ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs

Conference Paper · September 2019

CITATIONS
6

READS
560

6 authors, including:

Khoa Dang Pham
Advanced Micro Devices
54 PUBLICATIONS **606** CITATIONS
SEE PROFILE

Kyriakos Paraskevas
The University of Manchester
7 PUBLICATIONS **12** CITATIONS
SEE PROFILE

Anuj Vaishnav
Xilinx Inc.
30 PUBLICATIONS **337** CITATIONS
SEE PROFILE

Andrew Attwood
Liverpool John Moores University
24 PUBLICATIONS **228** CITATIONS
SEE PROFILE

ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs

Khoa Dang Pham[a], Kyriakos Paraskevas[a], Anuj Vaishnav[a], Andrew Attwood[b], Malte Vesper[a], and Dirk Koch[a]
[a]Department of Computer Science, The University of Manchester, Manchester, UK
[b]Science and Technology Facilities Council, UK

Abstract

This paper introduces *ZUCL 2.0*, which extends abstraction services for FPGA applications on ARM-FPGA hybrids. The *ZUCL 2.0* management services include 1) FPGA multi-tasking and context-switching based on dynamic reconfiguration and cooperative scheduling, 2) communication abstraction based on the ARM AMBA standard, and 3) memory isolation for privacy and security purposes. Moreover, FPGA applications deployed on *ZUCL 2.0* and the *ZUCL 2.0* kernel itself can be built and maintained independently. This is a crucial feature for higher design productivity and more flexible system updates. Prototypes were implemented for latest Xilinx UltraScale+ ZCU102, UltraZed and Ultra96 platforms to demonstrate the capabilities of *ZUCL 2.0*.

1 Introduction

Field Programmable Gate Array (FPGA) technology is becoming more popular in virtually all computing systems ranging from embedded devices to data centres [1, 2, 3, 4]. This is being driven by the size and performance of new FPGAs, that are targeting emerging workloads such as deep learning, blockchain, and computer vision [5]. Due to widespread availability of High-Level Synthesis [6, 7], designing for FPGAs has become easier than ever before. However, despite the availability of large capacity devices and demand for more complex applications, FPGA management is still rather rudimentary without abstraction layers to underlying resources. This renders FPGA acceleration similar to traditional bare-metal embedded application use cases (see Figure 1). Those applications are able to access the underlying hardware freely but are not able to switch to another application arbitrarily or to start new processes at run-time. For many computing systems, there is a need to adapt quickly to changing workloads and/or to a different number of tasks to execute [3]. These scenarios do not fit traditional methods of operating FPGAs. To achieve high utilisation, individual systems may have a number of active tasks, each operating without any impact or knowledge of each other, which is the core theme of multi-tenanted computing systems.

We believe that multi-tenancy is strongly needed and that this should be provided in fashion analogous as is provided in traditional software operating systems (OSs). This requires tackling the following questions:

- Design Productivity: How to speed up the design, implementation, and deployment of hardware applications?
- Process Management: How to launch and manage multiple hardware tasks transparently? How to load and unload hardware tasks when necessary without affecting the rest of the system? How to dynamically orchestrate hardware tasks for better performance and better utilisation of resources?
- Memory Management: How to manage and isolate the memory accesses of hardware tasks on shared platforms?
- Protection: How to protect data and execution of a specific user from another on a multi-tenanted platform?

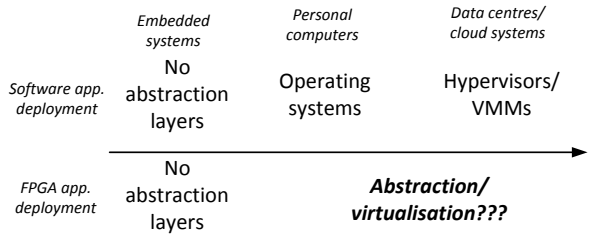

Figure 1 Analogue of software deployment and FPGA deployment roadmaps.

As a solution to this problem, this paper introduces *ZUCL 2.0*[1] — an extension of the ZUCL framework [12] that provides the following features:

1. *Decoupled Implementation*: static systems (often called *shells*) and reconfigurable modules are implemented independently (see Sections 3 and 4).
2. *Multiple Design Languages*: supports most application design techniques such as HLS (OpenCL, C/C++), RTL (Verilog/VHDL), and netlist (see Section 4).
3. *Bus Virtualisation*: supporting different AXI interfaces (Master/Slave/Stream as well as 32/64/128-bit data width) transparently (Section 5).
4. *Run-time Flexibility*: supporting different partial configuration styles: 1) island-style (where a module is placed exclusively in a reconfigurable region); 2) slot-

[1] *ZUCL 2.0* is available at `https://github.com/khoapham/fos.git`.

ISBN 978-3-8007-5045-0

Table 1 Summary of related works on various OS features.

OS features	FPGA version	Related work
Process isolation	Physical isolation	IDF [8], SDF [9], IPRDF [10]
	Dependency decoupling	PCIeHLS [11], ZUCL [12]
Access control	Protection rings	VMMs [13, 14], ReconOS [15], BORPH [16]
Multi-tasking	Run to completion	ReconOS [15], BORPH [16], SDAccel [17], SDSoC [18]
	Relocatable	PCIeHLS [11], ZUCL [12]
Standard interface	Hardware tasks	ReconOS [15], H-threads [19]
	Unix	BORPH [16]
Inter-process communication	Network-on-Chip	Yazdanshenas et al.[20], HopliteRT[21]
I/O virtualisation	High level	BORPH [16], AXI over Ethernet [22]
	Physical	not done
File system	File system access	BORPH [16]

style (where multiple modules can be daisy-chained within one shared region); and 3) 2-dimensional module placement as modules may occupy multiple adjacent regions (see Section 6).

