Technische Universität München
Lehrstuhl für Integrierte Systeme

Standalone Disaggregated Reconfigurable Computing Platforms in Cloud Data Centers

Jagath Weerasinghe

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Norbert Hanik
Prüfer der Dissertation:
1. Prof. Dr. sc. techn. Andreas Herkersdorf
2. Prof. Dr. Christian Plessl, Universität Paderborn

Die Dissertation wurde am 25.09.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 12.03.2018 angenommen.

Abstract

A wide range of applications are moving to data centers (DC) to benefit from the flexible and scalable compute infrastructures. Among them, applications that analyze big-data sets are becoming increasingly popular. However, the end of transistor scaling makes it difficult for current and future processor generations to provide sufficient performance and energy efficiency to perform such analyses. This has put a tremendous pressure on the DC computing infrastructures, which struggle to satisfy ever increasing processing demands.

To cope with this situation, and to realize steady performance increases, HW accelerators are making their way into the DCs. In particular, reconfigurable architectures, such as field programmable gate arrays (FPGAs), have proved to provide significant performance gains with high energy efficiency. However, as FPGAs have traditionally been deployed in CPU-centric infrastructures being tightly connected over a PCIe bus, the performance, the scalability, and the energy efficiency gains are limited. FPGA-centric infrastructures have the potential to overcome these limitations, where FPGAs are decoupled from the servers and deployed as standalone disaggregated resources.

First, this thesis shows a dynamic approach to build multi-FPGA fabrics by adding a fabric agent to standalone disaggregated FPGAs. It is also shown that these standalone disaggregated FPGA-based multi-FPGA fabrics outperform multi-CPU fabrics by 11x, 57x, and 39x in terms of network latency, latency variation and throughput. In order to experimentally validate this concept, a distributed text analytics application is ported onto this multi-FPGA fabric, showing that the applications can scale the number of FPGAs independently from the number severs, while improving the latency, latency variation and throughput by 40x, 18x, and 5x, respectively. Finally, it is shown that the proposed standalone disaggregated FPGAs have the potential to scale up to thousands of FPGAs in a standard DC rack by implementing an FPGA board, which outperforms the FPGA density of state-of-the-art CPU-centric approaches by a factor of two.

To my parents and the family.

Acknowledgment

First of all, I would like to thank Prof. Andreas Herkersdorf for giving me the opportunity to pursue a PhD at TU Munich. I greatly appreciate the discussions we had and his advice and guidance in shaping up this thesis. Also, I thank Prof. Christian Plessl for proofreading the thesis.

I am grateful that I had the chance to do this research work at IBM Zurich Research Laboratory (ZRL). I would like to thank Dr. Christoph Hagleitner and Francois Abel for their great support over the course of my PhD. The guidance I received from both of them in the discussions, in turning ideas into actions, and in technical writing was invaluable in making this thesis possible. Christoph introduced me to both Prof. Herkersdorf and Prof. Plessl. Also, he proofread the thesis and gave me feedback.

Raphael Polig is a wonderful colleague, always willing to help you. I would like to thank him for his help during the final phase of my thesis, particularly in terms of applications. I would also like to thank Mitch Gusat, Robert Birke, Francesco Fusco, Animesh Trivedi, Heiner Giefers, Kubilay Atasu, Mitra Purandare, Silvio Dragone, Jan Van, and Beat Weiss for sharing their expertise throughout the last few years. Thank you also to Christopher Ayala and Adrian Schuepbach for being nice office-mates in K042. I also want to thank Charlotte Bolliger and Anee-Marie Cromack for proofreading my publications.

Finally, I want to thank my parents-in-law and brother-in-law for their support from home, my parents and three brothers for their continuous touch, and my wife and the two kids for the hard time they went through during my PhD work.

Jagath Weerasinghe May 2018
IBM Research, Saumerstrasse 4, 8803 Ruschlikon, Switzerland

Contents

List of Tables

List of Figures

List of Acronyms

CPU	Central Processing Unit
CMOS	Complementary Metal Oxide Semiconductor
FPGA	Field Programmable Gate Array
LUT	Look Up Table
FF	Flip Flop
vFPGA	Virtual FPGA
GPU	Graphics Processing Unit
ASIC	Application Specific Integrated Circuit
DSP	Digital Signal Processor
HLS	High Level Synthesis
HPC	High Performance Computing
NSL	Network Service Layer
SDN	Software-Defined Networking
VM	Virtual Machine
VMDIO	Virtual Machine Direct I/O
CT	Container
SDN	Software-Defined Networking
SDF	Standalone Disaggregated FPGA
SDMFF	Software-Defined Multi-FPGA Fabric
MFFA	Multi-FPGA Fabric Agent
FMU	FPGA Management Utility

MFFC	Multi-FPGA Fabric Controller
UIMA	Unstructured Information Management Architecture
CAPI	Coherent-Attached Processor Interface
RDMA	Remote Direct Memory Address
CEE	Converged Enhanced Ethernet
ROCE	RDMA Over Converged Ethernet
IP	Internet Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ARP	Address Resolution Protocol
DHCP	Dynamic Host Configuration Protocol
PCIe	Peripheral Component Interconnect Express
QPI	Quick Path Interconnect
GFLOPS	Giga Floating Point Operations per Second
GMACS	Giga Multiplications and Accumulations per Second
HW	Hardware
SW	Software
RC	Reconfigurable Computing
DC	Data Center
HSDC	Hyperscale Data Center
TDC	Traditional Data Center
CDC	Cloud Data Center
HBM	High Bandwidth Memory

Chapter 1

Introduction

1.1 Motivation

In recent years, the way digital data is generated has been revolutionized by the rapid spread of mobile devices and social media platforms as well as the digitization of data records, such as scientific publications, medical reports and patents. Analyzing such vast amount of data is becoming increasingly important, as it reveals valuable insights for decision makers, such as scientists, marketers, medical practitioners and IT professionals.

Big data platforms analyze heterogeneous datasets by sorting, indexing, ranking or clustering. While the amount of data to be analyzed increases continuously, the acceptable time to produce the desired results is shrinking. Timely analysis of big data has a significant impact on productivity and on improving the user experience. To execute analytics tasks over large datasets, big data platforms have started to scale out [31] using distributed frameworks, such as Hadoop [32], Spark [33], Dryad [34], and UIMA [35], which are spread over a cluster of servers in data centers (DC). However, many distributed applications do not scale well on these infrastructures because of (i) the limited compute power of individual server nodes, (ii) the performance of the network that interconnects them, and (iii) the varying response times.

In scale-out applications, the compute capacity for running large-scale applications is increased by adding more servers to the DC. However, server expansion is usually hindered by the DC's power and cooling capacity [36] [37], as the energy consumption of DCs is increasing at an alarming rate, and energy costs start to exceed equipment costs [38]. This can only be resolved by drastically improving the energy efficiency of scale-out applications [39].

Even if the available compute capacity and power-efficiency is sufficient, inefficient networks hamper the scalability of IO-bound applications. When the number of servers is increased beyond a certain threshold, the training of a distributed deep

neural network becomes slow, as the network overhead starts to dominate [40]. Similarly, the scalability of TeraSort [41] [42] is hindered by the network's throughput performance, particularly in the data-shuffling phase. Distributed applications, such as search, online shopping, social networking, and high-frequency trading are interactive in nature with stringent latency requirements. When these applications run in DCs, a major cause of poor network latency is the SW-based packet processing in the server nodes [43].

In synchronous cluster applications, the performance is often impacted by the variance in processing times across different servers, leading to many servers waiting for the single slowest server to finish a given phase of computation [40]. As the variance in processing times is caused by unpredictable scheduling of CPU and IO resources, addition of virtualization layers makes the predictability even worse as scheduling must be executed in multiple layers [44] [45] [46]. To alleviate these issues, some distributed applications use straggler mitigation methods by cloning the same task multiple times [13], which, however, does not solve the fundamental issue.

To improve energy efficiency, performance, networking throughput, latency, and latency variations, server systems are increasingly relying on heterogeneous compute resources, such as graphics-processing units (GPUs), field programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs).

While special-purpose hardware such as GPUs and ASICs can effectively accelerate DC tasks, the rapidly changing nature of the DC application algorithms can quickly render a dedicated accelerator obsolete. This is especially true in the case of ASICs, where the design, verification and bring-up might take several years while the applications change on monthly basis. Meanwhile, FPGAs deliver the performance advantages of a fixed-purpose accelerator in a low-power, flexible hardware platform that can enable faster time to innovation.

Traditionally, in server systems, these FPGAs have been used as slave devices connected over a PCIe bus. This approach has two major drawbacks: (1) the boost in performance and energy efficiency that an FPGA-based accelerator can provide is weakened by the dominating static power of the server nodes [21]; (2) data center workloads are heterogeneous and run at different scales. Therefore, the scalability and the flexibility of the FPGA infrastructure are vital to meet the dynamic processing demands, as both the under-provisioning and the over-provisioning of FPGA resources affect the performance of the applications and the efficiency of the infrastructure. With PCIe attachment, a large number of FPGAs cannot be assigned to run a workload independently of the number of CPUs, and also those FPGAs cannot be connected on flexible user-defined topologies satisfying application demands.

