Analysis of the Usage Models of System Memory Management Unit in Accelerator-attached Translation Units

Kyriakos Paraskevas
School of Computer Science
University of Manchester
United Kingdom
kiriakos.paraskevas@manchester.ac.uk

Konstantinos Iordanou
School of Computer Science
University of Manchester
United Kingdom
konstantinos.iordanou@manchester.ac.uk

Mikel Luján
School of Computer Science
University of Manchester
United Kingdom
mikel.lujan@manchester.ac.uk

John Goodacre
School of Computer Science
University of Manchester
United Kingdom
john.goodacre@manchester.ac.uk

ABSTRACT

Including hardware accelerators to improve the performance while reducing energy consumption is becoming ubiquitous in computing systems ranging from large System-on-Chips (SoCs) for data centers to small embedded IoT devices. In accelerator-enabled environments, accelerator units are able to deliver a substantial increase in performance through increased programming effort and co-scheduling applications across the heterogeneous system. These accelerators need to have frequent access to the application data through the memory subsystem and therefore the need for efficient memory management is crucial. The memory subsystem needs to maintain the model of memory consistency between context switching, where resources such as accelerators are reallocated between applications and their defined memory space.

The consistency between the processor memory management and any System Memory Management Unit (SMMU) is either through the pinning of physical memory to specific virtual pages statically, or tracking page allocations dynamically and ensuring consistency at point of use of the memory between applications. Although the implication of these two methods has obvious consequences, the full implications on maintaining consistency of such translations have not been studied.

In this paper, we carry out use case measurements and analysis of three of the main usage models focused specifically on SoC level translation. We analyze the generic structure on SoCs given the example of the Zynq Ultrascale+ FPGA board that incorporates the state-of-the-art and widely used Arm System Memory Management Unit. We share our analysis and present the advantages and disadvantages to provide guidance in optimizing the integration of accelerators in a system. We also propose a usage model to utilize the embedded SMMU in order to enable accelerator devices to directly access application memory, through exposing a reusable API specific to the Zynq Ultrascale+ hardware.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422781

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; • **Software and its engineering** → **Software libraries and repositories**.

KEYWORDS

Arm SMMU, Memory management, Memory Virtualisation, Accelerators, FPGA, Memory isolation

ACM Reference Format:
Kyriakos Paraskevas, Konstantinos Iordanou, Mikel Luján, and John Goodacre. 2020. Analysis of the Usage Models of System Memory Management Unit in Accelerator-attached Translation Units. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3422575.3422781

1 INTRODUCTION

The wide adoption of heterogeneous architectures with accelerators capable of offloading workloads from a general-purpose processor led to increased application performance while reducing the energy consumption [12] [4, 13, 20, 26]. In the simplest usage model, the application running on the local processor utilizes the deployed accelerator(s) to achieve higher performance [24, 31, 32], but at the same time, the accelerator device might need to access or modify application data that reside in memory regions owned by the application.

Since accelerators are capable of accessing the memory space, the system must be able to cope with any malicious or accidental memory accesses. In the case of a single user application running on the system, there is no security or safety concern since most of the system resources such as available memory belong exclusively to the application, and it is assumed that it is the responsibility of the application to make reasonable use of it. But in cases where many applications are present in virtualized memory environments, the memory access model should provide memory isolation and guarantee non-interference between application's separate virtual

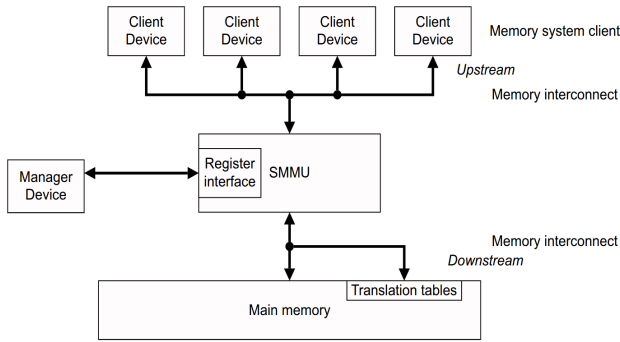


Figure 1: The implementation of the Arm System Memory Management Unit (SMMU). Client devices (accelerators) are connected through the memory interconnect to the SMMU in the upstream bus. The connection between the SMMU and the rest of the memory system is the downstream bus. When the SMMU is set, the client devices agnostically issue transaction requests to the SMMU. After a successful translation, the SMMU performs the memory access and returns a valid response, or fault otherwise. (Source: ARM [7])

address spaces, as shown in Figure 1. In such environments, accelerators may need to be redeployed fast to be assigned to multiple users.

Therefore, some features must be provided to enable virtualisation of accelerators, similar to the virtualisation of CPU resources. These include:

- Memory management, through the aspects of virtual memory and dynamic memory allocation-deallocation.
- Memory and hardware protection from malicious memory requests.
- Support for multiple application scenarios.

Accelerator-attached translation units can provide these features and have become widely popular. Dedicated management software can set up these accelerator-attached translation units on the event of user-switching. This enables multi-user support by providing the user virtual mappings through page tables to the translation units, while these units are protected by blocking any invalid or not permitted memory accesses issued from the accelerators. On the other hand, the solution to the general problem of maintaining memory consistency of virtual mappings between the accelerator translation units and the actual CPU MMU mappings in virtualised memory environments is not trivial. Userspace memory can change dynamically through allocations or deallocations, causing inconsistencies between the mappings in the CPU MMU and the attached translation units. This case raises questions regarding how user page table updates are provided to the translation units. Currently, it is addressed by pinning and updating the page tables on RAM or through dynamic memory tracking. But even when page tables are pinned, it is unclear how the hardware Translation Lookaside Buffers (TLBs) in the translation units are being updated.

Our work presents the potential usage models of the main consistency mechanisms, while trying to tackle the following questions:

- What are the implications of multiple applications running on the system (multiple users of accelerators) in maintaining consistency, through linking assets to virtual memory. An example is to attach accelerators to the application and making sure that any accesses to memory from the accelerators are within the application virtual memory space.
- How virtual address consistency is achieved.
- How memory protection is provided through the isolation of memory mastering devices.

