problem. First, accessing, using loads and stores, the registers/bitfields at the correct addresses/offsets in the correct order at the right time, which can be seen as a "syntactic" problem. Second, setting the correct values in these resources so that the device does what it is expected to do, which is more of a "semantic" problem. Overall, having a working hardware/software system is a tremendous amount of work and debug. It

thus comes at no surprise that the vast majority of bugs in operating systems comes from the device drivers (about 70% of the bugs, with drivers being 70% of the source code in Linux kernel), among which a third is due to the interface between the OS and the device [3]. Provisioning for detecting part of these bugs at run-time is very costly and lead to overheads in the order of 100% [4].

The problem is even worse in the context of FPGAs. Their high programability makes the number of IP version growing even faster. Indeed, it is very easy to implement an IP using either an Harware Description Language (such as verilog or VHDL) or High Level Synthesis (which makes IP development even easier). For instance, only on the Xilinx IP public catalog more than 800 IPs can be found in their last version. Updating an FPGA-based platform is also very fast compared to ASICs or System-on-a-chip. In a couple of hours (sometime minutes), it is possible to update an IP, run a couple of tests, implement a platform embedding the new version and finally program the FPGA with the updated platform. After that, a software update is often required and a huge amount of work can be required to update and debug drivers, even for tiny updates.

2.2 SoC approaches for HW/SW integration

In the SoC domain, easy and early HW/SW integration has been of interest for long. This is due to the fact that because of high cost constraints, the easy exploration of the hardware/software design space is required to optimize the target system for a given application or a class of applications. Taking as an example SoCs that handle flows of data, which is typical for consumer applications, at some point the application ends up specified as a set of interconnected tasks. A first step is then to decide which tasks will be realized in hardware and which will be realized in software. How this partition is done depends on many factors, the least of which is the existence of an HW IP that would be suited for the tasks. Once this choice done, the tasks, independently of their hardware or software nature, have to communicate, taking into account latency and bandwidth constraints. Communication synthesis aims thus at automating the generation of hardware and software required for the communication [5]. Taking the example of streaming applications, [6] proposes a layered approach in which application level, system level, OS level and physical level interfaces are predefined using templates. Synthesizing the communications boils down to instantiating the

right hardware and software elements and setting the appropriate parameters. The approaches based on these principles are well suited for an *ad-hoc* usage, for applications that can be specified using a static task graph and data-flow like communications [7, 8].

More general approaches have been proposed which also target shared memory communication, as reviewed in [9]. They actually build finite state machines for the hardware and software parts from high level specifications. An orthogonal strategy based on the concept of remote procedure call is proposed in [10]. The principle here is to describe the communication in the application through what the authors call "shared objects", that are able to serialize requests and perform actions, either internally if they are hardware IPs, or forward them to a software task otherwise. Although interesting as a different abstraction, the management of the shared objects is a bit unusual, and describing applications with this semantic cumbersome, which make the proposal difficult to generalize.

Overall, the SoC centric approaches are not concerned with the notion of driver per se, and do not deal with kernel interfaces, device sharing, process isolation, etc. They focus on communication abstraction, which is nice, but set aside the OS legacy that simply cannot be ignored when targeting actual processor centric products.

2.3 OS approaches to simplify device driver development

As far as operating systems are concerned, the issue is known and recognized, and different solutions have been attempted, but all under the assumption that the hardware/software boundary remains unchanged. Hardware is seen as a (possibly huge) set of registers that has to be read and written following the constraints expressed in a datasheet.

The first relevant work is automatic driver generation. Most approaches start by defining a domain specific language (DSL) in which the device specificities, described as a mix of hardware resources and behavior, are captured. Then, a tool, specific to an operating system (or even to a version of it to be fully accurate, for example Linux pre 2.6 had a big kernel lock, and now each device has its own set of locks), is used to generate the actual driver [11, 12]. The driver generation tool can be itself parameterized by a formal description of the OS interface [4, 13]. Figure 2.2 summarizes the idea.

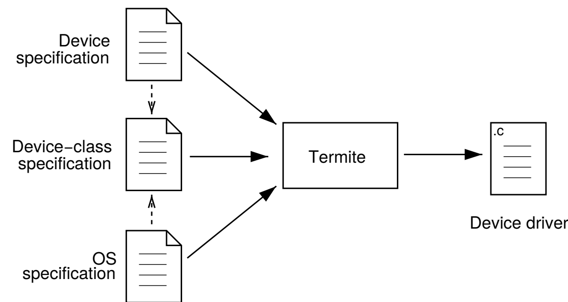


FIGURE 2.2: Principle of driver generation from a DSL (taken from [4])

Another approach has been to push for writing device drivers using high-level programming languages, targetting software challenges such as memory management or synchronization. The work [14] proposes to write drivers in Java, running a Java Virtual Machine inside the kernel. The work [15] even argues to rewrite the kernel entirely in a high-level language, in this case the Go language.

Using DSLs is a direction, but not the only one. Tools and DSLs are expected to help with the syntactic issues, not really the semantic ones. Consequently, they do not help that much with robustness with respect of the complexities of using a particular device. They do help however regarding the driver integration with the operating system by abstracting its API. They ease the following the programming rules defined by the operating system, in particular the complex ones regarding dynamic driver loading and unloading.

Other efforts of the software community have been pushing tools rather than frameworks. Tools like Coccinelle [16] or Coverity [17] are efforts to help write safer kernel code and better support code evolutions. The tools help on the usual software challenges of writing kernel code, such as memory management or synchronization, but the challenges of actually driving hardware devices are not addressed. The work [18] discusses the integration flows based on IP-XACT, an XML format that describes hardware components in order to facilitate their integration.

We see a possible limitation to these approaches. Indeed, all cases have to be thought of when defining the high level languages, which is not easy. So either the language is very abstract and simple to use, but may be limited in scope, or the language allows to describe many details in which case it is hard to use in a generic manner, and a lot of work is required to model accurately the behavior. Therefore, each evolution of a given OS requires new developments, and so does the introduction of a new OS.

The second work is focus at hiding hardware interfaces seen by the software. RIFFA [19], inspired by Microsoft Research's SIRC [20], proposes a reusable integration framework that targets the integration of accelerators deployed on programmable logic. RIFFA combines both software and hardware parts. Running on the processing system, RIFFA provides a C library and a Linux device driver. Deployed on the programmable logic, RIFFA provides a set of harware components. Both sides have been designed for a PCIe bus and a DMA engine used to pass data between the processing system and the accelerators on the programming logic. The RIFFA design is focused solely on accelerators with a design that is very DMA and PCIe specific, but the approach could certainly be extended to other bus technologies and other hardware devices.

It is interesting to note that the DMA transfers of the early RiFFA-1 [19] was later better abstracted by communication channels in RIFFA-2 [21], something very interesting that goes toward the idea of abstract conduits. But the RIFFA library is for applications, running in user mode. While this approach of dedicating accelerators on the programmable logic to a single application solves problematic issues, they do not integrate devices at the operating system level. Integrating it at the kernel level means that there would be a support for a device lifecycle since drivers may be loaded and unloaded and devices may fail. RIFFA does not include any lifecycle management which is something quite critical to address when integrating devices. Indeed, devices are sometimes more than just pieces of hardware booting at power-up and always running flawlessly. They can encounter issues, have some of their features unavailable for a certain period of time, or simply fail until they are reset.

Even though RIFFA has been of great help in its time, it is not maintained anymore. It has suffered a lot from Linux kernel updates making the amount of work to keep it up to date too much for the people still on the project. This shows that maintaining a software/hardware interface as they are today with all hardware and software updates they have to follow is a very hard task even for succesful projects such as RIFFA. It enlights once again that the complexity of this interface is a source of a lot of struggles, making us confident that reducing its complexity can help reducing the efforts required for driver maintenance and debugging.

The work MPRACE [22] follows a similar philosophy as RIFFA, very much centered on helping the development of PCIe devices that require very high bandwidth. The focus of the work is to help with being able to saturate a PCIe link, that is, being able to use all the throughput that a PCIe bus may provide. The C library aims at hidding the

details of the DMA engine from the software developers. The DMA engine is hidden behind FIFO interfaces for the hardware designers. Those choices are in line with FPGA vendors pushing for a DMA engine fronting AXI streams. This represents a great inspiration to push the idea further more to a full message-based solution.

Finally, the third, quite different, domain of operating system research that we believe is inspiring for our own objective is virtualization. Looking at it from the hypervisor or microkernel point of view reaches the same conclusion [23]: accesses to devices by the guest OS are performed through an inter-process communication (IPC) mechanism. However, the intent of these approaches is not to change the hardware-software boundary, but to wrap interrupts as messages, leaving drivers still loading and storing values in and out of hardware registers.

Figure 2.3 illustrates the *split driver* approach promoted by the Xen hypervisor to provide access to devices through an host OS (most commonly Linux). As explained in [24], the principle is to provide ”a simple, narrow, and idealized view of hardware” to the guest OS, which has to be modified so that the drivers are substituted by a front-end relying on this hardware abstraction. The guest OS has a back-end which receives the requests of the front-end, and calls the native drivers to actually access the device. This strategy is simple and clean: the hardware abstraction provided to the front-end allows to access the legacy drivers unmodified on the host through the back-end. Furthermore, a single (front-end, back-end) couple is required for a given driver class, which minimizes the modifications to be done on the guest OS.

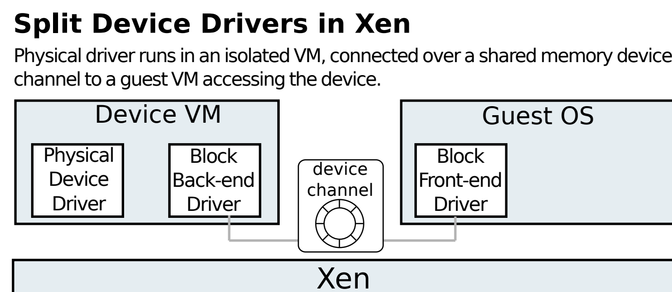


FIGURE 2.3: Xen’s split driver hypervisor architecture (borrowed from [24])

The work of [25] uses this concept to virtualize the devices on a SoC and provide access to them as OS services. The approach makes sense, but is very costly in terms of memory footprint and computing power, as it requires the use of the original Linux device drivers in the back-end of the split drivers.

2.4 Summary

Whatever the perspective taken, the addition of a piece of hardware into a computing system requires the software to be aware of its existence and detailed expectations. This makes the integration of Systems-on-a-Chip, even when using legacy hardware, very complicated. We believe that raising the hardware-software interface abstraction above the register map is a way to ease this integration. It does have a hardware cost, as some wrapping will be necessary to expose an other interface, but the hope is that it will make system software simpler. We choosed to tackle this problem from the FPGA perspective, not only because they are a great candidate for fast prototyping but also because the integration problem we desribed are even worse in this case. Indeed, as hardware is much faster to implement on FPGAs, new updates come at an even higher pace requiring a very efficient integration process. Improving the integration problem for FPGAs thus improves it also for System-on-a-chip in general and makes the adoption of a solution easier.

Chapter 3

Message-Based Integration for Embedded Systems

Chapter contents

3.1 The Current Integration Challenge **18**
3.2 Background **21**
3.3 Message-Based Integration Solution **23**
3.3.1 The Software Perspective 23
3.3.2 The Hardware Perspective 27
3.3.3 The Overall Lifecycle 29
3.4 Messages and Class-Genericity **30**
3.4.1 Using Messages 31
3.4.2 Class-Generic Protocols 34
3.4.3 Heterogeneity 35
3.5 Evaluation **39**
3.5.1 Overhead Evaluation 40
3.5.2 Small Embedded Systems 45
3.6 Summary **49**

The "Soc on a Programmable Chip" trend in which reconfigurable fabrics are integrated into System-on-Chip (or the other way around) started in the early 2000's [26]. It is now legacy, and this technology is routinely used to associate software running on processing systems with hardware components in programmable logic. In the embedded system world, these components are often controllers of external physical devices such as human interface devices, mass storages, or sensors. They may also be accelerators such as encryption or audio and video processing components. Moving such

functionalities from software implementations to hardware components is interesting when searching for better performance and lower power consumption, especially for small embedded systems. Moving more and more functionalities in hardware is thus the way to go to get the better out of a small system, but it comes with a huge trade-off: the integration challenge. Integrating hardware components is not only harder than implementing their software equivalent, but it spreads out the range of skills required by embedded system designers to make their system work. Indeed, the remaining software running on processor-centric systems have now to communicate with more hardware components than before, making the integration challenge a major concern for them.

The approach we propose to tackle the integration challenge is based on two principles: an abstract message conduit and class-generic protocols. Those concepts are the origin of the USB success story, showing how easy, low-cost and efficient it is to plug any device on any system. The philosophy of the approach is to provide to system designers a similar plug-and-play experience to integrate devices in their systems. It also provides device makers a consistent interface independent of the system in which their devices may be integrated. Finally it provides driver writers a generic interface simplifying the interface of devices by replacing hardware details with message protocols. This approach has been implemented and demonstrated for small embedded systems, including a bare metal software stack running on ARM based processing system and Xilinx programmable logic.

3.1 The Current Integration Challenge

The integration challenge consists of making hardware components functionalities exploitable by software. Those hardware components, also known as IPs (Intellectual Properties) are assembled and wrapped in devices waiting to be exploited by the software running in processing systems. The Figure 3.1 illustrates the integration process that embedded system designers are facing. The integration process requires both hardware and software skills.

Hardware skills are required to design a device and deploy it in the programmable logic. Designing and deploying a device consists of assembling one or more hardware components wired together and wrap them in an interface interconnected in turns to the processing system. To do that, the device must be implemented and integrated in

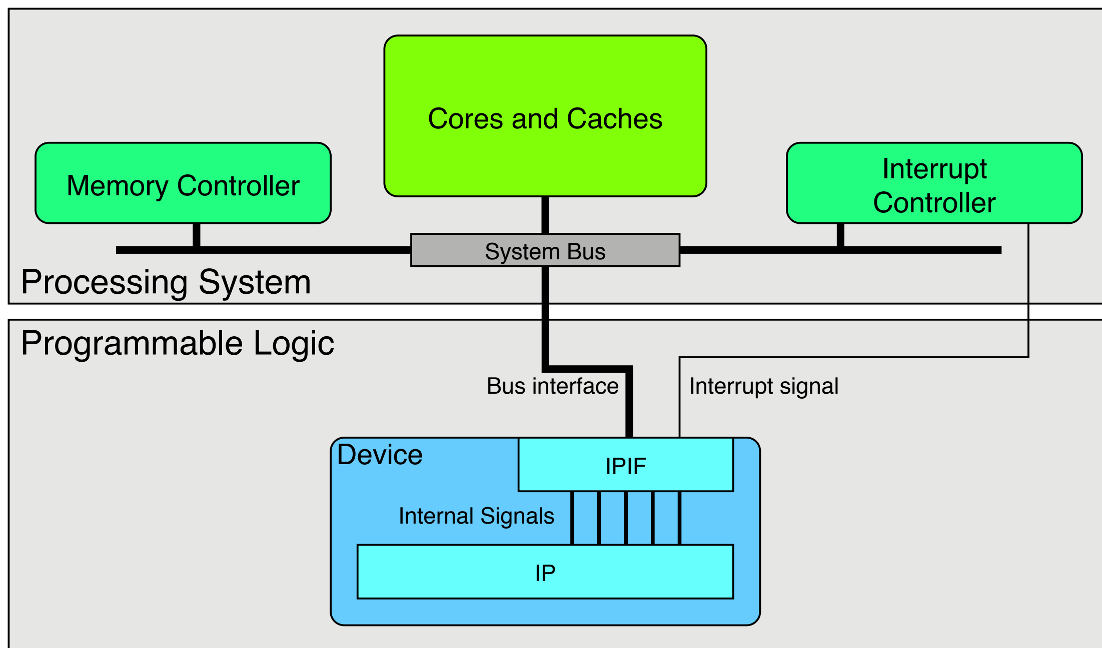


FIGURE 3.1: FPGA Device Integration

the system. The implementation phase consists of implementing and assembling all the components of the device using either Hardware Description Languages (HDL) or High Level Synthesis (HLS) tools [27–29]. The integration phase consists of putting the device in an hardware system typically consisting of a system bus and an interrupt controller. The system bus is used to send commands to the device, check its status or exchange data. The interrupt controller is used to notify processors of hardware events via single wires containing no other information than ”something happened, check it out”.

The software skills are necessary to allow the software stack running on the processing system to drive the deployed device. Each deployed device thus requires a device driver, a specific piece of software that knows how to exploit the specific features provided by the device. Device drivers are the vast majority of the time running in the bottom of the software stack, inside the operating system kernel. A typical example is a block device capable of storing blocks of data, exploited by its driver. The driver will perform read and write data block operations to its hardware device providing the foundation to a file system within an operating system.

A device driver drives its device through the system bus and the interrupt controller. This interfacing is done by a specific hardware component, often called an IP InterFace (IPIF), a hardware component acting as an interface bridge between the internal signals of the device and the system bus and interrupt controller. Some of these signals are

combined to generate interrupt signals to the interrupt controller while others are grouped as bit fields of hardware registers.

Hardware registers are then memory-mapped within the address space of the processor by the system bus. That way, the device driver can read or write these hardware registers by issuing regular load and store operations to the system bus. Traditionally, the memory mapping of a device reserves a contiguous range of memory addresses in the processor address space. This means that each device is associated with a base address, the first address of the reserved range. Each hardware register is then mapped at a specific offset from that base address.

Hardware registers have two main purposes: control and data transfers. Some control registers are used to issue commands to the device while others are used to read the current status of the device. In contrast, data registers are used to send and receive data to and from a device. This means that data registers are usually implemented in hardware as FIFO queues. FIFO queues are simple to drive from software, through load and store operations, but they require the CPU to take charge of data transfers, something not always the best for performances. Therefore, most modern devices embed a Direct Memory Access (DMA) engine to optimize data transfers. With a DMA engine, data transfers to and from memory are under the responsibility of the device, writing and reading to and from memory buffers. These buffers are setup in memory by the device driver.

Overall, this integration process is complex, hard to debug, requiring both hardware and software skills and thus costly. From a hardware perspective, the main challenge comes from the interface heterogeneity, especially the diversity and number of interfaces for system buses and interrupt controllers. As a consequence, this diversity almost requires to design a different IPIF for each embedded system. For instance, each bus specification usually defines its own burst capabilities, cache coherency, or frequency limitations. There are also different interrupt controllers, each with different interrupt types, using different signals with different signal constraints.

From a software perspective, the situation is not any better. Writing a device driver is known to be a complex task even for kernel-space software engineers. The main reason is the complexity of the nature of the software-hardware interface. Each bit of each hardware register must be understood, along with timing constraints or ordering constraints across hardware registers. Too often, a deep understanding of some internal details of the device is necessary to correctly drive that device, which requires

software developers to have hardware skills. Furthermore, error conditions are usually hard to understand and to handle properly from a software perspective. Finally, interrupts are completely asynchronous and requires an accurate handling of complex race conditions in software, especially when writing device drivers for high-performance devices.

All this is not good news knowing that device drivers account for 54% of the code base of the Linux kernel. It is no surprise then to see that 74% of bugs in operating system kernels are device driver bugs, most of them leading to a complete system crash [30, 31]. Worse, the integration process is a never ending struggle. New devices are constantly appearing requiring to repeat the whole process over and over. Indeed, new devices appear all the time and existing devices constantly evolve. Also, software stacks and kernel APIs are updated regularly requiring drivers to be updated to stay compatible. This requires writing new documentations or even worse, evolve existing documentations with many small edits scattered throughout pages.

This constant struggle makes it hard to have available drivers for the latest devices. Furthermore, it makes it hard to rely on safe drivers, unless a designer limits its design to integrate devices that are old enough to have well tested drivers. This leads to the unfortunate situation where a designer is trading innovation for safety.

3.2 Background

The integration challenge has been the subject of various related work, both on the hardware and software communities. Though, most of the efforts are rarely on both the software and the hardware side at the same time. Even more rarely has the message paradigm been advocated as an end-to-end game changer.

The Universal Serial Bus (USB) [32] standard is one exception. The work started on the hardware side, trying to solve the flood of plugs and cables in the 80's by adopting one unique physical standard to plug external devices to personal computers. In its hardware architecture, it is based on a master-slave serial bus, organized as a tree in which each node is a hub and each leaf a device. As a serial bus, the message paradigm seems logical and the USB standard built on that foundation to define device classes and promote class-generic drivers. We all know, including non expert people, how successful that approach has been.

The success of the USB standard is explain by two key elements: A message-based interface and class-generic devices. They demonstrated that a message-based interface can be designed at a very low cost (the additional chip that must be put in front of a PS2 mouse to makes it a USB mouse cost only a few cents). A message-based interface is great for device drivers development. Drivers do not see hardware registers and interrupts, they only see a set of pipes connected to device endpoints. In those pipes, drivers can send and receive messages to drive their devices. For instance, a USB mouse driver will receive messages containing a set of bytes describing button states and movements deltas without having to read or write any registers or reacting to an interrupt. For software developers this is not only very easy to do but it also removes a lot of points where critical bugs can be introduced. For instance, not having to write interrupt handlers removes potential bugs that almost always lead to system failure [33].

Class-generic drivers are key to the success of USB because they tremendously reduce the number of drivers require to support hardware devices. A class of device is a set of message protocols that all devices belonging to this class understand. For instance all storage devices follow the same protocol allowing drivers to use the internal storage of such devices. This explains why all USB sticks works without ever installing additional drivers.

Internally, the USB standard is based on a host-device design. The host controller regroups all device connections in a star topology (that can be extended using hubs). A host controller driver handles all basic communications to probe existing devices, get their description. This driver also handles data transfers and provides to higher level device drivers the ability to open pipes and send and receive messages to their device.

Unfortunately, the USB standard is a closed world, still very focused on cables and plugs, and it has not been adopted to connect devices internally in SoC, let alone deployed in programmable logic. Even though the key concepts could apply to the challenge we are tackling, the USB standard cannot be adopted as is. The main reason is that the USB standard is not abstract enough, as it is too tied in many ways to its master-slave serial bus design. Though, the key concepts are inspiring to design a proposal whose foundation would be an abstract message conduit, insulating both the device and the driver from the implementation details of that conduit. This opens then the path to class message protocols that can be specified as completely abstract protocols, independent of any underlying conduit, even ours.

This independence between the protocols and the conduit was inspired by the Small Computer System Interface (SCSI), an earlier attempt at physically connecting and transferring data between computers and peripheral devices [34]. SCSI has a much better separation between generic drivers, using abstract SCSI protocols, and underlying low-level drivers that are actually driving specific hardware devices. In fact, a SCSI device can be driven through any message conduit, even the Internet, with a mass storage located in a data center half way around the world. But unfortunately, the SCSI architecture does not address the challenges to write low-level drivers. Drivers are still written as they always have been, using registers and bitfields tied to hardware implementations. SPI and I2C are also examples of the SCSI story, even though they managed to offer abstract conduit, they are most of the time used the old way, using registers and bitfields [35].

In contrast, there has been quite a number of other efforts from the software community tackling the difficulty of writing device drivers. Most operating systems provide high-level frameworks for developing device drivers, helping with the integration within the surrounding operating system stack. Apple Corp. provides the *I/O Kit Fundamentals* [36]. Microsoft Corp. provides the *Windows Driver Framework* [37]. But again, none are improving the situation regarding the challenges of actually driving a hardware device. Nevertheless, these improvements would equally be applicable to ease the writing of message-based drivers.

3.3 Message-Based Integration Solution

Our solution proposes to organize the interaction between a device and its driver via bidirectional channels that permit either side to send or receive messages, each message containing a variable-size payloads of uninterpreted bytes. Channels are therefore software-hardware constructs, acting as conduits for messages between the software stack running on a processing system and hardware devices deployed in programmable logic.

3.3.1 The Software Perspective

From a software perspective, the solution extends a fairly standard framework for matching devices with suitable drivers, illustrated in Figure 3.2. Inspired by Linux,

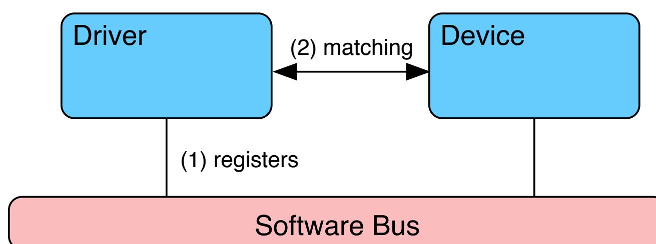


FIGURE 3.2: Focus on driver side

it is based on the same software constructs: a *bus*, *drivers*, and *devices*. Both devices and drivers are registered with the bus (1). For each available hardware device, the bus creates a corresponding *device*—a software construct that describes the hardware device. Once a device is created, the bus searches for a matching driver (2), picked amongst the drivers that are registered with the bus.