In summary, the scaling requirements of modern DC applications in terms of compute and communication capacity is exceeding the historical scaling rate of the server performance. Scaling up the DC performance simply by increasing

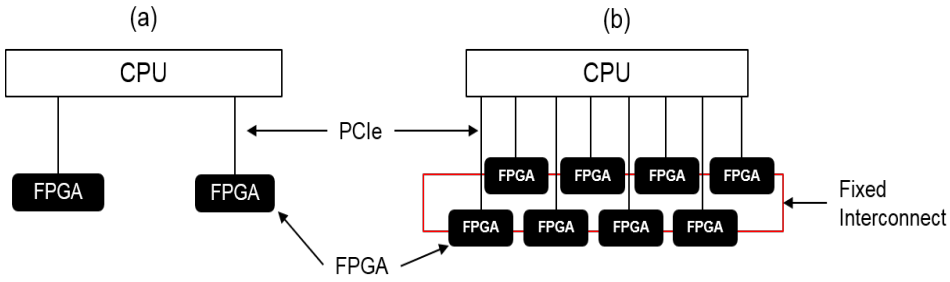


Figure 1.1: CPU-centric FPGA infrastructures with one or more PCIe-attached FPGAs: (a) FPGAs are not directly interconnected (b) FPGAs are directly interconnected in a fixed topology

the number of processor cores is no longer feasible. This creates an increasing mismatch between the application demands and the resources available, calling for novel big data processing technologies. Even though accelerators, such as FPGAs, have brought significant performance increases to the applications, the traditional approaches of using them in server systems have not been able to exploit the full potential of the FPGA resources.

1.2 Thesis Statement

FPGAs have been shown to provide significant performance and energy efficiency gains. Consequently, FPGAs are increasingly being used in DCs to improve the application performance and the overall DC energy efficiency. DCs are typically based on CPU-centric infrastructures. Traditionally, FPGAs are deployed on these CPU-centric infrastructures as slave devices in the form of accelerators. In FPGA-centric applications, most of the application processing is done in FPGAs, and CPUs are used minimally. Hence, the overhead of a CPU-centric FPGA deployment diminishes the performance, the scalability, and energy-efficiency gains for FPGA-centric applications. One approach to solving this issue is to deploy FPGAs in FPGA-centric infrastructures, by bringing the FPGAs to the rank of a standalone disaggregated compute resource. This thesis investigates standalone disaggregated FPGAs with the long-term vision to enable large-scale deployment of FPGAs in DCs for FPGA-centric applications.

1.3 Background

Server disaggregation refers to the separation of server components, such as CPU, memory, and storage into individual resources. These individual resources are allocated and interconnected to build server systems in a modular way. This modular approach allows to independently scale individual resources. When disaggre-

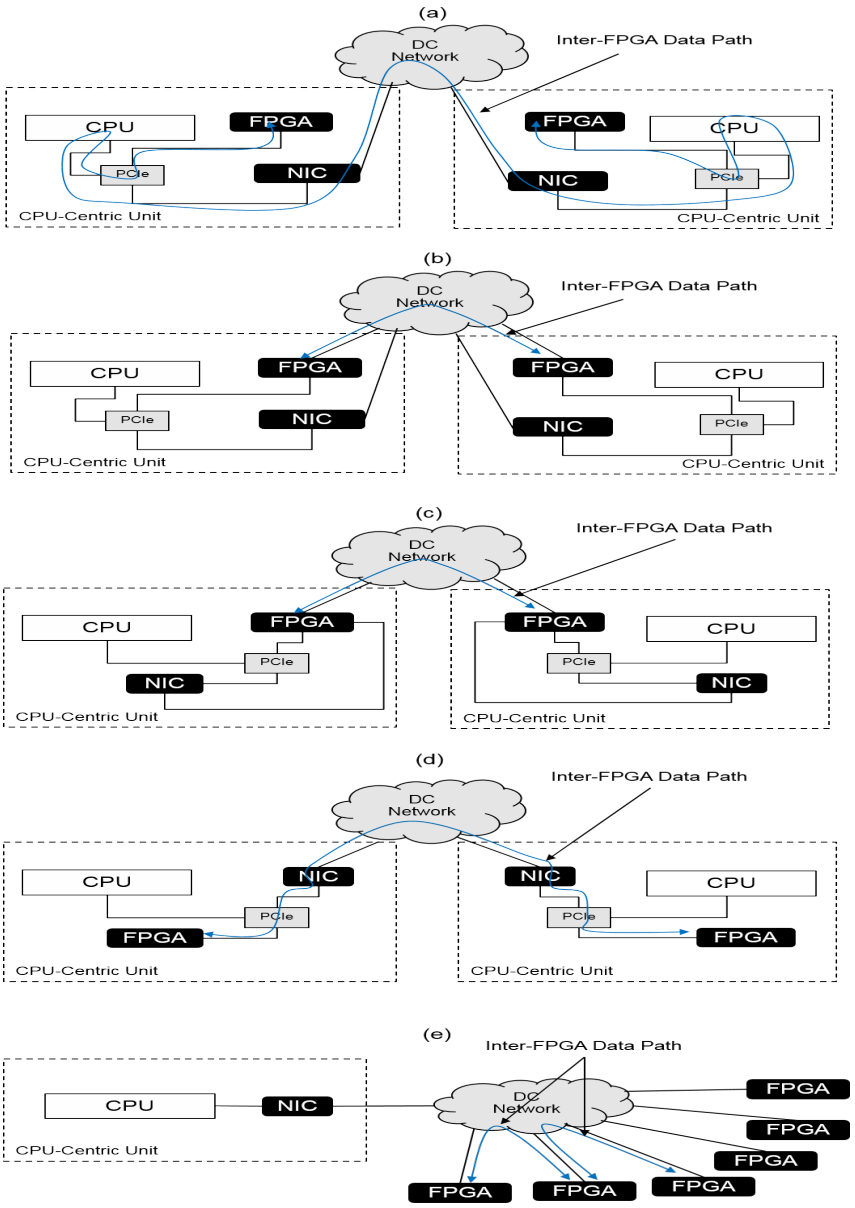


Figure 1.2: Inter-FPGA communication in CPU-centric (a, b, c, d) and FPGA-centric infrastructures (e): (a) Only NIC is connected to the DC network and the inter-FPGA data path traverses the CPU, (b) Both FPGA and NIC are directly connected to the DC network, (c) Only FPGA is directly connected to the DC network [1], (d) Only NIC is directly connected to the DC network and FPGA is connected to the NIC through PCIe, (e) FPGAs are decoupled from the CPUs and directly connected to the DC network as standalone disaggregated resources [2]

gation is applied to FPGA-based compute resources, we refer to it as a standalone disaggregated FPGA. We believe standalone disaggregated FPGAs have the potential to broaden the range of applications for FPGAs in DCs and progress towards the vision of large-scale deployment of FPGAs in cloud DCs.

In the existing solutions [47] [25] [1], FPGAs are deployed in CPU-centric infrastructures where they are tightly coupled to the host CPUs and to each other (Figure 1.1). When offering FPGAs as compute resources to cloud users, existing solutions face the following issues: (i) The CPU-centric approach requires the data path of the inter-FPGA communication to traverse a CPU for applications that use more FPGAs than a single CPU-centric unit can provide (Figure 1.2-(a)). (ii) The number of FPGAs that an application uses cannot be scaled independently from the number of CPU resources at the infrastructure level. (iii) Packaging approach associated with the tight coupling of FPGAs to CPUs limits the number of FPGAs per DC rack.

Above issue (i) can be alleviated by connecting the FPGA to the DC network in a way that does not require the inter-FPGA data path to traverse across the host CPU. There are three options (Figure 1.2-(b), (c), and (d)) in achieving this: First, both the FPGA and NIC are directly connected to the DC network (Figure 1.2-(b)), but this approach doubles the number of physical network connections to a single server. Second, only FPGA is directly connected to the DC network and the NIC is connected to the FPGA through PCIe or directly [1] [48] (Figure 1.2-(c)), in which the server is disconnected from the network when FPGA is undergoing a full reconfiguration. Third, only the NIC is directly connected to the DC network and the FPGA is connected to the NIC through PCIe (Figure 1.2-(d)). This form of communication has already been implemented for FPGA-GPU communication over PCIe [49] [50]. This is theoretically possible also for NICs by using for example SR-IOV [51] based vNICs, but for the best of our knowledge there are no complete system implementations available yet supporting this communication. Even if there are alternative solutions for issue (i), issues (ii) and (iii) cannot be solved with the CPU-centric approach.

FPGA-centric approach [2] (Figure 1.2-(d)) in DCs addresses above issues by decoupling the FPGA from the host CPU and deploying them as standalone disaggregated resources, which are directly connected to the DC network. The work covered in this thesis contributes to the further advancement of the FPGA-centric approach based on standalone disaggregated FPGAs. The main contributions of this thesis are:

1.4 Thesis Contributions

1. **Software-Defined Multi-FPGA Fabrics for Standalone Disaggregated FPGAs**
   In conventional approaches, interconnecting multiple-FPGAs for running