5. *Cooperative Scheduling*: allowing hardware context-switching to adjust resource allocation dynamically and transparently through a user-friendly API (see also Section 6).
6. *Memory Isolation*: enabling isolation of concurrent tasks in multi-tenant environment through memory management layer (Section 7).

2 Background/Related Work

There have been numerous efforts on building operating systems for FPGAs in the past with different aims and approaches as summarised in Table 1.

ReconOS [15] and BORPH [16] tackled the main problem of integrating FPGAs into standard software systems by providing hardware task interfaces to FPGA accelerators and to UNIX process interfaces respectively. This not only addresses the accessibility but also process isolation at software stack level. However, for a hardware developer, the physical interface for communication with the rest of the system is not yet standardised and tailored to a specific system. Current community-wide efforts in shell development address this issue by forcing applications to fixed well-defined interfaces (commonly AXI interfaces including master and slave). However, we believe that this is too restrictive and ZUCL 2.0 therefore allows a developer to choose from a wider range of interfaces entirely transparent.

NoCs (Network-on-Chip) have been proposed several times for implementing an OS Shell infrastructure (e.g., [20]). However, for ZUCL 2.0, we decided to support a plurality of buses (AXI masters or slaves at different bit-sizes as needed) and streaming ports for module-to-module communication [23]. With this, we are better matching an applications needs (e.g., when implementing an accelerator in OpenCL, that doesn't match most NoC protocols) and prevent an over-provisioning of the network which is common for NoC implementations.

State-of-the-art FPGA Shell approaches [20, 33, 13, 30, 31, 36, 37] are suffering in lengthy module implementation overhead. Moreover, as they do not allow users to build modules independently from the shell, a single update or modification of the underlying infrastructure will require the whole physical implementation process to be restarted. This is very different from the task deployment schemes that we know from the software world in which the OS kernels are freely updated without requesting their applications to be rebuilt.

Preemptive scheduling approaches have been proposed for accelerators using configuration read-back [24] and scan-chains [25]. These techniques have also been presented for allowing dynamic reallocation of resources for multi-tasking environments (e.g., [26]). However, these techniques often impose heavy context-switching penalties in terms of time (when using configuration readback [26]) and/or in terms of logic (when using scan chains [27]). Moreover, preemptive hardware execution enforces substantial restrictions on the hardware design. For example, multi-cycle paths, bursts on I/O transactions, DSP-blocks, etc. are commonly not supported for hardware preemption. Therefore, cooperative scheduling has been proposed for minimal context-switching overhead (e.g., [28]). In a cooperative scheduling system, a large compute problem is partitioned into chunks and context switching will only take place after a chunk had been processed (e.g., after a frame in an object classifier is processed).

Hardware resource elasticity is a very distinguished feature of FPGA-based systems which allow a hardware accelerator to change its resource allocation transparently to the task that is using it. When combined with cooperative scheduling, resource elasticity allows a trading of resources for throughput at run-time as proposed in [28, 36].

ZUCL [12] is a development framework for OpenCL applications only which is able to support cooperative scheduling as well as resource elasticity. However, it is lacking some essential levels of abstraction such as memory isolation and system bus abstractions. Ultimately, OpenCL kernels with their default direct memory access (DMA) engines can perform reads and writes to the whole physical memory, which has to be prohibited in multi-tenanted computing systems. Access control has been an

ISBN 978-3-8007-5045-0

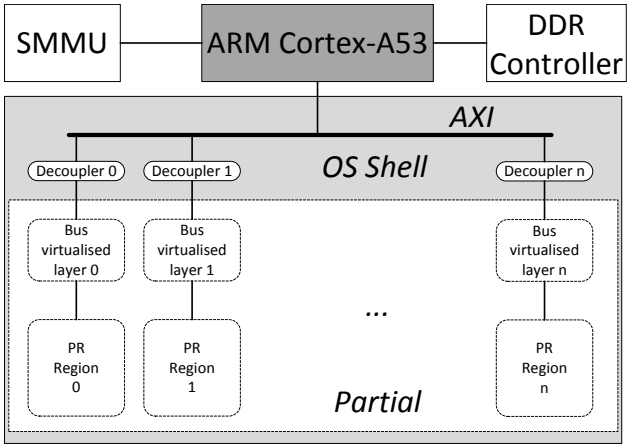

Figure 2 The overall organisation of ZUCL 2.0.

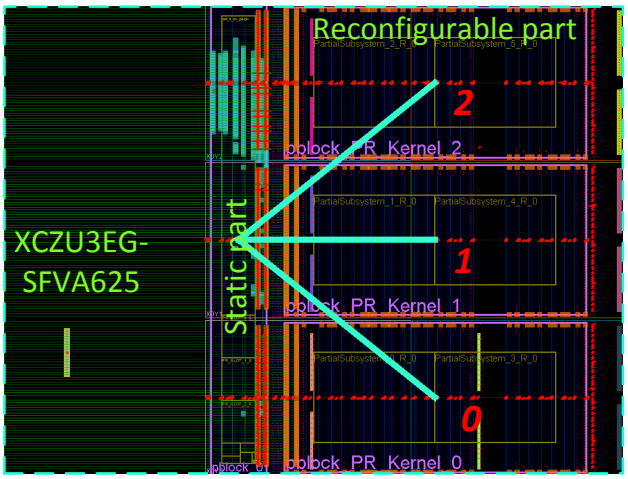

Figure 3 The physical implementation of *ZUCL 2.0* on the UltraZed and Ultra96 platforms. This version has 3 *slots* which can host up to 3 FPGA applications simultaneously.

important aspect for FPGAs that are to be deployed in cloud system for security and management purposes and is often addressed with various ring level privileges for stakeholders provided by Virtual Machine Monitors (VMMs) and run-time systems [13, 30, 31] and ZUCL 2.0 adapts these techniques.