The bus is therefore fully dynamic with respect to available devices and registered drivers. Devices are created when they become available and removed when devices fail or become unavailable. Drivers may be registered and unregistered at any time, using the two functions listed below that are part of the software bus interface:

```
void register_driver(struct driver* driver);
void unregister_driver(struct driver* driver);
```

Notice that each driver is described by an in-memory data structure, see the Listing 3.1 below. In that structure, the field *devices* specifies a list of compatible devices as an array of device identities, an array that is used by the bus to filter devices that can be matched with this driver, simply comparing device identities. The identity of a device is composed of the usual numbers provided by all devices: vendor number, device number, and release number. Of course, this limited device identity is not enough for an accurate matching of devices and drivers, it is only a pre-filtering. Therefore, a driver needs to probe and tries to configure a device before a match can be considered as valid. This requires a specific matching protocol between the bus and a driver, using the following two function pointers: `matched` and `unmatched`.

```
struct driver {
  const struct device_id* devices;
  int (*matched)(struct device*);
  void (*unmatched)(struct device*);
};
```

LISTING 3.1: "driver structure"

These two functions are defined by the driver and called by the bus. The bus calls the function `matched` to notify a driver that it has been matched with the given device, based on the simple device identity filtering. This is when the driver starts probing and configuring its device using messages. This is of course an asynchronous process that may require sending and receiving several messages. But as far as the bus is concerned, the driver and the device are matched. At any point in time, the driver may decide that the device is not a match after all and ask the bus to unmatch that device, calling the following function `unmatch` that is part of the bus interface:

```
  void unmatch(struct driver* driver, struct device* device);
```

Of course, the bus may also decide unilaterally that a device and a driver must be unmatched. This may be needed for several reasons. A device may have failed. A device may be controlling an external device that has been unplugged. Or it is an administrative decision to either unload the driver or to disable the device. To achieve this unmatching, the bus calls the driver through the callback `unmatched` setup in the driver structure, seen earlier in Listing 3.1. Once a device have been umatched from a driver, that device is never rematched with that driver until the next reboot of the system or until the driver is unregistered and then registered from the bus.

While a driver and a device are matched, they solely communicate through sending and receiving messages through channels. The available channels are managed by the bus and described in the following device structure:

```
  struct device {
    struct device_id id;
    unsigned int num_channels;
    struct channels* channels;
  };
```

Each channel provides the capability to both send and receive messages. The `send` function is setup by the bus and called by a driver to send a message through the channel. The `received` function is setup by the driver and called by the bus when a

message has been received from the device. Note that both sending and receiving messages are non-blocking asynchronous operations.

```
struct channel {
  void (*send)(struct message msg);
  void (*received)(struct message msg);
};
```

It is the responsibility of the driver to setup the `received` callback for all the channels defined by a device and this must happen when the driver is matched with that device. This means that a driver must, before returning from the `matched` call, setup the `received` callbacks in all the channels and then confirm this setup by asking the bus to connect the channels via the following bus function:

```
void connect_channels(struct device* device);
```

Once channels are connected, a driver can send and receive messages. As discussed earlier, this usually starts with probing and configuring the device, potentially negotiating which message protocol to use through which channel. Messages are variable-size payloads of uninterpreted bytes, described by the following structure:

```
struct buffer {
  uint8_t* bytes;
  size_t nbytes;
}
struct message {
  struct msg_channel* channel;
  struct buffer* payloads;
  void (*released)(struct message msg);
};
```

A message refers to the channel it is travelling through and it refers to a payload. The payload may be only one buffer or a sequence of multiple buffers. This interface allows to exploit scatter-gather DMA engines and avoids unnecessary memory-to-memory copies. This is useful not only to add headers without inducing unnecessary copies but also in permitting application messages to be across multiple buffers. Received messages may also be scattered across multiple buffers, which is interesting to let device to send multi-part messages. This also leaves the bus designer the required implementation freedom when it comes to implementing channels and the corresponding exchange of messages between the processing system and the programmable logic.

Note the `released` callback that permits an efficient memory management protocol that is compatible with our asynchronous send and receive operations. The `released`

callback notifies the creator of a message that the message has been consumed: either it has been sent or it has been processed once received. When sending, the callback is set up by the driver and until it is called by the bus, the driver may not modify or free the message or the message payload. When receiving, the callback is set up by the bus and until it is called by the driver, the bus may not modify or free the message or the message payload.

Thanks to this protocol, we can avoid memory-to-memory copies entirely when sending or receiving messages. For received messages, the bus gives a direct pointer to its internal buffer where the message was received directly from the hardware device. If the driver is able to handle the message within the `received` callback, it can call the `released` callback before returning. This is the most efficient scenario. However, if the driver needs to make a copy, it can use a memory-to-memory DMA engine, in which case it will call the `released` callback upon the completion interrupt of the DMA request.

3.3.2 The Hardware Perspective

The Figure 3.3 illustrates the hardware perspective. The device, deployed on the programmable logic as a set of interconnected hardware components, is interfaced to a device controller. The device controller acts as a bridge to the software world. The device controller exposes a set of channels and a reset signal. The device controller comes as a generic hardware component that is configurable at design time. The designer can choose the number of channels, with the channel `0` being always present. The designer can also choose between high-performance or low-performance streams, depending on the expected bandwidth.

This suggests that device controllers will be ultimately designed and provided by the vendors of the programmable logic, such as Xilinx or Altera. Until then, device controllers can be designed as regular hardware components, using standard design tools and leveraging standard interfaces to system buses and interrupt controllers. For instance, we were able to design our device controllers using hardware components available in the Vivado tool from Xilinx. Although we used existing components, nothing precludes the complete design of specific device controllers that would exploit specific hardware features.

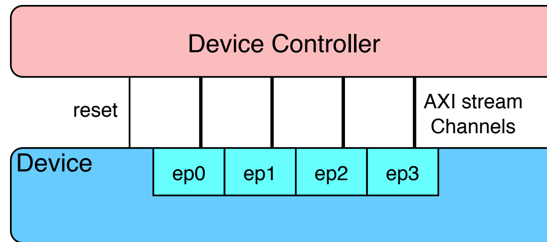


Figure 3.3: Hardware device perspective

From the device perspective, each channel is a pair of unidirectional data streams represented by endpoints, one endpoint being a stream slave to receive data and the other being a stream master to send data. These streams seem to require a full specification of the corresponding signals. While we suggest to use an existing standard, we do not impose any specific stream specification, leaving tools and boards designers with enough implementation freedom. For illustrative purposes, one could choose the AXI-stream standard that defines for each stream a set of data signals, a pair of control signals, and an end-of-packet signal that is useful to demarcate the last word of a message. We will discuss later the pros and cons of imposing a stream specification or letting each vendor choose their own.

Even though the details of the streams may be specialized, the concept of streams is not imposed. Furthermore, each stream is used to either send or receive messages, not both. This design is better than multiplexing channels onto a single pair of streams. Indeed, it simplifies the handling of messages for the device, with a simple automaton per endpoint. It also permits the hardware designer to tailor the buffering capabilities of each endpoints, better controlling resource consumption in the programmable logic. It also offers greater flexibility when it comes to bandwidth and latency. For instance, certain endpoints may be used to transfer large data messages while other endpoints are used for out-of-bound small messages. Finally, this allows to configure each endpoint as either high or low performance.

Messages have a very simple format. Each message includes a fixed-side header followed by a variable-size payload. The header contains two fields, one is the message type and the other is the length in bytes of the payload. The message type is used to differentiate control and data messages, a distinction that will become clear soon. Each message must be transmitted as a whole, therefore, each endpoint is a first-in-first-out lossless stream of messages. However, there is no timing constraints on the stream, a device may receive or send a message at its own pace.

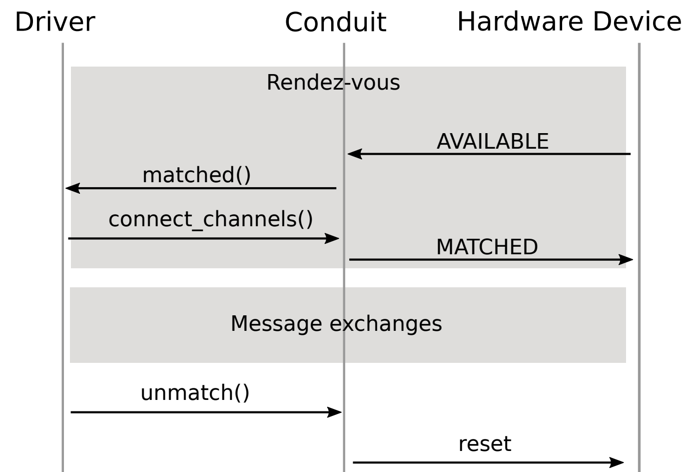


FIGURE 3.4: Device lifecycle

3.3.3 The Overall Lifecycle

The Figure 3.4 illustrates the lifecycle of a hardware device. Notice that this lifecycle involves not only the hardware device and its controller controller on the programmable logic but also the software bus and the device driver on the processing system. Three actors are involved: (1) the device on the programmable logic, (2) the device driver on the processing system, and (3) the message conduit that sits in the middle, across the programming system and the programmable logic. This conduit is thus a mix of software and hardware, composed of the software bus on the processing system and the device controllers on the programmable logic. The conduit is responsible to route messages through channels but it is also responsible to coordinate the lifecycle of both devices and device drivers.

Upon a general reset, all device controllers are held in the reset state, in turn holding their devices in the reset state. When a device controller wakes up from the reset state, it wakes up its associated device, releasing the reset signal. When a device wakes up from the reset state, the device initializes internally and then sends a first message to its device controller, through the endpoint 0. This is entirely asynchronous and each device may take as long as necessary to send this first message. This means a device is allowed to never send that first message, which may happen because the device is currently disabled or because it cannot recover from a previous failure.

This first message, *AVAILABLE*, contains the identification of the device and indicates to the device controller that the device has initialized properly and is now available. Upon the reception of this message, the device controller informs the software bus running on the processing system that it should reify the available device and attempt

to match it with a driver. At this point, the device is waiting for a driver to be matched. This matching will happen on the processing system at some later time, if it happens at all. If it happens, the matched driver will call the function `connect_channels` on the bus and the bus informs the device controller that in turn will send a message *MATCHED* to the device through the endpoint 1.

This message *MATCHED* finishes the rendez-vous between the hardware device and the software driver, both sides may now freely send or receive messages. Indeed, remember that channels are full-duplex and not master-slave, even though individual endpoints are master-slave streams. Note that sending messages advantageously replace the more traditional scheme of raising interrupts that was used to signal the processing system. The message can carry the right information describing the condition that requires the processing system attention rather than having the driver peek at the hardware register trying to discover what needs to be done.

This lifecycle is designed so that either party may break the rendez-vous. The most likely scenario is that a driver is unmatched from a device, either because the driver failed or because it is unregistered. In either case, the function `unmatch` is called on the software bus that will inform the corresponding device controller of the unmatch. The device controller will react by cutting all the data streams to and from all endpoints, preventing any further message transfers. Next, the device controller will reset the device, restarting a new lifecycle.

The driver may not be the source of the unpairing, it may be the device because of a failure. The failure may be a hardware failure or a failure resulting from requests from the driver. When a device detects a failure, it must send an *UNAVAILABLE* message to its device controller. Upon the reception of the *UNAVAILABLE* message, the device controller will cut all transfers through the streams to and from all endpoints, preventing any further message transfers. It will inform the software bus and then it will reset the device, restarting a new lifecycle. If the device cannot recover after being reset, it just does not send the *AVAILABLE* message.

3.4 Messages and Class-Genericity

The software-hardware interface proposed is based on messages. Even though this is not the only approach, it is superior to the traditional register/bitfield one when

dealing with more and more devices to integrate. The fact that USB is based on those concepts is a good hint but it is important to understand how important messages are.

3.4.1 Using Messages

For software developers using messages is much simpler. With the traditional approach, developers have to have both software and hardware skills. Furthermore, they have to be capable of reading and understanding the complex documentations of both the operating system kernel in which the driver is to be integrated and the complex, dense, constantly evolving, documentation of the device. This type of knowledge is relatively rare and requires highly competent engineers hence making drivers not only hard but costly.

With the message approach, almost any kernel developer can write a device driver. There are no hardware documentations, just a message-based protocol to read and understand, something that any software developer knows how to do. All the hardware details are hidden, it is just about receiving and sending messages, just like it would be over regular POSIX sockets, over the Internet, to a standard web server. Of course, it is still about writing kernel code, which is harder than user-space code.

Channels and messages are kept as simple as possible. Full-duplex channels and variable-size payloads is quite standard and feels familiar to software developers. Note in particular that channels hide the complex setup of scatter-gather DMA engines. Of course, a DMA engine may be used for performance, but it is an internal implementation detail of the conduit that channels provide. If there is a DMA engine, it is part of the device controller, it is no longer part of the device itself. This contrasts with the traditional approach in which each driver would have to manage a DMA engine that would be specific to each device since it was part of the device design. The approach is also safer since devices are no longer provided a direct access to memory.

Failures are addressed by adopting a simple and well-known fail-stop model [38]. In the real life of a device driver, device failures are not handled very well [39]. In particular, most device drivers do ignore the overwhelming number of failure-related status flags that most modern devices exhibit. Worse, some of these flags may only hint at an error, with the documentation telling you that the device may properly work, or may not. It explains why most drivers just reset their device at the very first hint of something not working as it is supposed to. A fail-stop approach makes it clearer for

software developers and hardware designers: a device or a driver either works correctly or it must stop without in-between status.

However, this simplicity does not prevent advanced error management, with complex error reporting. Indeed, there is a strong distinction between error reporting and lifecycle. From a lifecycle perspective, a device or a driver either runs correctly or stops. But regarding individual requests, the request may succeed or fail, without the device or the driver failing. For instance, a block device may report different error conditions when a write or read operation of a block fails, but this does not mean the block device has failed. The status report of a request is simply done by sending a response message once the request has been processed.

The use of messages provides a better separation of concerns, which impacts the size and complexity of the documentation. Today, the documentation of a device has two targeted audiences: the hardware community and the software community. This makes the writing of the documentation more complex than it ought to be. Also, it ties in one documentation the public interface of the device with the details of its implementation. With messages, the only public interface is the message-based protocol.

Additionally, this solution opens the way to have message-based protocols be specified independently from any given hardware implementations. As an example, open standard bodies such as the Linux Foundation[1] may become the reference to define such standards. Using message-based protocols also becomes really interesting when evolving existing devices with new functionalities. The new features will be designed as new messages, extending an already existing protocol. Therefore, it will be easy in most cases to preserve backward compatibility, it is just about supporting the non-extended protocol. This way, any previously existing driver would still be able to drive new releases of the device, just without the extended features. Furthermore, a message-based protocol is easy to extend, just add new messages or extends existing messages, in a backward compatible way.

In the case of major revisions, it may not be possible to extend the protocol, a new protocol may be necessary. This does not mean losing backward compatibility. New devices may support both the old and the new protocols. It can be done using different channels for the two protocols. It can also be done through a single channel, with the negotiation of the appropriate protocol during the probe and configure phase between the driver and the device.

[1] https://www.linuxfoundation.org

This is better than the current situation where evolving a hardware design usually changes the bit fields of the memory-mapped hardware registers. This usually means a rewrite of the documentation and of the code of all software drivers for all software stacks which always expose the risk of introducing errors and bugs. With larger and larger devices and more and more large software stacks, the situation is not sustainable in the long run. Specifying message-based protocols and offering the ability to negotiate protocols is just so much simpler.

For hardware designers, message-based protocols introduce some changes compared to a traditional approach, a device has a bunch of signals that can be easily mapped to bits in hardware registers and interrupts. This process is well supported by tools. The maintenance of the documentation is hard and even if the maintenance of the documentation is improved or automated, the challenge is just past down onto software developers that must update their drivers.

Using messages requires an extra hardware component bridging the world of messages with the world of hardware signals. When receiving a message, the message must be parsed in order to drive hardware signals of the internal device logic. In the opposit way, a message is composed using output signals and then transmitted. This wrapper will likely be written in HDL at register transfer level or using high-level synthesis. Even though it represent an additional component, it is straightforward in most cases, usually a few hundred lines of relatively simple HDL. The complexity of that extra component is dependent on the complexity of the message protocol implemented by the device.

This additional work imposed to hardware developers thankfully comes with major gains. Device designers no longer have to include a DMA engine, they are only dealing with stream-based endpoints. This means that hardware device designers are now insulated from all the details of the various bus interfaces and many interrupt controllers. Also, the details of their hardware implementations are now fully encapsulated and can be changed without impacting the public interface of the device. Hardware designers have also more freedom with a message-based interface. Indeed, the device logic is no longer so synchronous and entangled with the management of bus transactions. Using messages completely decouples the device from the system bus timing constraints or the different policies regarding interrupt signals.

3.4.2 Class-Generic Protocols

Adopting messages comes with great benefits, both for hardware designers and software developers. This is really important because message-driven devices are the key to defining class-generic protocols, which is the next game changer. The industry as a whole as already taken advantage of the power of message-driven devices and more specifically the power of specified class-generic protocols. Indeed, this has been at the heart of the success of the Universal Serial Bus (USB) specification.

By hiding the details of the connection between the processing system and the programmable logic, the solution allows to take advantage of standardized class-generic protocols. The idea is to identify classes of devices and to specify one standardized message-oriented protocol for each class. The revolution then comes from the fact that one driver is now capable of driving any device belonging to a class, not just one device from one vendor. In fact, many such specifications already exist under one form or another.

With class-generic protocols specified by an open body such as the Linux Foundation, any new device that belongs to an existing class can hit its market without any delay, knowing that there will be a driver already available. This also means that device makers do not have to define message protocols, they just have to select an existing class of devices and implement the corresponding protocol. However this may seems like it imposes limitations on new devices, limiting hardware innovation.

We all know that the industry suffers from a paradox. On one hand, they want new devices to be immediately usable and sold as soon as they are implemented. This requires to have a device driver available in all software stacks. This means the usual operating systems such as Windows, Mac-OS, and Linux, but this also means across all the proprietary and open-source software stacks for embedded systems. This objective is well-served by class-generic protocols. On the other hand, the companies also want devices to have unique features that will give them a competitive advantage among other competitors in the market. From that perspective, class-generic protocols may seem as refraining innovation.

A class-generic protocol must be seen as a minimum message-based protocol that ensures a device is usable across all software stacks because there will always be a corresponding class-generic driver. But this allows to have device-specific drivers as well. Such device-specific drivers would support the class-generic protocol but would also

be able to negotiate vendor-specific extensions with devices that they recognized. This is why it is so important that a driver can negotiate a protocol with a device before being matched. Device makers thus don't have to choose between standard compliance and a feature-rich competitive edge.

For instance, GPU makers that are known to hide the device specifications could now have GPU devices exposing both standard and proprietary protocols. Each GPU could therefore be handled by a standard driver and at the same time offer highly-optimized drivers using proprietary protocols. This is the direction highlighter by the Khronos group and more specifically the Vulkan GPU interface underlying an OpenGL pipeline. While this is future research, it shows that high-performance devices such as GPU have already adopted drivers that provide asynchronous queues for submitting work to a GPU.

Additionally, the adoption of class-generic protocols has an interesting side effect: improved safety. Drivers are the source of the most bugs and crashes in modern operating systems [40], except for class-generic drivers. There are several reasons for that. First, they are simpler to write and therefore have less bugs. Second, they do not manipulate DMA engines and interrupt handlers, where bugs usually induce a complete system crash. Third, they are widely used as one driver is used across a class of devices, not just one device, which means they are better field-tested through longer periods of time since standardized protocols rarely change.

The stability of class-generic protocols greatly improves device safety. Class-generic drivers will become safer and thereby ensuring that even the greatest and latest devices can be used, even if it requires to sacrifice using some of the latest features until their proprietary drivers become stable enough. This could be a real improvement for the open source industry and Linux in particular.

3.4.3 Heterogeneity

The conduit is a new abstract interface between the software and hardware worlds. This means that at least one conduit must be provided for each platform. Since the conduit starts at the software bus interface facing drivers and ends at device controllers facing devices, it combines both software and hardware components. Our proposal is completely abstract and therefore provides the required freedom to adapt the implementation to the specifics of each platform.

Embedded systems are often very heterogeneous systems letting think that a single software-hardware interface may come with too much restrictions. Indeed, such an heterogeneous environment might let think that it requires different interfaces. But even today, we are facing an abstract interface in between software and hardware, the one with hardware registers and interrupts. The message-based proposal is just proposing to build one more abstraction above, based on message-based conduit that could work as well across software stacks and hardware technologies.

The trade-off we are facing is that on one hand, we want design flexibility to handle heterogeneity and on the other hand we want a generic software-hardware interface standard based on messages. Flexibility is really important since our proposed approach must integrate with a wide range of software stacks and a wide range of hardware platforms. And yet we want one stable software interface available to drivers and one stable hardware interface available to devices. Though the message-based approach offers a great balance between those two concerns.

From the hardware side, our proposal specifies a device controller that requires a stream interface, used by channel endpoints. We could specify a dedicated stream specification or even better pick an existing one such as the AXI stream. Assuming we adopt the AXI-stream interface, an interface that is easy to support on any programmable logic, we could argue that we have a generic interface between devices and their device controllers. But even with this specified interface, device vendors will still have to port each of their devices across different programmable logic technologies, mostly likely using different vendor-specific tools to do so. This is already the case today, but this difficulty is compounded with the heterogeneity of the system bus interface and the heterogeneity of the interrupt controllers. With our approach, with a unique stream specification, the hardware integration would be simpler.

But there is no need to impose a unique stream interface. In fact, the solution allows a specific vendor to propose a proprietary stream interface, rather than adopting the AXI-stream specification. Indeed, the strength of the proposal is not in the stream details, it is in the message-oriented specification that can be implemented over any data stream interface. This means the added value is in the general adoption of the overall message-based approach, across different platforms from different vendors. It would therefore be best if each vendor would integrate in its own tooling a general-purpose configurable device controller, adopting the stream interface of their choice, but one device controller that adheres to our proposal.

Fortunately, we do not have to wait until programmable logic vendors do so before our proposal can matter. Indeed, a device controller is a regular hardware component, so it could also be developed and provided by an open-source community. As a regular hardware component, it would be connected to the interrupt controller and the system bus, using memory-mapped hardware registers, even potentially using a DMA engine. It would be written in an HDL or using high-level synthesis. And as all current hardware components available across platforms today, the device controller would need to be ported across platforms and tools from different vendors. This is where heterogeneity will place the most burden on our proposal. However, this would also be an opportunity to adapt the device controller and the conduit to the specifics of different classes of platforms. This represents an extra work, but this is valuable work.

Considering the different platforms and corresponding designs, we start simple with considering small embedded systems first with a combo of a programmable logic and a processing system. At the very minimum, the processing system will include a small processor, a system bus, and some memory, the programmable logic will probably be small so the size of the device controller matters. For those platforms, a small implementation of our message conduit is possible. In fact, the device controller can be reduced to almost nothing.

The minimum device controller is a multiplexer/demultiplexer component connecting the endpoint streams and a pair of memory FIFO queues. In this case, most of the conduit is implemented in software. In particular, the lifecycle is entirely managed in software, with channels being easily implemented above a pair of FIFO queues, one to receive messages and one to send messages. Each message is composed of a header and a payload as explained earlier. However, the header has an extra field: the endpoint number. This extra field allows the device controller to route messages to the correct endpoint. Conversely, the device controller will add the endpoint number in front of messages sent by the device.

This design allows for very small implementations. For instance, if a device only has one endpoint, this multiplexer/demultiplexer component is not even necessary and the extra field in the message header may be removed. But nevertheless, all implementations fully respect our software-hardware interface based on messaging. This improves the portability of hardware devices, even though there is still some required work to port a device across different vendors using different programmable logic technologies and tools.

A similar situation is faced with the class-generic drivers. Across platforms, a class-generic driver could conceptually be reused without any modifications, but in practice, this is not entirely the case since a driver is dependent upon the surrounding software stack, even potentially using different programming languages. But this is not the hard part, software portability is a well-mastered problem and the problem exists with or without using messages. So drivers will need to be ported and using messages make it simpler as using messages is less entangled with the surrounding software stack than using hardware registers, interrupts, and DMA engines.

But how difficult is it to port our software library that provide the framework for the bus, devices, and drivers. This is not free, especially that the traditional software-hardware interface is visible internally. Fortunately, most vendors provide a thin hardware abstraction layer (HAL) for standalone embedded applications written in C or C++. This is all we need to design and implement our proposal on the processing system. Reusing the Linux kernel entirely is obviously another option, giving us a portable HAL, but it is only suited for larger embedded systems. Since our framework is based on the Linux concepts, it can be easily integrated within the Linux kernel as regular kernel modules, something addressed in the next chapter.

Remains the question of driving the device controllers. Fortunately, the requirements of our solution are quite simple and quite standard. Device controllers may use simple FIFO queues on small embedded systems for sending or receiving messages, a simple design that is low-cost. For larger embedded systems, potentially integrating high-performance devices, device controllers behind FIFO queues may no longer be suited. To speed up data transfers, we need to replace FIFO queues by a scatter-gather DMA engine, relying on a pair of in-memory rings containing buffer descriptors.

With that design, one ring is used to send messages and the other to receive messages. Both operations are entirely asynchronous, which is well supported by the asynchronous design of our interface. Supporting asynchronous scatter-gather DMA was a primary design goal when we designed our buffer managements regarding messages and the asynchronous notification through the `released` callback we previously discussed. Although we just introduced a DMA engine, we preserved the overall safety.

Indeed, although we used a regular DMA engine, the DMA engine is not part of each device, it is part of each device controller. In other words, the DMA engine is hidden to both the driver and the device. This means that the device is never granted memory access directly. The only interface the device sees is the stream interface, not the DMA

engine. This is a radical change from the traditional approach where the DMA engine is an integral part of the device and where an IOMMU would be required to preserve the same level of safety.