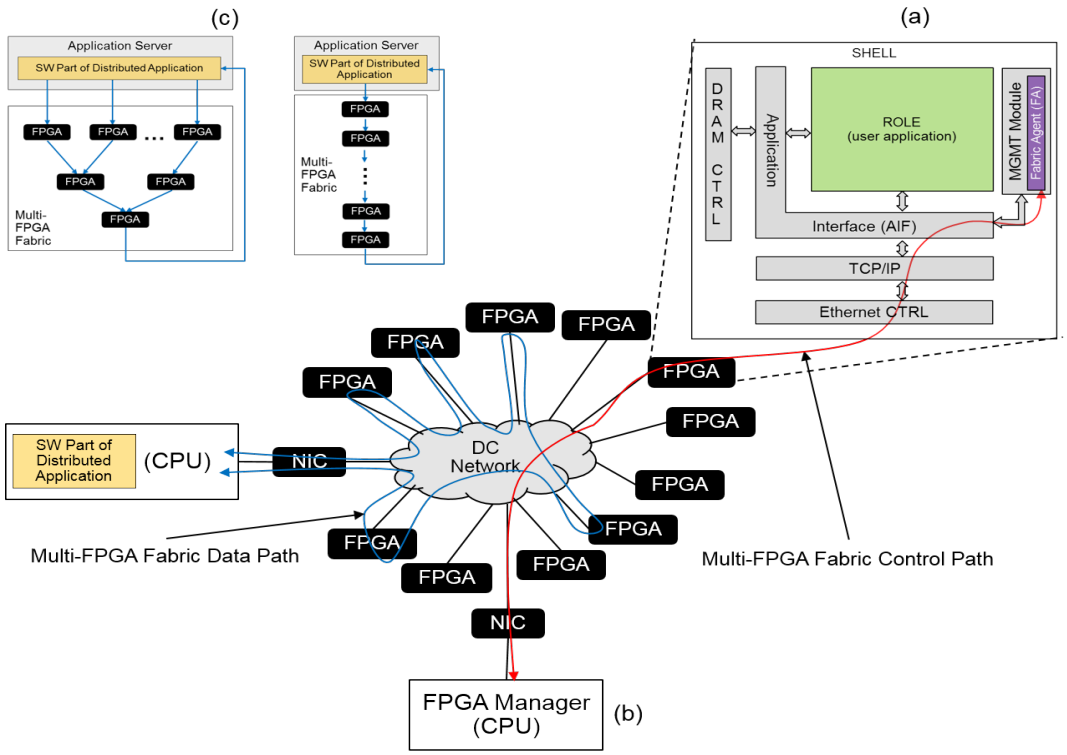


Figure 1.3: Using the shell-based fabric agent and the FPGA manager to build multi-FPGA Fabrics on an FPGA-centric deployment: (a) Shell-role architecture with Fabric agent (FA), (b) FPGA manager, (c) Example multi-FPGA fabrics

distributed applications and changing the inter-FPGA data path on demand requires a new bit stream (full or partial bit stream) to be generated and the FPGA to be reconfigured. New bit stream generation and reconfiguration disrupt the operation of the distributed applications and also introduce a high latency in the control path of the distributed applications. This is not suitable for inherently dynamic cloud environments. To address this issue, a "fabric agent" is introduced to the standalone disaggregated FPGA shell-role architecture (Figure 1.3-(a)). In the shell-role architecture, the SHELL abstracts the FPGA I/O, such as FPGA-CPU and FPGA-DRAM communication, whereas the ROLE hosts the user application. The fabric agent allows to form inter-FPGA connections on demand at application-level, without requiring FPGA reconfiguration. Application awareness in forming inter-FPGA connections allows to dynamically optimize the amount of FPGAs needed for a particular application, such as HTTP load-balancing. A centralized "FPGA manager" (Figure 1.3-(b)) uses the fabric agent in the SHELL to dynamically build multi-FPGA fabrics by interconnecting multiple FPGAs over the DC network in software-defined manner (Figure 1.3-(c)).

The feasibility of this approach is validated through a HW prototype in a 10 GbE-based DC network [52] [53]. It is demonstrated that the fabric agent in combination with the FPGA manager allows the data path of distributed applications to be changed dynamically without generating a new bit stream and reconfiguring it to the FPGA, which reduces the multi-FPGA fabric control path latency down to a sub-millisecond range from tens of minutes. To form a multi-FPGA fabric with 2 FPGAs, the software-defined approach took 0.754 ms, whereas the conventional approach of bit-stream generation and reconfiguration took 29 minutes in our development platform [54]. Out of 29 minutes reconfiguration over DC network took 9 seconds. Even when compared only with reconfiguration time of conventional approaches, the software-defined approach is 12x better in control path latency. We believe that this flexible and scalable approach will allow to seamlessly scale FPGA applications to the size of cloud DCs with thousands of compute nodes. Further, standalone disaggregated FPGA based multi-FPGA fabrics improve the data path performance in terms of latency, latency variation, and throughput by 11x, 57x, and 39x, respectively, compared with multi-CPU compute fabrics interconnected over the DC network [52].

2. **Application Integration and Evaluation**
   Application1: RESTful (Representational State Transfer) web service, which runs on top of HTTP, is an approach to provide interoperability between computer platforms and programming languages on the Internet. In today's cloud environments, many applications can be accessed via RESTful APIs. A RESTful web service (HTTP and REST layer) was ported on to a standalone disaggregated FPGA, and a natural language processing application was demonstrated as the web service application. The performance was compared with a pure SW implementation and a SW implementation accelerated with PCIe-attached FPGAs. Standalone disaggregated FPGA outperformed both of those implementations by 308x and 20x in terms of application throughput, and by 175x and 4x in terms of latency [54].

   Application2: UIMA (Unstructured Information Management Architecture) is a popular distributed computing framework for applications, such as text analytics. The UIMA framework was modified to integrate the standalone disaggregated FPGAs with their software-defined multi-FPGA fabric. Next, a real-world distributed text-analytics application was ported onto this multi-FPGA-fabric-enhanced UIMA framework with 2 standalone disaggregated FPGAs. The results are compared to a pure CPU-based implementation and a CPU-based implementation accelerated with PCIe-attached FPGAs. Multi-FPGA-fabric-enhanced UIMA outperforms both of those implementations by 40x, 18x, and 5x in terms of application latency, latency variation and throughput [53].

   In the above applications, both the network and application processing are

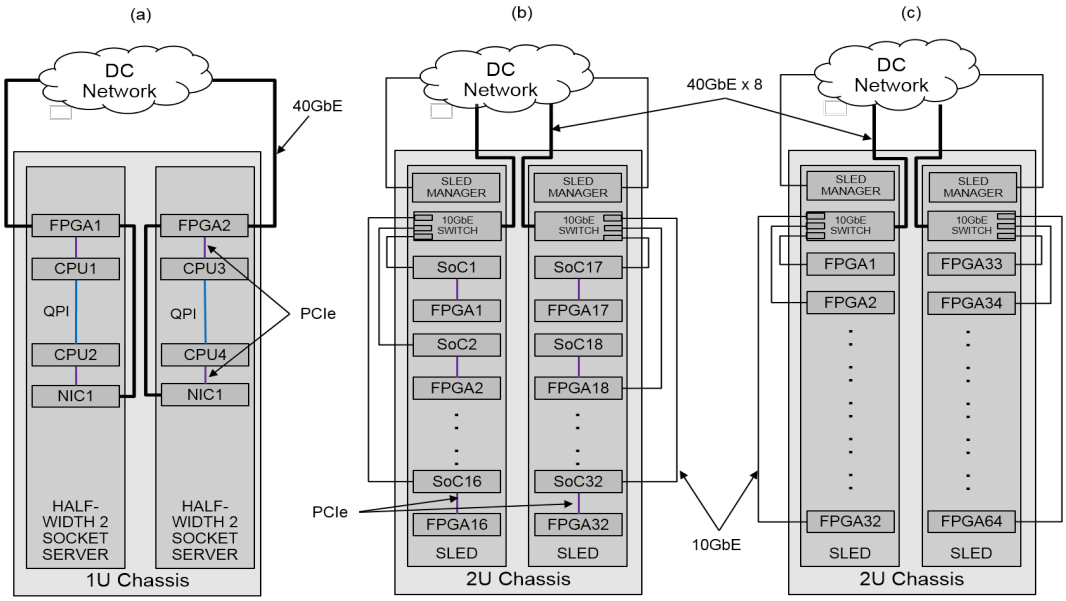


Figure 1.4: DC FPGA Deployment Architectures: (a) Traditional CPU-Centric approach by Microsoft Catapult V2 [1], (b) Traditional CPU-Centric approach on DOME μServer [3] infrastructure, (c) FPGA-Centric approach on DOME μServer infrastructure based on Standalone Disaggregated FPGAs

entirely executed in FPGAs, which eliminates most of the CPU involvement in the distributed application processing.

3. **Densely-packaged Rack-scale Architecture for Standalone Disaggregated FPGAs**
   The FPGA infrastructure density (number of FPGAs/Rack) of state-of-the-art CPU-centric deployments [1] [25] is currently limited to approximately 100 FPGAs per rack. In the state-of-the-art Microsoft Catapult V2 implementation [1], one FPGA is attached to a CPU in a half-width two socket server scaling up to 96 FPGAs per rack (Figure 1.4-(a)). Using the same CPU-centric approach, the DOME μServer [3] based DC infrastructure increases the FPGA density by 5x compared to the Microsoft Catapult V2 implementation. DOME μServer infrastructure achieves this by deploying SoCs and Networking (10 GbE Switch) on small-form-factor motherboards (140×62 mm) in a water-cooled environment, where one FPGA is attached to each SoC over PCIe. In this infrastructure (Figure 1.4-(b)), two sleds (1 SLED = half-width chassis unit) each consisting of 32 boards, a 10 GbE switch, and an SLED manager are deployed on a 2U (1U = 44.45 mm) rack chassis. We implemented an FPGA board [23] based on the concept of standalone disaggregated FPGAs and the shell-role-architecture that increases the infrastructure density of the FPGA deployments by further 2x (Figure 1.4-(c)), achieving

overall 10x density compared to the state-of-the-art deployments in cloud DCs. This approach allows to deploy up to 1024 FPGAs in a standard 42U data center rack at any granularity [2] [23].