Following extensive literature reviews, we found no analysis of the performance or complexity evaluation of implementing the aforementioned. We leverage the latest Xilinx Ultrascale+ FPGA which includes the Arm System Memory Management Unit (SMMU) and we exploit its capabilities to evaluate how it can satisfy the needs of virtualisation. Additionally, other than measuring the time required to set up the SMMU to align the accelerator access rights with the applications address space, we also contribute by creating a testbed on real hardware and measured the overhead of using the memory management subsystem on this generic FPGA platform in various scenarios. The modified CPU agnostic allocator driver and the implemented library that utilizes the driver has been released as a Bitbucket repository [19].

This paper is organised as follows: Section 2 provides a description of the main usage models of the IOMMU, while also discussing some of the implications and challenges faced in various scenarios. Section 3 presents the platform and the methodology used to meet the demands of virtualisation in a widely used SoC-FPGA platform. In Section 4 we present and discuss the results of the experiments, in Section 5 we discuss issues on mapping consistency on heterogeneous systems, current approaches and system considerations and lastly, in Section 6 we conclude with remarks and discuss our next steps.

2 USAGE MODELS OF AN IOMMU

The fundamental purpose of an IOMMU is to allow DMA-capable masters such as accelerators to move or access data within a specified memory space. These devices can access several IOMMUs included in a system design. Their purpose is to provide system-level protection rather than application-level protection. Several IOMMU specifications were published such as Intel Virtualization Technology for Directed I/O (Intel VT-d) [3, 22], AMD-Vi [5] or Arm SMMU [7] that enable device virtualization by controlling any inbound-outbound transactions to memory or system devices. Vendors extend the concept of protection domains in which each I/O device in the system can be assigned to a specific protection domain and a distinct set of page tables, shareable to applications that belong to that domain. Each protection domain can be defined by a set of translation-protection policies. When an I/O device attempts to read or write system memory, the IOMMU intercepts the access, determines the domain to which the device has been assigned, and uses the TLB entries associated with that domain or the I/O page tables associated with that device. That helps to determine whether the access is to be permitted as well as the actual location in system memory that is needed to be accessed. The OS can restrict the device accesses to specific memory addresses by configuring the IOMMU appropriately, thereby protecting the system from errant [15] or malicious devices [11, 36] or even drivers that utilize system devices which are malfunctioned. [14].

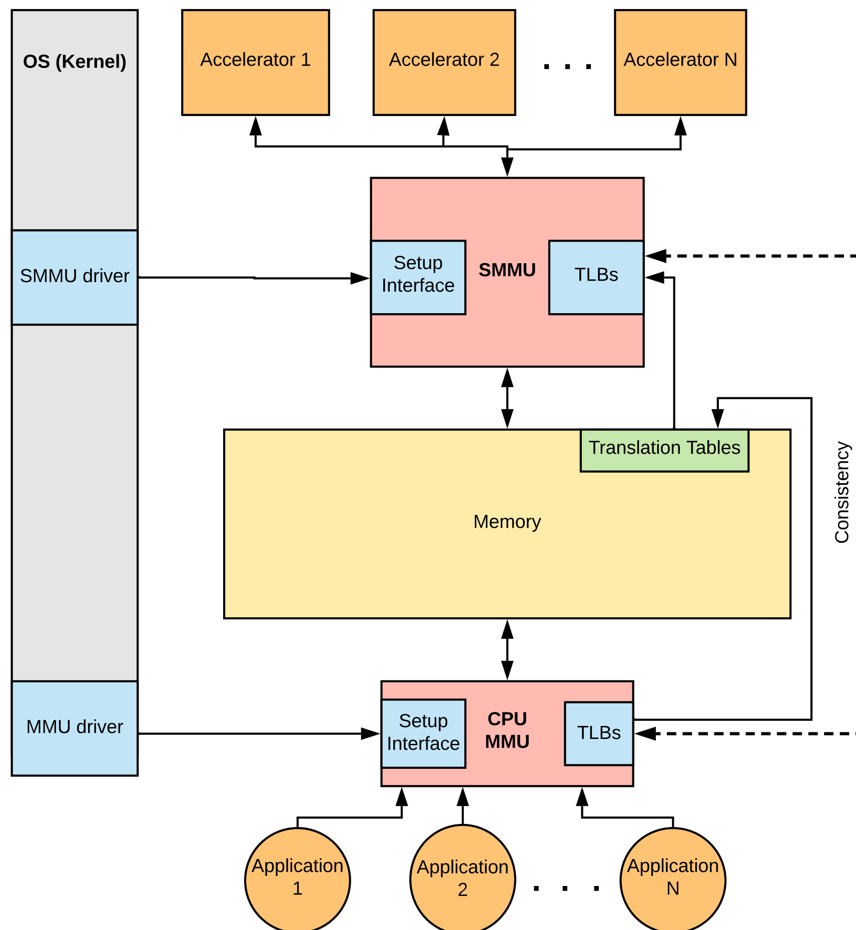


Figure 2: The implementation and utilization of the Arm hardware memory management units (in bold). Applications (on the bottom) are using system calls to utilize the SMMU driver and register accelerators, as well as specialized malloc functions. The kernel utilizes the SMMU driver to 1. bind-unbind accelerators to the application, 2. enable the accelerators to access the applications address space and to maintain consistency between the application PA and VA addresses by pointing the SMMU to the application translation tables. 3. Dynamically associate and de-associate resources on a context switch event. The SMMU extends the system security by blocking any illegal accesses issued to or from the accelerators. The dashed arrow between the MMU on the Arm CPU and the SMMU highlights where the mappings consistency must be present.

Figure 4 depicts the usage model where applications on the local CPU are offloading computations to accelerators while also accessing memory. In response, the DMA capable accelerators may require access to the application memory space to read or modify data. This depicts a usage model for both accelerators and SMMU.

In another usage model, accelerator cards have their own private memory distributed to accelerator devices deployed in the card, as shown in Figure 3. A memory management unit (MMU) can be set up by a hypervisor to allow each accelerator to access a specific portion of the private memory by pointing the SMMU to pinned page tables in memory. When an accelerator is redeployed to another application, or when multiple applications are using multiple accelerators, the memory management unit blocks any intrusions and ensures non-interference between the applications' memory spaces.

This paper discusses three use cases: i) where SDSoC tool, provided by Xilinx, is used to create a project that the application uses DMA to move data between buffers. ii) where I/O page tables are separate page tables used by each I/O devices and managed by each application. iii) where the OS exposes and provides the application page tables used by the MMU on the CPU to the Arm SMMU while also keeping them consistent throughout the application lifespan. This is accomplished through our implemented framework. In each case, the main questions are around the consistency model and how memory virtualization is maintained.