Using a DMA engine is the first optimization, but our design can be further improved by moving the lifecycle management from the processing system to the device controller. On small system which use FIFO queues it is likely that the entire lifecycle management will be done in software. But for larger embedded systems, with larger programmable logic, one may consider to move the lifecycle management in the device controller. This trades better performance for a larger footprint on the programmable logic, but the footprint increase is very reasonable, less to a few hundred look-up-tables (LUTs), the elementary logic element of FPGAs. Also, this design frees the processor from the lifecycle management, which is a design that scales up much better. Supporting the lifecycle management in device controllers is a small development effort, with a few hundred lines of straightforward HDL.

Having device controllers that manage the lifecycle opens up the path for hot plug and unplug of devices at the hardware level. This may not be seen as a required capability for embedded systems, but it actually is the case for several reasons. First, even if the logic of the device may be deployed once for all, that logic may control an external device that may be available or not (plugged in or not, powered up or not). Second, dynamic management of programmable logic is increasingly relevant as programmable logic makes its way to larger systems. This means that the loading and unloading of devices would be handle through our available/unavailable lifecycle. This is really interesting when considering programmable logic in data centers where rented services are associated with applications on demand. This opens the question of how our solution suits in hypervisors, a question addressed in Chapter 5.

3.5 Evaluation

To evaluate our solution, we built several prototypes and conducted several experiments. The first experiment focuses on the evaluation of the overheads introduced by the solution. The second experiment highlights the behavior of the solution with very small embedded systems, showing one part of the flexibility of our solution.

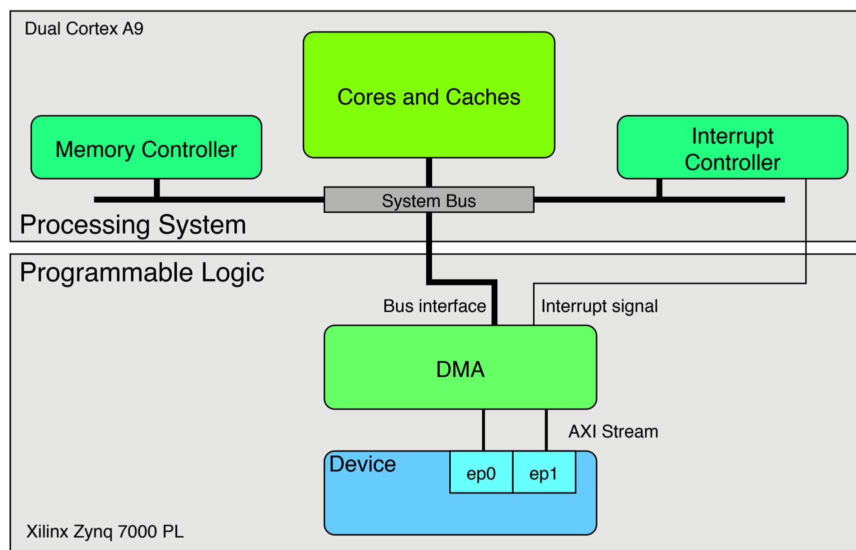


FIGURE 3.5: Baseline design

3.5.1 Overhead Evaluation

The first experiment is to determine the overheads introduced by using messages. These overheads are overall negligible and we had to design a specific experiment to expose these overheads. We designed a no-delay block device, a device that drops all written blocks and always reads zero-filled blocks. Our device is therefore the hardware equivalent to the combo of `/dev/null` and `/dev/zero`.

Given this design, since there is no device overheads, any experiment will highlight the communication overhead: creating requests and receiving responses. We therefore integrated this device in two ways. First, we integrated that device using a traditional integration based on MMIO registers, interrupts, and DMA engine. This represents our baseline for all our measurements, it gives the performance numbers that one can expect on our hardware platform.

Figure 3.5 illustrates this baseline for the Zybo board from Diligent with the Xilinx Zynq 7010 System-on-Chip (SoC) integrating a dual Cortex-A9 processing system at 700MHz with 512 MB of DDR memory and a Xilinx programmable logic around 20K LUTs. Our block device was written using VHDL (236 lines), leveraging the optimized scatter-gather DMA engine provided by Xilinx with a small footprint (1500 LUTs). This DMA engine has two AXI stream interfaces connected directly to the device and two rings in memory, one to transmit and one to receive.

The device can handle read requests and write requests, each request being sent as one buffer, described as one descriptor in the DMA transmit ring. A read request is only encoded on 5 bytes, with the following format: `READ_REQUEST` type (1 byte), the offset of the block (4 bytes). A read response is encoded as a `READ_RESPONSE` type (1 byte), offset of the block (4 bytes), a status (2 bytes), and the data block. A write request is encoded on the same 5-byte header followed by the data block. A write response is encoded as a `WRITE_RESPONSE` type (1 byte), the offset of the block (4 bytes), and a status (2 bytes).

As a software stack, we adopted a bare metal approach, avoiding the inherent noise when measuring large software stacks. This bare metal stack is based on the small Hardware Abstraction Layer (HAL) provided by Xilinx as a foundation for small embedded system development. We added a small event-oriented scheduler (516 lines of C), a driver for our block device, and the small code of our benchmarks. The software stack cannot be smaller, therefore any experiment will measure the communication performance between the driver and our block device. Our benchmark is a simple throughput benchmark, a never-ending loop sending the same request over and over, either a request to read a block or a request to write a block. All requests are pipelined rather than using a synchronous request-response scheme.

- For each write request, a buffer is initialized with the write request and a data block. The buffer range of memory addresses is then flushed to DDR memory from the Cortex A9 caches before the corresponding entry in the DMA transmit ring is given to the DMA engine to send. An interrupt will signal the asynchronous availability of the status response in the DMA receive ring. For each response, the memory range corresponding to the response is invalidated from the Cortex A9 caches before the status response is checked, bringing the status response in the Cortex A9 L1 cache.
- For each read request, a buffer is initialized with the read request. The buffer range of memory addresses is then flushed to memory from the Cortex A9 caches before the corresponding entry in the DMA transmit ring is given to the DMA engine to send. An interrupt will signal the asynchronous availability of the status response in the DMA receive ring. For each response, the memory range corresponding to the response is invalidated from the Cortex A9 caches before the status response is checked and the block contents is read, bringing its contents in the Cortex A9 L1 cache.

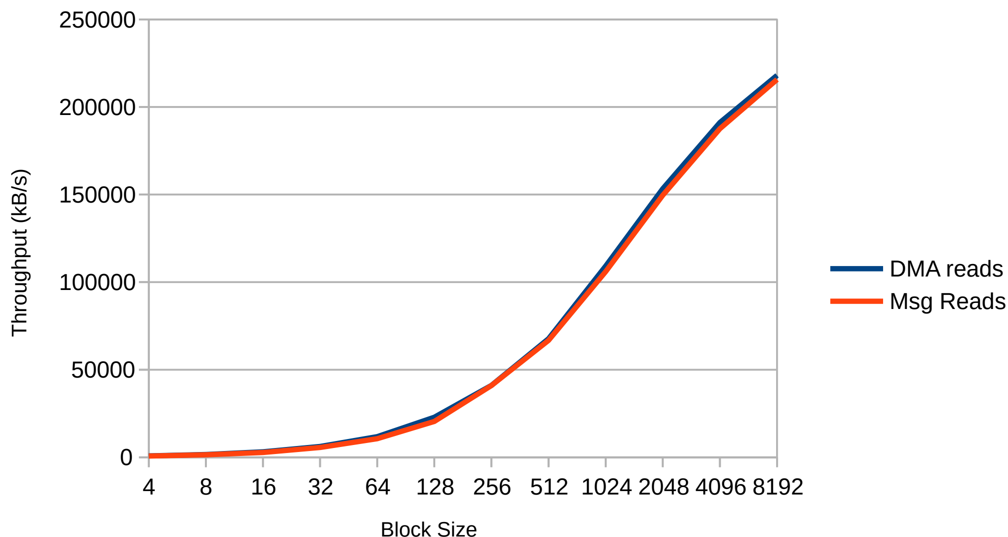


Figure 3.6: Read Requests

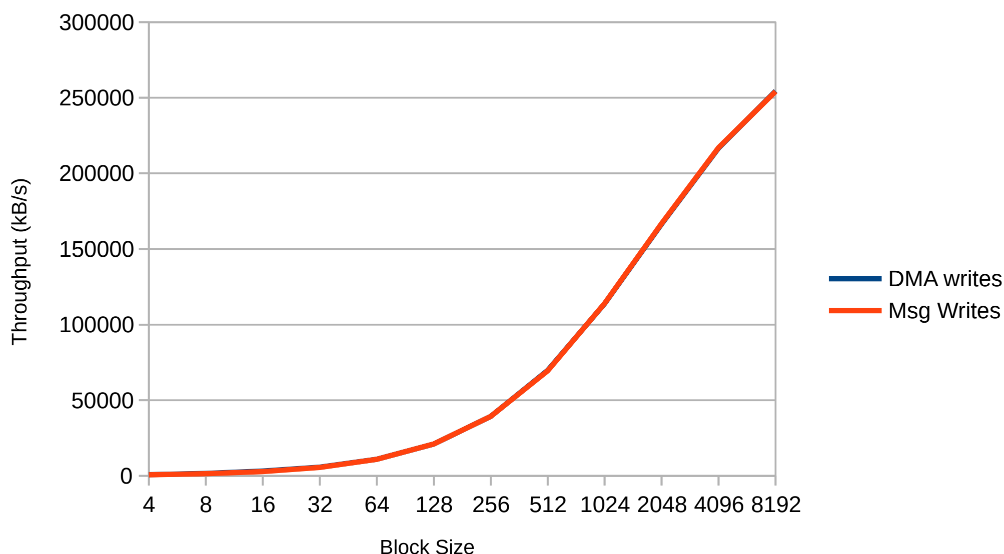


Figure 3.7: Write Requests

Periodically, the benchmark computes the achieved throughput in kilo-bytes per seconds (kB/s). The results are given in Figure 3.6 and in Figure 3.7, comparing the baseline implementation described above versus the implementation of our proposed solution. As we can see, the throughput overhead is negligible. The latency overhead is by design only a few cycles.

Figure 3.8 illustrates the details the hardware design of our message-based block device. The block device has 4 endpoints bound with 2 RX-TX software channels, One

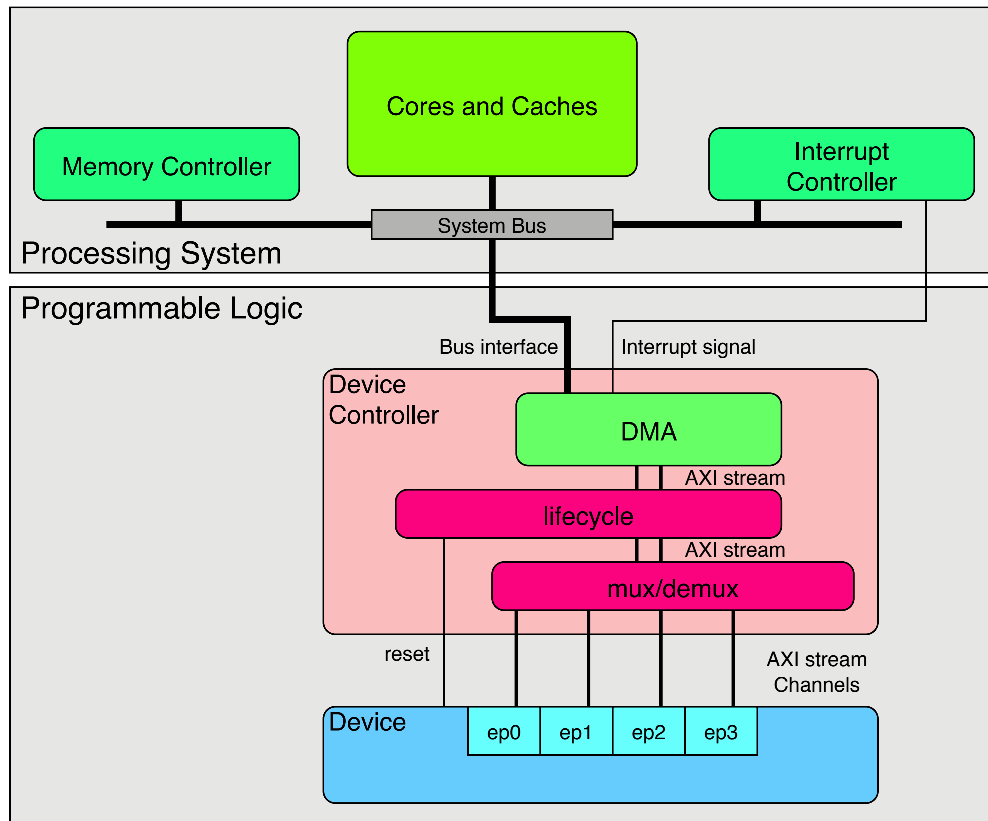


FIGURE 3.8: Our Hardware Design

mandatory to manage the lifeycle, the other to submit read and write requests. The device controller is composed of three hardware components: a DMA engine, a lifecycle controller, and a multiplexer/demultiplexer. For the DMA engine, we reused the same Xilinx DMA engine as for the baseline. We wrote in VHDL the lifecycle controller and the mux/demux component. The mux/demux component is necessary to route messages to and from endpoints since all channels are multiplexed through a single DMA engine.

The lifecycle and demux/mux components add 810 lines of VHDL to the 236 lines of VHDL for our device. Remember that our device does essentially nothing. For read request, it parses them and sends back a zero-filled data block. For write request, it parses them and just consumes the data block. The synthesis of our device controller VHDL adds 450 LUTs to the Xilinx DMA engine that takes 1500 LUTs, for two channels. Each new channel adds about 10 lines of VHDL.

The presence of the device controller is of course only half of the story, there is also a different software stack. Like for the baseline, the software stack is based on the same HAL from Xilinx and the same event-oriented scheduler. However, the driver

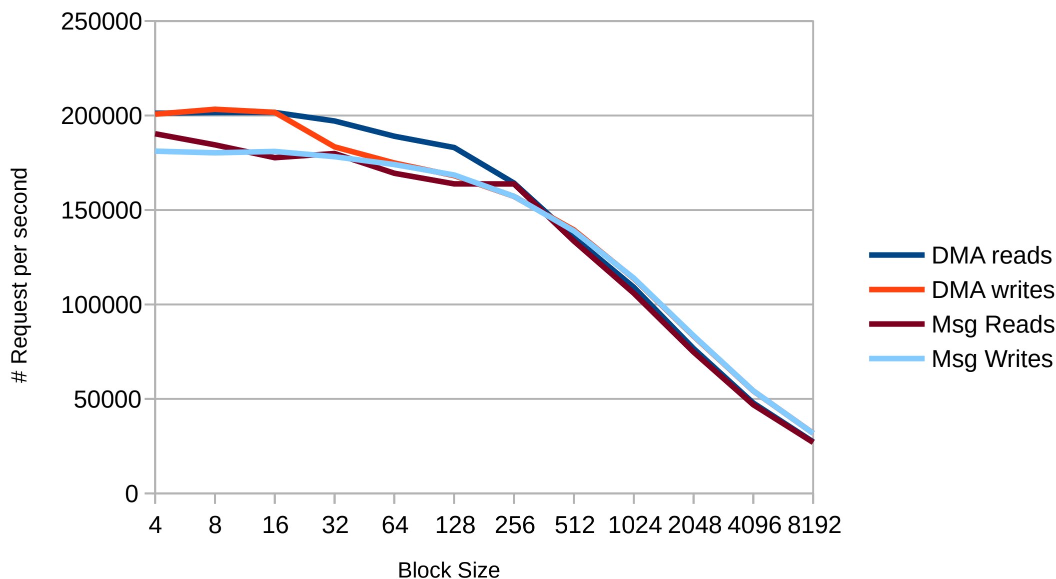


FIGURE 3.9: Number of requests per second

is now a class-generic driver, sending and receiving messages through channels. This means that the Figure 3.6 and the Figure 3.7 include both our hardware and software overheads. The global overhead resulting from using messages seems negligeable at first glance, but the figures are not telling the entire truth due to their scale. At high throughput, our proposal does induce negligeable overhead, but it is not the case at low throughput where the overhead is higher but always remain below 12%.

The Figure 3.9 plotting the number of requests per second reveals the evolution of the overhead of using messages. It is logical that the impact becomes negligeable as the payload size increases as the time to write, read, and transfer the payload increases while our overhead is not proportional to the payload size, but rather fixed software and hardware overheads, with the usual variability due to architected processor optimizations such as caches and also bus arbitration and memory controller conflicts.

This overhead is acceptable and representative of any device, as there is nothing specific to the concept of block in our device or in the given performance numbers. Our device receives and sends messages. The read requests are short messages but the response messages are large as they contain large payload. For the write requests, this is the opposite. The write requests are large as they embed a large payload and the responses are small as they only embed a status report. By varying the "block size", we are effectively getting performance numbers that are representative of any device, not just block-oriented devices.

The overhead of using messages rather than a traditional DMA is higher with small requests and smaller with larger requests. This means that the overhead is small for high-performance devices that are block-oriented devices, such as network cards or mass storage. Indeed, these devices will have message payloads much larger than 256 bytes, which means that the performance of high-performance devices will not be affected. The performance is essentially governed by the DMA engine performance. For small messages, the overhead is larger but we believe it to be of no consequences. Most devices using small messages may be expected to be low-performance devices, such as mouse or keyboards, sending and receiving few messages per seconds. Therefore, we believe that 12% overhead is not going to affect the performance of such devices. Also, let's not forget that the overhead will melt away with real devices. Remember our device does nothing, but real devices take time to process requests. The longer the device processing time, the more negligible our overhead becomes.

3.5.2 Small Embedded Systems

In the previous section, we evaluated a high-performance design. The software stack was a bare metal design, with minimal software overheads. The Zybo board embeds the Zynq-7010 SoC integrating a programmable logic with a high-performance Cortex-A9 processing system. We used the high-performance DMA engine provided by Xilinx. But our proposal is not only intended for high-performance embedded systems running bare metal software stacks. In this section, we want to assess really small embedded systems.

As an example of an emerging class of small embedded systems we thought it would be interesting to consider systems with just a programmable logic and no processing system. So we went somewhat to the extreme with the Xilinx Arty board, a board with only the Artix-7 programmable logic. For the processing system, we deployed the Xilinx MicroBlaze on the programmable logic, a low-performance soft-core processor. The corresponding design illustrated by Figure 3.10.

The block device is the same as before. Notice that there is no device controller since the Microblaze soft-core has direct support for the AXI-stream. This means that the device is entirely driven by the software running on the MicroBlaze, handling both lifecycle and channel messages. The software stack is minimal, reusing our small event scheduler (516 LoC), our software bus (2205 LoC) and a block driver (348 LoC), running on top of the Xilinx standalone platform. The overall software footprint is less than

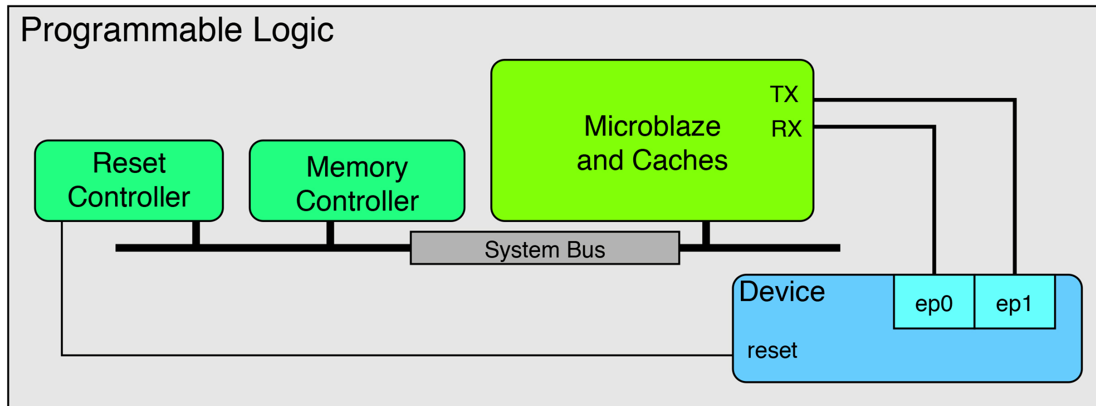


FIGURE 3.10: Embedded MicroBlaze design

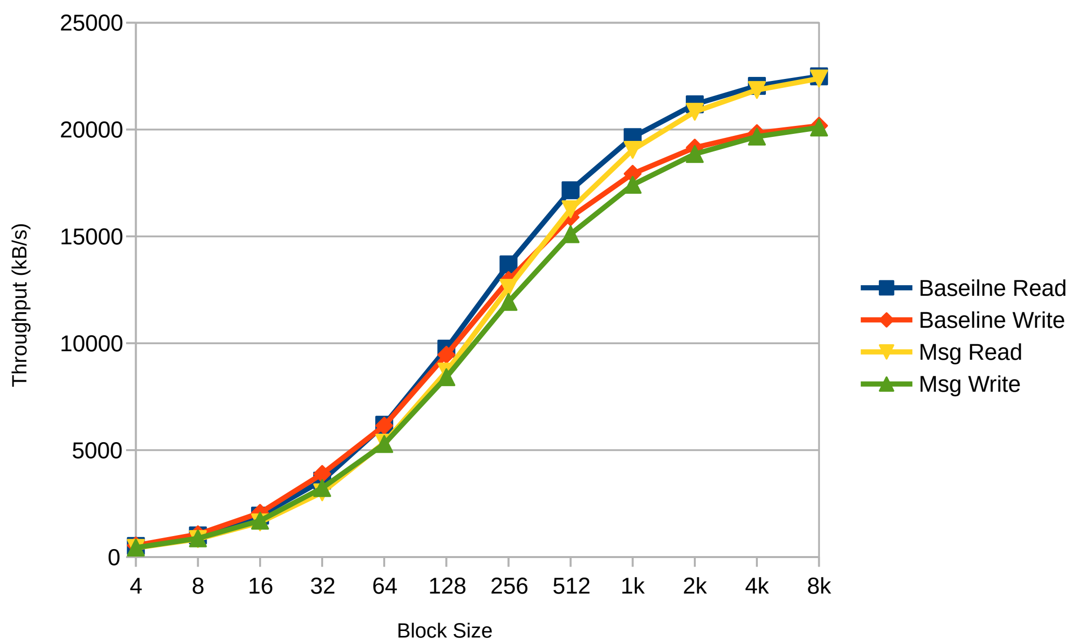


FIGURE 3.11: Stream vs Message Throughputs

30K bytes. Note that there is no rings in memory since there is no DMA, only one statically allocated buffer is necessary to receive messages from the device.

We ran the same experiment as before, with the same block-oriented device as before. We also used the same bare metal software stack based on the Xilinx-provided HAL and our small event-oriented scheduler. The baseline uses a small driver directly reading and writing through AXI-stream ports. Our proposal includes a full implementation of our bus and channels. The driver uses messages sent and received through channels, as before. However, the implementation of our bus is radically different. We do not use any DMA engine, we directly read and write to the AXI-stream ports. Since we do not have a device controller in hardware, the entire lifecycle is managed in software.

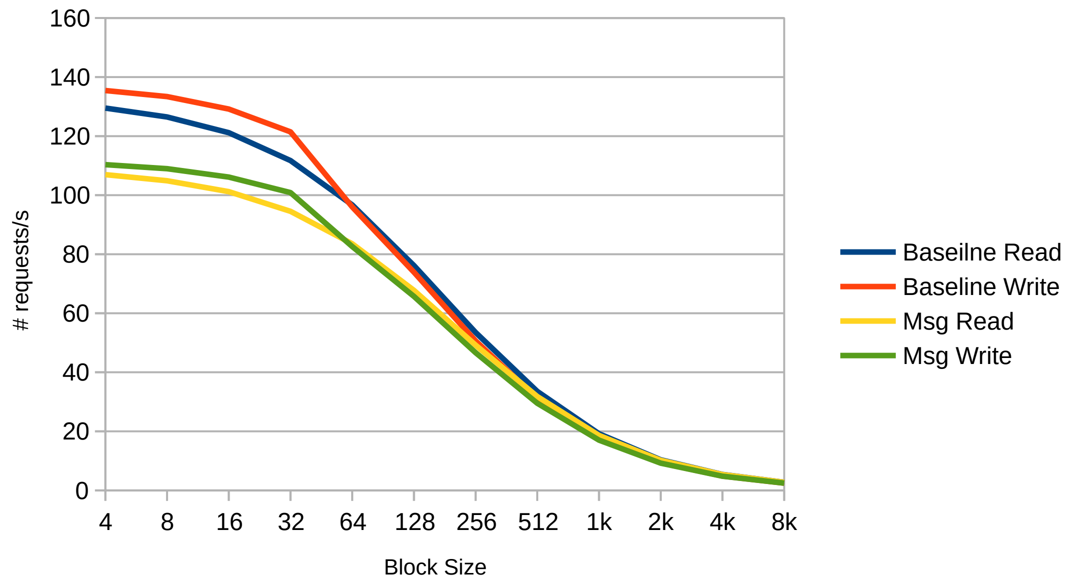


FIGURE 3.12: Number of Requests per second

The Figure 3.11 shows the throughput achieved using directly the AXI-stream or through our message-oriented channels. We see a similar behavior as before on the Zybo board, using messages does not introduce any significant overhead. Something that is confirmed by looking at the number of requests per second, in Figure 3.12.

Of course, we see again the larger overhead with smaller messages, which is confirmed with the throughput losses in Figure 3.13. The overhead remains below 20%, which is higher than our previous 12%. But it is not surprising as our overhead is mostly on the processing system and one can expect a soft-core processor to be slower. But again, let's remember that real devices take time to process requests and we are benchmarking here a device that does nothing. Nevertheless, it is clear that the approach puts more strain on the processing system for smaller embedded systems.