1.5 Thesis Outline

This thesis is organized as follows:

Chapter 2: State of the Art
First, this chapter discusses the emergence of heterogeneous computing. Next, it elaborates on the role of FPGAs in heterogeneous computing. Finally, the state of the art FPGA deployments are reviewed focusing on the architectural aspects and applications.

Chapter 3: System Architecture
First, the system requirements for deploying FPGAs in DCs are analyzed. Then, an FPGA-centric system architecture is proposed based on standalone disaggregated FPGAs. Finally, the proposed system architecture is compared with the state-of-the-art FPGA deployments in terms of FPGA infrastructure density (number of FPGAs / DC rack).

Chapter 4: Standalone Disaggregated FPGA.
This chapter elaborates on the standalone disaggregated FPGA architecture and shows its implementation in a commercial FPGA. Next, it shows the performance comparison of the standalone disaggregated FPGA based multi-FPGA fabrics with those based on state-of-the-art CPU-based approaches in terms of latency, latency variation, and throughput.

Chapter 5: Software-Defined Multi-FPGA fabrics (SDMFF).
This chapter introduces the concept of software-defined multi-FPGA fabrics and explains its SW and HW framework. Next, it explains how SDMFFs are built using standalone disaggregated FPGAs over the data center network. Finally, the control path latency in building SDMFF is compared with the traditional approach of building multi-FPGA fabrics by FPGA reconfiguration.

Chapter 6: Experimental Validation by Applications.
As the first application, a RESTful web service application ported on to a standalone disaggregated FPGA is demonstrated. As the second application, a text analytics application is demonstrated on SDMFF-enhanced UIMA distributed computing framework. The results of both the applications are compared with a pure CPU implementation and a CPU implementation accelerated with PCIe-attached FPGAs.

Chapter 7: Conclusion.
Concludes this thesis and gives an outlook for future work.

Chapter 2

Background and State of the Art

This chapter is organized as follows: Section 2.1 provides an overview of emergence of heterogeneous computing and Section 2.2 elaborates particularly on FPGAs. CPU-FPGA attachment options and state-of-the-art FPGA deployment architectures are reviewed in Section 2.3. The chapter is summarized in Section 2.4.

2.1 Emergence of Heterogeneous Computing

Every data processing platform mainly contains 3 subsystems: (i) logic circuitry, (ii) compute architecture, and (iii) applications. Over the past few decades, the logic circuitry is mainly built using CMOS-based semiconductor technology. When main-stream computing is considered, the compute architecture in those platforms are driven by the general-purpose CPUs. The applications which run on the computer architecture are the general-purpose SW tightly coupled with the underlying CPU technology. Rapid trends in the application subsystem have been demanding ever more performance increments and have put a tremendous pressure on the underlying two subsystems. This pressure has led to the technological innovations in those two subsystems, which drove the success of the computing industry over the past few decades. However, the new trends in the application sub-system, such as big data, have increased the performance requirements to an unprecedented level. Meanwhile, further improvements from the underlying two subsystems (logic circuitry and compute architecture) are becoming hard to achieve due to the technological scaling limits. In the next subsection, we review how the limitations of technological scaling have affected the semiconductor technology.

2.1.1 Technology Scaling

The semiconductor industry has been driven by two scaling laws: (i) Moore's Law and (ii) Dennard scaling. It is these two scaling trends that have resulted in the popularity of CMOS technology and subsequent advances in computing technology over the past few decades. Moore's law states that the number of transistors that

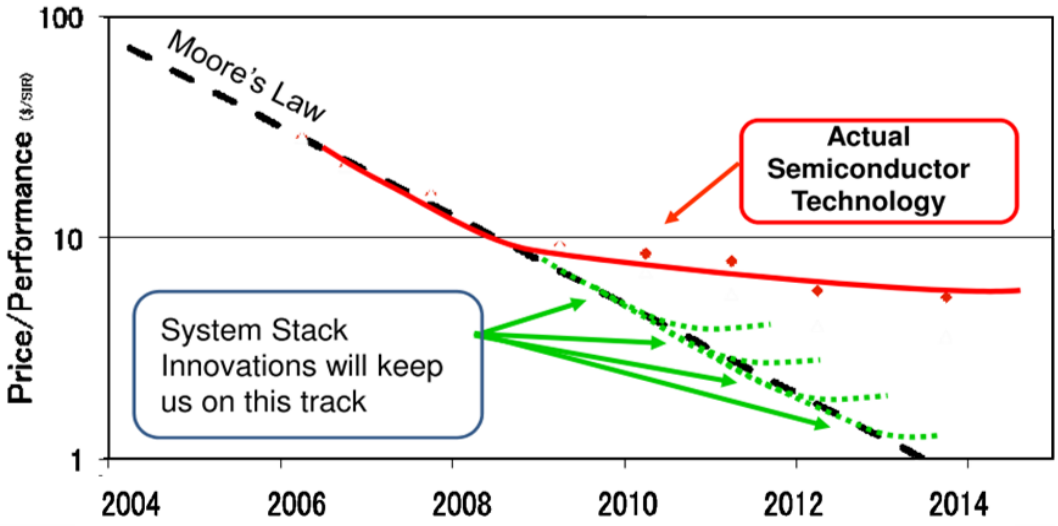


Figure 2.1: Anti-Moore's Law Behavior and System Stack Innovations [4]

can be placed onto an integrated circuit doubles every year. This law is a scaling prediction, which is purely an observation of the progress of the technology rather than a scientific finding. Although moore's law states about the need for making transistors smaller to increase the chip density, only dennard scaling explains how the transistors can be made smaller. Both moore's law and dennard scaling continued to work until the last decade, however now both have started to cease as: (i) doubling transistors increases the power density and (ii) scaling transistor size is no longer going to be possible because of the CMOS physical limitations, such as leakage current. This leads to innovation in two directions: (i) alternatives to CMOS technology and (ii) innovations in system stack (Figure 2.1). Although not much credible success has yet been achieved in the first approach, the second approach has resulted in improved extensions. These extensions evolved in multiple paths: (i) low-level architectural extensions such as multi-core, SMP, SIMD, MIMD, SoC in the processor road map, (ii) advanced memories, (iii) improved system software integration and cloud in the application space, and (iv) workload acceleration. However, the stringent physical limitations of CMOS technology inhibit exploiting increasingly more performance out from the CPUs except for the case of workload accelerators that are built using customized HW.

2.1.2 HW Acceleration

The customized architectures deliver order of magnitude higher performance and energy efficiency benefits at a lower cost compared to general-purpose architectures. Figure 2.2 shows the energy efficiency comparison of general-purpose CPUs, DSPs and ASICs for DSP applications [5]. Compared to general-purpose microprocessors, DSPs deliver up to 100x more energy efficiency, while dedicated

application-specific HW are 1000x more energy efficient. The authors collect the data from 20 different chips across heterogeneous architectures, which were published at the International Solid-State Circuits Conference (ISSCC) between 1998 and 2002.

The mobile computing industry were the first to embrace the HW accelerators on specialized architectures, mainly due to the stringent constraints in mobile environments in terms of power and area [20]. In the recent past, we observe an increasing number of mainstream computing domains start to use HW accelerators. In the domain of mainstream computing, the HPC community were the first to embrace HW accelerators and at the time being the cloud data center community is also slowly progressing towards HW accelerators [55] [1] [56] [47] [57] [58] [59]. With the advent of HW accelerators, the traditional data processing platform has changed its original shape and become heterogeneous in the computer architecture sub system.

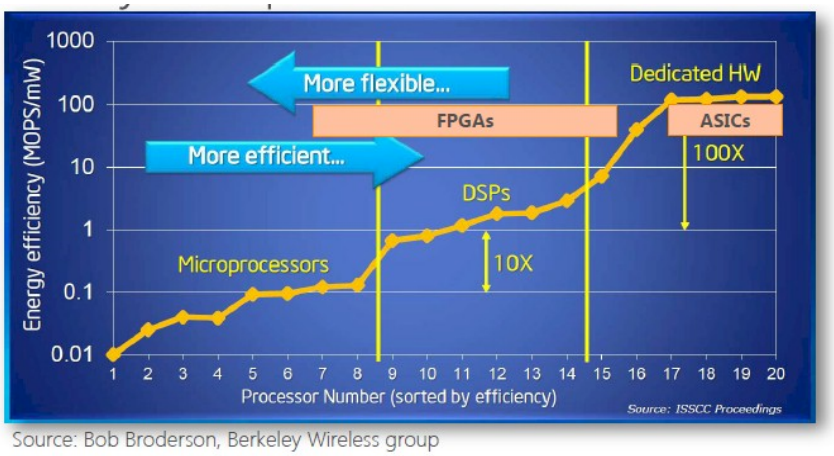


Figure 2.2: Specialization for Performance Improvement [5]

2.1.3 Specialized HW

2.1.3.1 GPU

GPUs have taken the multi-core trend of the CPUs to the extreme and the HW is semi-specialized by adding thousands of smaller cores (Figure 2.3-(b)), which perform arithmetic operations in parallel, onto a single device. At the time of this writing, state-of-the-art Tesla P100 GPUs from NVIDIA contains 3584 cores [60]. These large number of cores collectively offer performance of 5.3 and 10.6 TFLOPS, respectively for double precision and single precision. GPUs have been commonly used in graphics add-in cards and have been specifically popular with the gaming industry. However, over the recent years, GPUs were adopted for accelerating computing because of large quantities of cores and the ability to massively paral-

lelize operations. The use of GPUs for accelerated computing has been explored for many applications, such as deep learning [61] [62] [63], image processing [64] and scientific computing [65] [66]. One of the major drawback of GPUs compared to CPUs is their high power consumption. Tesla P100 GPU consumes around 300 W [60], whereas a state-of-the-art general purpose CPUs from Intel consumes around 165 W [67].