2.1 Maintaining consistency between translation units

Any application access to memory is monitored by the processor MMU. In case of violation, the application is terminated by the OS. On the other hand, the accelerator devices deployed to the application do not natively use the application page tables when they access memory. Without any imposed restrictions, they can access the whole memory space, causing significant security issues in environments hosting many applications e.g. linux, by corrupting system memory or accessing memory of another application. Monitoring of accelerator memory accesses requires additional hardware through SMMUs that stand between the accelerator and the memory system.

Problems of maintaining this consistent view of application memory mappings between MMU and SMMU occur due to the dynamic memory allocation or deallocation throughout the application lifespan. While the processor MMU is automatically updated, the translation unit may not. This can cause significant problems. If the translation unit resources are not updated when any update on the application page table occurs, the consistency is broken, and the accelerator can now access memory that has been recently freed by the application and could have been reallocated to another application. Similarly, the translation unit could be unaware of any new memory allocation to the application, and therefore block any accelerator access to this memory.

Another challenge arises in the event of a context switch when an accelerator device is reassigned to another application. In this case, the software must ensure that the SMMU contains the new applications page tables to enable accesses to the new application's memory space.

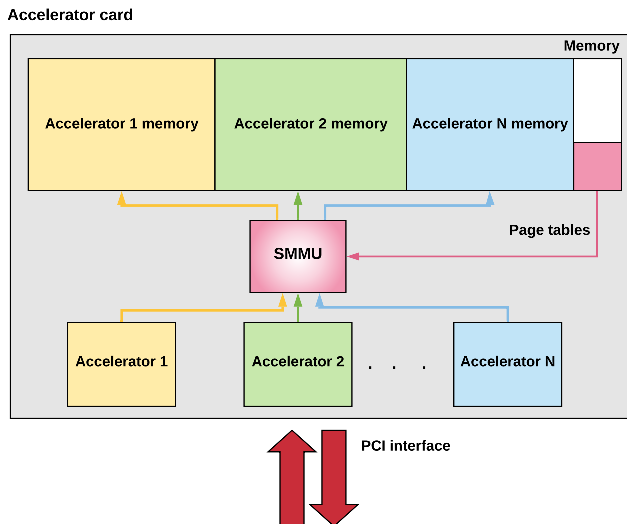


Figure 3: An overview of a SMMU usage model. Deployed accelerators on an FPGA accelerator card need to access local memory resources. The memory can be partitioned to accelerators by loading the page tables to a SMMU

2.1.1 *Coherent page tables.* In a coherence-enabled system, the TLBs are considered coherent with a page table in memory if each mapped entry of the TLB refers to the same place where the corresponding entries in the page table data reside. Multiple TLBs for example, the TLBs of the SMMU and the TLBs in the processor MMU are considered coherent with each other if each TLB has a coherent view of the same page table. UNITD [28] is a unified hardware coherence framework that integrates translation coherence into the existing cache coherence protocol. In UNITD coherence protocols, the TLBs participate in the cache coherence protocol just like the instruction and data caches, without requiring any changes to the existing coherence protocol, but without provision for virtualized systems and also high energy consumption. HATRIC [38] provides translation coherence atop existing hardware cache coherence protocols but it requires modifications to the translation structures such as TLBs and MMU caches. Generally, coherence between translation structures require complex hardware and is also not practical in heterogeneous NUMA systems with frequent page remapping. The Zynq Ultrascale+ does not provide any coherent TLBs or coherent page table walker.

2.1.2 *Software-based consistency.* It is known that the coherence of I/O TLBs with their associated page tables can be maintained through software, a common practice. Software maintenance of coherence consumes significant processor cycles, affecting performance. It is therefore desirable to improve, indeed automate, the maintenance of TLB coherency. I/O page tables can be maintained by the guest OS, but any update triggers a series of actions. In case of any memory-related event, the page table is updated. Due to the lack of coherence between TLBs, the TLBs of the accelerator-attached translation units need to be invalidated. The Arm SMMU contains specific registers that invalidate the TLBs of a selected context bank. That also means that software on the host OS or the hypervisor is responsible for the bookkeeping of the contents on each context bank, invalidating when necessary. For example, in the case when freeing a buffer, or when a context switch occurs and another application uses the accelerators.

2.1.3 *The Distributed Virtual Messaging bus.* Lastly, Arm implements the Distributed Virtual Memory or DVM, as part of the AMBA 4 ACE protocol [30]. DVM messages are generated from a DVM source, such as an Arm CPU, and are broadcast to other compatible DVM destinations for example, to the SMMU. The DVM messages are distributed by a coherent interconnect and/or a DVM network. Responses from the DVM sources or destinations are merged into a single response, which is returned to the sending DVM source. In the Zynq Ultrascale+, the DVM bus can be used to broadcasting TLB maintenance operations to the SMMU. Clearing TLB entries through broadcast messages can improve system performance by freeing-up TLB entries compared to memory-mapped invalidations, through automatic message broadcasts issued by the CPU. Unfortunately, it is unclear how to enable the interface in the Zynq Ultrascale+. For that reason, software managed invalidations through writes to the SMMU registers are used instead.

Table 1 displays the methods for achieving mapping consistency, along with the latency and implementation complexity factors.

	Latency	Complexity
Distributed Virtual Messages	Low	Medium
Software Solution (framework)	High	Small
Coherent page tables	Low	Large

Table 1: A display of the methods for achieving mapping consistency between the units. The factors of functional latency and the implementation complexity are shown.

2.2 Multi-tenancy on FPGAs

In addition to maintaining memory consistency between applications in a virtual memory environment and an accelerator, this issue is further compounded when the machine is further partitioned to support multiple independent tenants, each with their own independent virtualized memory environment.

In such environments, access control has been an important aspect of resources deployed in the cloud system for security and management purposes. It is often addressed with various ring level privileges for stakeholders provided by Virtual Machine Monitors (VMMs) and run-time systems [10, 18, 34]. These hypervisors manage a two-stage memory management translator. This translator provides a two-stage translation that allows the hypervisor to control a view of memory in a VM through intermediate translations.