Now, there is another interesting question: how to we manage multiple devices? There are two different approaches possible. One is to leverage the fact that a MicroBlaze can be configured to support more than a pair of AXI-stream, allowing to connect more than one device directly. There is another interesting way, one with deploying a tree of devices connected to one pair of AXI streams on the MicroBlaze. We choose to pursue this second design because it illustrates the flexibility of using messages and is clearly more scalable.

The Figure 3.14 illustrates the corresponding hardware design, using AXI-stream interconnect hardware components from the Xilinx Vivado tool. In terms of latency, this

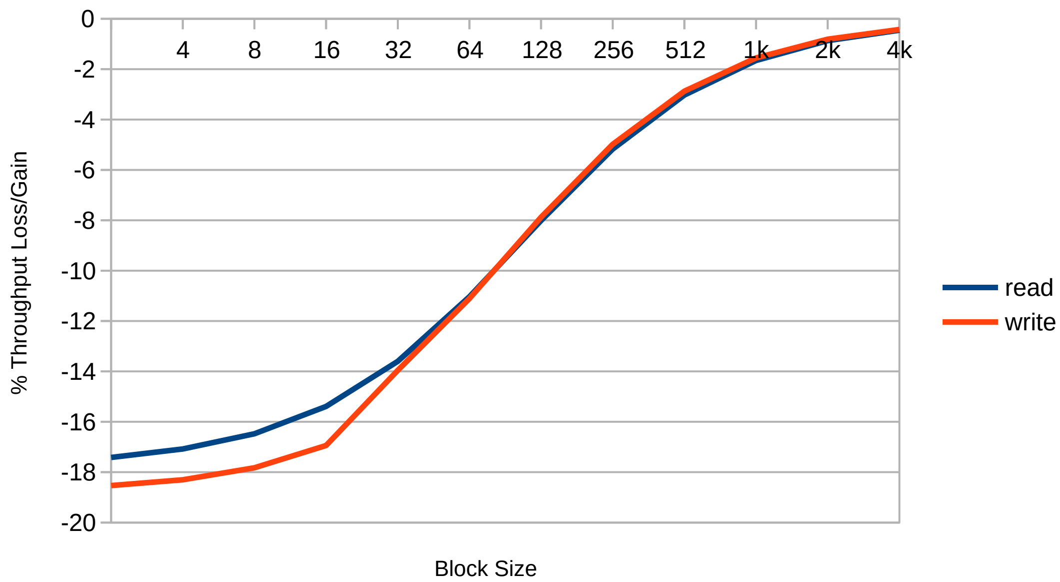


FIGURE 3.13: Throughput Gain/Loss in percents

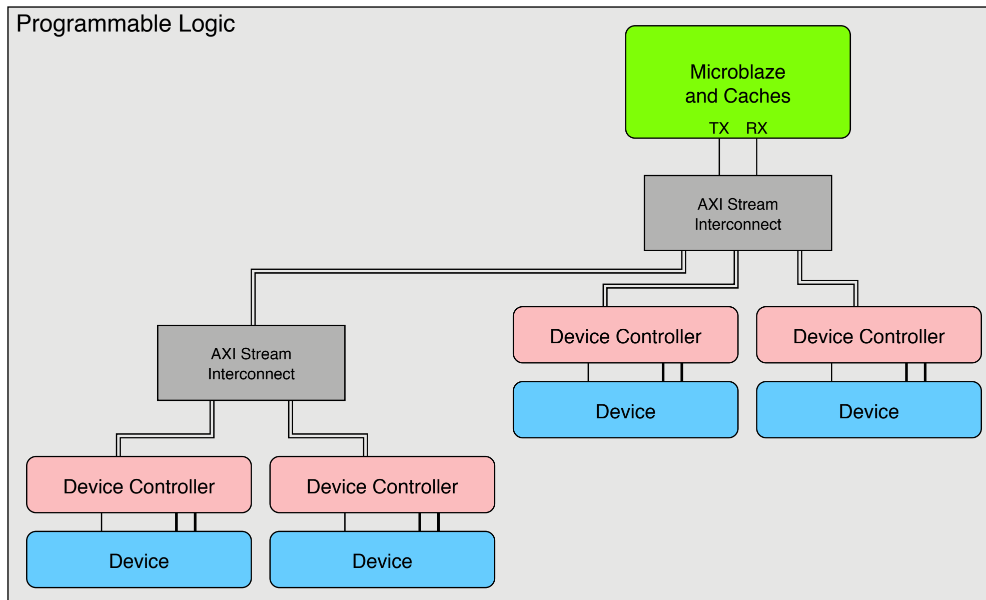


FIGURE 3.14: Device Tree

design is efficient with a few extra cycles. Vivado power analysis tool estimate that power consumption is not impacted, which is understandable as it is mostly forwarding hardware signals without much calculation. When N drivers communicate with N devices, the available throughput is divided between the N interactions. When only 1 driver communicates with its own device, the throughput is not impacted by the added hardware required to connect multiple device, it simply act as if it was not there.

3.6 Summary

In this chapter we showed that the integration challenge of FPGAs can be greatly improved using messages. They make a clean separation between the software and the hardware worlds, understandable by both side. Using messages greatly simplifies driver development by removing hardware implementation details from the programing interface of devices. This leads to easier device integrations and maintenance. Finally messages enable the concept of class generic devices, allowing devices to be more easily available when they fit in already existing classes. The next step in improving FPGA integration is to move toward bigger systems in particular Linux based systems.

Chapter 4

Linux Integration

Chapter contents

4.1 Background **52**
4.1.1 Linux kernel modules 52
4.1.2 User-space interfaces 54
4.1.3 Linux Device Driver Model 56
4.1.4 Software-Hardware communication in Linux 61
4.1.5 Some bus implementation examples in Linux 62
4.2 The Extension Proposal **67**
4.2.1 A new message bus for Linux 68
4.2.2 Driver API 69
4.2.3 Low Level Driver API 71
4.3 Experiments **75**

Following our journey in the integration of message-based design, the logical next step is full fledge operating systems, Linux in particular. Linux is a major actor in the embedded world and must be considered when proposing a solution for this community. That's what we will do in this chapter by proposing the design and validating of our solution in a Linux environment. Our solution is meant to be as well integrated as possible with existing Linux concepts. This means that we don't want to reinvent what already exists so that existing Linux users can take advantage of our solution without having to choose between our world and the Linux world.

4.1 Background

Device drivers in Linux are designed with modularity in mind. It means in particular that most of the time, a driver in Linux is a kernel module, a piece of code that can be loaded and unloaded dynamically. Drivers are part of a stack that eventually ends up giving a functionality to a user-space process. To do this, Linux uses the file system as an interface between the user-space and the kernel-space. Special files, like character device files, can be opened, read and written by user processes to exploit a functionality provided by the kernel and its drivers. Inside the kernel itself, drivers are organized around what Linux calls the device driver model [41]. This model is based on three type of elements: devices, drivers and buses. Devices are software constructs, most of the time representing a real hardware device or an abstraction of it. Those devices are registered within a bus to be found by drivers. This model provides the basic elements to allow drivers to find their device and is used by almost all drivers in the kernel nowadays.

4.1.1 Linux kernel modules

Linux kernel modules are binary object files that can be dynamically loaded and linked within the kernel. Kernel modules define functions that can be exported and used throughout the kernel. They can also invoke functions exported by other loaded modules. A kernel module can thus be seen as a plug-in, being able to extend kernel features at runtime. A key advantage of using kernel modules is that a running kernel can be updated without having to shutdown and rebuild the whole system. Another advantage is that kernel images are highly configurable by enabling desired modules and disabling others. Device drivers are typical examples of kernel modules. They are dynamically loaded when needed to drive a newly plugged-in device, unloaded when no longer necessary.

Linux kernel modules are written in the C language, as is still (but how long for? [42]) the majority of user space application. Even if basic C programming concepts stay the same when programming a kernel module, there are several important differences. First, modules run in kernel space, which grants modules unlimited privileges and the responsibilities that come with it. Second, if usual applications define a `main()` function and run from start to end, modules register themselves with the kernel and wait for other modules to call functions they define. Every module defines a `init()`

function, invoked during its loading process. This function has to perform any memory allocation, resource allocation such as registering interrupt handlers, or registration with other kernel frameworks that it might need. A module also has to define an `exit()` function, called when the module is unloaded from the kernel, to release any resources it has allocated.

Modules are also built differently. Module object files are classical Executable and Linkable Format (ELF) files, using however the `.ko` extension. They are built with the Linux build system. Modules can not be linked to user space libraries. Indeed, a module is linked directly with the kernel symbol table and loaded into the kernel memory. A module can only use functions and global variables defined by other modules, primarily declared in the `linux/include` directory in the Linux source tree. When a module is linked, its exported functions are added into the kernel symbol table. This table contains addresses of global functions and variables allowing modules to call functions and use global variables defined throughout the kernel. This linking mechanism allows for stacking modules, each module providing features used by modules in upper layers. These dependencies are identified at build time, not at load time, and are used by the `modprobe` utility to load modules in the proper order.

The programming model of kernel modules is also different, it is not based on multithreading, it is fundamentally event driven. Indeed, modules register themselves with the kernel and wait to be notified of events. Modules typically have two ways of receiving events: either they define functions and export them to the kernel using the `EXPORT` macro and wait for other modules to call them, or they give pointers to functions they define to other kernel frameworks. These functions can be called at any time. In contrast to single-threaded applications that run sequentially from start to end, kernel modules have to be programmed being aware that many functions can be called at once. There are various sources of concurrency, the most obvious one is multiple processes concurrently calling module functions via system calls. Interrupt handlers can also run at the same time than other module functions, as well as other software abstractions such as timer or tasklets. Fortunately, the Linux kernel provides all the necessary synchronization tools, in particular locking mechanisms.

Modules also differ from user-space applications with respect to failures. User applications are running in processes that can be stopped and removed from the running system by the kernel. Developers can rely when exiting their process on an ultimate cleanup by the kernel. In contrast, modules are not isolated from the rest of the kernel, they can easily corrupt the system memory, even because of simple mistakes such as

copying a non zero terminated string using `strcpy()`. Segmentation faults and memory leaks are other forms of module failures. These failures are much harder to track and debug than user space applications failures. Although, it is possible to use a kernel debugger, debugging the kernel is not as easy as debugging applications. The kernel could corrupt the debugger itself. Furthermore, if the kernel is halted, the debugger, the graphical interface or the terminal would freeze. Of course, virtual machines like Qemu have drastically improved the situation for kernel debugging at large, but not so much for device driver debugging because device drivers often need to run on real hardware. Indeed, running on real hardware increases the number of interwining events. This is due to the fact that real hardware devices run their logic purely in parallel rather than virtual emulated devices running on CPU. It makes in particular forgotten or unpredicted race conditions or wrongly timed events appear. For instance, in some cases a sequence of two write operations in a device register bank may need to be made with a minimal delay between them. It is often the case that running in a simulated hardware makes that kind of condition invisible because the time taken to switch from one software process to another makes this minimum delay always respected. Though, running on real hardware will make this issue appear as register writes will be performed with a much lower latency. Another example is interrupt triggers. In some cases a hardware device may trigger two interrupts very quickly with a very small delay between them. It can happen that on a simulated hardware, the second interrupt always appears after the execution of the handler of the first one has finished, where on real hardware, the second interrupt may be triggered during the execution of the handler of the first interrupt. If that case, there's a chance that the second interrupt is simply ignored as that particular case doesn't appear on the simulated hardware.

4.1.2 User-space interfaces

Linux kernel modules can interface with user space applications using various mechanisms. The most common are character and block devices. Character and block devices are special files, visible by the user in the file system. Users can use them as usual file, using `open()`, `close()`, `read()` and `write()` system calls. In addition, users can use the ioctl() function to send requests to character or block device files.

To implement a character device, a module has to define a character driver. A character driver is a set of functions, stored as pointers in a structure called `file_operations`.

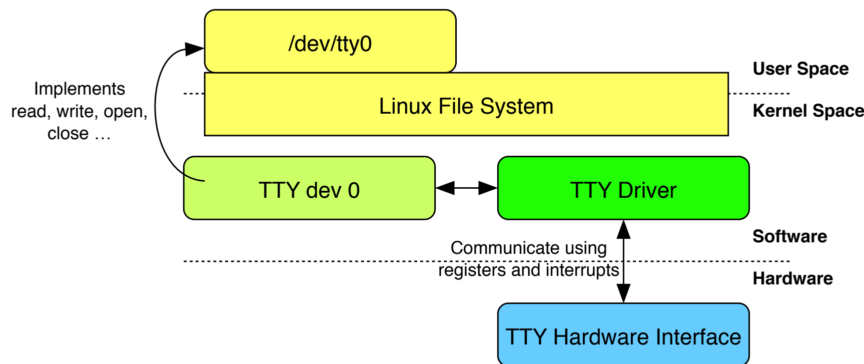


FIGURE 4.1: TTY character device

The main functions are `open`, `release`, `read`, `write`, and `ioctl`. The `open()` and `release()` functions are called when a user opens a file and when the last user closes it. The `read()` and `write()` functions are called upon reading and writing the file. The `ioctl()` function provides special interaction capabilities. A module may also implement asynchronous variants of these functions, for instance the `aio_write()` that asynchronously perform a write operation on the character device. Figure 4.1 illustrates the principle on a teletypewriter (tty), a character device. Its driver directly communicates with the hardware interface, issuing loads and stores in hardware registers to configure and transfer data and handling interrupts to be notified of changes.

Block devices are implemented in similar way. Block drivers fill a `block_device_operations` structure. The major changes between character drivers and block drivers are how `read()` and `write()` functions and additional operations called by the kernel are implemented . The `block_device_operations` structure doesn't contain any pointer to a `read()` or a `write()` function. Instead, block drivers have to define a `gen_disk` structure, representing a virtual disk, and add it to the kernel. Within the `gen_disk` structure block drivers add a request queue. This queue contains a `make_request()` callback, invoked when an operation (read or write operations for instance) has to be performed on the disk. The `make_request()` callback must process BIO objects (Block Input Output) that contain information on the operation to perform, such as the type of the request (read or write), the location of the operation on the disk, and a buffer containing the data to write, or where to put the read data. This operation is most of the time asynchronous, meaning that the `make_request()` callback will initiate the operation then return before it actually ends. Later when the read or write operation has been completed, the `bio_endio()` function will be called to notify the block layer that the request has been completed.

Network devices are similar to block devices. They also have to implement the `open()` and `close()` functions but they vary on data exchanges. To manage packets, the kernel provides a special buffer structure called `sk_buff`. This structure contains not only data buffers but also control informations such as packet datagrams or cells. This structure is defined with ring buffer in mind, meaning that they can be linked together in very efficient ways. This is especially important when dealing with high performance network cards that organize their reception and transmission buffers in rings.

All these special files have another interface in common called the IOCTL interface. As its acronym let it think, they are used as I/O control and are used mostly to configure or get the status of a device. Although it was not intended at the origin, it is also seen as a generic interface used when the read/write operations do not suit the behavior of the device. It consists of an `ioctl` function that has two important parameters: the code of the command to perform and some data. The command code is always specific to the device. It can be a configuration command, for instance setting the baud on a serial device. The data parameter can be either a scalar or a user pointer that can be used to send bigger data structures.

4.1.3 Linux Device Driver Model

Device drivers are organized in Linux following the concept of hierarchical buses. Some of these buses are matching hardware buses, such as PCI or USB. Other buses are about software abstraction such as the bus grouping devices that are Human Interface Devices. For each bus, three steps are important: reifying devices, registering drivers, and matching.

First, for each bus, devices must be reified. By reification we mean that a hardware device be instanciated in software. It must be made handleable using its own software structure instance[1]. Indeed, in this kind of model, devices are more than just a base address so there is a need for a structure containing useful data such as its identification, pointers used to remember dynamically allocated data related to the device and sometimes function pointers. This reification can happen as the result of processing a given description of which devices are plugged on that bus. An example of such description is the device tree given to the kernel on ARM that describes the different hardware buses and the hardware devices that they host. A bus can also support hot

[1] As we are using the C language, we talk about structures but the linux kernel is almost object oriented as it happily uses function pointers

plugging of devices, such as PCI buses, allowing to enumerate hardware devices. The enumeration is used by the bus implementation to reify devices that are represented on the bus at any given time.

Second, for each bus, the necessary device drivers must be registered. In Linux, each device driver is written to register to a given bus, something that happens when the module containing the device driver is loaded. Modules can be pre-loaded or they can be loaded on demand as the result of enumerating new devices. This is the case for example with the USB bus, looping back to the udev mechanism that controls which modules are loaded for driving devices. Third, each bus matches locally drivers to devices. To do this matching, the bus relies on devices describing what they are and on drivers describing which devices they can drive. This is typically done through different identifiers such as device IDs and vendor IDs.

Then, Linux supports stacking buses, where device drivers in one bus reifies themselves as devices in other buses. For instance, the driver of an USB mouse, on the USB bus, might reify itself as an HID device in the HID bus. Stacking buses helps organize the overwhelming number of devices and their drivers into a hierarchy of concepts that separate concerns, encapsulate details, and help the necessary composition. For instance, the HID bus can compose different pointing devices as a single pointer that can be used by the window manager to drive the mouse icon on the screen.

4.1.3.1 Devices, Drivers, Buses

Linux bus model articulates three core concepts: drivers, devices, and buses. To understand them, it is just easier to look at a PCI bus, which initially motivated the Linux concept of a bus. Buses, drivers and devices are all visible in the system filesystem (of type `sysfs`, usually found at `/sys` in Linux), the virtual file system Linux uses to expose informations about kernel subsystems. The Figure 4.2 illustrates this model with a PCI device example. In this bus, a software construct of the PCI device has been reified so that its driver has been able to find it. The driver, also registered within the bus, is able to communicate with traditional registers and interrupts once it has been matched with its device. This driver then interfaces with higher stacks of the kernel to eventually provide the device functionality to user processes.

Each device represents a software view of hardware devices such as PS/2 mice, Ethernet controllers, audio controllers, or USB controllers. Devices can be very close to

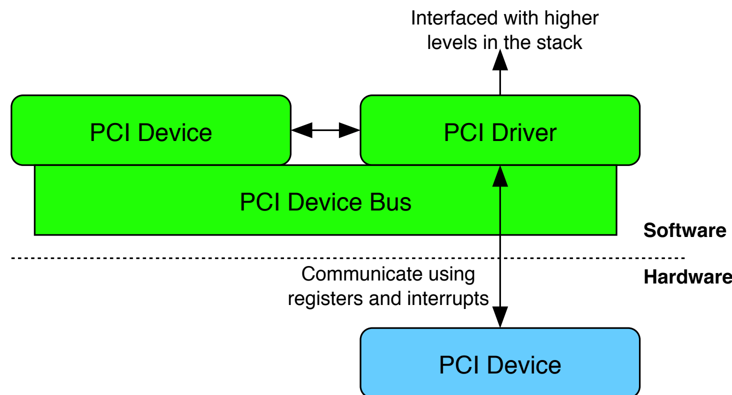


FIGURE 4.2: PCI bus example

their hardware device or very abstract, or even software devices with no corresponding hardware device. To declare and register a device, a kernel module has to fill a `struct device` structure. Listing 4.1 shows the `struct device` fields. The parent field is used to build device hierarchy. The bus field is the pointer over the bus in which the device needs to be registered. The driver field is the driver that the device is bound to. This field is automatically set by the Linux bus framework. The driver data field is a pointer that can be freely set by the device driver as a cookie allowing the driver to retrieve its own data structure for that device. Finally, the `release` function is called by the Linux bus framework when the device is unregistered from the bus it was sitting on. To register or unregister a device, a module has to call the `device_register()` and `device_unregister()` functions.

```
struct device {
    struct device *parent;
    struct bustype *bus;
    struct device driver *driver;
    void *driver data;
    void (*release) (struct device *dev);
    /* . . . */
}
```

LISTING 4.1: Linux device structure

To declare and register a driver, a module has to fill a `struct device_driver` structure, depicted in Listing 4.2 . The field `name` is the driver name set by the module and shown in the `sysfs`. The bus field, as for the device structure, is the pointer to the bus that the driver will be registered with. The devices field is a linked list of devices bound with the driver, a list managed by the kernel. The `probe()` function, set by the module, is called when a device is bound to the driver. In this function, the driver must

check if it can indeed handle the device and perform the needed initialization in order to prepare the device to be used. The `remove()` function, also set by the module, is called when a bound device is removed from the system. To register and unregister a driver, a module must use the `driver_register()` and `driver_unregister()` functions. Once a driver is registered, and bound to a device, it typically communicates with the device to implement higher-level operations such as implementing a write operation by communicating with a hard drive.

```
struct device_driver {
    char *name;
    struct bustype *bus;
    struct list head devices;
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    /* . . . */
}
```

Listing 4.2: Linux device driver structure

Buses are also dynamically created by modules, the modules that have the device driver for that bus controller. To create and register a bus with the kernel, a module has to fill the `struct bus_type` (shown in Listing 4.3) structure and register it using the `bus_register()` function. There are very few fields a module must set to define a bus, most of them being handled by the Linux bus framework. The `match()` function is called when the kernel has found a device that could be bound with a driver. The function has to check if the driver can handle the device, returning a non zero value if the bus determined that it is the case. The kernel handles all the generic operations such as updating driver and device lists, calling the right callbacks for instance the driver `probe()` callback when a device has been bound, making the interface with the `sysfs` to exposed an updated view of the bus at any time.

```
struct bus {
    char *name;
    /* Sets of devices and drivers */
    struct kset drivers;
    struct kset devices;
    int (*match) (struct device *dev, struct device_driver *drv);
    /* . . . */
}
```

Listing 4.3: Linux bus structure

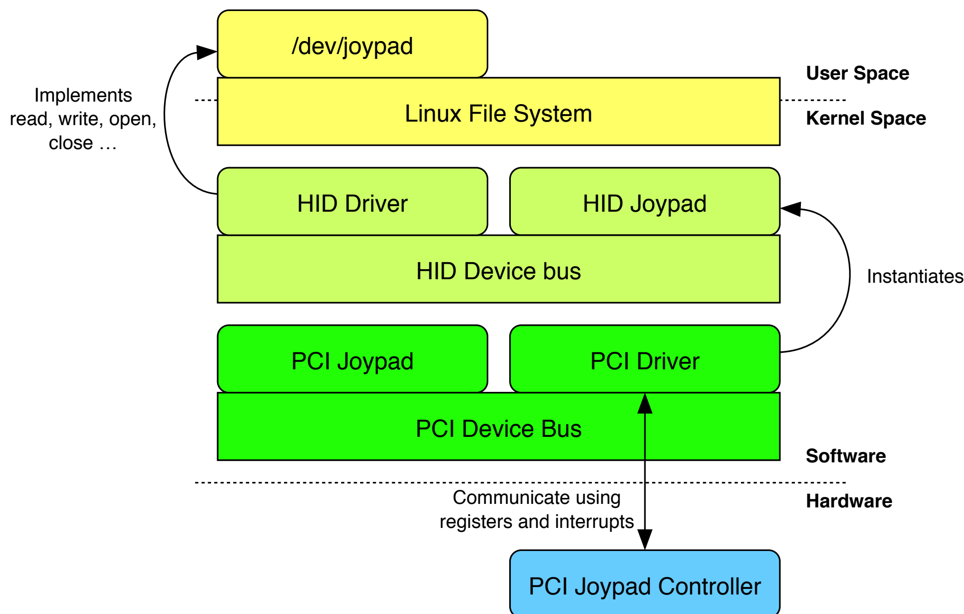


FIGURE 4.3: Bus stacking example

Finally buses can be stacked on top of each other as illustrated by Figure 4.3. In this example, a PCI device implementing a hardware Joypad controller sits on the PCI hardware bus. As it is a PCI device, it is logically reified within the PCI device bus and matched with its PCI driver. The PCI driver then turns around and registers a Human Interface Device (HID) within the HID bus. This device contains informations about the device and callbacks to implement its behavior. In particular it contains callbacks that are called by the PCI driver when buttons are pressed. This device is handled by a generic HID driver. To let user processes be notified when buttons are pressed, the HID driver creates a character device file. Those user processes can listen to input simply by reading the file as if it was a serial interface. Going bottom up on the stack, when a button is pressed by a human a series of event happen. First, the PCI driver is being notified by an interrupt and check which button has been pressed by reading the device registers. Then, one of the HID device callback to notify of the event. The event is handled by the generic HID driver that will in turns push the code of the button in the character device file that user processes are listening into.

Bus stacking is maybe one of the best features in the Linux device model as it allows devices that have similar features to have abstract representation in high level buses while each of them can come from a different hardware bus. In our case, this is a concept we will take advantage of to be able to design a generic message-based interface, stacked on top of specific bus technologies.

4.1.4 Software-Hardware communication in Linux

In addition to being interfaced with user space processes through character and block devices, kernel modules can communicate with the hardware interfaces. This communication is performed by issuing loads and stores in the device's hardware registers and by handling their interrupts. To to this, Linux defines I/O ports, I/O memory and interrupts which represent part of the Linux Hardware Abstraction Layer (HAL).

Hardware devices are controlled by reading and writing their hardware registers. These registers are usually accessed in a consecutive address space. I/O ports and I/O memory are an abstraction concept, defined by Linux to allow modules, and more specifically device drivers, to access these hardware registers. I/O ports and I/O memory provide portability across various hardware architectures. I/O ports are used to access hardware I/O registers. To use I/O ports, a module first needs to allocate a range of addresses containing them in order to guarantee that the module will have exclusive access to the I/O registers in that range. To read and write these ports, a module must use different functions related to different size of registers such as: `outb()` to write into a one byte port, or `inl()` to read into a 32-bit port.