2.1.3.2 FPGA

FPGA architecture is vastly different from GPUs and CPUs, which is based on reconfigurable logic blocks (Figure 2.3-(c)). The flexibility offered by the FPGAs at the HW level enables the designer to realize almost any computer configuration that can be imagined and use any form of parallelism. As a result, the end users who owns the applications have the opportunity to make efficient application-specific machines by customizing the HW according to the application requirements [68]. Today, FPGAs are used heavily in communications, but are making ways into the general-purpose computing market. The most significant advantage that FPGAs have over CPUs and GPUs is computing efficiency. In many applications, higher efficiency means faster operations that consume less energy.

2.1.3.3 ASIC

An ASIC is extremely similar in function to an FPGA. However, ASICs do not have the ability to be reprogrammed and the HW is fully-specialized. The program is burnt (wired) directly onto a piece of silicon and is packaged into a chip (Figure 2.3-(d)). Because of the startup costs, ASICs make the most sense in applications where large volumes or large performance increases are needed. Most importantly, FPGA designs can be ported to ASIC designs for increasing computing efficiency. Although ASICs are not suitable for frequently changing general-purpose DCs, application specific DCs can afford to build ASIC clouds [69] to optimize the overall efficiency.

2.1.3.4 FPGA vs CPU

The CPUs are based on temporal computing paradigm, where in a software application, the program source code is transformed into a list of instructions for a particular processor, which is then loaded into the memory attached to the processor. Data and instructions are read from memory into the processor core, where operations are performed, and the results are written back to memory. Modern processors contain many levels of caching, forwarding and prediction logic to improve the efficiency of this paradigm; however, the model is inherently sequential with performance limited by the latency of data movement in this loop (Figure 2.4-(a)).

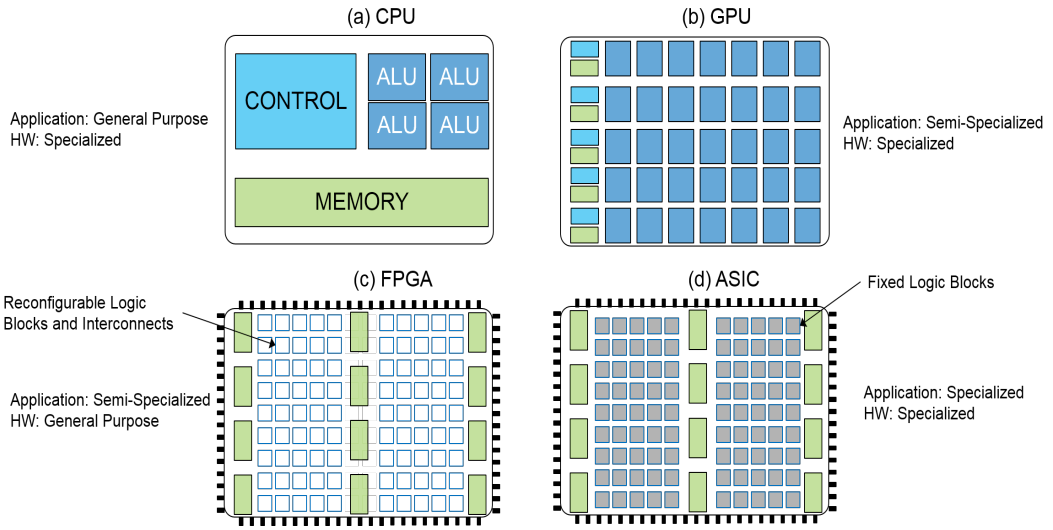


Figure 2.3: High-Level Architecture of (a) CPU, (b) GPU, (c) FPGA [6] and (d) ASIC

In contrast, FPGAs are based on spatial computing, where the program source code is transformed into a compute engine configuration file, which describes the operations, layout and connections of a compute engine (Figure 2.4-(b)). Data can be streamed from memory into the chip where operations are performed and data is forwarded directly from one computational engine to another, as the results are needed, without being written to the off-chip memory until the chain of processing is complete [7]. These fundamental architectural characteristics inherent to FPGAs have resulted in application performance improvement over CPUs in the order of hundreds [9] [70].

2.1.3.5 FPGA vs GPU

As both FPGA and GPU can be programmed to run a particular application, the performance in terms of application speed up and energy efficiency depends on the nature of the target application [71] [72] [73] [74] [75] [76]. However, in terms of qualitative characteristics, they differ in many aspects, as shown in Figure 2.5, which shows an updated radar graph by adding ASIC characteristics to an FPGA and GPU comparison from [77].

GPUs gain advantage when considering total floating-point processing power, development effort, backward compatibility, device cost, and flexibility in terms of modification of already developed applications. FPGAs also provides huge processing capabilities but with a greater power efficiency. This allows the integration of FPGAs in small housings, on-board equipments, or in extreme temperature environments. Interfacing and latency are two strong points of FPGAs compared to GPUs. GPUs are limited to PCIe, but FPGAs are flexible, offering a wide class

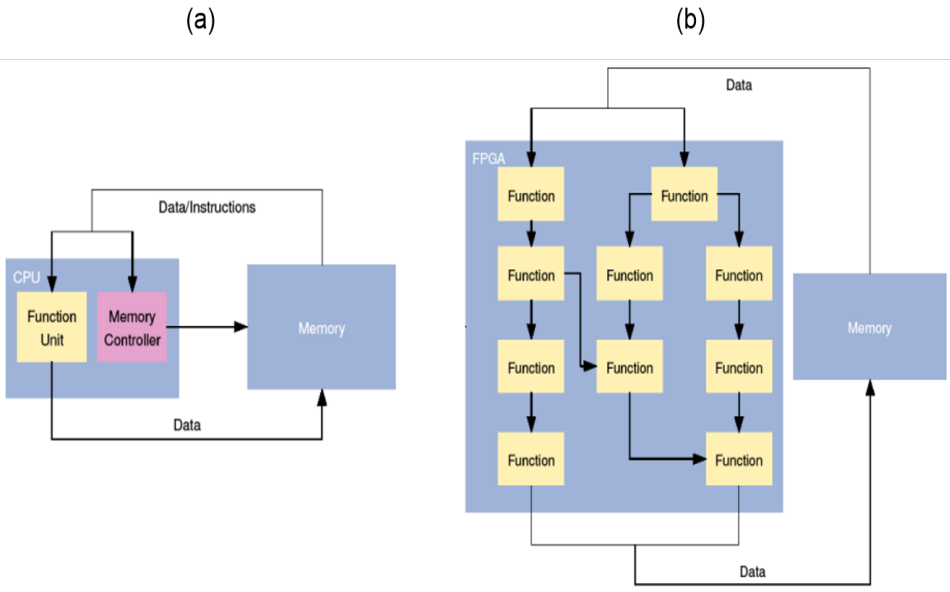


Figure 2.4: CPU Computing vs FPGA-based Data Flow Computing [7]

of possible interfaces. In terms of latency, GPUs improve CPU performance but FPGAs provides deterministic timing in the order of nanoseconds. [77]. When comparing FPGAs and GPUs there is no clear winner as of now [78], but FPGAs offers a wider scope in terms of interfaces and virtualization to be integrated in different applications. Even though FPGAs are inherently built to be used as slave HW devices over PCIe, network [79] and virtualization [80] support is being investigated by the research community.

2.1.3.6 FPGA vs ASIC

Since ASICs are devices built to run a specific function, they are inherently not flexible and does not provide backward compatibility. However, they are highly efficient in terms of energy consumption, logic density and speed compared to FPGAs, as the architecture is extensively tailored to a specific application [81]. With respect to FPGAs, the key differentiation is around two factors: (i) the cost and (ii) the time to market. The ASIC functionality determined by custom mask tooling, for which customers paid with an up-front non-recurring engineering (NRE) cost. Since FPGAs have no custom tooling, the up-front cost is reduced by making one custom silicon that can be used by thousands of customers. The high up-front NRE cost ensures that FPGAs are more cost effective than ASICs up to a certain amount of volume. But, if the cost of each unit is compared, the ASICs are much cheaper than FPGAs. Hence, beyond a certain volume, ASICs become much cheaper than FPGAs (Figure 2.6 dashed line). As the process technology improved, the unit cost of both ASICs and FPGAs reduced, but the up-front NRE

cost for ASICs increased, which made ASICs less cost effective unless the volume is extensively increased (Figure 2.6 dashed line) [8].

Even though ASICs provide much more performance at higher energy efficiency compared to FPGAs and are more cost effective beyond a certain volume, they take two or more years to develop and deploy in a production environment. Because of this long production process, the applications can make them obsolete when they are actually ready to be used. Therefore, deploying ASICs in dynamic compute environments, such as DCs, is not practical in terms of cost, time to develop and the rapid changes in applications.