Recent implementations of FPGA runtime systems have provided multi-tenanted placement of FPGA accelerators but have not provided security mechanisms to protect from rogue hardware elements [8, 23] yet. In [23] Ng et al. provided FPGA memory virtualization via the implementation of an MMU attached to a PCIe bus. This enables memory translation for the FPGA accelerators but without the provisioning of multiple partial reconfigurable FPGA regions. In addition, this approach consumes FPGA resources and increases latency due to low clock speed achieved for the MMU on the FPGA. In [8], memory virtualization and isolation is achieved by using Isolators and ID checkers on the PCIe bus. An improved approach is to use the hardware I/O Memory Management Unit, such as System Memory Management Unit (SMMU) [6], that provides virtualization and protection facilities between system components.

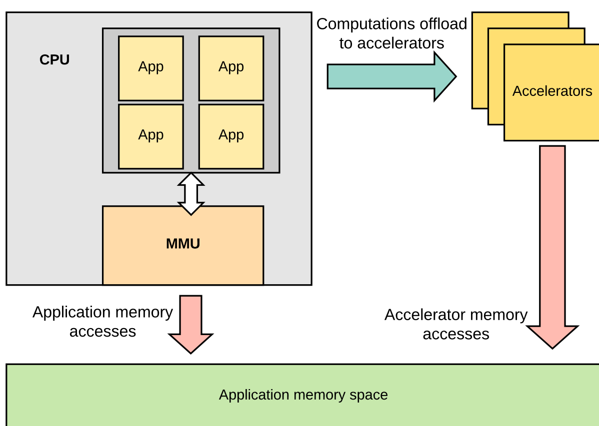


Figure 4: An overview of another accelerator usage model. Applications on local CPU are offloading computations to accelerators and also access data on memory. Accelerators may also need to access or modify the application data

FPGA hardware security for multi-tenanted heterogeneous systems is still an open research area. Integrating FPGAs into multi-tenanted heterogeneous systems along with CPUs and GPUs is opening a large surface of attack which needs to be studied intensively. Attacks are ranging from Denial-of-Service attacks [9, 17], which can bring down FPGA services, to remote side-channel attacks [16, 27, 29, 39], which aim at stealing data from other users in multi-tenanted environments. Malicious users have utilised the ability of FPGA dynamic reconfiguration to launch such attacks. From a practical point of view, using a SMMU protects against unauthorized accesses to shared memory. This concept has been successfully exposed recently in multi-tenant enabled frameworks[25, 33], but without provision for mapping consistency in cases of dynamic changes in application memory mappings.

We expect that the analysis and the conclusions drawn on a single translator can be directly applicable to such two-stage translators, improving memory performance.

3 DESIGN METHODOLOGY

To evaluate the use cases and expose any implications of maintaining mapping consistency in each of the three cases, we selected the Zynq Ultrascale+ FPGA as a modern integrated device, it is widely used from many users, and it incorporates a hard block of the Arm SMMU that can be set to meet the demands of virtualized memory environments. The CPU of the MPSoC runs on 1.2 GHz. The SMMU was designed to use I/O page tables pinned on memory, but through our implemented framework in the third case, we are able to expose the user application page tables used by the local processor, while maintaining the mapping consistency in every case.

3.1 Using the SDSoC

SDSoC Development Environment is a platform from Xilinx [1] which provides easy development of applications for heterogeneous Zynq SoC and MPSoC deployment. SDSoC offers application and system level profiling, automated software acceleration in programmable logic, automated system connectivity generation, and ready-to-use libraries that offer to developers the opportunity to implement SW/HW co-design solutions. For end-users SDSoC is an HDL based on C-Code and uses PRAGMA directives to direct the compilation and synthesis of the hardware kernel or to optimise the function of the data mover operating between the processor and the hardware logic. By specifying the board type and the operating system (bare-metal environments, linux or a FreeRTOS real-time operating system), the user can create a project. All the device drivers are handled by the tool. In addition the communication between the Processing System (PS) side and the Programmable Logic (PL) side, is generated from the development environment.

The SDSoC Development tool uses the notion of data movers in order to move data from the PS to PL. Typically every data mover is defined by the amount of data that is responsible to transfer, the access pattern expectations that the hardware function creates, and the characteristics of the memory being transferred. The users

of the tool can specify the behavior of the data movers by setting PRAGMA in their source code. For instance, there are PRAGMA that specify that the hardware function will access the data from shared memory through an AXI master bus interface. It is obvious that this environment provides an enhanced advantage to the developers and tackles the problem of generating the peripheral components in order to create all the levels of an SW/HW co-design [2].

3.2 Leveraging the Arm System Memory management Unit

The SMMU core is an industrial standard provided by Arm [6] which provides a common view on the memory to all system components and that takes charge of all memory management issues including caching and memory virtualisation.

It is an Arm implementation of an IOMMU [6], a computer hardware unit in which all memory references pass through, performing the translation of virtual memory addresses to physical addresses and providing at the same time memory protection and isolation when configured. If left unconfigured, no checks are performed and the SMMU is essentially bypassed. The SMMU implementation in Zynq Ultrascale+ supports a 48-bit physical address width in various page size granularities. It performs address translation of an incoming AXI address and AXI ID (mapped to context) to an outgoing physical address. It also supports the concept of translation regimes such as two-stage translation which can be set up by hypervisors.

Each client SoC device to the SMMU generates a StreamID which is unique for each client device and may be associated with an SMMU context (the context bank) that contains the configuration of the SMMU on how transactions should be processed. The implementation on the Ultrascale+ supports 16 context banks. Stream matching is used to find the appropriate context for a particular StreamID inside the SMMU. By inserting StreamIDs in different Stream Match Registers (SMR), the dynamic association of SMR registers to different context and lastly the ability of having a different setup for each context are allowing us to achieve several configuration combinations, such as fully isolated contexts or even shared memory regions between client devices. In Ultrascale+, 48 such registers have been implemented. The Arm SMMU driver is responsible for the handling of the SMMU. It can also call functions that create I/O page tables and I/O groups, implemented for the Arm architecture. These groups are the smallest sets of devices which can be considered isolated from the perspective of the IOMMU. Devices within a group can also share the same page tables.

Since all transactions to and from the accelerator devices pass through the SMMU, we ran measurements using the Integrated Logic Analyzer (ILA) to check if the usage of the SMMU induced any overhead. The ILA is a customizable IP core that monitors the signals in a hardware design. We monitored the signals responsible for the initiation and completion of a transfer, and compared them versus a design where the SMMU was set to bypass.