Recent implementations of FPGA run-time systems have provided multi-tenanted placement of FPGA accelerators but have not provided security mechanisms to protect from rogue hardware elements [32, 33] yet. In [32] Ng et al. provided FPGA memory virtualisation via the implementation of an MMU attached to a PCIe bus, thus enabling memory translation for the FPGA accelerators but with no provision for multiple PR regions. In addition, this approach consumes FPGA resources and increases latency due to the low clock speed attainable due to the presence of the MMU on the FPGA. In [33], memory virtualisation and isolation is achieved by using Isolators and ID checkers on the PCIe bus. An improved approach is to use the hardware IO Memory Management Unit (IOMMU), such as System Memory Management Unit (SMMU) [34], that provides virtualisation and protection facilities between the Programmable Logic (PL) and the Processing System (PS) sub-systems, as presented later in this paper for ZUCL 2.0.

From a practical point of view, the two most important issues ZUCL 2.0 is protecting against are 1) accesses to shared memory (through using an SMMU [34]) and 2) ensuring that configurations are protected. For the latter, ZUCL 2.0 ensures that a process can only partially reconfigure resources in its allocated FPGA region without being able to accidentally compromise other modules or the static infrastructure.

3 Architecture Overview

Although ZUCL [12] is supporting the development and deployment of partially reconfigurable systems, ZUCL 2.0 is providing important novel features towards building a complete FPGA OS. ZUCL 2.0 is targeting the latest Xilinx ZYNQ UltraScale+ devices that includes hardened ARM CPUs in the Processing System (PS) part, coupled with a UltraScale+ FPGA fabric in the Programmable Logic (PL) part. Other hardened primitives in ZYNQ UltraScale+ FPGAs are the AXI interfaces, memory controllers, and clock domain crossing which ZUCL 2.0 is utilising to build a light-weight but feature-rich FPGA operating system services.

The FPGA resources are divided into 2 parts: 1) the fixed part is analogous to an OS kernel, called *shell*, which provides all essential infrastructure, and 2) the changeable part, which is analogous to software applications to perform the actual compute work, as illustrated in Figure 2. The fixed part includes the ARM cores, other hardened primitives in the PS, and a small portion of reconfigurable resources. In ZUCL 2.0, the PL provides the AXI system bus and PR decouplers for bus communication enabling/disabling which are forming the FPGA *shell* for providing basic services for deploying partially reconfigurable FPGA applications. The PR decouplers switch off modules from the rest of the system when the FPGA is partially reconfigured in order to prevent possible glitches which may compromise the AXI interconnect. The changeable part is reserved for acceleration and provides about 83% of the available FPGA logic on UltraZed and Ultra96 platforms and contains multiple PR regions, called *slots*, which can host FPGA applications (see Table 4) as well as bus virtualised layers if necessary (see Section 5 for details).

Building partially reconfigurable systems comes with some extra complexity for implementing critical physical constraints, partitioning the system into static and dynamic parts, and floorplanning the FPGA resources. ZUCL 2.0 addresses this by providing compilation scripts for static systems (shells) and the modules. Figure 3 shows an implementation of *ZUCL 2.0* on the UltraZed platform. Like it is common in the software world to use an operating system without changes, our ZUCL 2.0 shells are designed to meet the requirements of most users.

Memory isolation is achieved by utilising the available System Memory Management Unit (SMMU) core which is available in the hardened PS. The SMMU core is an industrial standard developed by ARM [34] which provides a common view on the memory to all system components

ISBN 978-3-8007-5045-0

Table 2 Designs implemented on ZUCL 2.0.

Program	Category
Discrete Cosine Transform*	Signal processing
FIR filter*	Signal processing
Sobel filter*	Image processing
3D Normal Estimation*	Image processing
Sparse Matrix-Vector Mult.*	Machine learning
Histogram*	Data analytics
AES*	Security
Vector addition*	Arithmetic
Matrix Multiplication*	Arithmetic
Mandelbrot set**	Arithmetic
SHA-3***	Security

* in OpenCL; ** in C; *** in RTL.

and that takes charge of all memory management issues including caching and memory virtualisation. In order to use the SMMU, we have built and integrated a kernel driver onto the ZUCL 2.0's run-time execution and management system (see also Section 7).

4 Module Compilation

The ZUCL 2.0 module compilation is carried out by TCL scripts which it adopted from the ZUCL [12] framework. These scripts take care of all the low-level FPGA details of the partial design and the user just has to provide a module specification that meets any of our supported (AXI) interfaces. This also implies that a user of ZUCL 2.0 does not need any extra licenses for partial reconfiguration from FPGA vendors. *In addition, the module implementation scripts are agnostic to specific ZUCL 2.0 shells so that modules can even be used with different ZUCL 2.0 shells.*

Our experiments show that the implementation phase of the module compilation takes less than 6 minutes per 1-slot module on a Windows 7 machine with Intel Core i7-4930K CPU at 3.4GHz, 64GBs RAM and 512GBs SSD. It has been verified with 11 different designs in various domains such as signal processing (DCT and FIR filter), data analytics (Histogram), machine learning (SPMV), security (AES and SHA-3), and arithmetic (VADD, Mandelbrot set, and MM), as illustrated in Figure 4. Note that the SHA-3 source code is in RTL, the Mandelbrot set is written in C, while the remaining are made in OpenCL from the academic Spector benchmark [35], as shown in Table 2.