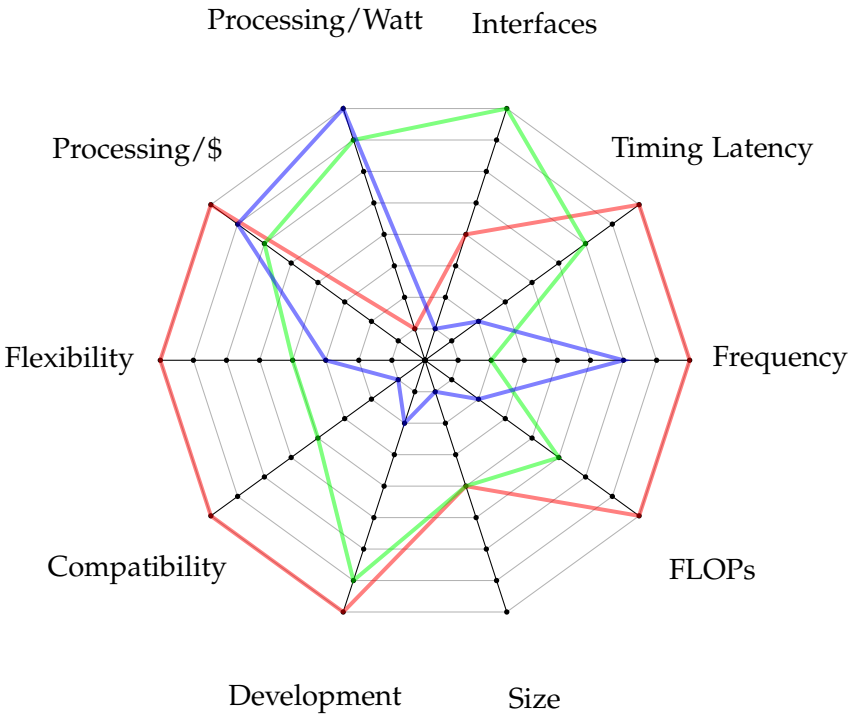


Figure 2.5: Qualitative Characterization of GPU (Red), FPGA (Blue) and ASIC (Green)

2.2 FPGA

2.2.1 Architecture

Due to their fine granularity at boolean logic level, FPGAs provide very high flexibility in what and how to implement a desired task. The basic idea of an FPGA is a set of configurable logic cells referred to as configurable logic block (CLB) (Xilinx) or adaptive logic module (ALM) (Altera) that can be arbitrarily interconnected via a programmable mesh of wires. A configurable logic cell is

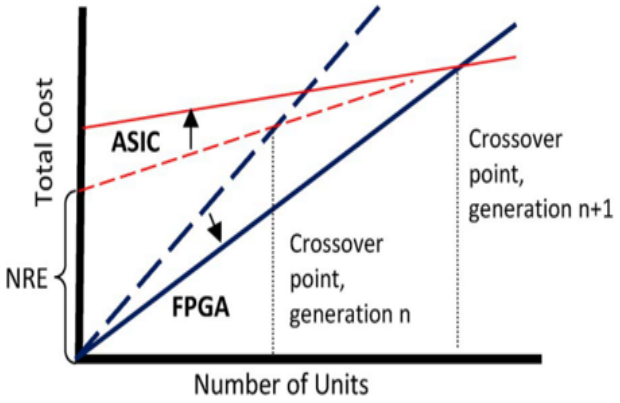


Figure 2.6: FPGA vs ASIC: Production Cost [8]

made up of a set of look-up tables (LUT), which can be configured to implement any desired logic function such as OR, AND or XOR. The LUTs are followed by a full adder (FA) structure, which can be used to create larger arithmetic or logic functions. The outputs of either LUTs or FAs can be routed to a flip-flop or register to create synchronous designs. Figure 2.7 illustrates this generic architecture of a logic cell in an FPGA.

To enhance the efficiency and performance of FPGAs, modern architectures include special hard macros implementing a wide range of functions. Common block types are embedded memory, often referred to as BlockRAM, or digital signal processing (DSP) blocks, which consists of a multi-bit wide multiplier followed by an adder or accumulator stage. Furthermore, complex input/output (I/O) blocks are added such as memory controllers for high throughput external memory access or PCI-Express controllers to provide a high-speed interface to a host processor. Especially for embedded systems, some FPGA products include a full processor core such as an ARM A9 to perform less critical but more complex operations.

The mean to describe a task that should be executed on an FPGA is to write code in a hardware description language (HDL), such as VHDL or Verilog. Sophisticated electronic design automation tools are provided by chip vendors to compile such code and generate the contents of the configuration memory. The tools map the code to vendor-specific logic elements and perform placement and routing of these elements on the chip. This means that the design-flow for an FPGA application is closely related to designing hardware on a logic level. A designer needs to be aware e.g. of the limited amount and different types of available resources and the depth of logic between register stages to achieve timing closure.

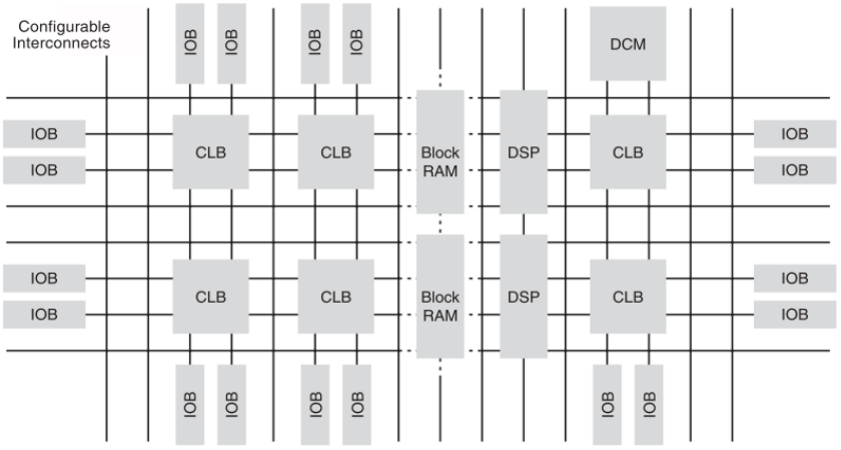


Figure 2.7: FPGA Internal Architecture [9]

2.2.2 Advances in Technology

This section looks at how the technology in FPGA evolved to what it is today, based on the statistics collected from data sheets from Xilinx and Altera. The first ever FPGA built by Xilinx (XC2064) in 1984 was based on 2.5 μm process technology and it had only 64 logic blocks, each of which held two three-input LUTs and one register [8]. Over the last decade, FPGA industry saw a rapid growth in manufacturing process technology (Figure 2.8-(a)) and as a result today's FPGAs consist of millions of logic blocks (Figure 2.8-(b)) inside a single chip. Amount of multiplications and accumulations are another important factor that measures the capability of arithmetic performance. Early FPGAs did not have such a capability, but gradually DSP blocks started to be integrated. Nowadays, state-of-the-art FPGAs contains close to 12000 DSP blocks (Figure 2.8-(c)). The block ram determines the capacity of fast memory inside the FPGA chip. Over a decade ago the amount of block ram available in FPGAs was only few mega-bits, but today state-of-the-art FPGAs consists of few hundred mega-bits (Figure 2.8-(d)), facilitating applications with low latency access to the memory. One of the key difference between FPGAs compared to CPUs and GPUs is the DRAM throughput. Historically, FPGAs were not performing well in this aspect. However, over the last few years this has been changing (Figure 2.8-(e)) and with the introduction of HBM, which is DRAM integrated inside the FPGA chip, now offers a bandwidth of around 256 GBs. This is a major breakthrough in FPGA technology when catching up with the application performance of GPUs and CPUs. Ethernet Speed is another factor that distinguishes FPGAs from other compute resources, as FPGAs are extensively used in networking applications. Figure 2.8-(f) shows that Ethernet speed has improved from 1 Gbs to 100 Gbs over the last decade. Another interesting move from the FPGA vendors is to harden the MAC and PHY layers in the chip, leaving more reconfigurable resources for the applications. In summary, technology in FPGA has