3.3 Utilising the Virtual Function Input-Output (VFIO) framework

The linux kernel includes a Virtual Function Input-Output (VFIO) driver that enables us to virtualise and expose direct device access in userspace. We choose to enable this framework and see how it can satisfy the needs of virtualisation and memory mapping consistency. The VFIO [21] [35] driver is an IOMMU/device agnostic framework for exposing direct device access to userspace in a secure IOMMU protected environment. It allows the modern IOMMUs to drive a device directly from the userspace without any additional specialized kernel drivers. The framework is utilizing the Arm SMMU API, an implementation specific to the Arm architecture. The API enables device region mapping and DMA mapping. Before the user can set up the VFIO to bind a device to a virtual machine or a userspace driver, all the devices that are under the same IOMMU group must be unbound from the host kernel and their respective drivers and bind to the VFIO. A description of the VFIO flow can be found on [21]. It can be considered as an extension of the aforementioned "group" concept of the IOMMUs, by implementing the concept of "containers". Several groups can be added to a single container and share the same configuration or page tables. We used VFIO because of its user friendly interface that makes use of the SMMU driver and it can also satisfy the needs for virtualization on heterogeneous systems.

3.4 The FPGA hardware design

The testbench hardware design deployed on FPGA was implemented in Vivado[37], a software suite by Xilinx for synthesis and analysis of Hardware Description Language (HDL) designs. An overview of the designs is found in Figure 5. We leverage the accelerator IP generated by the SDSoC tool to compare the performance between use cases. The aim is to retain the same application functionality as if using the SDSoC, while measuring the performance (execution time) of the application.

The design contains the generated IP, which is an IP block that does DMA transfers. The accelerator is using I/O virtual addresses that correspond to DMA buffers allocated by the application. The addresses are being translated by the SMMU before reaching the physical address. To measure the impact of using the SMMU in system performance, an ILA core was deployed to measure the latency of the issued transactions by the accelerator.

3.5 A direct approach

In conjunction with the aforementioned hardware, we designed a framework that provides to accelerator devices secure and direct access to the virtual address space of the application. While most of the aforementioned hardware capabilities have been discussed by various researchers and engineers in several online forums, our work integrates them into a usable framework providing a rich and expanding tool that can be used in any potential usage model. The framework includes both kernel drivers and userspace libraries. The kernel driver can be called by the wider application flow to request and associate resources to the application by satisfying any requirements for consistency.

The testbench application utilizes the framework to bind accelerator devices to it and to allocate heap memory, through standard malloc() and free() functions, that is consistent with the accelerator memory view. In our testbench, the application configures the accelerator to do DMA transfers between two buffers on the application virtual address space. To deploy the framework we used Xilinx

Petalinux, which includes a tool to customize, build and deploy embedded linux solutions on Xilinx processing systems. We used the linux image provided by Xilinx and customized it by adding our framework. The following sections describe each framework component and the challenges addressed.

3.5.1 Kernel Driver. The driver layer provides an interface to allocate and associate resources such as memory and accelerator device stream mappings to the application. The driver creates a `mmap` endpoint in the `/dev` file system that is restricted to those accelerators owned by the user. The driver is responsible for:

- Associating new StreamIDs or invalidating existing entries in the SMR registers of the SMMU.
- Configuring the page table pointer of a particular SMMU context to point to the exposed application page table on a similar way to a context switch.
- Setting the appropriate cacheability attributes for the user page table. This is exceptionally useful in cases where cache coherence is needed. By changing the cacheability attributes of a page table, the behaviour of data accesses is changed and any modification on the data can be coherent with the CPU caches.
- Flushing the page table from the CPU cache, due to lack of coherent page table support by the platform. By default, the page table structure is cacheable. This poses a concern for the memory mapping when other translation units try to access it, or when it is modified, as argued below.

During the SoC design process, IOMMU implementations such as the SMMU on the Zynq MPSoC utilize a non-coherent Page Table Walker (PTW). This decision is usually taken to conserve resources on the die as the coherency mechanisms require additional complexity in the cache coherent interconnect. In addition, memory regions accessed via the IOMMU are usually static and long-lived in e.g., kernel allocated ring buffers for devices and also, the memory regions committed to the accelerators are usually non-cacheable and therefore coherency of the PTW is deemed unnecessary. This poses a challenge when providing accelerator access to userspace memory where allocations can be dynamic during the life cycle of the application. To overcome this we provide the userspace with an API for page table management.

3.5.2 Userspace library. The userspace library utilizes the kernel driver and implements the API that is exposed to the user. The user can use the library to bind to the endpoint created by the driver and wraps a number of system calls to provide an abstraction. The purposes of the library are to:

- Create handles that are used to associate the application page table and accelerators to the SMMU.
- Allocate and pin cacheable memory that can be coherently used by both application and accelerators.
- Free and de-associate application memory and accelerators.

The user-level library utilizes the driver to extend the functionality of the malloc() and free() functions so that any modifications to the application memory space are also consistent with the SMMU.

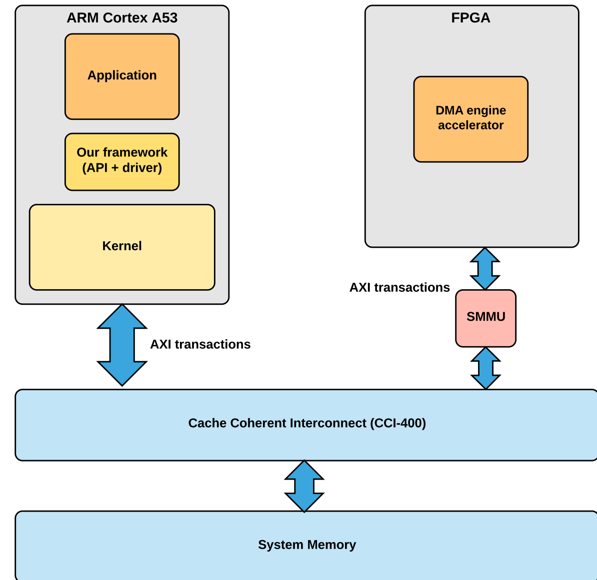


Figure 5: A depiction of the hardware design. An accelerator device (Xilinx Central DMA IP) is deployed on FPGA. The application running on local CPU (A53) is configuring the accelerator device to issue DMA transfers within the application virtual address space. The configured SMMU proceeds with the translation of the virtual addresses into physical. The Cache Coherent Interconnect fetches the requested data directly from the processor cache if cached, or directly from the system memory, and updates the cache accordingly.