I/O memory is used to access large size of device address spaces. They are used for various purposes such as sending and receiving Ethernet packets, sending video data or receiving data blocks. Despite the fact that accessing hardware device address spaces is highly hardware dependent, the principles are the same, allowing Linux to provide a unified abstraction as I/O memory. I/O memory, as I/O ports, must be allocated by a module to be sure it has an exclusive access on the region it wants to use, using the `request_mem_region()` function. Before using the allocated I/O memory region, modules must ensure it is accessible by the kernel, by first setting up a mapping using the `ioremap()` function. This function returns a pointer, used to access the I/O memory. However, directly using the pointer is not considered that portable, and modules must instead use special functions such as `ioread8()` or `iowrite32()` to respectively read 8-bit and write 32-bit of data.

Some devices have to access big chunks of the main memory. Instead of letting the CPU writing or reading data to and from the device, drivers setup buffers within the main memory to be accessed by the device. This way, the CPU is free to perform other tasks while data is transmitted from or to the main memory by the device itself. The main concern with this situation is cache coherency. When the driver writes data in a buffer to be read by the device, it must ensure that all cache lines storing data from

that buffer are flushed to the actual memory. The other way around, when the device has written data in the buffer, cache line targeting this buffer must be invalidated in order to let the driver see the most recent data. All these operations are performed in Linux by a subsystem allowing to map, unmap and handle IO coherency.

Besides using I/O ports, I/O memory and DMA buffers to control devices, read and write into their memory, modules may need to access interrupts. Interrupts are 1-bit signals, sent by devices, to notify of a state change. They are used for instance by clocks to notify of time changes, or serial ports, notifying that incoming data is pending. Using interrupts avoids polling devices, wasting CPU time and incurring power consumption. Linux provides a software abstraction for interrupts, allowing module to request to be notified of interrupt signals. To request an interrupt, modules use the `request_irq()` function, specifying the interrupt number they target and an interrupt handler that will be called when the interrupt signal is triggered.

4.1.5 Some bus implementation examples in Linux

The Linux bus model offers a generic way to organize devices and drivers, but it does not get in the way of drivers interacting with their devices. In fact, some details of this interaction are often bus specific. For instance, drivers often need more details on their devices than a generic framework can provide. PCI drivers need to access device configurations. USB drivers need to list device interfaces and endpoints. Moreover, traditionally, drivers communicate directly with their devices through I/O ports, I/O memory, and interrupts. This is what makes drivers so hard to write and utterly device specific. The PCI bus is a typical example of this philosophy. Other buses, such as USB or I2C, have adopted a different philosophy based on messages, yielding simpler and safer drivers.

4.1.5.1 PCI drivers

We will see how PCI drivers look with Listing 4.4 that shows how to define and register a PCI driver. The `ids` variable is an identification table containing ID structures allowing the PCI bus to bind to the driver the devices it can really handle. The `struct pci_driver` structure embeds a `device_driver` structure, filled by the `pci_register_driver()` function. In particular, the function will set the bus field of the device driver structure to the pci bus pointer (usually bus pointers are global variable). When a PCI

driver is bound to a PCI device, it first needs to access its configuration registers. To do that, many function that are part of the PCI bus API provide this access. For instance the `pci_read_config_byte(struct pci dev* *dev, int loc, u8 *val)` allows to read a byte with the value `val` at the location `loc` in the configuration registers of the device `dev`. To access I/O locations a PCI driver uses the PCI resource management to get PCI device memory addresses. These addresses are then used with the I/O memory Linux API to read or write in the device address space. To be notified of device events, PCI drivers can use interrupts. Interrupt numbers are stored in PCI device configuration register. They can be accessed using the `pci_read_config_byte(pci_dev, PCI_INTERRUPT_LINE, &irq_number)` call. The interrupt number can now be used to request interrupts as usual in Linux.

```
/* Source taken from the book Linux Device Driver Thrid edition */
static struct pci_device_id ids [] = {
    {PCI_DEVICE(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82801AA_3)},
    {0}
};

static int probe(struct pci_dev *dev, const struct pci_device_id *id)
{
    /* various initializations:
       e.g. allocating I/O memory, requesting interrupts, etc.  */
    return 0;
}

static void remove(struct pci_dev *dev)
{
    /* clean-up any allocated resources:
       e.g. releasing I/O memory and interrupt lines */
}

/* The PCI driver structure */
static struct pci_driver pci_driver = {
    .name = "pci_skel",
    .id_table = ids,
    .probe = probe,
    .remove = remove
};

static int pci_skel_init(void)
{
    /* register the PCI driver with the PCI bus */
```

```
    return pci_register_driver(&pci_driver);
}
```

Listing 4.4: PCI bus example

4.1.5.2 USB drivers

USB is a good example to be compared with PCI. Listing 4.5 shows how to defines and register a USB driver. As with PCI drivers, USB drivers define an ID table used to bind them with devices, and register themselves using the similar usb register() function. The key difference between USB and PCI buses is the way devices are described and how drivers communicate with them. The USB device structure contains a list of interface, each interface holding a list of endpoint. The Linux USB API allows drivers to connect to these endpoints in order to send or receive message through them. To send or receive a message through an endpoint, USB drivers use USB Request Blocks (URBs). A URB represents a transfer request that a driver can create, submit, and cancel. A URB can perform the four data transfer types of the USB specification (Control, Interrupt, Bulk and Isochronous). URB transfers (more precisely, bulk transfers) are asynchronous. A completion callback is set before submitting them to allow the driver to be notified when the transfer will be completed. Listing 4.6 shows an example of a USB driver submitting a bulk transfer to a device.

```
/* Source inspired from the book Linux Device Driver Thrid edition */
/* Define these values to match your devices */
#define USB_SKEL_VENDOR_ID  0xfff0
#define USB_SKEL_PRODUCT_ID 0xfff0

/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
    {USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID)},
    {}
};

static int skel_probe(struct usb_interface *interface,
                      const struct usb_device_id *id)
{
    /* Allocate any needed resources */
    return 0;
}
```

```
static void skel_disconnect (struct usb_interface *interface)
{
    /* Release any allocated resources */
    return;
}

static struct usb_driver skel_driver = {
    .name = "skeleton",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect
};

static int usb_skel_init(void)
{
    int result;
    /* register this driver with the USB bus */
    result = usb_register(&skel_driver);
    return result;
}
```

LISTING 4.5: USB bus example

```
static void bulk_completed(struct urb *urb)
{
    /* Transfer has been completed */
}

void start_bulk_transfer(struct usb_device *dev,
                         int endpoint_address,
                         char *buffer, size_t count)
{
    struct urb *urb = NULL;
    char *buf = NULL;
    /* create the urb */
    urb = usb_alloc_urb(0, GFP KERNEL);
    /* initialize the urb properly
     * create a pipe to the endpoint,
     * sets the buffer to send,
     * and the completion callback */
    usb_fill_bulk_urb(urb, dev,
                      usb_sndbulkpipe(dev, endpoint_address),
                      buffer, count,
                      bulk_completed,
```

```
                    NULL);
    urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
    /* send the data out the bulk port */
    retval = usb_submit_urb(urb, GFP KERNEL);
}
```

Listing 4.6: USB bulk transfer example

The communication model between USB devices and drivers follows a master-slave scheme. It means that every data transfer, either transmitting or receiving a message, is initiated by the driver itself. Before USB 3.0, this meant that drivers had to poll devices like mice or keyboard. Indeed, there was no such mechanism as interrupts coming from the device to let the driver know that a message is ready to be received. From USB 3.0 on, when a device has to send a message to its driver, it triggers a notification signal from its USB controller. This notification is then forwarded to the device which will then start the transfer as usual.

Even though USB has been a real game-changer in the way drivers are communicating with their devices, the software interface still lacks a level of abstraction independent from the hardware bus. As an example, some parts of the API are related to the new class of USB type C cables. USB defines the notion of bulk, isochronous, interrupts and control transfers. They are very tightly linked to the specification of the USB hardware. Actually there are two different transfer types: asynchronous and periodic. Bulk and control transfers are asynchronous, which means that the driver initiates a transfer and will be notified when the transfer completes. Isochronous and interrupt are periodic transfers, which means that the USB controller is set up in such a way that these transfers are initiated periodically.

We are not arguing that USB should be a multi-master bus or should change its specification to shrink the four transfer types into two, asynchronous and periodic. In fact the USB is even completely compatible with our solution as it is a message-based bus, and a very successful one regarding how it evolved. The potential offered by USB has been a great motivation for the new Linux device model in the 2.6 version of the kernel. They also showed the full potential of stacking buses. Indeed, a USB mouse is actually stacking drivers of, from bottom layers to upper layers, a platform driver for the PCI controller, a PCI driver for the USB controller, a USB HID generic driver for the USB mouse controller, a HID driver managing all HID devices either coming from USB, bluetooth or other buses, and finally the input driver of Linux and even a character driver when showing the device in the file system.

There was an opportunity to aggregate every message-based buses behind a unique API, not only integrating USB as yet another bus. Of course, for the industry, the implementation effort was not worth it, for very good reasons. They had to keep the entry cost as low as possible and be integrated as fast as possible. Asking Linux to change was not the best way to sell it rather than being the smallest addition possible to the kernel. They also have interests in keeping as much people as possible using their tools and technologies which is completely understandable from a business perspective.

4.2 The Extension Proposal

The extension proposal consists of taking the concepts we develop for small embedded system and validate that they also fit within a Linux environment. Doing that consists of taking all the parts of our solution that are common with Linux and merge them in our extension, then add the new features to it. Linux did such a tremendous amount of work on their block subsystem, it would make no sense to simply throw it out to rebuild it. Now, merging our concepts with Linux's is not a piece of cake, it would be much simpler to simply exist beside and integrate as yet another kernel module. The problem is that it would have no future. Simply no one could accept such a partition inside the kernel, so our choice is to integrate as much as possible with the existing, while still bringing our new ideas. In particular, the notions of bus, device, driver but also bus stacking are fully reused as is. The high-level concepts such as char, block and network devices are also reused, we don't specify a new user/kernel interface.

What we add is our communication model, meaning the abstract message-based channels. Today, every bus is basically defining its own communication model with its own API, even for those exchanging events or messages. So not only our channels are here to support our new devices, but they also offer a consistent generic message-based interface that basically any existing or future bus technology may use to take advantage of it. In particular this is very helpful for class-generic devices where devices belonging to the same class, supporting the same message protocol, may sit on very different hardware buses.

4.2.1 A new message bus for Linux

As we have seen, the Linux kernel is quite powerful when it comes to integrate device drivers and in particular aggregating, discovering and matching devices. The aspect of the interaction between a driver and a device that Linux does not specifically define is a communication abstraction between drivers and device. Linux separately define useful abstractions to register interrupt, access hardware registers and even manage DMA buffers, but no such standard communication layer have been attempted at that level. Though, Linux demonstrated it is possible. The best example is the block device layer where every single storage device is represented, no matter how they are implemented or on which hardware bus they are.

The idea of our extension is to bring the same level of abstraction as the block layer did in Linux. It brings the concept of message-based communication between devices and drivers as an addition to the discovering and matching process that already exists. The goal of doing that isn't really to help Linux solving an integration issue, the goal is to show that our approach is feasible not only for small embedded system but also for larger systems.

Large systems, other than Linux, suffer the most from the device integration challenge and thus would benefit the most from our solution. They don't have the titanic code-base of driver of Linux even though they must be running on more than one small embedded system. This requires a great level of portability among various hardware, something that is achievable only when writing a virtual ton of device drivers.

Our extension is not intrusive in existing implementations. It allows to progressively aggregate compatible devices and hardware buses, while letting unchanged ones working as usual. In other words, what's already working continues to work. In our case, we integrate only our new message-based devices with the hope that others will follow.

In order to be integrated inside Linux, our extension exists as a kernel module, compiled and linked separately from the kernel itself. This allows users to insert or remove our module from the kernel as needed. To let other systems work as before, the bus implemented by our module is seen as yet another bus among the others. It allows to match our devices and drivers only inside our extension but still being integrated with the rest of the kernel. Drivers compatible with our extensions registers like in

any other buses, meaning that the matching process is very similar to the PCI and USB we have seen in the background section.

Our extension is divided into two APIs: the driver API and the Low Level Driver API. The driver API is basically the same API as we presented in the chapter 3. The Low Level API is an additional API allowing bus controller drivers to register new devices in our extension. Those API are made to integrate our extension using the pervasive stacking mechanism, providing all the Linux flexibility. The driver API stacks under high-level software buses, for instance the block subsystem while the low level API stacks above low-level drivers, for instance the device controller driver transmitting messages to devices.

As an example we take a stack of 3 subsystems including our own integrating a hardware AES device. The AES functionality provided by this hardware is ultimately provided to user-space processes through a character device file in the `/dev` directory. As depicted in Figure 4.4, starting from the hardware, the AES device possesses two channels. The channel 0 receives the data to encrypt while the channel 1 sends the encrypted data. The AES is as usual in our solution connected to its device controller. In this case the controller is designed using a scatter-gather DMA for maximum performances. The device controller is seen as a platform device discovered through the device-tree and driven by its controller driver. This controller driver instantiates the device structure and registers it to our bus. This device is then matched with its driver, the one that will send and receive messages through the two channels and implement the character device interface. This character device interface is then shown to userspace processes as a file in `/dev/aes0`. To use it, userspace processes would simply have to write the data to encrypt, and read the encrypted data. Multi-process accesses can be managed in two ways. Either only one process can open the file at a time, or the driver manage transactions putting processes in sleep while others are waiting for their data to be processed.

4.2.2 Driver API

The C API is identical on almost all aspects, with very little coloration from Linux quirks and features. This shows that whether being on a very small embedded system running on a Microblaze, or running the large and powerful Linux kernel on a general purpose architecture does not impact the way devices should communicate. The only difference is the transmission buffer management. Linux asks its drivers to

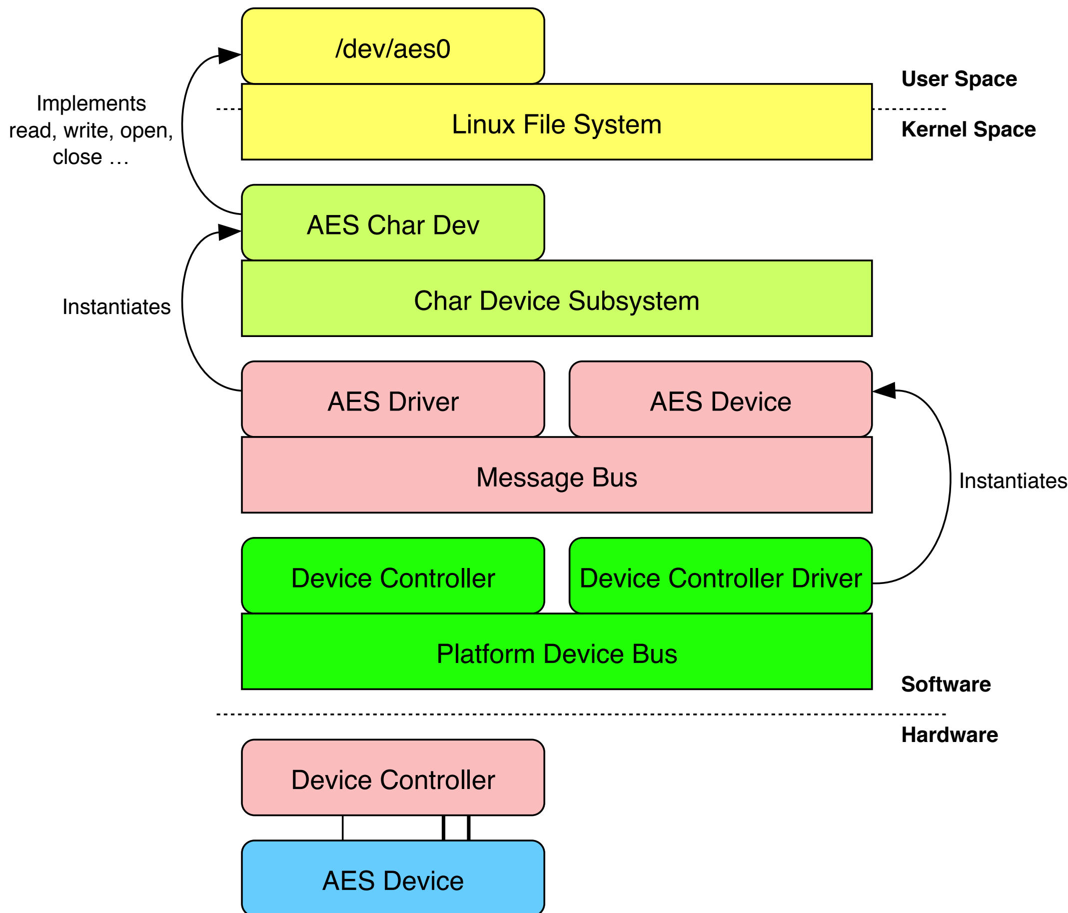


FIGURE 4.4: The full stack integrating an AES device as a character device

use its DMA-mapping interface when dealing with buffers being accessed by DMA. In particular, the owner of DMA buffers must call map and unmap functions when allocating and releasing its buffers, and call synchronization functions when having to flush or invalidate corresponding cache lines. For that reason, drivers have to manage the mapping and unmapping process of their transmission message buffer. This will depend on the way they choose to manage their buffer but in general they will have to map them before starting using them with their device and unmap them when they become useless.

Having the same API makes drivers really looking like the one running on our standalone solution. We reuse the same exact functions, types and callbacks, without additional constraints. This may let think that drivers are portable from system to system. While it may be the case for the small portion of code interacting with our API, it is not true for the whole driver. For instance, interfacing with the Linux block subsystem to register a new block device may be a complete different story on another system. But that's our exact goal here. We have reduced and simplified the portion of code

interacting with the hardware device to let the experts concentrate on interfacing the driver with the kernel.

Our API heavily makes use of bus stacking in Linux. Drivers are stacked under upper layers such as block or HID buses. They execute as any kernel module, registering themselves within their initialization function, the only change is that they interface with their device via sending and receiving messages through channels. Our API never puts them in interrupt context, even when calling their callbacks, making it easier to interact with the kernel without worrying about it. Once matched with their device, they can configure it and register a new device in an upper bus.

The simplest stacking example for a driver is when registering a character device, as we illustrate taking our AES example again. The driver defines its `open` and `close` function as usual. The `send` function copies the data to be encrypted from the user-space buffer and send it through the corresponding channel. The `read` function waits until all the encrypted data has been received and copy it to the user-space buffer.

4.2.3 Low Level Driver API

To help various hardware implementations of our solution to integrate with our extension, we allow them to stack under our bus using a low-level API. This API allows any driver to register devices in our bus and implement its channels. It also allows multiple implementation of device controllers to coexist using each their own specific driver. Each of these controller drivers can register its own set of devices in our bus. We call Low Level Drivers (LLDs for short) all these driver stacking under our bus and registering devices. This type of stacking is not new and exists for instance with the HID bus. Both bluetooth drivers and USB drivers have the ability to create HID devices and register them as abstract devices. [2]

The Low Level API is important as we want our extension to be as much integrated in the kernel as possible. Not only new drivers can be easily integrated as other kernel modules registering message-based drivers in our bus but also other hardware bus technologies can be integrated without too much pain. The Figure 4.5 illustrates this with a block device. This block device sits on a hypothetical future hardware bus to show how one can integrate new hardware ideas in our extension. As usual the device

[2] They are not actually entirely abstracted as some of the USB or bluetooth details are visible to HID drivers

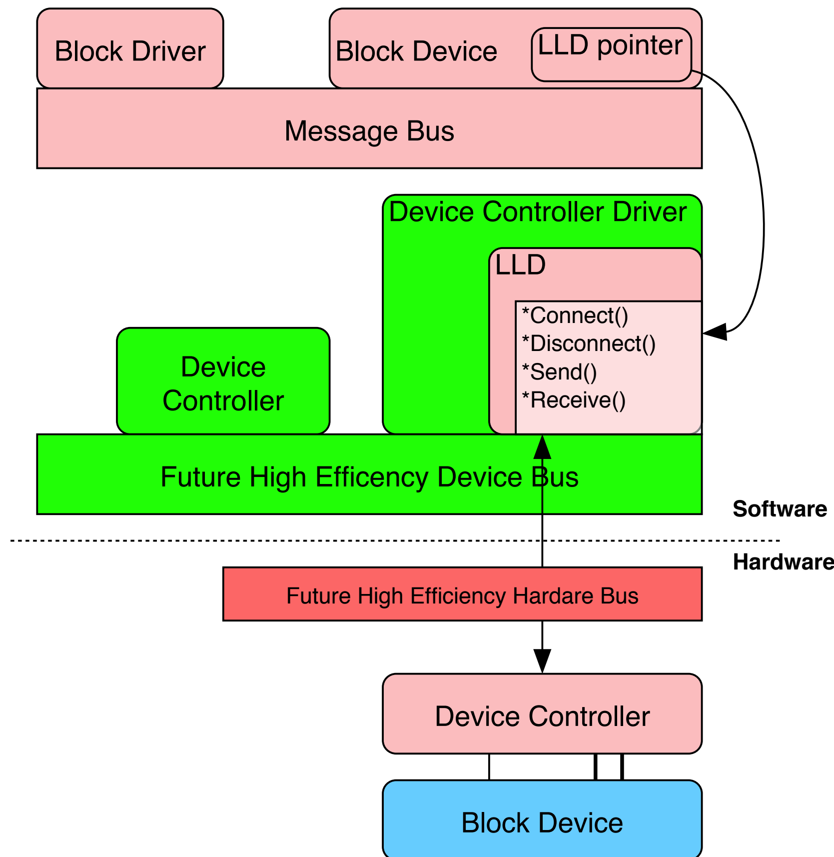


FIGURE 4.5: LLD Example

controller is instantiated within the future software bus and matched with its driver. But this device controller is new and has very specific features so it must implements its own way of sending and receiving messages and perhaps also to connect and disconnect channels. To integrate this new implementation of channels, it defines its own Low Level Driver containing function pointers to connect and disconnect channels and also to send and receive messages. When instantiating devices in our message bus, the controller driver will provide a pointer to this LLD so that every channel operation will leads to a call of one of these callbacks.

Not only this design helps new hardware technologies to integrate with our bus, helping the extensibility of our design, but also allow very different hardware designs to coexist. Indeed, a PCI integration of our bus could implements channels its own way with its own LLD, taking advantage of the last four generations of PCI express features. Beside, a very small implementation relying on FIFO registers may also implement low throughput, low latency channels through its own LLD. Both would registers devices having the exact same channel interface. Each channel implementation would behave

the same way with the same lifecycle but with very different performance characteristics, in the same system. In addition, one cannot disrupt the other as they exist separately, being only aggregated at the message interface level.

The last advantage of such a design is to facilitate the aggregation of class generic devices. Indeed, in the USB world, class generic devices are all sitting on the same hardware bus, making it not a real problem. In our case though, devices belonging to the same class could be sitting on very different hardware buses. Using LLD the whole stack doesn't have to be replicated, the only change is the channel implementation at the device controller driver level. This means that a class generic driver may drive two devices belonging to the same class, one being integrated on a high performance hardware bus, while the other on a low performance one. Both devices would behave the same, only with a performance gap between them.

The low-level driver API follows the exact same life-cycle we described in the previous chapter. Low level drivers create devices when discovered on their hardware bus. The device discovery is initiated by the driver when it sends its *AVAILABLE* message, starting the rendez-vous with its controller. This message will eventually notify the device controller driver that it should create a new device and registers it with the bus. At any point, if the device becomes unavailable, the controller will unregister the driver from the bus, starting the unmatching process. Devices are registered and unregistered using the two following functions of the LLD API.

```
void register_device(struct device* device);
void unregister_device(struct device* device);
```

As usual, the bus manages the matching process and calls the right callbacks to notify of device events to the driver. When matched, the driver sets-up channel callbacks and calls the `connect_channels()` function, allowing it to start exchanging messages through its device channels. Implementing those channels is performed through a structure called `low_level_driver` that sits in the device structure. This structure is part of the low level interface and is defined by device controller drivers. This structure consists of the set of functions and callbacks described next:

```
struct low_level_driver {
    void (*connect_channels)(struct device* device);
    void (*disconnect_channels)(struct device* device);
    void (*channel_send)(struct message *msg);
    void (*channel_received)(struct message *msg);
};
```

The `connect_channels()` function is setup by the LLD and called when the driver matches its device. As defined by our device lifecycle, when executing this function the LLD must inform the device controller and eventually the device that a driver is matched to the corresponding device. This is done by sending the *MATCHED* message to the device, completing the rendez-vous.

The `disconnect_channels()` function is setup by the LLD and called when the driver is unmatched from its device. The unmatching process can starts from two different sides. When a device become unavailable the controller driver unregisters it from the bus. The bus will then notify the driver through the `unmatched()` callbacks, telling it to stop using the device. The controller driver will then be informed that is can starts disconnecting the device through this `disconnect_channels()` function. The unmatching process can also start from driver side when it unregisters itself from the bus. Again the driver will be called with the `unmatched()` for all its device and `disconnect_channels()` will be called by the bus for each of them.