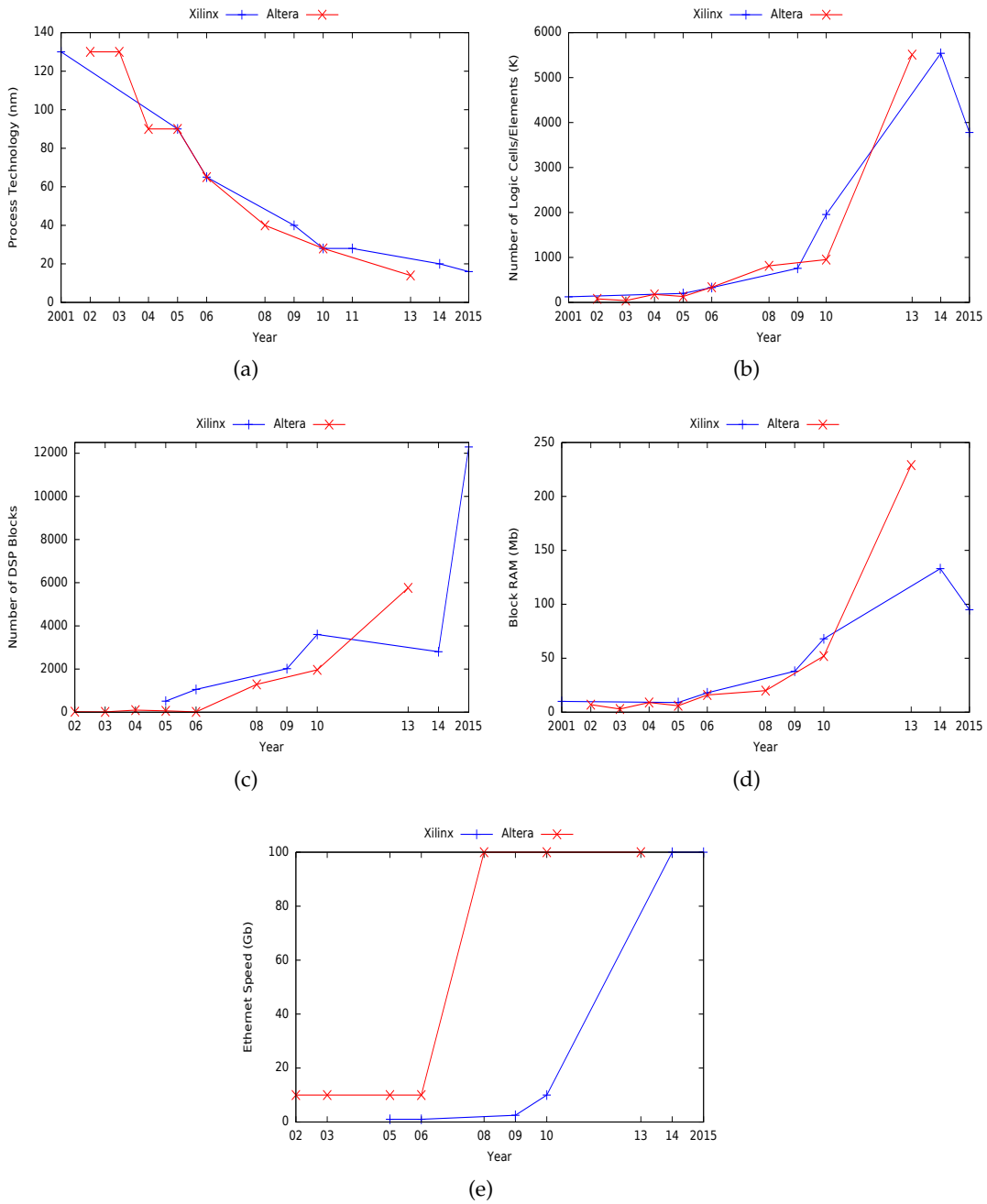


Figure 2.8: Advances in FPGA Technology Over the Last Decade: (a) Evolution of FPGA Manufacturing Process Technology, (b) Evolution of FPGA Logic Density, (c) Evolution of DSP Capacity in FPGAs, (d) Evolution of FPGA Block RAM Capacity, (e) Evolution of Ethernet Support in FPGAs

significantly advanced over the last decade, making them a promising compute resources for cloud data centers.

2.2.2.1 Programmability

One of the disadvantages of FPGAs over CPUs and GPUs is its programmability, which needs expertise in low-level RTL programming, such as Verilog and VHDL. To close the gap with CPUs and GPUs in terms of programmability, research community and industry has turned to high-level synthesis (HLS) [82]. In HLS, the design flow does not start from a hardware description but rather from an algorithm description in a programming language such as C or C++. Although some restrictions apply, this allows software programmers to experiment with FPGAs in a way they are familiar with CPUs. A wide range of industry products [83] adopted this flow and allowed different kind of inputs, such as C, C++, SystemC or even Java and MatLab code. With the increasing heterogeneity of computer systems, the urge was strong to find a common way to program and communicate with different processing units. By the end of 2008, the first OpenCL technical specification was published by a consortium consisting of both hardware and software companies to tackle this challenge. OpenCL standard defines a hierarchical memory layout with different access permissions for different parts of the code. While the code itself is very C-like, the programmer has to put some thought into how to partition the application into data-parallel and task-parallel pieces. This enabled hardware vendors to produce compilers that are able to compile such kernels to run on their devices. Initially CPU, GPU and DSP vendors provided such compilers and the required API library implementation to run it. But in 2013, Altera released the software development kit for OpenCL with their 13.0 tool suite [84]. Although all these design technologies raised the interest of application designers, a last hurdle remains: long compilation times. Despite the use of incremental compilation and hard IP blocks, the compile times of modern FPGAs can become easily multiple hours long.

2.3 FPGA Deployment Architecture

There are many possible options for incorporating FPGAs into systems to build heterogeneous computing infrastructures, including how it interfaces with the CPUs, and how the FPGAs communicate with each other. This section reviews those options for attaching an FPGA to the CPUs based on the placement. The closer the integration with the CPU, the finer-grain the problem that can be offloaded to the FPGA and the lesser the independence FPGA has for running heterogeneous applications. As shown in the Figure 2.9, an FPGA can sit (i) on the chip, (ii) on the package, (iii) being bus-attached, (iv) being pcie-attached, or (v) network-attached in the system hierarchy.

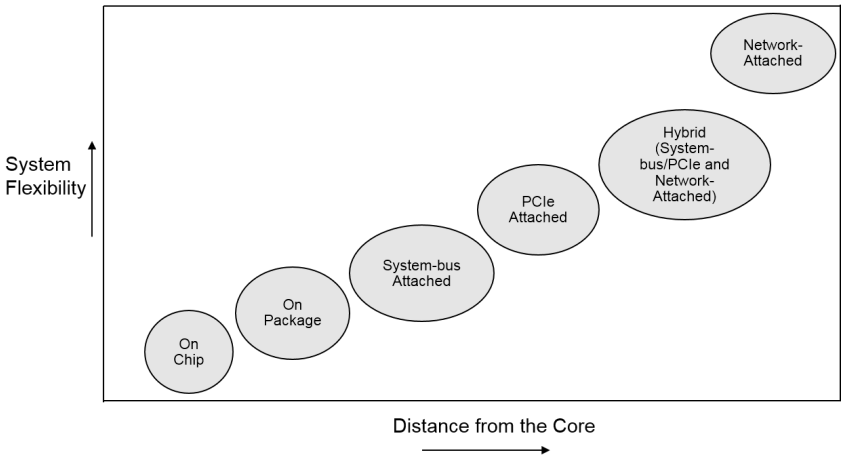


Figure 2.9: Accelerator Attachments Options

2.3.1 On Chip

In state-of-the-art SoCs that integrates CPUs and FPGAs, typically an FPGA is attached to a weak CPU, such as ARM. In this arrangement, FPGA is much more prominent than the CPU and the CPU is just there to implement the control path. Both the two major FPGA vendors have adopted this SoC approach. Xilinx releases their SoCs under the code name of Zynq [10], whereas Altera goes with the name of their FPGAs codename, namely stratix, arria, and cyclone [85]. Figure 2.10 shows a Zynq SoC from Xilinx, which combines dual ARM Cortex-A9 processors with FPGA programmable logic, operating in a small power envelope of 0.5-2 W. Each A9 core has 32 KB of instruction and data caches, a shared 512 KB L2 cache and a variety of peripherals, including FPGA fabric. The FPGA fabric is coherent with the cores' L1 and L2 caches through the Accelerator Coherency Port (ACP). Both the cores and the FPGA have access to 1 GB of DRAM through the AMBA AXI bus.

2.3.2 On Package

Processor industry led by Intel, IBM, and AMD and the FPGA industry led by Xilinx and Altera were working on two independent paths. However, with the acquisition of Altera in 2015, Intel started to integrate their CPUs and FPGAs in the same package. One key difference with the SoC approach of FPGA vendors explained in Section 2.3.1 and this approach is that Intel integrated their high-end server-class CPUs with high-end FPGAs from Altera, targeting data center workloads. Figure 2.11 shows this multi-chip package from Intel, which integrates a Xeon CPU and an Arria 10 FPGA.

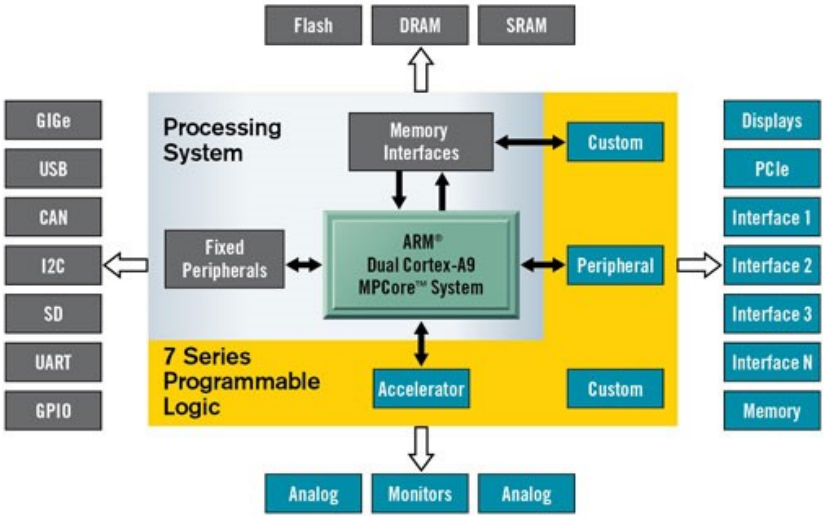


Figure 2.10: Xilinx Zynq SoC: FPGA with Dual Core ARM [10]

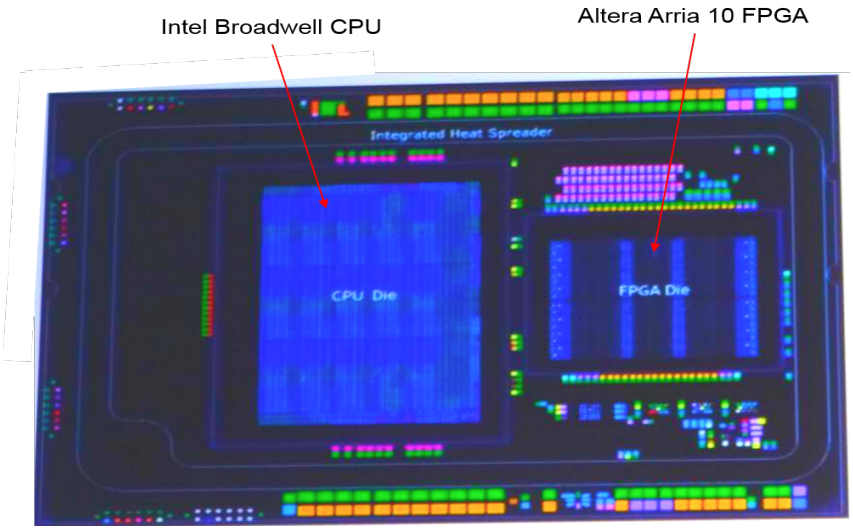


Figure 2.11: Intel Broadwell and Arria 10 Multi-Chip-Package [11]

2.3.3 System Bus-Attached

For low-latency, tight and coherent coupling between CPUs and FPGAs, CPU vendors opted to attach the FPGAs over the system bus: Intel attaches FPGAs to their CPUs over QPI [86] [87], whereas IBM provides a cache-coherent processor interface over PCIe [88]. Nallatech developed a front side bus accelerator for intel Xeon CPUs [89]. Although AMD has not officially announced about such interfaces for their processors, there are several research attempts for attaching FPGAs over hyper-transport [90] bus to AMD CPUs [91] [92]. On the other hand,

when low-latency tight coupling is needed for inter-FPGA interaction, multi-FPGA boards [93] [94] [95] [96] [97] [18] [98] used with several FPGAs connected over a shared bus on the same PCB. The topology of FPGAs in such cases varies from mesh, torus, and to crossbar [30].