5 Bus Virtualisation

Operating a hardware accelerator needs communication with the host CPU to issue commands as well as access to memory for the data to process. The original ZUCL framework provides a 32-bit AXI-Lite interface for control register access and another 32-bit AXI4 Master port for memory access. Even though this interface corresponds directly to the default interface of OpenCL kernels when compiling with Vivado HLS, it is neither compatible with wider 64/128-bit AXI4 Master ports nor the other AXI Stream interface standard. Moreover, OpenCL kernels are equipped with DMA engines for fetching data from memory by default, which is not always the case with hand-crafted RTL or customised netlist accelerators. This limits either the system throughput or the deployment flexibility as we need to replace or adjust the current static infrastructure to support another interface.

We have tackled this issue by providing another level of abstraction for bus interfaces between the FPGA applications and the ZUCL 2.0 shell, where the interface of the 32-bit AXI-Lite protocol and the 128-bit AXI4 protocol are fixed. Depending on what exact physical interface will be used by a module, ZUCL 2.0 provides a set of *bus adaptors* that will be instantiated in a module wrapper such that a module can communicate with the rest of the system as required by the given individual FPGA application. An example of this process is illustrated in Figure 5. With this, the ZUCL 2.0 shell can remain light-weight, operational and unchanged while the FPGA applications can be wrapped up with the provided *bus adaptor* at design-time or run-time. ZUCL 2.0 supports up to 128-bit wide datapaths for memory accesses because this is the native width to the ARM SoC.

We also use a *bus adaptor* to translate between AXI Master and AXI Stream protocols. ZUCL 2.0 provides different versions of AXI Stream adaptors to be used depending on the AXI Stream channel width.

A ZUCL 2.0 user can either re-compile their modules with a logical wrapper of the appropriate bus adaptor at design phase or stitch their modules with a pre-built binary of that bus adaptor at run-time. However, because a fixed portion of the slot is allocated to those bus adaptors in all scenarios, the resource overhead is significant, as shown in Table 3. Figure 6 shows the implementation of the bus adaptor with control register, AXI MM2S, and AXI DMA services for the module which has 32-bit AXI-Lite and 32-bit AXI Stream interface.

6 Run-time Management

In the ZUCL 2.0 run-time management layer (shown in Figure 7), the configuration controller is inherited entirely from the ZUCL framework [12] while the hardware task scheduler is extended to support more advanced scheduling policies such as Resource Elastic Scheduling (RES) [28] and Heterogeneous Resource Elastic Scheduling (HRES) [29], along with the existing Round Robin policy. The overhead of the Round Robin scheduler is 1μs while the HRES takes 2.9μs on the UltraZed board with a Quad-core ARM Cortex-A53 CPU at 1.5GHz and 2GB of DDR4 memory. This time needed for taking a scheduling decision is much less than the configuration time which is in range of milliseconds for a slot on ZUCL 2.0 platforms. Moreover, the memory management stack is newly integrated into this ZUCL 2.0 run-time management layer.

7 Memory Isolation/Management

Since the FPGA interface is capable of addressing the entire memory space of a system, simple zero-copy data

ISBN 978-3-8007-5045-0

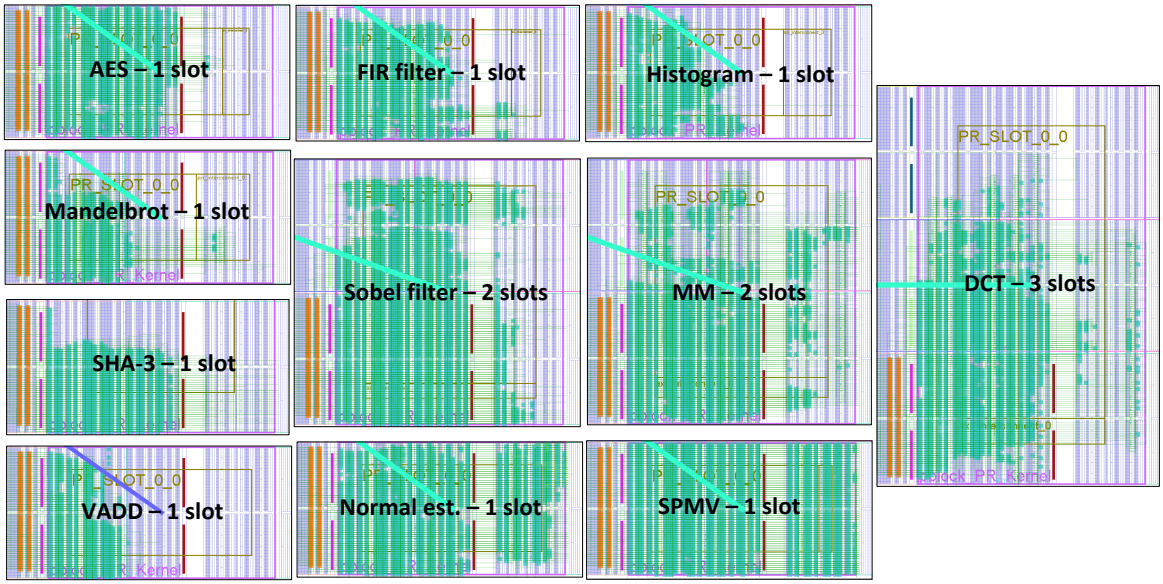

Figure 4 11 designs, which are written various design languages such as RTL, OpenCL, and C, are physically implemented by ZUCL 2.0 compilation flow on UltraZed and Ultra96 boards.