4 EVALUATION

The main contribution of this paper is to evaluate the functionality and performance in every design methodology that described in the previous versions. In parallel, we explore how the necessary memory mapping consistency between translation units is achieved by each framework, an also see how multi-tenancy requirements can be satisfied. It is also important to measure any induced overhead when using the SMMU, by monitoring the memory transactions issued by accelerators. We, therefore, evaluate the following use-cases:

1. Device memory allocation using the Xilinx SDSoC development environment.
2. Device memory allocation using the VFIO framework to attach hardware devices to applications and allocate memory. This method creates I/O page tables used by the device.
3. Allocating memory through our implemented framework. In this case, the device can access application memory directly, without any additional buffers, by using the existing application page tables.

4.1 Use-cases

In this section, we present, explore, and quantify the different design methodologies discussed in the previous sections by implementing simple use-cases. We measure the induced latency of binding

devices, allocating memory and setting up the SMMU, to measure different parameters of the design and quantify the user effort of every approach.

The target of this use-case is not to create a computationally intensive hardware accelerator but a simple design that will help us to explore the memory limitations and apply the design methodologies. For our experiments we implemented a simple hardware accelerator which copies an array of integers from one memory location to another. Specifically, on the software side of the testbench application we allocate two buffers of twenty 4K pages each, with data. The hardware side retrieves the buffer and copies the data to another buffer. After this step, the application reads the data written to the destination buffer to ensure that the data is written correctly.

Figure 6 presents a breakdown of the application total running time for each of the three use cases. The measurements are indicative of the overheads of the testbench application that do data copying by using DMA accelerators. Each use case is described in the following paragraphs.

SDSoC Implementation. As mentioned above, SDSoC is the state-of-the-art tool from Xilinx. With this approach, we port the application to the tool's editor and the tool decides on the number of DMAs that will be used, automatically instantiates them, and moves the data from software to hardware. For our design we allocate our buffers with the `sds_alloc` which yields better performance due to the data being allocated and stored in physically contiguous memory. We allocated physically separated contiguous memory spaces for each buffer. The sds compiler creates separate DMAs for each buffer in order to improve the performance. The compiled C code which produces the hardware IP is shown below. For this case, the latency of using the `sds_alloc` function is measured.

```
#pragma SDS data zero_copy
(inputPtr[0:ARRAY_SIZE],
outputPtr[0:ARRAY_SIZE])

void simpleDMA(int inputPtr,int outputPtr)
{
        int buf[ARRAY_SIZE];
        // copy the data from the first buffer
        for (int i=0; i < ARRAY_SIZE; i++)
                buf[i] = inputPtr[i];
        // copy the data to the second buffer
        for(int j=0; j < ARRAY_SIZE; j++)
                outputPtr[j] = buf[j];
}
```

Listing 1: The C code compiled by the sds compiler to generate the hardware design. The pragma `SDS data zero_copy` means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.

VFIO and I/O page tables. In this use case, we used the hardware IP generated by Vivado High Level Synthesis tool and used by the SDSoC, to achieve identical functionality compared to the SDSoC implementation described above. An application was integrated on Petalinux with the same source code as on the SDSoC case. Since setting up the SMMU and the devices required `ioctl()` system calls, we measured the individual latency for setting up the VFIO framework and binding the devices to it. That includes:

(1) The setting up the VFIO framework to be used by the device. This latency occurs only once in the beginning of the application.
(2) The latency of binding accelerator devices to the VFIO driver and to the testbench application.

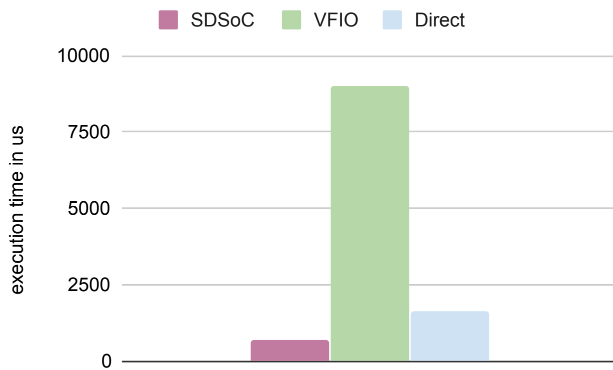
It was also important to measure the latency of mapping and pinning or unmapping memory buffers.

Application native page tables - The direct approach. Lastly, in this case we demonstrate the capabilities of our implemented framework. For results to be consistent, we used the same hardware design as in the previous cases. We modify the application page tables and expose them directly to the SMMU. The difference in this approach is the bypass of software layers, such as the creation of new I/O page tables and the association of these to the SMMU. Instead, the application uses the userspace API to:

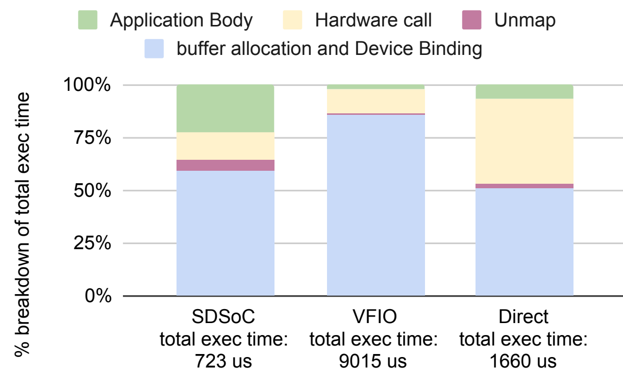
- Create "handles" that contain a pointer to a buffer, the size of the buffer as well as the value of the StreamID. The StreamID is exposed by combining the base address of the FPGA port with the value provided in the Zynq Ultrascale+ Technical Reference Manual. The PS interconnect assigns the master ID bits and transfers these bits on the AxUSER bits of the associated AXI transaction.
- Configuring the page table pointer of the SMMU context that the application is using. Upon completion, the pointer points to the exposed application page table on a similar way to a context switch.
- Setting the appropriate cacheability attributes for the pages on the user page table. This is exceptionally useful in cases where cache coherence is needed. By changing the cacheability attributes of a page table, the behavior of data accesses is changed. For instance, the pages can be marked as *device* memory type, which by default is non cacheable, or *normal* type, that stands for normal cacheable memory.
- The last and most important step is the flushing of the page table from the CPU cache, due to lack of memory coherence support by the platform. By default, in contrast to I/O page tables, the application page table structure is cacheable. This poses a concern for the memory mapping when other translation units try to access it, or when it is altered, breaking the mapping consistency between the translation units.