The `channel_send()` function is setup by the LLD and called when the driver sends a message on a channel to its device. When executing this function, the LLD will start sending the given message. When using a scatter-gather DMA for instance, this is where the LLD will setup the next buffer descriptors, setup the message header and flush the payload buffer [3]. Once the transmission is complete, it will also call the `released()` function of the message, allowing the driver to know it can reuse the message buffer. This callback must not be called in the interrupt handler of the DMA as specified by our driver API. The controller driver must thus call it in its bottom interrupt handler.

The `channel_received()` function is setup by the bus and called by the controller driver when it has received a message from the device. It also initializes the `released()` function pointer to allow the driver to tell when it has finished using the corresponding payload buffer. Again this function must not be called in interrupt context as specified by the driver API.

[3] Flushing the buffer is done through the Linux dma-mapping API, calling a synchronization function. This is the reason why we ask drivers to map and unmap their buffers

4.3 Experiments

To validate the Linux integration we integrated our solution and experimented with a block device. The hardware platform is the same Zybo board as in the previous chapter, integrating an FPGA and running Linux on the Cortex-A9 processor. The hardware design integrating the device is the same as before using the same device controller implementation. The only change is really the software part where the device controller driver has been rewritten for Linux. The rest of the stack is an unmodified Linux kernel 3.6 provided by Xilinx; The only difference with a standard Linux kernel is the addition of Xilinx drivers to support the Zybo board.

We chose to design a ramdisk, a block device that stores blocks in the DDR memory, rather than driving a real external mass storage, with the intent to compare our ramdisk with the Linux builtin ramdisk. We reserve a region of memory in the memory map and use it as storage for our block device. This means that our device reads and writes blocks in the DDR memory from the programmable logic. The device was developed with high-level synthesis, using the Vivado tool from Xilinx. The raw performance are 114MB/s for reads and 13MB/s for writes, which is overall slow compared to the performance of our message conduit with around 200MB/s throughput. It seems therefore that the Xilinx DMA engine is far better optimized than the logic produced by high-level synthesis.

Once the Linux module packaging the driver for our block device is loaded in the Linux kernel, our block device appears as `/dev/blka` and we could create a file system and mount it. We compared our performance with the Linux ramdisk `/dev/ram0` and we got encouraging results. The command `mksf.vfat` ran in 11ms on Linux ramdisk and 12ms on our device. The command `mount` took 7ms versus 8ms and the command `umount` took 41ms for both devices. To evaluate the throughput, we compare the Linux ramdisk and ours using the `dd` command with a block size of 4KB.

The Figure 4.6 gives the throughput when reading from the ramdisks and writing to `/dev/null`. The line `ram0` shows the throughput for the Linux ramdisk while the line `blka` shows the throughput for our ramdisk. Overall, we see that throughput improves as the size grows, which is logical as there are fixed overheads. With small sizes, these fixed overheads dominate. With larger size, we are measuring a more realistic throughput. With larger sizes, we can see that our ramdisk has comparable performance but there still seems to be an increasing overhead as the size grows.

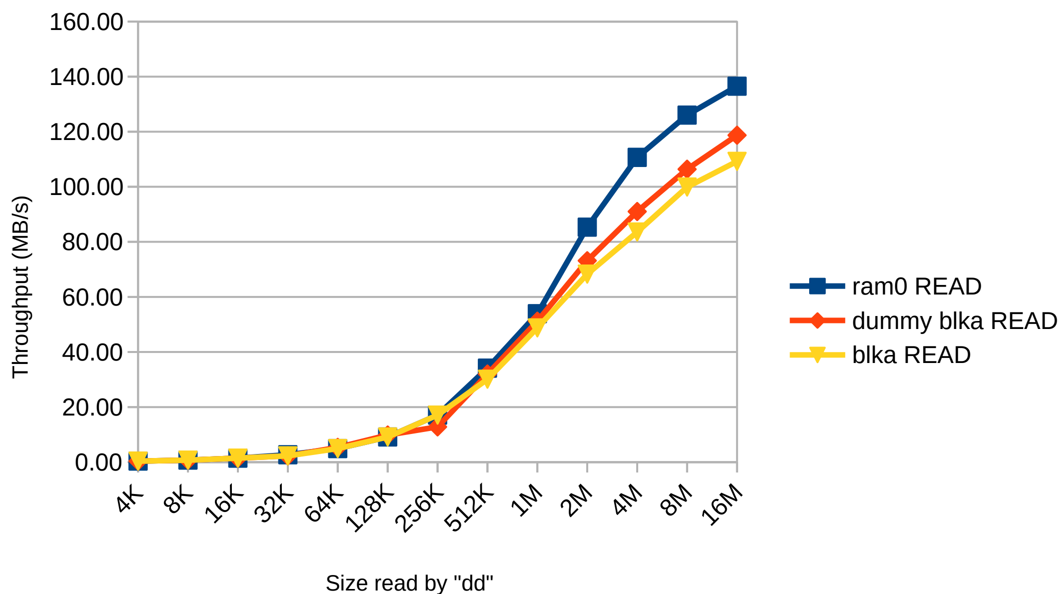


FIGURE 4.6: Read Throughput

To identify this overhead, we also measured our dummy block device, the one behaving as `/dev/zero`" the throughput is slightly better, so the limiting factor is not the reading of blocks from memory by the HLS part of our device. So it could be the overhead of messaging, but it is not. The overhead comes from cache operations, in this case the cache invalidate operation done by the driver before it can read the block contents. This is something that any block driver would have to do with a real device on the programmable logic. The Linux ramdisk can do without because it reads blocks using a memory copy operation done by the processor, not an external DMA engine. When removing the cache invalidate operation, our ramdisk performs as the Linux ramdisk.

The Figure 4.7 gives the throughput when writing to the ramdisks and reading from `/dev/zero`. Again, the line `ram0` show the throughput for the Linux ramdisk while the line `blka` shows the throughput for our ramdisk. We see the same overall shapes as for the read throughput, with the same impact of the fixed overheads inherent to Linux. Yet, we notice that the write throughput of our ramdisk quickly peaks at 12MB/s. After investigation, the culprit is the HLS part of our device, with a maximum throughput of 12MB/s when writing blocks to memory. This is confirmed by the line showing the throughput for our dummy device, behaving like `/dev/null` and thus without the HLS part writing blocks to memory.

Like for reads, we see a performance penalty due to cache operations between our dummy block device and the Linux ramdisk. Again, the Linux ramdisk can do without

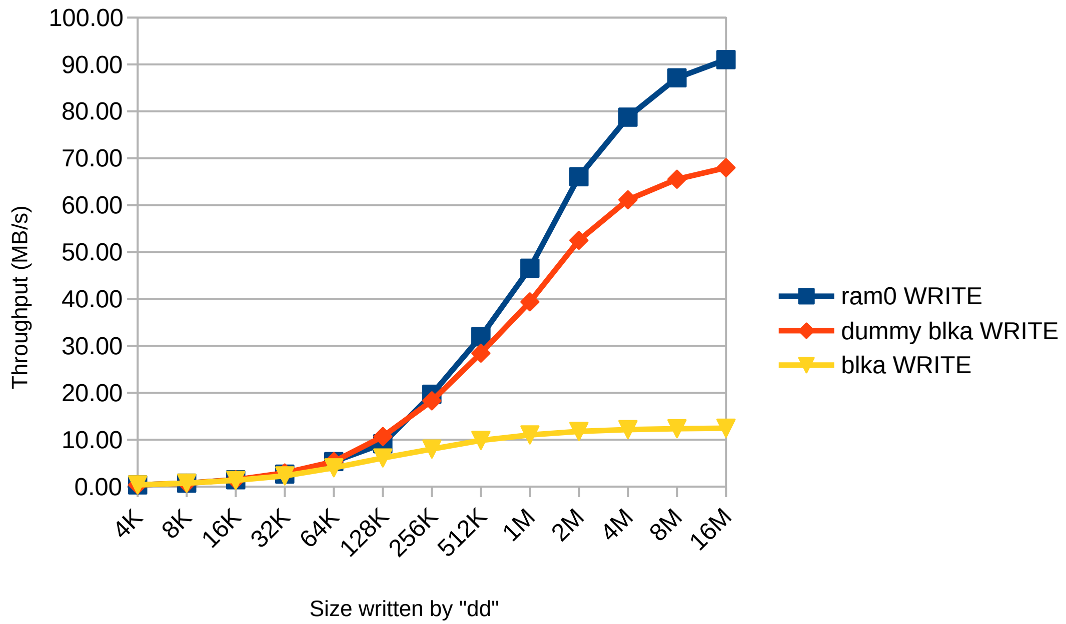


FIGURE 4.7: Write Throughput

because it writes blocks using memory copy operations done directly by the processor. In contrast, our ramdisk must flush caches before writing blocks, again something that any block driver would have to do with any real device on the programmable logic. When removing the cache flush operations, our ramdisk performs as the Linux ramdisk.

Chapter 5

Xen Integration

We have seen how using messages can improve the integration process of hardware devices for small embedded systems and bigger Linux based systems. The next logical step is to see if our solution can also offer solutions for cloud environments. In this chapter we will see how the cloud is architected and how devices are integrated into it. We will see what is the state of FPGA adoption in the cloud and how it may evolve in the future. Finally we will see how we designed and integrated our solution in an hypervisor and what might be the benefits for the cloud.

5.1 The FPGA Adoption in the cloud

The cloud is a very heterogeneous domain in which various technologies coexist to provide standardized services. Recently, FPGAs have been integrated in many cloud solutions [43, 44], providing more hardware programmability, better energy efficiency, and overall cost efficiency. Despite all those advantages, the adoption of FPGAs is not fully completed, as understanding how to integrate them seamlessly still requires a decent amount of work. Indeed, FPGAs are not simple to exploit within applications while they need to be fully integrated to deliver the benefits they promise [45, 46].

5.1.1 The architecture of the cloud

The core feature of cloud systems is to provide services through a network, either publicly on internet or on private networks. Those services can be split into three main

categories: softwares as a service (SaaS), platforms as a service (PaaS) and infrastructures as a service (IaaS). SaaS regroup all software applications accessible from a web browser or a desktop interface. These applications usually access database services to both run the application and store user data. PaaS provide a full operating system environment. In this type of service, the operating system and all underlying storage, network and operating systems are the responsibility of the cloud provider but the user can install any application or library. IaaS provide one or several virtual machines on which the user can install any operating system he needs while the underlying hypervisor and hardware platform is managed by the provider.

Services are accessible to clients through a network and can be accessible through a web application, an SSH connection, an FTP server and such. For instance, the famous application Netflix is a public service, running on a cloud infrastructure, providing a video and audio streaming service accessible through a web application. Internally, this service needs other services such as video and audio encoders that are not directly accessible by users. Some services are even full operating systems executing on what we call virtual machines. There is thus a high heterogeneity in cloud infrastructures. Small services interconnected through a network are used internally and assembled to provide a final service.

Services are implemented in very different ways. They can be implemented as a unique piece of software, running on one unique machine. More realistically, big services are implemented by many pieces of softwares and hardware services, running on many different machines. For instance, the Netflix service is replicated all around the world to offer maximum performance to all users across the globe. Those services are using different software and hardware services. We don't know the exact details of the implementation of Netflix but we can guess that they use hardware services to efficiently encode audio and video streams. They surely also use software services to manage user accounts, libraries and high-performance storage. In the end, many different software and hardware entities interact to implement the end service.

Cloud systems are by nature very heterogeneous. They are geographically spread out at different scales from a server rack to the entire world. Entities composing a cloud system are implemented in a variety of manners, both in software or hardware. Hardware accelerators can themselves also be implemented in many ways, either in ASIC chips or FPGA fabrics. No matter how those accelerators are implemented, they must be exploitable by software programs. Those software programs can be running on a lot of different operating systems. Programs must communicate locally on the same

machine but also distantly using network protocols. In the end, the cloud is a very heterogeneous world consisting of software and hardware modules working together.

At the very bottom of every cloud stack, there are physical machines and hardware devices. This bottom layer has three roles: 1) Execute software, 2) Accelerate some functions (video encoding, neural network inference, data-mining, etc.), 3) communicate with the outer world. The first role is performed by CPUs and coprocessors including GPUs. The second role is performed either by ASIC chips or FPGAs. The last role is performed by network interfaces.

Until now, there is no big difference with usual computers except some very high performance devices. These resources are potentially exploited by software application programs, running on an operating systems. This is where a big change happens. In a cloud environment, there is not only one operating system exploiting directly hardware resources. Most of the time in the cloud, for efficiency and cost reasons, operating systems share the same physical machine. Doing so offers a lot of flexibility. For instance, when two physical machines are running at 50% of their computing capacity, half of the computing load can be moved from one machine to the other. This allows to turn off the second machine and save both energy and hardware wear.

To allow different operating systems to share the same physical machine, we add an additional layer between operating systems and the hardware platform called hypervisor [47]. Figure 5.1 shows an example of an hypervised system using Xen. At the least, an hypervisor offers an execution scheduler and a virtual memory space. The scheduler allows each operating system to run on the same CPU, exactly as a traditional operating system would schedule processes. The virtual memory space allows each operating system to have its own memory space preventing one to access the memory of the other, again exactly as virtual memory space in operating systems isolate one process from the memory accesses of an other. To make the distinction, we call machine addresses the addresses used by the hypervisors, physical addresses the ones used by operating systems and virtual addresses the ones used by processes.

This is what is called virtualisation. Hypervisors virtualize a physical machine to offer a virtual machine in which an operating system can run as usual. These virtual machines are also known as guests. Some guests run without being aware that they are hypervized. Each guest managed by the hypervisor is then running without being aware of other guests, again in the same principle as processes run without knowing other ones exist. In some guests, the operating system and its processes are running

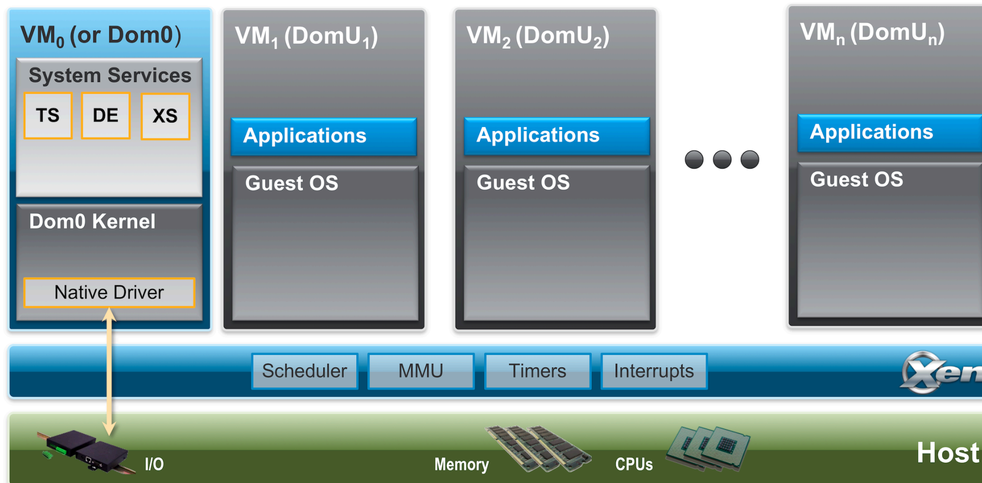


FIGURE 5.1: Xen Architecture (gathered from https://wiki.xenproject.org)

as usual on a physical machines, without even knowing the existence of the hypervisor [48]. This allows the operating system running in those guests to be unmodified.

Some other guests, though, can be aware that they are hypervized, a feature called paravirtualisation. By knowing that it is paravirtualized, a guest can exploit features offered by its hypervisor. The two most important ones are shared memory and guest-to-guest interrupts. Hypervisor features are accessed through an hypercall interface. Hypercalls are similar to system calls (or syscalls), they are traps from a guest to the hypervisor as syscalls are traps from a process to the operating system.

Having a running environment is not enough to run operating systems, guests need to access hardware devices such as consoles, displays, network interface, etc. There are multiple ways to allow hardware devices to be exploited by software services. The first and very radical way is to fully dedicate hardware devices to one service [49]. This has the obvious drawback to prevent all other guests to use the same device.

The second way is to split drivers between the guest running the service and a special guest accessing the hardware [50]. This requires paravirtualization and thus to modify the operating systems that are meant to be guests. The guest running the service runs what we call a front-end driver communicating with a back-end driver running in the special guest to access hardware features. They usually communicate by exchanging with an abstract message protocol made of requests and responses, implemented through shared memory.

The third way is to virtualize the hardware device using a hardware support. This is for instance the case in SR-IOV [51]. The idea is to virtualize physical PCI functions into

virtual functions. Virtual functions can be shared to guests allowing SR-IOV compliant devices to offer interfaces to different guests.

5.1.2 The state of FPGA adoption in the cloud

FPGAs are more and more integrated in the cloud complementing GPUs solutions. Indeed, compared to GPUs which are programmed using only software, FPGAs are facing some issues, in particular regarding programmability and a lack of dynamicity (meaning that they are not easy to reprogram on the fly). But tools are making progress, in particular HLS solutions which are more attractive to software developpers. HLS offers easier development, debugging and maintenance reducing the cost of implementing complex features on FPGA making them more and more attractive compared to GPUs for some classes of application.

Though, programmability is not the only issue to solve; Integrating FPGA is also a big challenge. Two point of views regarding the integration of FPGAs can be distinguished: 1) FPGAs as a placeholder for devices, 2) FPGAs as programmable logic that must be shared to end users as such from guest operating systems. The first view is quite well addressed by some works [52–54] and is focused on sharing the programmable resources of FPGAs themselves to guests. We chose to target the second view as we believe that the two are complementary. Even though we see FPGAs as devices placeholders, it does not prevent works sharing resources of FPGAs to be compatible with our solution.

In a cloud context, FPGAs are mainly populated with accelerators such as video encoding, signal processing and such in order to accelerate software tasks. Integrating such devices to give guests an access to them has not really been done for FPGAs specifically but rather for all devices in general. VIRTIO (Virtual Input Output) is a Linux kernel API in which communications between a guest and its hypervisor are abstracted [55]. Guests contain front-end drivers for block devices, network devices, PCI emulation, a balloon driver (used to manage guest memory), and a console driver. These front-ends communicate with backends in the hypervisor (for instance Linux KVM [56]) using queues of requests. This design is very inspiring and resembles the Xen split-driver design. Both influenced our design as they are well proven designs that works in production systems.

Another solution to share devices, though limited to PCI interfaces, is Intel's SR-IOV (Single Root I/O Virtualization) [57]. With SR-IOV, a physical device, referred as a Physical Function (PF) appears as multiple Virtual Function (VF). Each VF has its own configuration space and can be accessed with its own IDs. It also has its own memory space used to map its register interface. Virtual functions are accessed by guests exactly in the same way as if they were physical functions. Not all devices are SR-IOV compliant and this is often used with network cards to improve network performances [58]. This solution is also an inspiration for us and shows in particular that device supporting different contexts of execution (hence multiple guest accesses) is feasible and can be considered in our design choices.

5.2 The impact of our proposal for the cloud

To help FPGA adoption progress further in the cloud, programmability must be improved [59] as well as device integration [46, 60]. The programmability issue is well addressed [61–63] and the improvements are made at a good pace [59, 64]. The device integration issue though needs improvement to help reducing the cost of FPGA integration. Moreover, interesting works [65–68] such as dynamic partial reconfiguration of FPGAs open new usages and bring more dynamicity [69, 70], and therefore more challenges, in the lifecycle of devices. Finally, cloud systems have migration capabilities where guests can move from one hardware platform to another, potentially changing available devices at runtime. This shows that this dynamicity must be kept in mind when designing an integration solution for the cloud.

Device integration must be improved to follow these evolutions, although it needs to accommodate some constraints to keep the solution feasible. These improvements must follow the needs of cloud systems. It means that direct hardware access is not a viable solution as devices must always be integrated keeping in mind that they will be shared among multiple guests. Modifying all drivers to support paravirtualisation is not sustainable neither, this requires too much work to be done for every existing operating system. Hardware side, we also can't make a major revolution by asking to change neither system interfaces, be it memory, peripheral or processing, nor adding specific CPU extensions. Instead, we need a solution that fits well in the existing cloud systems and helps improve FPGA integration with a smooth transition.

We believe that our solution can help to improve the integration of FPGAs considering those constraints. The two main strengths of our proposal, message-based communication and class genericity, are key to address the problem. Indeed, they not only allow more efficient device integration but fit well in existing hypervisors without the need of modifying them.

We focus our solution on paravirtualized guests. Indeed, non-paravirtualized guests can hardly use messages. Their interface with devices can only be the traditionnal register-based one, they thus can't be aware of other guests and their channels. The only types of devices they can handle are thus either fully dedicated hardware devices (making these devices exclusively usable by one guest) or emulated devices. This type of guest is interesting to reuse existing device drivers but is less and less common because devices are hard to share among them and device emulation is not suitable for high-performance systems.

5.2.1 Message-based communication

Message-based communication is well suited for driver-device communication in cloud systems. This is in particular confirmed by existing drivers in paravirtualized operating systems such as the Linux Kernel which already has paravirtualized drivers using message-based interface, e.g. the block and network drivers. Indeed, compared to reading and writing in hardware registers, sending and receiving messages is an easier task to multiplex and demultiplex among multiple guests. Several messages coming from various guests and going to the same device will simply be sent one after the other and processed in the same order. In the other way, a device may send multiple messages to various guests, those messages being routed to the right destination. This can be done efficiently through producer-consumer queues.

The good news is that message-based drivers are not huge pieces of code. For instance, in Linux, the frontend part of the paravirtualized block driver is only a little more than 2.7k lines of code long and the backend part is around 3k lines of code. It shows that with a reasonable implementation work, a paravirtualized message-based interface can be implemented and integrated within an operating system.

Hardware side, the impact of messages depends on the programming interface of devices. There are two possibilities here; either devices are integrated with the traditional register and interrupt based interface, or they are able to process messages themselves.

In the first case, it means that using messages will require a piece of software that interpret messages coming from guests and perform the appropriate register reads and writes. In the opposite way it will also produce messages for guests when an event happens on a device. Even though this works, it may become a problem as it requires specific drivers interpreting messages differently for each existing device. Linux mitigates this issue by having this piece of software interfaced with higher-level layers. For instance, a message containing a block operation on a storage device will be translated in a Block I/O request pushed in the right block device request queue. As the kernel already has drivers for a lot of storage devices, this is not a problem for Linux.

In our solution though we are in the second case, our devices have a message-based interface and will process messages themselves. It means that software side, small message proxies making the multiplexing and demultiplexing tasks are needed but no real message content processing is required. This allows only one generic backend driver allocated to message transfers, removing the issue of writing many complex backend drivers.

5.2.2 Class genericity

Using messages not only helps reducing the complexity of backend drivers but also unlock class-generic drivers. For cloud systems, this is key to reduce the number of required paravirtualized frontend to develop. This helps to aggregate similar devices, mitigating a lot the heterogeneity of device hardware implementations. It means that all devices belonging to the same class are to be driven by the same frontend. For instance only one frontend for every storage device is required, reducing the amount of work to interface this frontend with the kernel.

Drastically lowering the number of drivers to implement in order to support a lot of devices is really the key to help operating systems to be ported in hypervized environments. Indeed, the less work it requires to write frontends, the more likely operating systems will be able to run on hypervisors with reasonable hardware support. This offers great possibilities for less popular operating systems to grow up in cloud systems with a reduced cost.

In a way, Xen related drivers in Linux already have class-generic drivers. Indeed, block or network devices can be seen as generic classes having their own message-protocol. The block class uses Block I/O requests while the network class uses `sk_buff` structures.

Though, communicating with a block Linux backend from another operating system is quite hard as it would need to gather a lot of code from Linux. However, it shows the benefits of grouping devices in classes even for cloud systems.

5.3 Feasibility

With the benefits that our solution can bring to cloud system, we will now see that it is also feasible to implement it in today hypervisors. We will see what are the base features of every hypervisors and how our solution can be built on top of that.

5.3.1 Concepts offered by hypervisors

Hypervisors give guests a context of execution that can be compared to a process execution contexts. This execution context consists of giving guests a set of vCPUs (for virtual CPUs) as execution units. The number of total vCPUs given to all the guests on a specific platform can be lower or higher than the number of actual CPUs. These vCPUs are mapped to actual CPUs by the hypervisor scheduler to equally allow each guest to advance in its own execution. It means that in opposite to an operating system running on an actual hardware, a guest operating system can sometimes be completely halted for a certain period of time while others are running. This must be taken into consideration in particular when communicating with the hardware. In some cases, it is required that a guest (for instance communicating with hardware devices) is never scheduled out. To do that, it is possible to force a vCPU to be always active (or to be pinned) on an actual CPU.

The memory space is shared between guests in also the same way as with processes but with an additional layer of address translation. The machine space (the actual hardware memory being on the system) is shared with a guest by giving the physical address spaces. Physical addresses are then translated by guests themselves into virtual addresses to allow their processes to share this physical space. This means that to access the actual data on hardware memory from an address in a process, this address must be translated to a physical address and then to a machine address. This prevents guests to corrupt their data and also to access hardware they don't have the privilege to interact with.

Hypervisors also allow vCPUs to receive interrupts. These are virtual interrupts and are different from actual hardware interrupt. Indeed, as multiple guests can run alternatively on the same CPU, we have to make sure that hardware interrupts are triggering the right guest while it is currently running. There are two ways of mapping real interrupts to virtual guest interrupts. The first and easy one is when a guest has its vCPU pinned on a particular CPU. In this case, the hardware interrupt can directly trigger the guest making the situation the same as if the guest operating system was not hypervized. The other solution is to route the interrupt from the hypervisor. In this case, the hypervisor receives the interrupt by installing its own interrupt handler. In its handler, the hypervisor will find to which guest the interrupt is designated (this can have been setup in various ways, usually at guest startup) and trigger a virtual interrupt to its vCPU. Finally, virtual interrupts can also be used to send guest-to-guest notifications, allowing a guest to trigger an interrupt in another one, generally notifying that a shared resource has been updated.