Table 3 Resource overheads for bus virtualisation at the logical and physical levels.

Module interface	Shell Interface	Bus adaptor's services	Resource overhead		
			Primitives	Logical Level	Physical Level
32-bit AXI-Lite and 32-bit AXI4 Master	32-bit AXI-Lite and 128-bit AXI4 Master	AXI Inter-connect	LUTs	153	2400
			FFs	284	4800
			BRAMs	0	12
32-bit AXI -Lite and 32-bit AXI Stream	32-bit AXI-Lite and 128-bit AXI4 Master	Control register, AXI MM2S, and AXI DMA	LUTs	1952	2400
			FFs	2694	4800
			BRAMs	2.5	12

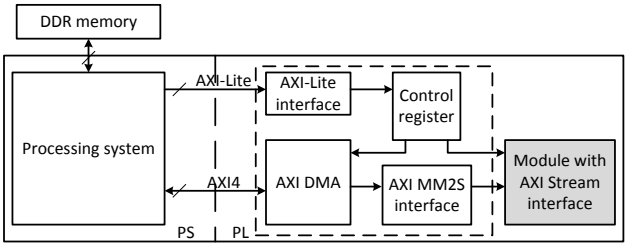

Figure 5 An example for bus virtualisation: the module has a 32-bit AXI-Lite interface and a 32-bit AXI Stream interface without DMA engine. In this case, the *bus adaptor* (see the dashed box) with AXI DMA and AXI MM2S IPs are chosen to carry out the communication with the rest of system.

transfers of control between software and accelerators can remove the induced communication overhead of intermediate software layers. Moreover, in the case where there are multiple applications running on the system, the memory access model should provide memory isolation and being able to guarantee non-interference between applications' separate virtual address spaces, as shown in Figure 8.

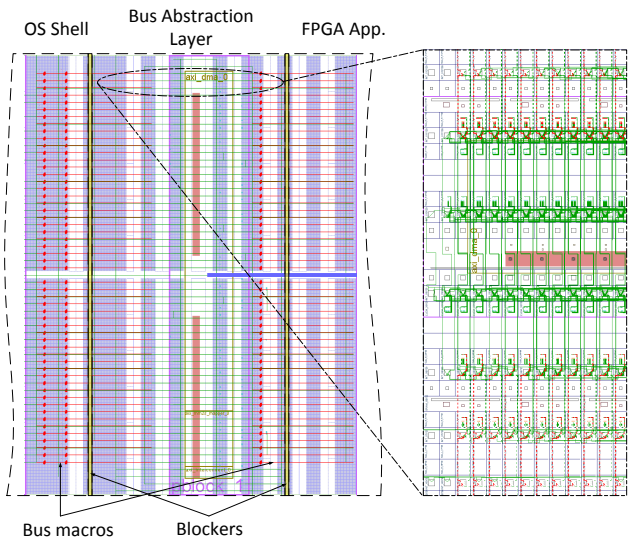

Figure 6 Implementation of a bus abstraction layer on ZUCL 2.0. The bus adaptor is implemented as a module binary and stitched to the system at run-time by partial reconfiguration. The adaptor is a partial module that in turn interfaces to other partial modules. This technique helps users avoid re-compiling of modules at some logic overhead for the bus adaptor.

ISBN 978-3-8007-5045-0

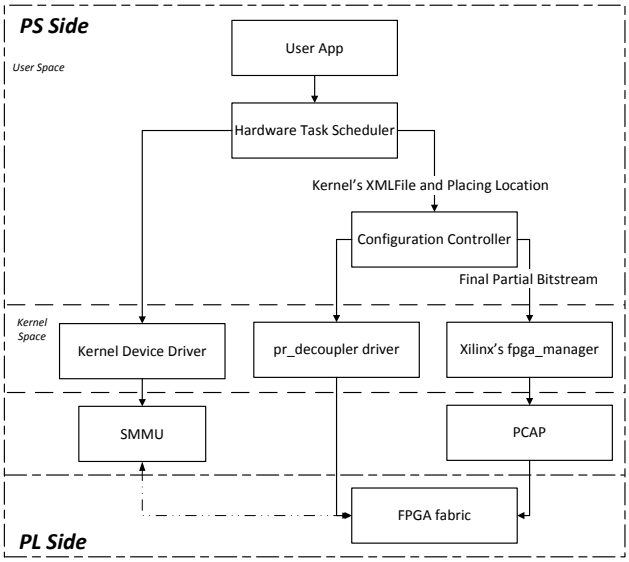

Figure 7 Run-time execution and management. The configuration controller is taken from the ZUCL framework [12] while the hardware task scheduler is featured with more advanced scheduling policies. The memory management stack is newly integrated in the ZUCL 2.0 platforms.