We measure the latency of using the driver, by putting timestamps before and after using our API, that corresponds to the time required to allocate memory, register the device to the application and unmap the memory.

Lastly, by using the ILA deployed in the design we monitored the signals of a DMA transaction to measure the impact of using the SMMU. Table 2, presents the results of DMA iterations on the same source and destination buffer. It shows that for the very first transaction of a DMA transfer that is consisted of many transactions, the latency of the first transaction is almost five-fold higher than



(a) SDSoC, achieves a 12-fold decrease in total execution time



(b) Breakdown of the total execution time

Figure 6: Total execution time of the three use cases and a % breakdown of the total execution time

	With SMMU (ns)	Without translation (ns)
First transaction	~900	~200
Next transaction	~200	~200

Table 2: The overhead of using the SMMU. In the first iteration of a DMA transfer, DMA read was completed in 90 PL cycles (900 ns total, 10 ns per cycle using a 100 Mhz clock) on average, where all following iterations took 20 cycles (200 ns) to complete. When SMMU is not used, the completion times are identical, but the anomaly of the first iteration is absent.

the rest. This is attributed to the page fault that triggers the SMMU to fetch the page table that the table pointer points to. After the first "cold miss", all the following transactions take the same time as if the SMMU was not configured. In the latter case, all the transactions would again go through the SMMU, without any checking or translation.

5 DISCUSSING THE RESULTS

This section will expose the implications and the consequences of maintaining the mapping consistency across the use cases described above. We will also discuss and present the advantages and disadvantages for each aforementioned use case, providing guidance in optimizing the integration of accelerators in a system.

In Figure 6(a), there is a large differentiation in the total execution time measured in each case. Breaking down this time reveals that:

- The SDSoC provides the best performance. The total execution time is the lowest of all three cases, while still providing memory virtualisation through the SMMU. While the exact mechanism of binding accelerator devices to applications is unclear, the **sds_alloc()** function utilises the Arm drivers to map and pin memory to the application. Also, the SDSoC deploys additional data movers to ensure optimal performance. This explains why the execution time of the application body is lower than the other cases, something verified by the ILA core deployed in the hardware design.
- VFIO consumes a considerable amount of the total execution time in memory and device mapping to the application. Ioctl and mmap system calls are used to map, pin and mark memory buffers as DMA, initiate the VFIO, unbind devices from their associated drivers to bind them to VFIO so that, in turn, be mapped to the user application.
- Our direct approach provides direct access of application's memory space to accelerators, while leveraging the SMMU to prevent memory corruption. The measured execution time of the application body is identical to the VFIO use case, since they share the same hardware, but the total execution time is lower compared to VFIO. This is attributed to the removal of redundant system calls, such as mapping and marking memory as DMA. It is also possible though to meet the low application body execution time of the SDSoC approach, by also using additional hardware such as data movers.

5.1 Achieving mapping consistency

All use cases were able to maintain mapping consistency. The Ultrascale+ platform does not provide a coherent PTW that can track any page table changes, therefore, the utilised SMMU relies on the SMMU driver provided by Arm for any invalidation. The driver can invalidate a range of entries or all the entries inside a TLB. After invalidation, any page fault will trigger a PTW to update the TLB with the missing translation information. This stalls the transaction for 700 nanoseconds, as measured.

For the SDSoC, Xilinx provides a level of abstraction to the user, between the communication of the software and the hardware through data movers. Due to the nature of the SDSoC, it is not trivial to track the system calls when using the sds_alloc, since Xilinx uses the `xlnk` drivers. The memory management, accelerator control and the data movement performed from the SDSoC tool without any end-user involvement. The tool through its compiler interacts with the SMMU driver to set it up, achieving a software mapping consistency.

In our direct approach, the implemented framework is using a custom driver to manage the SMMU. Whenever the application

leverages our API, the driver keeps the mapping consistency by flushing any page table entries out of the cache, and invalidating the TLBs on the SMMU. Instead of mapping DMA memory, a costly procedure, we allocate memory from the heap. This is a much faster approach than both the VFIO and the SDSoC approach, where mmap and file descriptors are used instead. However, flushing the application page tables takes a 97% (1.26 out of the 1.3 milliseconds) of the total time required to allocate buffers and bind the device. Nevertheless, albeit slower than the SDSoC use case, is much faster than if VFIO was used.

5.2 Enabling memory virtualisation

The reason why both SDSoC and VFIO are allocating DMA-mapped memory instead of directly accessing application memory is the SMMU itself. The main reason of using the SMMU is to prevent memory corruption, rather than accelerating access. This satisfies the needs of memory virtualisation, but at a performance cost due to the system calls required. Instead, the direct approach utilises the SMMU to provide direct access of memory to accelerator devices through our API. Additionally, the direct approach provides a good compromise between performance and flexibility. The API can be used by a scheduler to support acceleration binding to multiple applications running on the system, a feature of the VFIO, while removing much of the VFIO induced overhead. We therefore hope that it can be useful on many cases, as argued in Section 6.

6 CONCLUSIONS - FUTURE WORK

Concluding, the SDSoC is a potent tool that can ease the programming effort to generate a optimised system design that meets the specific application need. However, there is no native provision for virtualisation due to the restrictive nature of the drivers provided by Xilinx. It also requires the movement of data between buffers during allocation. On the other hand, VFIO is a more flexible interface that can be used to share a FPGA across Virtual Machines. However, it is much slower since it heavily relies on system calls to make memory available to accelerators as well as to bind devices to applications.

Lastly, the direct approach that we propose is able to merge accelerator memory access with the actual memory space of an application, removing the cost of setting up and tearing down the bindings, while enabling virtualisation by setting up and using the SMMU. The accelerator is able to interact directly or through additional data movers to further optimise the application as done in SDSoC. With the application memory space accessed directly rather than using dedicated buffers we remove the overhead from buffer allocation and memory registration required when using a DMA engine. We also released a git repository [19] of the direct scenario to act as template for the development community. As a future work, it will be interesting to see how our proposed direct method can be overlaid into the VFIO framework. The benefits of VFIO can then be extended to support the direct placement of page mappings, merging the full flexibility of VFIO regarding virtualised memory mapped environments with the benefits of direct memory mapping.