The other important feature that hypervisors allow is shared memory. The hypervisor can setup page translation so that two different physical addresses from two different guests target the same machine address. This allows two guests to share data and communicate together. This is in particular very useful when using the split-driver model, where frontend drivers have to communicate with backends in different guests.

5.3.2 Description of our solution

Our solution is architected as depicted Figure 5.2. In the FPGA, there is a device integrated into the hardware interface of our conduit. This is the same architecture as in the previous chapters, nothing new here, the device exchanges messages with its driver through our conduit. On the software side however, the situation is a bit different due to the fact that we are in a hypervized system. The device driver sits in a guest operating system (guest1 in Figure 5.2). This driver exchanges messages with its device using our software interface also sitting in the guest.

Between the driver and its device, there are still channels moving messages between both ends. To connect channels to its device, the driver has to follow the exact same procedure as before, with exactly the same API. It first registers itself as a driver with the software bus to be matched with its device. When matched, it connects channels and starts exchanging messages with its device. The device lifecycle is also unchanged,

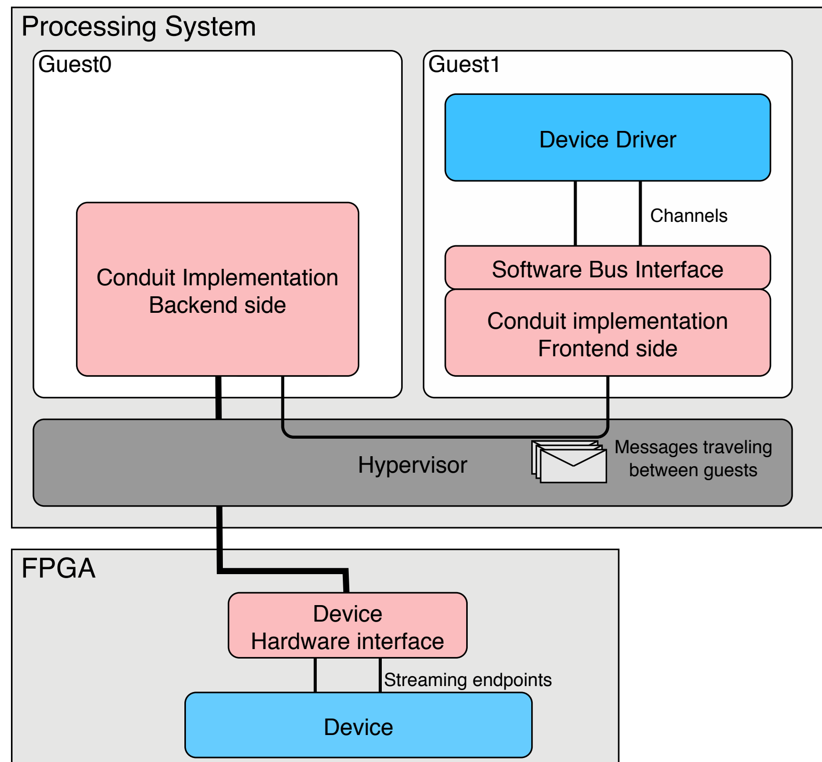


FIGURE 5.2: Our solution architecture in the cloud

any disconnection event is notified to the driver and it can also disconnect from its device at any time.

On the hardware side, the device is connected to its controller through the same interface made of streaming endpoints pairs. It sends and receives messages through these endpoints and honors the same lifecycle protocol we described in Chapter 3. It exchanges with its device controller a set of generic messages allowing it to follow its lifecycle. Upon reset, the device sends a first message and exchanges a few messages to describe itself, and waits for a driver to be matched. The device can also be unmatched as usual, either because it sent a message notifying it has become unavailable, or because the driver has been unmatched from the software side.

In a supervised system, our solution thus makes the situation of both ends unchanged in its principles, both ends keep the same interfaces. The difference stays in the middle, in the implementation of the channels. Guests have a limited access to the hardware, they can't directly access device controllers all at the same time. To do that we need a unique point used to multiplex hardware accesses, a special guest allocated to this task, called guest0 in Figure 5.2. This side is commonly called the backend side and contains the first half of the conduit implementation. The guest0 accesses the hardware and forwards messages to other guests such as guest1 without interacting with device

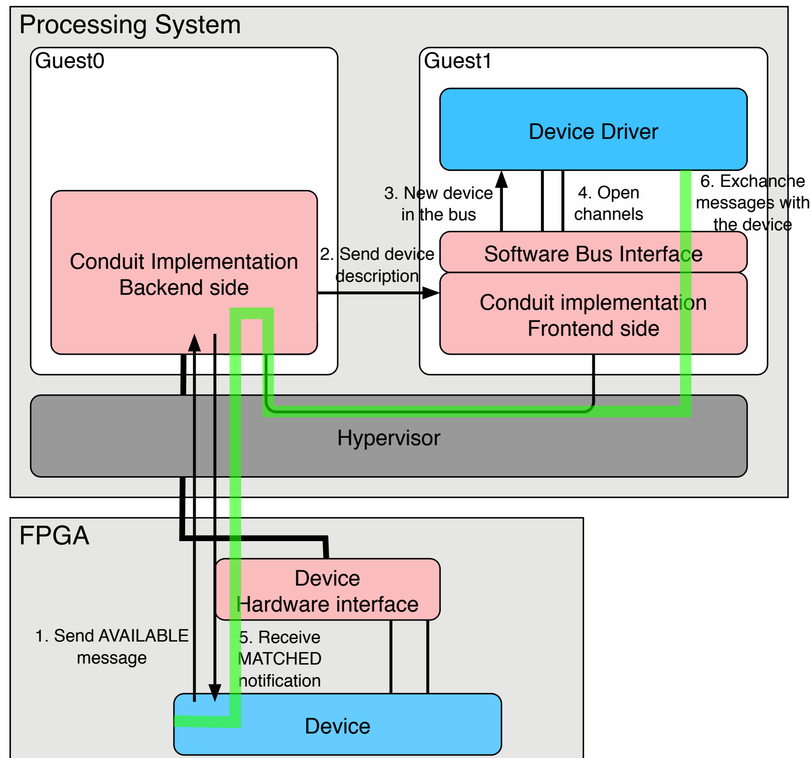


FIGURE 5.3: New device probing

features. These messages are exchanged with the help of the hypervisor and paravirtualized guests in which our conduit implementation sits. In guests, the side commonly called the frontend side, there is the second part of the conduit implementation making the interface with drivers.

Figure 5.3 shows the scenario of a device waking-up and opening channels with its driver. When the device wakes-up it sends its first (*AVAILABLE*) message containing its description to the guest0. The guest0 is being notified of the new device and forwards the device description to the guest1. With this description, the software bus can reify the device and allow drivers to match it.

Once matched, drivers will open their channels. This will result in the guest0 sending a *MATCHED* notification to the device. As multiple drivers among multiple guests can access the same devices, a device in this situation can receive multiple *MATCHED* messages. Thus, each one of these messages contains an extra field, a unique guest-ID integer used to allow both the device and its controller to track all the guests accessing it.

Once channels are open, the driver starts sending messages to its device (through the green path in Figure 5.3). Those messages are sent by the guest1 to the guest0 and

then forwarded to the device by the guest0. In the other direction, messages are sent by the device, caught by the guest0 and forwarded through the guest1 to the driver. The header of messages contains an extra field, the guest-ID used by the device to know which guest a message is coming from, or which guest a message is going to.

In the case of unmatching a driver from its device the situation is similar as before. When the unmatching starts from the driver, a notification is sent to the guest0 which starts shutting down all channels. From here the situation is a little different as multiple drivers from multiple guests access the same device at the same time. This means that we cannot simply cut all streams and reset the device. Instead, the device controller will only cut the channels of the driver being unmatched using the ID used in the *MATCHED* message. The device will then only be reset by its controller when all drivers have been unmatched to start a new lifecycle.

In the case of a device becoming unavailable, the situation is also similar as before. The device sends its *UNAVAILABLE* message, making its controller cutting all streams. The guest0 is then notified of the event by the controller and forwards this notification to all guests (only guest1 in Figure 5.3). In the guest1, the driver is then notified that it has lost its device to clean up all resources that are related to it.

There are two stages of communication between devices and drivers: A first stage between devices and the guest0 and a second stage between the guest0 and other guests. To allow the guest0 to communicate with devices, we use the exact same hardware architecture as before. Devices are interfaced with device controllers which typically contain a DMA engine. The guest0 is given the full access to these controllers allowing it to drive the DMA engine in order to exchange messages with devices.

Figure 5.4 shows this architecture using a DMA-based controller with ring buffers to send and receive data. Messages coming and going from and to guests are multiplexed and demultiplexed through the DMA. When a message comes from a driver, its header is filled, in particular with the guest-ID and then is put in the transmit (TX) ring. The DMA consumes messages from the ring and sends them through a stream-based bus (i.e. an AXI-stream interface) going to the device. The device then receives those messages from its reception endpoints, processes them and prepares a response. This response is filled with data and the guest-ID of the destination and then sent through a transmission endpoint. These responses are then put in the receive (RX) ring by the DMA. To demultiplex messages, we use the guest-ID field in message headers to know to which guest the message must be forwarded to ultimately reach the right driver.

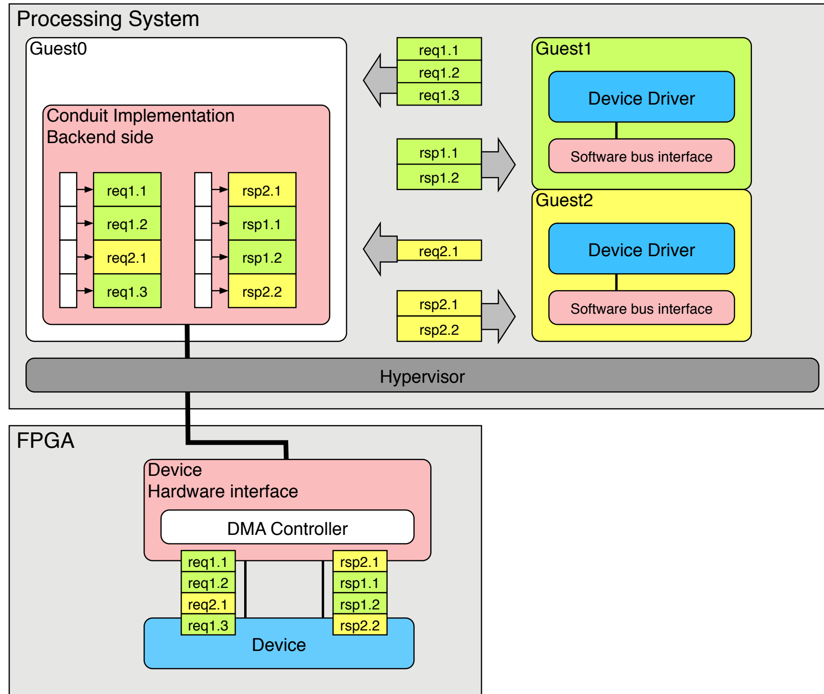


Figure 5.4: DMA based architecture

When there are multiple devices on the same FPGA, two main designs principles are possible: 1) Using a full device controller for each device, multiplying the number of DMA controllers by the number of devices, 2) Sharing one DMA controller among multiple light device controllers. In both case, the situation stays the same from the point of view of devices and their drivers. In the first case, there is not much changes compared to using one device. There is one pair of RX and TX rings per device controller and messages are put in their respective TX or RX ring. In the second case, all devices share the same pair of rings. It means that we need a way to link each message with its corresponding device. To do that, we add an extra field in the message header containing a unique device identification number or device-ID. This ID is set statically at hardware design time because we assume that there is no modification of the architecture that resides on the FPGA at runtime. Messages are then put with the right device-ID in the shared pair of rings.

Figure 5.5 shows an example of a multi-device architecture with multiple guests accessing them. In this architecture, each device has its own full device controller with its own DMA controller. Guest1 has drivers connected with devices 0 and 1 and guest 2 has a driver connected with device 1 only. In this example, the rings of the device controller of the device 1 are shared with two guests. When a message comes from a guest, it must be put in the right pair of rings to go to the right device. To do that,

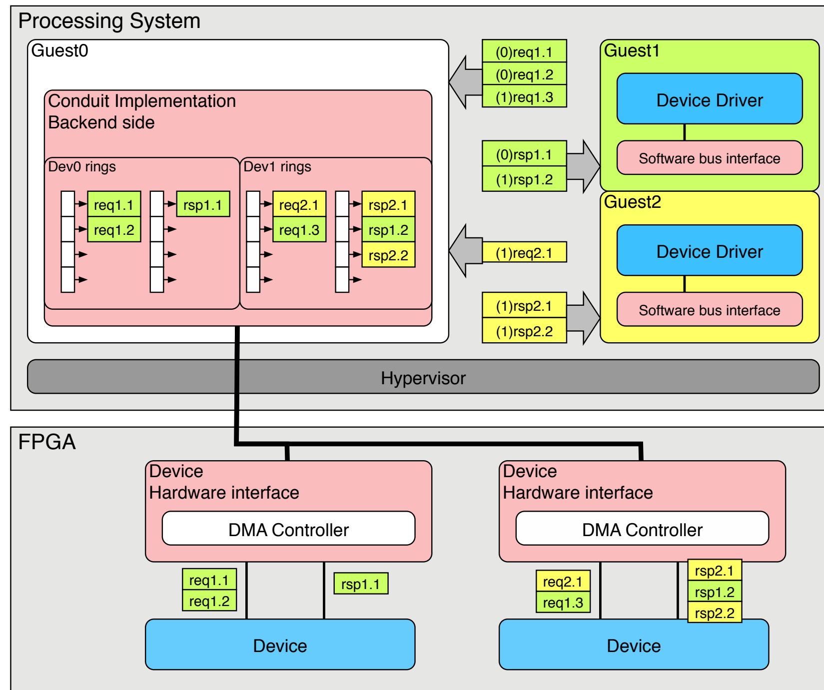


FIGURE 5.5: Multiple device in a DMA based architecture

guest0 keeps a table associating each pair of rings with an internal device number. This device number is set beforehand and given to each guest with device descriptions. Then, when a message is sent from any guest to the guest0, this device number is given at the same time, allowing the guest0 to know in which TX ring the message must be put. When receiving a message, this device number is given when forwarding the message to the right guest. This will allow the guest to find for which driver the message is intended.

The second stage of the communication takes place between the guest0 and other guests. Guests communicate using shared memory and interrupts, but we need to build a design on top of that to allows messages to be exchanged easily between guests. The design we choose is suitable for a wide range of devices and is based on shared memory ring buffers, similar to what we have seen with DMA controllers. For each guest, we setup a pair of ring buffers to allow it to communicate with guest0. GuestX to guest0 communication is made in the TX ring and guest0 to guestX is made in the RX ring. Ring buffers are great lockless structures for producer-consumer communication. They allow the producer to push any message it wants without locking the structure while the producer is consuming them at the same time.

Our ring buffers are made of request/response elements. These elements are written by both sides. The producer writes a request in the ring and the consumer reads and

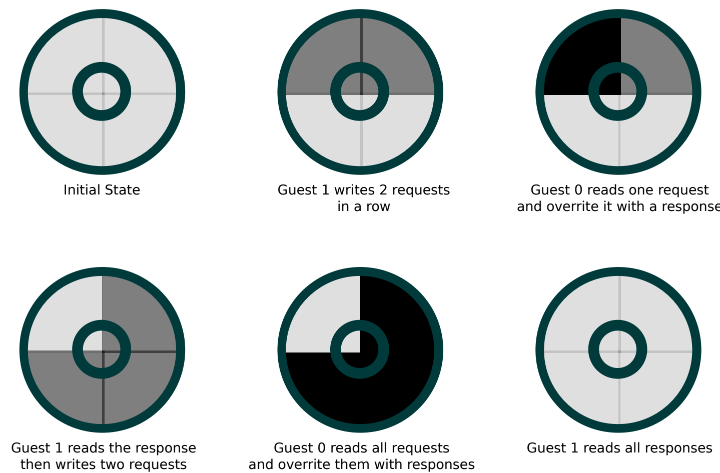


FIGURE 5.6: Ring buffer example

processes them then overwrites them with a response. We allow that by defining each element as a union of requests and responses such that the size of an element is the biggest size among a request and a response.

Figure 5.6 shows an example of a simple request/response scenario where guest1 is the producer and guest0 the consumer. Guest1 first fills two requests in a row. After writing them, it increments a counter indicating which part of the ring has been used (in other words, this counter indicates the head of the ring). Guest0 reads one request, processes it and overwrites it with a response. It then increments another counter indicating that it has filled one response in the ring. Guest1 reads the response and increments a counter indicating where the tail of the ring is, making the element available for a new request. It then writes two additional requests in the ring. Guest0 reads all requests and overwrites them with responses, updating the response counter. Guest1 consumes all responses and empties the ring by updating the tail counter.

We see that there are three relevant counters here, the head counter, the response counter and the tail counter. The head counter is incremented when the producer writes a request in the ring. This counter is only incremented by the producer and never decrements. We also don't have to take care of overflows because we make the buffer size a power of two such that the lower bits of a counter indicates an index in the ring. The response counter indicates where the last response is. It thus indicates that the request segment is contained between the response counter and the header counter. The consumer can then process any request and write any response between

those two counters. The response counter is used by the producer to know the limit to where it can store requests. It only reads that counter to ensure not to overflow responses with new requests. When it reads responses, it increments the tail counter indicating where the end of the read responses is, allowing it to add more request in the ring.

This type of ring can be full in two cases. Either the ring is full of requests, meaning that the consumer should consider processing some of them and write responses in place. This will allow the producer to process these responses and leave place for more requests. Or the ring is full of responses meaning that the producer should consider processing some responses before pushing more requests.

Having each pair of guests access these ring buffers requires that they both are able to access the same data in memory. Thankfully this is a feature that hypervisors give us using shared memory, allowing us to store ring buffers in pages accessible by both ends. To do that, the guest0 sets up the pages to store the ring buffers and grants an access to the other guest using the hypercall interface. The other guest then allocates a corresponding range of virtual addresses to map them to the shared space. Both ends can then start using the rings when they know each part is ready.

Having the ring buffer set up in shared memory is not enough to efficiently use them. Indeed, each end has to know when the other has updated the counters, otherwise, they would have to poll them indefinitely. To do that, we use guest-to-guest notifications that we map on interrupt handlers. For each ring, we setup a notification channel in both ways, allowing each end to notify the other when it has made an update to the ring.

With ring buffers and notifications, messages can travel between guest0 and other guests setting up one pair of ring and one two ways notification channel per couple of guest0/guestX. Ring buffers are setup as scatter-gather rings containing chunks of messages traveling between guests. Ring buffers elements are filled with a message buffer descriptor and booleans indicating if the buffer is the first or the last part of a message. Message buffer descriptors contain a reference to a shared page containing the chunk of the message, an offset and a length. The shared buffer containing the chunk is shared exactly in the same way as pages containing rings themselves are.

Message buffers must be allocated and granted to be shared and transported between two guests, leading to the question of which side allocates those buffers and when it grants them. This can be managed in two ways. Either they are allocated by each

end and granted to the other on the fly, or all buffers are allocated and granted by the guest0. Both solutions have advantages and drawbacks and choosing one over the other isn't a trivial choice.

In the first case, each end allocates its own buffers and grants them to the other. The main advantage is that it avoids copies from the sender of messages. It also is the easiest solution to implement as each end never runs out of buffers (except when running out of physical memory). Though, it can induce copies in the guest receiving messages. For messages coming from a guestX, going to the guest0 and eventually to the device, buffers must be in an address space accessible by the hardware. This kind of space is very often not the whole physical memory space and buffers must thus be copied in the right space. For messages coming from the hardware, data contained in them are often copied in guestX to other previously allocated buffers. For instance user owned buffers, used to interface the kernel space with the user space, are allocated prior to receiving the message containing the data. Thus, all or part of the data contained in the message must be copied in this user buffer. The other problem is that each time a new buffer is allocated, it must be granted on the fly adding non negligible overheads. Indeed, granting pages ends up in page table manipulation and thus cache flushes impacting performances. Even though this can be mitigated by smartly granting multiple buffers in one burst, it is not guaranteed that it can be done in optimal ways all the time.

In the second case, buffers are all allocated and granted beforehand by the guest0. This solution is a little bit more complex to implement. The guest0 must manage a pool of buffers and their ownership. Some buffers must be given to guests beforehand in order to let them write messages to send to devices, giving back the ownership to the guest0. Some other buffers are given to guests when messages are coming from devices, letting guests process the data into them and give back the ownership to guest0. This has the advantage that all buffers will be accessible by the hardware, removing the need to copy buffers before sending them to devices. Though, pre-granted buffers may still induce copies when sending messages from guestX when the data to send cannot be put directly into those buffers. This is the case again for user buffers that are allocated by other layers in the kernel. But even though this is still not always ideal, it still removes potential copies as we are certain that these buffers are accessible by the hardware saving some performances. For buffers coming from devices, the situation is unchanged as the only difference with the former solution is that a pool of buffers is granted beforehand instead of on the fly.

It is important to note that this ring buffer design is an example of implementation, suitable for a lot devices, but is not meant to be a standard. Indeed, some devices may have very specific throughput requirements, in particular regarding video or audio streaming devices. In this case, a smarter design may be necessary, for instance using time allocation (USB call this isochronous). In this type of design, a period of time is allocated regularly to each devices, such that they are all guarranteed to transmit messages even when others are saturating the bandwidth. Multiple designs can coexist in the same system (the low-level driver design is meant to aggregate them) and the one we just presented is a good start for general-purpose devices.

5.4 Deeper Analysis

To further analyze how our solution can integrate with existing hypervisors we verify the compatibility with the Xen hypervisor. We will also see what are the consequences of our solution for device makers and cloud systems.

5.4.1 Integration in Xen

Xen architecture is similar to most hypervisors. It offers guests in which operating systems are running separately as well as all the features we introduced before. We chose Xen as it is fairly common in the industry and is quite well documented but our solution is still compatible with hypervisors offering shared memory and guest-to-guest interrupts.

Paravirtualization in Xen is architectured around the concept of split-driver [71]. A normal guest, called DomU (for domain U), has its own little frontend communication with a backend running in a privileged guest called the Dom0 which has all access to the real hardware. From now, guests will be called domains as it is the preferred term in the Xen environment. This Dom0 typically runs a Linux kernel in various available distribution such as Fedora, Ubuntu, CentOS, etc.

In addition to that, Xen offers a feature called the xenstore. The xenstore is a database shared between all domains to store and access configuration and status information. Data in the xenstore are organized as a tree in which nodes are an association of a key and a string value. It is maintained by the Dom0 which manages read and write

accesses. At startup a domain is already connected to the xenstore so that it can read its configuration right at the beginning of its execution. Listing 5.1 shows an example of the xenstore when only the domain 0 is running. The syntax is fairly simple, each line associate a key with a value, both being strings and an indentation step represent one level in the tree.

```
# xenstore-ls : pretty prints current xenstore configuration
local = ""                     # Local machine
 domain = ""                   # Starting domain definition
  0 = ""                       # Dom0
   name = "Domain-0"           # Named as expected
   device-model = ""           # Path to the device model program
                               # (Default used if empty)
   state = "running"         # Guest state (running, halted, etc)
   memory = ""                 # Domain memory configuration
    static-max = "524288"      # Maximum domain memory in MB
    freemem-slack = "1254331" # Amount of memory left free
                               # (a minimal amount is necessary
                               # to let other guests boot)
```

Listing 5.1: Xenstore example, borrowed from https://wiki.xenproject.org/wiki/XenStore

The xenstore is useful in particular when two domains want to share data. Both domains will use the xenstore to store informations about the sharing as well as their status along the setup of this sharing. This is illustrated by the three following listings which report the evolution of the state of the xenstore during this task. Typically, things would happen as follows. Let us say that Domain 1 and Domain 2 want to share a memory page. Domain 1 sets up a space in its xenstore space to allow other guests to add their domain number when they require a shared page. Then, Domain 2 writes in the new field `connections` of Domain 1 to notify that it would like to setup a shared page as shown in Listing 5.2. Domain 1 then asks xen to grant the page to Domain 2, giving a granted page number (432 in the example). Domain 1 then writes this number in the xenstore with its own status as shown in Listing 5.3. This number is then read by Domain 2 which maps the page in its own address space and update its status in the xenstore. Finally, Domain 1 updates its status one last time, making its xenstore space looking like Listing 5.4

```
local = ""
 domain = ""
  0 = ""
   name = "Domain-0"
```

```
1 = ""
 name = "Domain-1"
 connections = ""
   2 = "WaitingForPage"    # The state of domain 2
2 = ""
 name = "Domain-2"
```

Listing 5.2: Setting up a shared page in Domain 1

```
1 = ""
 name = "Domain-1"
 connections = ""
   2 = "WaitingForPage"
     PageNumber = "432"
     Status = "WaitingOtherEnd"
```

Listing 5.3: Intermediate state of the sharing procedure. Here Dom2 is waiting for Dom1 to send the reference to a shared page

```
1 = ""
 name = "Domain-1"
 connections = ""
   2 = "Shared"
     PageNumber = "432"
     Status = "Shared"
```

Listing 5.4: Finalization of the sharing procedure

Xen also implements event channels. They are notification channels allowing a guest to notify another guest. They are similar to interrupts in the way that they store one bit of information triggering an event by transitioning from 0 to 1. Also as interrupts, notifications following an event are masked until the bit is cleared back to 0. Event channels can also be used to receive hardware notifications, IPIs or VIRQs (the latter being often use in timers or other CPU events).