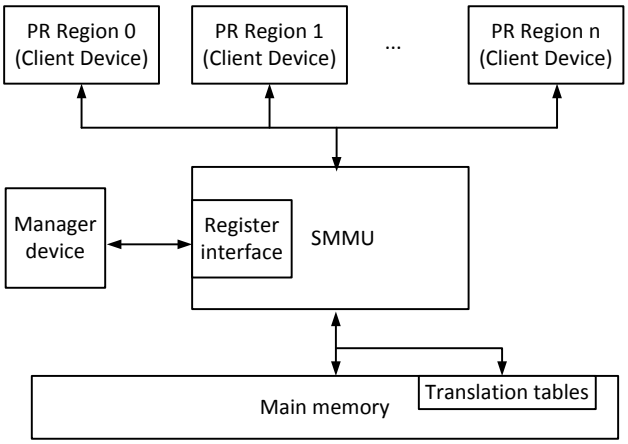

Figure 8 The implementation of the ARM SMMU in the memory system. Client devices in the PR regions are connected through the memory interconnect to the SMMU in the upstream bus. The connection between the SMMU and the rest of the memory system is the downstream bus. When the SMMU is set, the client devices agnostically issue transaction requests to the SMMU. After a successful translation, the SMMU performs the memory access and returns a valid response, or faults otherwise.

7.1 Accelerator Memory Management Requirements

It is essential that the memory management subsystem is flexible, thus allowing a resilient reconfiguration, with the most common case being the allocation or de-allocation of FPGA resources to applications in a multiple memory context approach. One context should not be able to block another context, except where this is desired for the operation of a device. Contexts must also have the ability to be reset, so when devices are reassigned to different applications. In any case, being able to dynamically configure the memory management subsystem to match or even update the actual application user-space with the address space that the accelerator is allowed to access poses a challenge. Failing to do so may lead to performance slowdown such as the case when there are no available translation contexts left; the applications would then contend for limited resources, that may lead to stalling of application execution, until the required resources become available.

7.2 Input-Output Memory Management Unit

The System Memory Management Unit (SMMU) is an ARM implementation of an IOMMU [34], a computer hardware unit in which all memory references pass through, performing the translation of virtual memory addresses to physical addresses while also providing 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 physical address width of 48 bits in various page size granularities.

Each client device generates a Stream ID which is unique for each client device and may be associated with an SMMU context that contains the configuration of the SMMU on how transactions should be processed. Stream matching is used to find the appropriate context for a particular Stream ID inside the SMMU. By inserting Stream IDs in different Stream Match Registers (SMR), the dynamic association of SMR registers to different contexts and consequently 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 on discrete PR regions*. In addition, any local external memory directly connected to the FPGA that is associated to a portion of the total PL address space can be treated as a shared memory region with an appropriate SMMU configuration as described above.

7.3 Memory Management Framework for FPGA

By utilising the aforementioned hardware, we designed a framework that provides secure virtualised access to the user-space virtual address by the application accelerators. The framework consists of two components: the kernel driver and the user-space library. The kernel driver can be called by the wider application flow to allocate memory resources to the application, virtual machine, or container that the accelerators can access. The following sections describe each component and the challenges addressed.

7.3.1 Kernel Driver

The driver layer provides an interface to enable the reconfiguration controller to specify the region and associated stream mappings that have been occupied by the accelerator provided by the user, given the possibility that multiple PR regions are occupied. The driver creates a `mmap` endpoint in the `/dev` file system that is restricted to those accelerators owned by the user and this reference is then

ISBN 978-3-8007-5045-0

passed back to the PR controller to apply the appropriate group ownership settings for user access. The driver is responsible for:

- Associating new Stream IDs 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 userspace application page table.
- Setting the appropriate cacheability attributes for the user page table.
- Flushing the Page Table Entries from the CPU cache, due to lack of memory coherence support.

Usually during the SoC design process, IOMMU implementations such as the SMMU on the ZYNQ MPSoC utilise a non-coherent Page Table Walker (PTW). This decision is usually taken to save resources on the die as the coherency mechanisms require additional complexity in the cache to implement a 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. Additionally, 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 user-space memory where allocations can be dynamic during the life cycle of the application. To overcome this we provide the user-space with an API for page table management, such as flushing.

7.3.2 User-space library

The user-space library binds to the endpoint created by the driver and wraps a number of system calls to provide an abstraction to the user in order to:

- Create handles that are used to associate the user's page table and accelerators with the SMMU.
- Allocate memory and pin it to RAM so it can be used by user accelerators.
- Free and de-associate user-space memory and accelerators.

8 Evaluation

We evaluate the development and deployment overheads of ZUCL 2.0 in this Section. Design trade-offs made for ZUCL 2.0 platforms in terms of FPGA resources are reported in Section 8.1, while the cost to deploy the memory management is analysed in Section 8.2. Finally, the run-time overhead of a deploying application is examined in Section 8.3.

8.1 Analysis of Resource Overhead

Available resources for slots are summarised in Table 4. On the ZCU102 platform, ZUCL 2.0 introduces exactly the same number of resources for partial modules as ZUCL framework as the integration of memory management is taken only in the hardened ARM cores and its software stack. It means that we do not need to compromise FPGA primitives in the PL for the enhanced memory protection.

Table 4 Available resources for 1 slot of the *ZUCL 2.0* versions on the ZCU102 platform and the UltraZed & Ultra96 platforms. The version on ZCU102 has 4 slots in total while the other platforms provide 3 slots in total.