REFERENCES

[1] [n.d.]. *SDSoC Development Environment Guide.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1027-sdsoc-user-guide.pdf

[2] [n.d.]. *SDSoC Environment Optimization Guide.* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug1235-sdsoc-optimization-guide.pdf

[3] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel Virtualization Technology for Directed I/O. *Intel technology journal* 10, 3 (2006).

[4] Amazon. [n.d.]. Amazon EC2 F1 Instances. ([n. d.]). https://aws.amazon.com/ec2/instance-types/f1/

[5] I AMD and O Virtualization. 2007. Technology (IOMMU) Specification. (2007).

[6] ARM. [n.d.]. System Memory Management Units. ([n. d.]). https://developer.arm.com/ip-products/system-ip/system-controllers/system-memory-management-unit

[7] ARM-Holdings. 2013. ARM system memory management unit architecture specification—SMMU architecture version 2.0.

[8] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A Fahmy, and Paolo Ienne. 2017. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *IEEE Access* 5 (2017), 1900–1910.

[9] C. Beckhoff, D. Koch, and J. Torresen. 2010. Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration. In *FPL*. https://doi.org/10.1109/FPL.2010.117

[10] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *FCCM*.

[11] Brian D Carrier and Joe Grand. 2004. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation* 1, 1 (2004), 50–60.

[12] Fabien Chaix, Aggelos Ioannou, Nikolaos Kossifidis, Nikolaos Dimou, Giorgos Ieronymakis, Manolis Marazakis, Vassilis Papaefstathiou, Vassilis Flouris, Mihailis Ligerakis, Georgios Ailamakis, et al. 2019. Implementation and impact of an ultra-compact multi-FPGA board for large system prototyping. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 34–41.

[13] Seonil Choi, Ronald Scrofano, Viktor K. Prasanna, and Ju-Wook Jang. 2003. Energy-efficient Signal Processing Using FPGAs. In *FPGA*. https://doi.org/10.1145/611817.611850

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 73–88.

[15] John Criswell, Nicolas Geoffray, and Vikram S Adve. 2009. Memory Safety for Low-Level Software/Hardware Interactions.. In *USENIX Security Symposium*. 83–100.

[16] Ilias Giechaskiel, Kasper B. Rasmussen, and Ken Eguro. 2018. Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires. In *ASIACCS*.

[17] D. R. E. Gnad, F. Oboril, and M. B. Tahoori. 2017. Voltage Drop-based Fault Attacks on FPGAs using Valid Bitstreams. In *FPL*.

[18] J. Weerasinghe et al. 2016. Network-Attached FPGAs for Data Center Applications. In *FPT*. https://doi.org/10.1109/FPT.2016.7929186

[19] Andrew Attwood Kyriakos Paraskevas. 2018. arm-user-space-page-table-project. https://bitbucket.org/kiriakos1992/arm-user-space-page-table-project/src/master/.

[20] Iakovos Mavroidis, Ioannis Papaefstathiou, Luciano Lavagno, Dimitrios S Nikolopoulos, Dirk Koch, John Goodacre, Ioannis Sourdis, Vassilis Papaefstathiou, Marcello Coppola, and Manuel Palomino. 2016. ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems. In *DATE*.

[21] Antonios Motakis, Alvise Rigo, and Daniel Raho. 2014. Platform device assignment to KVM-on-ARM virtual machines via VFIO. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*. IEEE, 170–177.

[22] Jun Nakajima. 2007. Intel virtualization technology roadmap and VT-d support in Xen. *Intel Open Source Technology Center* (2007).

[23] Ho-Cheung Ng, Yuk-Ming Choi, and Hayden Kwok-Hay So. 2013. Direct Virtual Memory Access from FPGA for High-productivity Heterogeneous Computing. In *FPT*.

[24] Kyriakos Paraskevas, Nikolaos Chrysos, Vassilis Papaefstathiou, Pantelis Xirouchakis, Panagiotis Peristerakis, Michalis Gianniudis, and Manolis Katevenis. 2018. Virtualized Multi-Channel RDMAwith Software-Defined Scheduling. *Procedia Computer Science* 136 (2018), 82–90.

[25] Khoa Dang Pham, Kyriakos Paraskevas, Anuj Vaishnav, Andrew Attwood, Malte Vesper, and Dirk Koch. 2019. ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs. In *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*. VDE, 1–9.

[26] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron

Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA*.

[27] Chethan Ramesh, Shivukumar B. Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel Holcomb, and Russell Tessier. 2018. FPGA Side Channel Attacks without Physical Access. In *FCCM*.

[28] Bogdan F Romanescu, Alvin R Lebeck, Daniel J Sorin, and Anne Bracy. 2010. UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[29] F. Schellenberg, D. R. E. Gnad, A. Moradi, and M. B. Tahoori. 2018. An Inside Job: Remote Power Analysis Attacks on FPGAs. In *DATE*.

[30] Ashley Stevens. 2011. Introduction to AMBA® 4 ACE™ and big. LITTLE™ Processing Technology. *ARM White Paper, CoreLink Intelligent System IP by ARM* (2011).

[31] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2019. Heterogeneous Resource-Elastic Scheduling for CPU+ FPGA Architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 1–6.

[32] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. 2018. Resource Elastic Virtualization for FPGAs using OpenCL. In *FPL*.

[33] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A Modular FPGA Operating System for Dynamic Workloads. *arXiv preprint arXiv:2001.09990* (2020).

[34] W. Wang et al. 2013. pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment. In *CODES+ ISSS*.

[35] Alex Williamson. 2012. VFIO: A user's perspective. In *KVM Forum*.

[36] Rafal Wojtczuk et al. 2008. Subverting the Xen hypervisor. *Black Hat USA* 2008 (2008), 2.

[37] Xilinx. 2014. *UG910 - Vivado Design Suite User Guide*.

[38] Zi Yan, Ján Veselỳ, Guilherme Cox, and Abhishek Bhattacharjee. 2017. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 430–443.

[39] M. Zhao and G. E. Suh. 2018. FPGA-Based Remote Power Side-Channel Attacks. In *SP*.

ACKNOWLEDGMENTS

This work is supported by the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the EuroEXA project (grant agreement 754337).
K. Iordanou is funded by an Arm Ltd. & EPSRC iCASE PhD Scholarship. Prof. Mikel Luján is funded by an Arm/RAEng Research Chair Award and a Royal Society Wolfson Fellowship.