Our solution has been implemented for experimentation and debug purpose of top of all these features. Although we could not achieve proper performance evaluation, we can still confirm that the solution can be integrated in a Xen environment. The architecture follows the split-driver model. Front-ends in DomU are implemented as low-level drivers in our solution as described in Chapter 4. It sits behind the software bus API implementing channels by communicating with the backend sitting in the Dom0. These two drivers are fairly small, no more than 1.5k lines of code each. Both domains run the Linux kernel but we will see later that other solutions are possible.

To allow both domains to communicate we implemented the ring-buffers we described in the previous chapter on top of xen features. One pair of RX-TX channels (RX being from Dom0 to DomU) is setup for each pair of Dom0-DomU domains using the xenstore. When a DomU starts, the frontend writes in a special xenstore field setup by the Dom0 to notify that it requires a connection. The Dom0 then sets up an event channel, binds it to a handler and publishes its reference to the xenstore. It also grant a page for the RX ring and publish its reference as well. It does not prepare the TX ring page itself as it is usual that the owner of the page is the one sending requests and reading responses. When notified of these updates, the DomU connects to the event channel, maps the RX page and sets up the TX ring page. The setup phase ends when the Dom0 maps the TX ring page and sends a first event channel to notify that the communications can start.

The content of the messages between both ends is almost exactly the same as received by the hardware. This allows the frontend driver to react in the same way as if it was receiving them directly from the device without any backend in-between. Other specific requests such as message buffer granting and release are transmitted as other types of requests. Listing 5.5 shows the structures of both requests and responses traveling in both rings. It also shows the `shared_buffer_descr` used to store granted page references as well as their status to keep track of the current owner of a page.

```
struct msg_request {
    uint32_t type;

    union {
        struct msg_device_id dev_id;

        struct {
           /* Small space to store message headers
            * Avoid scattered messages to generate
            * too much small page sharing */
            uint32_t buffer[MSG_HEADER_SIZE/4];

           /* Buffer containing the message */
            struct shared_buffer_descr buffer_descr;
        };
    };
};

struct msg_response {
    uint32_t type;
```

```
    struct shared_buffer_descr buffer_descr;
};

struct shared_buffer_descr {
    unsigned int status;

    grant_ref_t ref; /* Shared page reference */
};
```

Listing 5.5: The structure defining messages between domains

When a new device is probed by the Dom0, the backend driver sends its description through the RX ring filling the `dev_id` field. Receiving this description, the frontend instantiates a new device structure and registers it to the software bus, allowing its driver to connect to it. Once a driver matches its device, the frontend sends the notification that the device has been connected, ending in the backend notifying the device itself. At this point, messages can start being exchanged through the rings.

Messages are transferred along the way between devices and drivers through a decent amount of layers. From the device, messages sent by it are put in memory by a DMA (in most practical cases) which triggers an interrupt. This interrupt is normally caught by the hypervisor and routed to the right guest. This is not a good solution for performance, so what we do, as well as a lot of hypervized systems, is to pin down the guest accessing the hardware on a specific core, so that the interrupt can be triggered directly on it. That interrupt will trigger a top handler that will in turn schedule a bottom handler. This bottom handler will look in the DMA ring to find received messages. These messages are then put in the right frontend ring to forward the message. At this point two solutions are possible: copying the message in an already shared buffer or sharing the message buffer on the fly. In both cases the frontend will be notified by an event triggering a top handler. This handler is as for hardware interrupts in an interrupt context and will thus schedule a bottom handler that will read all incoming messages. These messages will be then sent to drivers through the software bus API triggering their `received()` callback.

In addition to integrate our solution in guests running a Linux kernel, we experimented integrating our solution in a smaller operating system called MiniOS. MiniOS is provided by Xen and can be seen as a hardware abstraction layer, useful to easily implement a small and quick paravirtualized guest. It was first intended to be used to disaggregate the Dom0 into smaller privileged domains to improve security, isolation

and reliability in general. In practice, this idea has not been transformed to a final product but thankfully MiniOS is still maintained enough so that it is possible to integrate our solution in it. This allows small projects to benefits from our solution to have an easy entry point to integrate in a Xen environment. It also helps to open the way to move hardware accesses into MiniOS as a Dom0 module and benefits from its small footprint. Indeed, as good as the Linux kernel is, it is a huge piece of code that may slow down execution, especially when reacting to hardware interrupts and forwarding messages to DomUs. Even though a proper performance analysis is required, we are confident that such work can be beneficial to remove part of the backend overhead running inside the Linux Dom0. This solution does not force to change everything, the Dom0 can still be used as before and is actually crucial as it runs the xenstore. It would require this MiniOS backend to get a direct access to the FPGA interface (typically the PCI interface in most cases), the difficult part of this work being to write the DMA driver in MiniOS.

5.4.2 Consequences of our solution for cloud systems

For hardware devices, our solutions means that they have a message-based interface with multiple entities (i.e. guests) with which they communicate. We already argued about the fact that using messages for a device does not increase to much its complexity, but managing multiple contexts for a device may become complex. This complexity depends on the nature of the device and how it works, more specifically on whether or not the device is stateless. A stateless device has a communication model in which the same request will have always the same response, no matter how many times it has been sent before. In other words, a stateless device does not keep a memory of previous events, its responses only depends on the data contained in requests. These devices can be sensors, accelerators such as video compression or signal processing. Such devices are easy to share among multiple guests, multiplexing requests and demultiplexing responses will do the job. If some global configuration is required though, it is still possible to reserve a specific message channel to a privilege guest, tagging that channel as an ”exclusive” one.

For stateful devices, the situation is a little more complex, as it will require for the device to remember the context of each guest. This implies an additional implementation effort but the benefit is that the device will be available for cloud systems. The amount of work is hard to evaluate in general; it really depends on the complexity of the device

itself and whether or not the context memorisation and switching can scale enough to allow a various number of guests to use this device. Though this is not impossible as devices under SR-IOV already exist, this particular question needs more research to evaluate how existing devices can be turned into guest-aware devices.

To mitigate the issue with stateful devices, the idea of context-switching in FPGA [69, 70] can be a solution. This idea is already well developed and offers reasonable performance with a low enough latency to be considered interesting to try out. The idea is to gather the state of the device (i.e. the values of the set of registers that are needed to characterize its state), store it in memory and replace it with a previously saved state. As it can take some time (up to a few milliseconds), this is a solution that suits devices with a non-critical latency but not real-time devices. The impact of the context switch latency can also be reduced by having a smart message multiplexer that will reorganize messages to optimize the number of switches to perform.

Another consequence of the solution that exists for every split-driver based solution is the latency of the Dom0. Indeed, transferring a message from the device to the end driver in a specific guest requires to schedule the Dom0 first, transfer the message in its own ring, and schedule the right guest. As we said, reducing the footprint of the Dom0 by having a small hardware proxy guest can help but there is still room for improvement. A possible idea is to implement the message proxy in hardware directly and make it send and receive messages directly in guest rings. This solution could either perform data transfers in the right addresses by itself or use an IOMMU. Doing so would still leave the issue of interrupts as all hardware interrupts will wake up only the Dom0 that must in turn wake up the right guest leaving some latency in the Dom0. To reduce this latency, it is possible to use a tiny piece of code running exclusively on one core and dedicated to interrupt routing.

Overall, we believe that our solution is good for cloud systems. It reduces the size of frontend drivers improving its maintainability and its safety. Our solution also takes advantage of the concept of device classes, allowing a small backend driver that only act as a proxy. This makes devices more easily available to guests as the same backend can be used for future devices.

An interesting side effect about using messages is that frontend drivers can be in a lot of cases identical, whether in a hypervized or non-hypervized context. Indeed, as the communication with the hardware is performed either by a paravirtualized low-level driver under the software bus, or by a low-level driver communicating with the

hardware directly, the actual device driver will have the same interface in both cases. This reduces a lot the implementation effort to run an operating system in an hypervized environment, reinforcing our confidence in the fact that our solution eases driver development, whatever the context.

Chapter 6

Conclusion

FPGA technology opens a lot of great possibilities for silicon-based systems from small embedded systems to big cloud computing infrastructures. Though all those possibilities come with a major drawback: it worsens the overall integration process of devices into processor-centric platforms. Indeed, to fully exploit the dynamicity of FPGAs a new integration approach is necessary. An approach that allows devices to come and go at any time on the FPGA, having the right responsiveness to load and unload their drivers at the right time. An approach that simplifies the development of device drivers by hiding the hardware implementation details of devices. Finally, an approach that allows devices to share the same drivers, making them readily available and thus reducing their time-to-market and integration cost.

6.1 Summary

The approach we designed advocates to use messages as the communication paradigm between devices deployed on FPGAs and their drivers running on the processing system. This approach is built on existing software and hardware technologies and concepts. Though, it moves from existing frontier concepts, interrupts and memory mapped registers, to a simpler, safer and more cost effective frontier made of communication channels. The move can be made incrementally as the approach stays compatible with existing solutions and does not require a complete move by throwing out the current practice.

One of the key design point of our solution is an abstract conduit for sending and receiving messages. This conduit has abstract interfaces both with the hardware and the software, hiding the implementation details to both sides. This leaves only a message protocol to follow by both sides. Software side, the interface that faces drivers provides communication channels. These channels are easy to use to simply send and receive messages with devices. The interface also allows to discover new devices at runtime, allowing to dynamically load and unload the corresponding drivers and resources needed to drive devices. Hardware side, the interface that faces devices is made of simple stream-based endpoints. Each software channel is connected to two stream-based endpoints, one to receive and one to send messages. Those stream-based endpoints can be implemented with various standards, allowing to suit the needs of every user. For instance, we built our demonstration platform with the popular AXI-stream standard with success. Reusing such popular standards also allows to exploit existing hardware components and tools and still follow our design rules.

Between both interfaces travel messages made of variable-sized payload. These messages are general purpose, meaning that the payloads they transport are not interpreted by the conduit itself but encapsulated in a generic message protocol. This generic message protocol defines a generic lifecycle for every device. The lifecycle allows each device to announce itself at startup (either at FPGA reprogramming time or at reset time), to announce when it fails, reboots or leaves the FPGA.

Messages and the generic message protocol are the foundations of our solution. They enable the definition of classes of devices, leading to the specification of class-generic message-based protocols. This improves time to market of new devices because new devices belonging to an existing class will likely have an available driver to integrate them with the operating system. Our approach does not limit next innovations as message protocols can be extended. This can be useful to allow devices to follow their own very specific protocol without belonging to any class. It also allows users to define their own private class or proprietary extensions to existing classes. Even though that is not something we wish, we acknowledge the fact that some device providers may want to hide some details of their technology. With proprietary extensions, a device will still be usable by generic drivers exploiting generic features, but will require the proprietary driver to exploit all the features it offers.

Following our approach is thus applying the principles behind the USB ecosystem to the world of programmable logic. It brings enough confidence as messages and class-generic devices have proven how well they allow easy and cheap integration of a huge

number of different devices. The key elements that allowed the USB success story are 1) A simple and cheap interface for devices, convincing device providers that it is worth to make the small effort to move to the new interface, 2) Simple software concepts allowing simpler drivers by hiding hardware details behind a message-based interface and 3) Defining generic classes of devices allowing one driver to drive thousands of different implementations of the same type of device. As we retain all those elements, we have an additional confidence that messages are a very good way to integrate the device in the context of FPGAs.

Using messages improves the stability and the safety of embedded systems. Indeed, drivers are simpler to use when using messages to the point of view of a software developer. Software developers are more used to message protocols and software interfaces rather than to hardware interfaces. In our approach the hardware interface seen by the software is hidden in our message conduit, a piece of software that is written once for every implementation of the conduit, meaning much less often than the number of drivers and devices in systems. Systems following our approaches are also safer because the message conduit itself is safer. Indeed, only this conduit has to make DMA accesses, handle interrupts and have memory mapped registers. It hides this hardware-software frontier, reducing the risks of bugs leading to wrong memory accesses from devices, or wrong hardware accesses from the software. Additionally, the systems are also more secure as devices no longer make direct memory access by themselves, they can only send and receive messages through stream interfaces connected to our message conduit. This means that, intentionally or not, a device cannot make direct memory accesses to wrong addresses, reducing the risk of memory corruption. Even if messages are made of completely wrong data bytes, they will simply be treated as faulty devices without risking the integrity of the system.

We have demonstrated that our proposal is suitable for various environments, showing that the approach is flexible across a wide range of systems from small embedded systems to large cloud computing systems. Small systems with low processing power and small FPGA resources are suitable as the overhead we add for such systems is small enough. In those small systems, we built our conduit using existing suitable rather chip technologies such as memory mapped FIFO queues without big DMA engines. Our solution is also compatible with bare metal systems since it only adds a small layer of abstraction including the software side implementation of our message conduit. It can be adapted with existing hardware abstraction layers offering the basics to access the hardware, setup interrupts and cache memory management.

Bigger systems with high processing power and huge FPGA resources are also suitable. In such systems, data transfers are made by full scatter-gather DMA engines. Our impact on those DMA engines is negligible allowing to fully exploit the performances of existing technologies. Our approach is compatible with the Linux kernel and thus any Linux distribution. It integrates as a new software bus hosting message-based devices and drivers following our design rules. For cloud environment, we demonstrated that our approach not only suits well the architecture of the cloud but also offers the benefits of using messages to enable class-generic devices. Indeed, it allows guest operating systems to have a generic frontend for each class of devices bringing better device availability and thus a better hardware support in general.

6.2 Perspectives

As a future point of interest, we believe that our approach could benefit for partial reconfiguration of FPGAs, which might be of interest e.g. for cloud providers. In such technologies, part or all the FPGA can be reconfigured at runtime by the software, bringing additional dynamicity to systems integrating FPGAs. It means that devices deployed in such systems may start, be replaced at runtime by another device, and comeback later. As our approach allows devices to come and go at any time, it is completely suitable and may help integrating devices deployed in a partial reconfiguration context. Each block of partial reconfiguration may be interfaced with a set of endpoints on which devices can connect and start their lifecycle. Once the first messages exchanges helped recognize the device the right driver notified that a new device it can handle has appeared and interface it with upper layers of software. Clearly, the delay of partial reconfiguration can be critical, it means that a detailed investigation of the delay taken by the first exchanges of messages and potentially find solutions to reduce it.

Another future point of interest is device-to-device communication. For now we follow a pure host/device view but we see more and more systems in which device-to-device communication can be useful. For instance, when processing signal, multiple layers of filters may be used, and some filters may even be used multiple times. Having multiple layers integrated as devices may benefit from a device-to-device feature instead of using the CPU and memory between each layer to transfer data between them. Though, developing such an idea requires to review our message conduit, allowing channels to be not only from a device to a driver but also between two devices. This is

not something trivial to do as it means that architectures will become more and more symmetrical, slowly making the concept of host disappear. It will require to review device lifecycles, ensuring that rendez-vous can be made and broken without loosing messages or receiving ghost messages from previous channel instances. It will also require to determine how messages will be buffered. For instance, they can be buffered in a large external memory, ensuring that messages can fit entirely, or in small buffers between devices. The topology will also be important. For instance, a star topology may lead slow devices to slow down all other device communication but is efficient in term of area usage while a fully connected graph will ensure to have highly parallel communications but will consume a lot of area. It may also require an additional layer of generic message protocol to allocate device addresses and negotiate channel operations between devices. Finally, it means that devices will definitely have the ability to talk to multiple devices at the same time, leaving the ”one driver to one device” Linux scheme to something closer to what we have seen in the cloud computing context where multiple drivers in multiple guests can access the same device.

Bibliography

[1] Kurt Keutzer, A Richard Newton, Jan M Rabaey, and Alberto Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[2] Piyali Goswami, Sushaanth Srirangapathi, Chetan Matad, and Stanley Liu. Re-target-able software power management framework using soc data auto-generation. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 141–146, 2016.

[3] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 275–288, 2009.

[4] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 73–86. ACM, 2009.

[5] Tarek Ben Ismail and Ahmed Amine Jerraya. Synthesis steps and design models for codesign. *Computer*, 28(2):44–52, 1995.

[6] Jean-Yves Brunel, Wido M. Kruijtzer, Harjan J. H. N. Kenter, Frédéric Pétrot, Laurent Pasquier, Erwin A. de Kock, and Wim J. M. Smits. Cosy communication ip's. In *Proceedings of the Design Automation Conference*, pages 406–409, 2000.

[7] Antoine Fraboulet and Tanguy Risset. Master interface for on-chip hardware accelerator burst communications. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 49(1):73–85, 2007.

[8] Pieter van der Wolf and Ruud Derwig. Modular soc integration with subsystems: The audio subsystem case. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 157–162, 2013.

[9] Rolf Ernst. Codesign of embedded systems: Status and trends. *Design & Test of Computers*, 15(2):45–54, 1998.

[10] Philipp A Hartmann, Kim Grüttner, Philipp Ittershagen, and Achim Rettberg. A framework for generic hw/sw communication using remote method invocation. In *2011 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6. IEEE, 2011.

[11] Mattias O'Nils and Axel Jantsch. Operating system sensitive device driver synthesis from implementation independent protocol specification. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 562–567, 1999.

[12] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An idl for hardware programming. In *Proceedings of the 4th USENIX Operating System Design & Implementation Symposium*, pages 17–30, 2000.

[13] Shaojie Wang, Sharad Malik, and ReinaldoA. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In Ahmed Amine Jerraya, Sungjoo Yoo, Diederik Verkest, and Norbert Wehn, editors, *Embedded Software for SoC*, pages 69–82. Springer, 2003.

[14] S. Chen, L. Zhou, R. Ying, and Y. Ge. Safe device driver model based on kernel-mode jvm. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC '07)*, pages 1–8, Nov 2007. doi: 10.1145/1408654.1408657.

[15] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 89–105, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/cutler.

[16] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *USENIX Annual Technical Conference*, 2018.

[17] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[18] Wido Kruijtzer, Pieter Van Der Wolf, Erwin De Kock, Jan Stuyt, Wolfgang Ecker, Albrecht Mayer, Serge Hustin, Christophe Amerijckx, Serge De Paoli, and Emmanuel Vaumorin. Industrial ip integration flows based on ip-xact standards. In *Design, Automation and Test in Europe*, pages 32–37. IEEE, 2008.

[19] M. Jacobsen, Y. Freund, and R. Kastner. Riffa: A reusable integration framework for FPGA accelerators. In *Field-Programmable Custom Computing Machines (FCCM)*, 2012.

[20] K. Eguro. Sirc: An extensible reconfigurable computing communication api. In *Field-Programmable Custom Computing Machines FCCM*, 2010.

[21] M. Jacobsen and R. Kastner. Riffa 2.0: A reusable integration framework for FPGA accelerators. In *Field Programmable Logic and Applications (FPL)*, 2013.

[22] G. Marcus, W. Gao, A. Kugel, and R. Männer. The mprace framework: An open source stack for communication with custom fpga-based accelerators. In *2011 VII Southern Conference on Programmable Logic (SPL)*, pages 155–160, April 2011. doi: 10.1109/SPL.2011.5782641.

[23] Gernot Heiser and Ben Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM Asia-Pacific Workshop on Systems*, pages 19–24. ACM, 2010.

[24] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the development of soft devices. In *Proceedings of the USENIX Annual Technical Conference*, pages 379–382, 2005.

[25] V. Srinivasan, N. Parihar, V. Khurana, and A Gavrilovska. A split driver approach to soc virtualization - challenges and opportunities. In *Proceedings of the 39th International Conference on Parallel Processing*, pages 50–57, September 2010.

[26] Steven JE Wilton and Resve Saleh. Programmable logic ip cores in soc design: Opportunities and challenges. In *Proceedings of the IEEE 2001 Custom Integrated Circuits Conference (Cat. No. 01CH37169)*, pages 63–66. IEEE, 2001.

[27] Michael C McFarland, Alice C Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[28] Daniel D Gajski, Nikil D Dutt, Allen C-H Wu, and Steve Y-L Lin. *High—Level Synthesis: Introduction to Chip and System Design.* Springer Science & Business Media, 1992.

[29] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008.

[30] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 305–318, 2011.

[31] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. Faults in linux 2.6. *ACM Transactions on Computer Systems*, 32(2): 1–40, 2014.

[32] Universal serial bus. http://www.usb.org.

[33] John Regehr. Random testing of interrupt-driven software. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298, 2005.

[34] X3T9 technical committee of the American National Standards Institute (ANSI). Small computer system interface, 1994.

[35] Laurentiu-Cristian Duca, Anton Duca, and Aurel-Sorin Lup. Real-time linux drivers and latency evaluation system for ti omap4 mcspi peripheral. In *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–4. IEEE, 2020.

[36] Introduction to i/o kit fundamentals. https://developer.apple.com/library/archive/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/Introduction/Introduction.html.

[37] Windows driver framework. https://developer.microsoft.com/en-us/windows/hardware.

[38] Fred B Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984.

[39] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 41–50. IEEE, 2007.

[40] Tapasweni Pathak. Faults in linux 3.x. *Login Usenix Mag.*, 43(1), 2018.

[41] Patrick Mochel. The Linux kernel driver model. In *Proceedings of the Linux.Conf.Au*, 2003. https://www.kernel.org/doc/html/latest/driver-api/driver-model/overview.html.

[42] Tiobe index for november 2020. https://www.tiobe.com/tiobe-index/. Accessed: 2020-11-11.

[43] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[44] Feifei Li. Cloud-native database systems at alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[45] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435. IEEE, 2015.

[46] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation*, pages 991–1010, 2020.

[47] Harry Katzan Jr. Operating systems architecture. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, pages 109–118, 1970.

[48] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[49] Gary R Allred. System/370 integrated emulation under os and dos. In *Proceedings of the May 18-20, 1971, spring joint computer conference*, pages 163–168, 1971.

[50] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 125–134, 2006.

[51] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, volume 2, 2008.

[52] S. A. Fahmy, K. Vipin, and S. Shreejith. Virtualized fpga accelerators for efficient cloud computing. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 430–435, 2015. doi: 10.1109/CloudCom.2015.60.

[53] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328708. doi: 10.1145/2597917.2597929. URL https://doi.org/10.1145/2597917.2597929.

[54] Wei Wang, M. Bolic, and J. Parri. pvfpga: Accessing an fpga-based hardware accelerator in a paravirtualized environment. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–9, 2013. doi: 10.1109/CODES-ISSS.2013.6658997.

[55] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[56] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[57] Byron Gillespie, Marc Goldschmidt, Terry Sych, and Bruce Young. Method and apparatus for interfacing a device compliant to a first bus protocol to an external bus having a second bus protocol and for providing virtual functions through a multi-function intelligent bridge, May 12 1998. US Patent 5,751,975.

[58] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In *2010 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2010.

[59] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. Best-effort fpga programming: A few steps can go a long way. *arXiv preprint arXiv:1807.01340*, 2018.

[60] Fritjof Steinert, Philipp Kreowsky, Eric L Wisotzky, Christian Unger, and Benno Stabernack. A hardware/software framework for the integration of fpga-based accelerators into cloud computing infrastructures. In *2020 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 23–28. IEEE, 2020.

[61] Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic l'Hours, Nicolas Simon, Steven Derrien, et al. Gecos: A framework for prototyping custom hardware design flows. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 100–105. IEEE, 2013.

[62] Konstantinos Krommydas, Ruchira Sasanka, and Wu-chun Feng. Bridging the fpga programmability-portability gap via automatic opencl code generation and tuning. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 213–218. IEEE, 2016.

[63] Dirk Koch, Frank Hannig, and Daniel Ziener. *FPGAs for software programmers*. Springer, 2016.

[64] Christian Fibich, Stefan Tauner, Peter Rossler, Martin Horauer, Herbert Taucher, and Martin Matschnig. Preliminary evaluation of high-level synthesis tools-xilinx vivado and panda bambu. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–4. IEEE, 2018.

[65] E. J. McDonald. Runtime fpga partial reconfiguration. In *2008 IEEE Aerospace Conference*, pages 1–7, 2008. doi: 10.1109/AERO.2008.4526368.

[66] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb. Fpga partial reconfiguration via configuration scrubbing. In *2009 International Conference on Field Programmable Logic and Applications*, pages 99–104, 2009. doi: 10.1109/FPL.2009.5272543.

[67] W. Lie and W. Feng-yan. Dynamic partial reconfiguration in fpgas. In *2009 Third International Symposium on Intelligent Information Technology Application*, volume 2, pages 445–448, 2009. doi: 10.1109/IITA.2009.334.

[68] P. Sedcole. Modular dynamic reconfiguration in virtex fpgas. *IEE Proceedings - Computers and Digital Techniques*, 153:157–164(7), May 2006. ISSN 1350-2387. URL https://digital-library.theiet.org/content/journals/10.1049/ip-cdt_20050176.

[69] Arief Wicaksana, Alban Bourge, Olivier Muller, and Frédéric Rousseau. Demonstration of a context-switch method for heterogeneous reconfigurable systems. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[70] S. M. Scalera and J. R. Vazquez. The design and implementation of a context switching fpga. In *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*, pages 78–85, 1998. doi: 10.1109/FPGA.1998.707884.

[71] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the development of soft devices. In *USENIX Annual Technical Conference, General Track*, pages 379–382, 2005.