Resources on ZCU102	Number of resources (1 Slot)	Slot utilisation (%)	Total utilisation (%)
CLB LUTs	32640	11.70	46.80
CLB Regs.	65280	11.90	47.60
BRAMs	108	12.10	48.40
DSPs	336	13.30	53.20
Resources on Ultra96 & UltraZed	**Number of resources (1 Slot)**	**Slot utilisation (%)**	**Total utilisation (%)**
CLB LUTs	17760	25.17	75.51
CLB Regs.	35520	25.17	75.51
BRAMs	60	27.78	83.33
DSPs	96	26.67	80

However, as we have less FPGA primitives on UltraZed and Ultra96 boards than on ZCU102 counterpart, we decided to reserve as many of these precious resources as possible for the partial modules while making the ZUCL 2.0 infrastructure much more compact and lightweight.

8.2 Analysis of Memory Management Overhead

The total overhead of the procedure of registering an application to the SMMU by using the driver is measured at 1.3ms, where the driver function of Page Table flushing takes up the significant portion of 1.26ms. Given the fact that page table flushing takes place only once before any translation occurs, we consider that this is outside the critical path of the overhead.

The overhead of using the SMMU was determined in two experiments by measuring the number of clock cycles in the FPGA logic for the read part of a DMA transfer. The source and destination addresses are provided by the application associated with the accelerator (DMA engine) and the FPGA logic clock frequency was set to 100MHz. In the first experiment, the DMA uses virtual addresses and the SMMU is configured to translate, where in the second run, the SMMU was set to bypass and physical DMA addresses were used. The results that are shown in Table 5. The increased latency of the first iteration ("cold miss") can be attributed to the initial page table miss from the TLBs of the SMMU and the subsequent fetching operation by the PTW.

8.3 Analysis of Run-time Overhead

To evaluate the performance improvement and system overhead of ZUCL 2.0, we use a memory bound application rather than compute bound application as the performance of a compute bound application is primarily tied to the logic resources and not to the run-time overhead caused by the shell. To this advent we use the application with

ISBN 978-3-8007-5045-0

Table 5 The overhead of using the SMMU. In the first iteration of a DMA transfer, DMA read was completed in 90 cycles (900 ns) 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.

	With SMMU (ns)	Without translation (ns)
First iteration	∼900	∼200
Next iterations	∼200	∼200

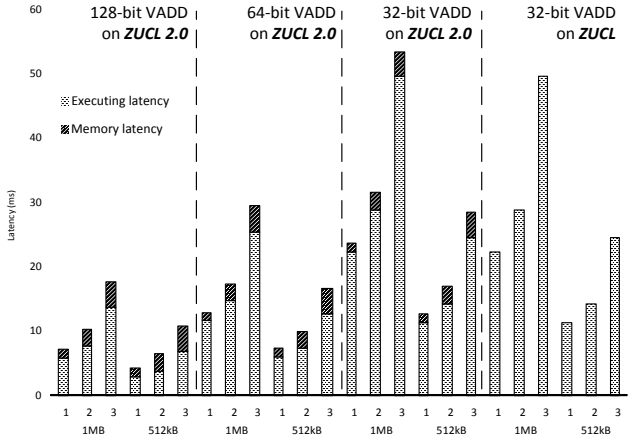

Figure 9 The Vector Addition (VADD) design which has 2 inputs and 1 output (i.e. conducting 2 operand reads and 1 operand write at a time) is being used for this experiment.

the most common access pattern of 2 read operation and 1 write operation behaviour: vector addition (VADD). We compare the results of ZUCL 2.0 with the ZUCL framework [12] to identify and quantify the worst-case overhead caused by the communication interface and the memory virtualisation.

We test the system with 32-bit, 64-bit, and 128-bit bus-width memory bound VADD accelerators, over memory access of 512kB and 1 MB and numbers of users ranging from 1 to 3 which access the memory concurrently of the same application type. Note, the 32-bit bus-width application are deployed on both ZUCL 2.0 and ZUCL while the 64-bit and 128-bit ones are available on ZUCL 2.0 only. Results of these experiments are presented in Figure 9.

The end-to-end time of module operation includes overheads of the hardware scheduler, configuration controller, memory management, and the module execution itself, as summarised in Equation 1:

$$t_{total} = t_{sched} + t_{conf} + t_{mem} + t_{exe} \quad (1)$$

Overhead of the hardware scheduler, t_{sched}, is the time to make its decision of which hardware module is selected to launch. t_{sched} ranges from 1μs in ZUCL to 2.9μs in ZUCL 2.0, as mentioned in Section 6. However, this t_{sched} is marginal for the total overhead.

t_{conf} summarises the configuration time and software overhead to load a hardware module to the FPGA. The throughput of the Processor Configuration Access Port (PCAP) is measured at 256MB/s while the software overhead is measured at 13ms on average. Because these overheads are the same on both ZUCL and ZUCL 2.0 platforms, we omit them in the final comparison.

t_{mem} is the penalty we pay to integrate the memory isolation stack into the run-time management layer. The overhead of registering a new application is measured at 1.3ms while the overhead of using SMMU is 0.9μs for the first iteration is negligible, as mentioned in Section 8.2.

The module execution time, t_{exe}, is measured from fetching the processing data to its completion.

As mentioned above, VADD computation is lightweight but memory intensive. Thus, by expanding the communication bus-width from 32-bit to 128-bit, we are able to boost the performance up to $3.1\times$ compared to the previous ZUCL framework. This noticeable speed-up is achieved even when we have already paid a penalty for the memory isolation, which helps to enhance the security and robustness of a system. The exact benchmarking results are shown in Figure 9.

9 Conclusion

We have introduced *ZUCL 2.0* — Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs, which offers many levels of vital resource abstraction and virtualisation for heterogeneous multi-tenanted reconfigurable computing systems. This is achieved with a combination of hardware static systems (Shells) and a software stack. The ZUCL 2.0 shell provides 1) variable width and protocols with bus communication virtualisation, 2) relocatable and re-adjustable acceleration modules, and 3) decoupled accelerator development from the shell using a custom PR flow. In addition, the software stack provides 1) standard I/O virtualisation with SMMU integration, 2) multi-tasking and 3) dynamic management of FPGA with a cooperative scheduler and a slotted shell architecture. Overall, these key features provide easier, faster, maintainable and secure development for FPGA accelerators in embedded systems and provide a direction for FPGAs virtualization in the cloud.

The complexity of ZUCL 2.0 is embedded in a freely available distribution offering user-friendly interfaces for implementing partial modules and deploying them. With this, ZUCL 2.0 allows an application-centric implementation and execution in the ARM-FPGA heterogeneous computing platforms with ease.

Acknowledgement

This work is kindly supported by the National Cyber Security Centre of the UK through the project *rFAS - reconfigurable FPGA Accelerator Sandboxing* (grant agreement 4212204/RFA 15971) and by the European Commission through the H2020 projects ECOSCALE EuroEXA (grants 671632 754337). We also thank the Xilinx University Program for tools and boards.

10 Literature

[1] S. Choi et al., "Energy-efficient Signal Processing Using FPGAs," in *FPGA*, 2003.

[2] A. Putnam et al., "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *ISCA*, 2014.

[3] I. Mavroidis et al., "ECOSCALE: Reconfigurable Computing and run-time System for Future Exascale Systems," in *DATE*, 2016.

[4] Amazon Web Services, "Amazon EC2 F1 Instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/. Accessed: 11 July 2019.

[5] S. Jin et al., "FPGA Design and Implementation of a Real-Time Stereo Vision System," *IEEE TCSVT*, 2010.

[6] A. Canis et al., "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM TRETS*, 2013.

[7] J. Cong et al., "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE TCAD*, 2011.

[8] Xilinx, "XAPP1222 - Isolation Design Flow for Xilinx 7 Series FPGAs or Zynq-7000 AP SoCs (Vivado Tools)," 2016.

[9] Altera, "AN 567: Quartus II Design Separation Flow," 2009.

[10] K. D. Pham et al., "IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs," in *MCSoC*, 2018.

[11] M. Vesper et al., "PCIeHLS: an OpenCL HLS framework," in *FSP*, 2017.

[12] K. D. Pham et al., "ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications," in *FSP*, 2018.

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

[14] A. K. Jain et al., "Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform," *JSPS*, 2014.

[15] M. Happe et al., "Preemptive Hardware Multitasking in ReconOS," in *ARC*, 2015.

[16] H. K.-H. So and R. Brodersen, "A Unified Hardware/Software run-time Environment for FPGA-based Reconfigurable Computers Using BORPH," *ACM TRETS*, 2008.

[17] Xilinx, "SDAccel Development Environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. Accessed: 11 July 2019.

[18] Xilinx, "SDSoC Development Environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html. Accessed: 11 July 2019.

[19] W. Peck et al., "Hthreads: A Computational Model for Reconfigurable Devices," in *FPL*, 2006, pp. 1–4.

[20] S. Yazdanshenas and V. Betz, "Interconnect Solutions for Virtualized Field-Programmable Gate Arrays," *IEEE Access*, vol. 6, 2018.

[21] S. Wasly et al., "HopliteRT: An efficient FPGA NoC for real-time applications," in *FPT*, 2017.

[22] W. Kamp, "AXI over Ethernet: a Protocol for the Monitoring and Control of FPGA Clusters," in *FPT*, 2017.

[23] ARM Ltd., "AMBA Specifications." [Online]. Available: https://www.arm.com/products/system-ip/amba-specifications. Accessed: 11 July 2019.

[24] M. Happe et al., "Preemptive Hardware Multitasking in ReconOS," in *ARC*, 2015.

[25] A. Bourge et al., "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow," *ACM TRETS*, 2016.

[26] H. Simmler et al., "Multitasking on FPGA coprocessors," in *FPL*, 2000.

[27] D. Koch et al., "Efficient Hardware Checkpointing – Concepts, Overhead Analysis, and Implementation," in *FPGA*, 2007.

[28] A. Vaishnav et al., "Resource Elastic Virtualization for FPGAs using OpenCL," in *FPL*, 2018.

[29] A. Vaishnav et al., "Heterogeneous Resource-Elastic Scheduling for CPU+ FPGA Architectures," in *HEART*, 2019.

[30] J. Weerasinghe et al., "Network-Attached FPGAs for Data Center Applications," in *FPT*, 2016.

[31] S. Byma et al., "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," in *FCCM*, 2014.

[32] H.-C. Ng et al., "Direct Virtual Memory Access from FPGA for High-productivity Heterogeneous Computing," in *FPT*, 2013.

[33] M. Asiatici et al., "Virtualized Execution run-time for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017.

[34] ARM, "System Memory Management Units." [Online]. Available: https://developer.arm.com/ip-products/system-ip/system-controllers/system-memory-management-unit. Accessed: 11 July 2019.

[35] Q. Gautier et al., "Spector: An OpenCL FPGA Benchmark Suite," in *FPT*, 2016.

[36] A. Khawaja et al., "Sharing, Protection, and Compatibility for Reconfigurable Fabric with AMORPHOS," in *OSDI*, 2018.

[37] F. Chen et al., "Enabling FPGAs in the Cloud," in *CF*, 2014.

ISBN 978-3-8007-5045-0