2.3.4 PCIe-Attached

2.3.4.1 IBM SuperVessel Experimental FPGA Cloud

IBM SuperVessel [99] [100] is a heterogeneous computing infrastructure, which offers FPGAs over the cloud for research. It offers both PCIe- and CAPI-attached [88] FPGAs (Figure 2.12) for experimental purposes. CAPI is a feature specific to the POWER8 processor. It leverages the use of system-level accelerators by providing a high-bandwidth and low-latency interface built on top of the physical specification of PCIe. CAPI allows accelerators to operate in the same virtual address space as the processor cores allowing them to act as part of program execution. This eliminates the need for device drivers and simplifies the software integration of the accelerator. The enabling component for CAPI is the Coherent Accelerator Processor Proxy (CAPP) unit of the POWER8 processor chip. It is connected to the PCIe interface and acts as a representative core for the accelerator on the internal processor bus. It participates in the coherency protocols and maintains a directory of all cache lines held by the accelerator [88]. Although initially CAPI was proprietary and vendor specific solution from IBM, with the introduction of OpenCAPI [101] [102], other vendors are also able to benefit from this technology.

2.3.4.2 Convey

The Convey Wolverine Application Accelerator (Figure 2.13) is a PCIe Express form factor coprocessor that provides application specific hardware acceleration. The coprocessor incorporates the latest high-density Xilinx FPGAs. It consists of two FPGAs: low-end one for control logic of the card and high-end one for application engines. Further, it logically shares the virtual address space of the host through Convey's unique Globally Shared Virtual Memory (GSVM). GSVM reduces development efforts by removing the "abstractness" and programming complexities of treating the PCIe card as an I/O device [12].

2.3.4.3 Nallatech FPGA-Accelerated Compute Node

Nallatech FPGA-accelrated compute node is 1U rack server, which has 4 PCIe-attached Nallatech 510T FPGA cards each consisting of 2 Arria 10 FPGAs.

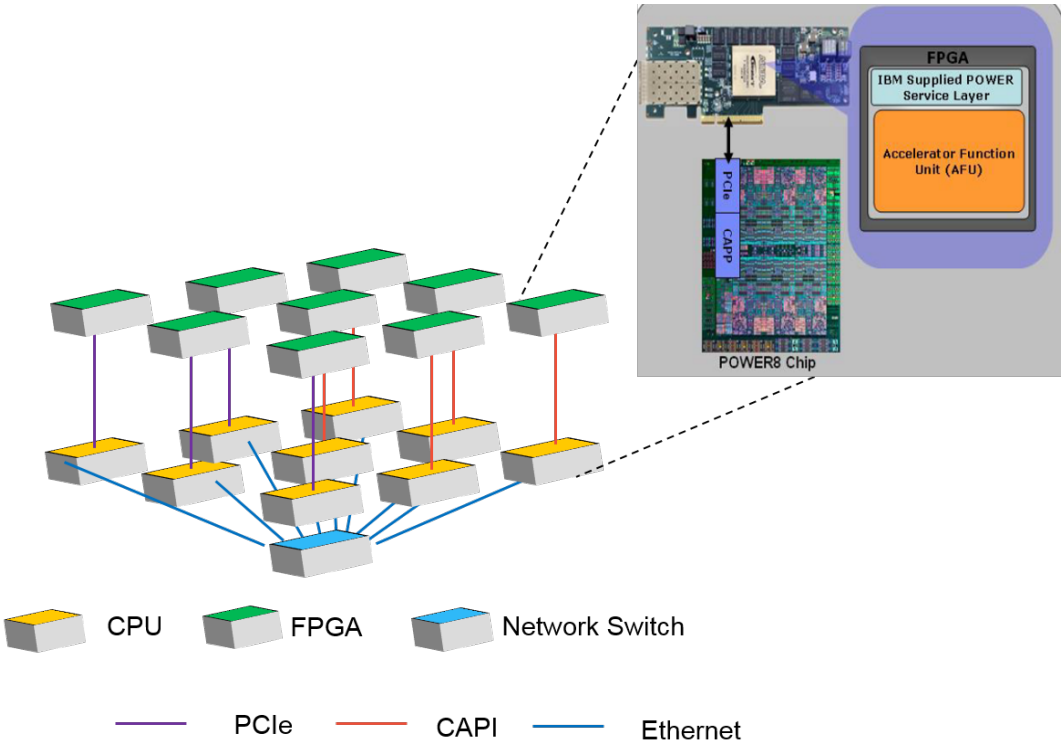


Figure 2.12: IBM SuperVessel Hardware Infrastructure

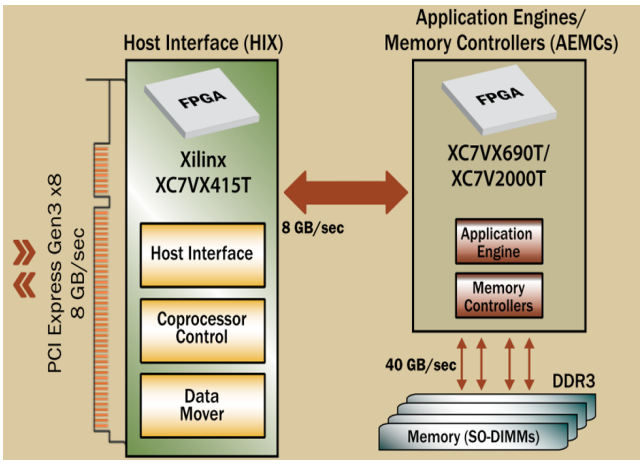


Figure 2.13: Convey Wolverine Accelerator Card [12]

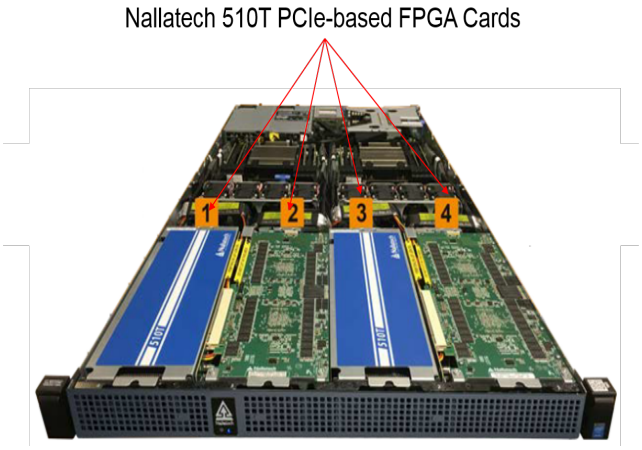


Figure 2.14: Nallatech FPGA-Accelerated Compute Node [13]

2.3.5 PCIe- and Network-Attached

2.3.5.1 NetFPGA

NetFPGA is a project designed specifically for the research and education communities. It provides software, hardware and community as a basic infrastructure to simplify design, simulation and testing, all around an open-source high-speed networking platform [103]. The first public NetFPGA platform, NetFPGA-1G [104], was a low cost board designed around Xilinx Virtex-II Pro 50. The successor introduced in 2010, NetFPGA-10G [105], expanded the original platform with a 40 Gb/s, PCIe Gen.1 interface card based on a Xilinx Virtex-5 FPGA. The latest NetFPGA provides a platform for rapid prototyping of 10 Gb/s and 40 Gb/s applications, and a technology enabler for 100 Gb/s applications, focusing on bandwidth and throughput. It is based on a Virtex-7 FPGA, along with peripherals supporting high-end design, which includes PCI Express (PCIe) Gen.3, multiple memory interfaces and high-speed expansion interfaces [103].

2.3.5.2 Microsoft Catapult V1

Catapult [25] (Figure 2.15) is a highly customized application specific FPGA-based reconfigurable fabric designed to accelerate page ranking in the bing web search engine. From the perspective of cloud computing, the acceleration provided is integrated in the SaaS layer and is hidden from the users when the bing search service is used over the Internet. In catapult, 1632 FPGAs are connected in a 2D torus network where each FPGA with 8 GB of off-chip DRAM is node attached to a server over the PCIe bus. The page ranking algorithms are executed in a pipeline of 8 FPGAs, and the inter-FPGA communication occurs over 10 Gb SerialLite III [106] protocol. Compared to a software-only approach, catapult has achieved 95% improvement in ranking throughput for a fixed latency, and when the throughput

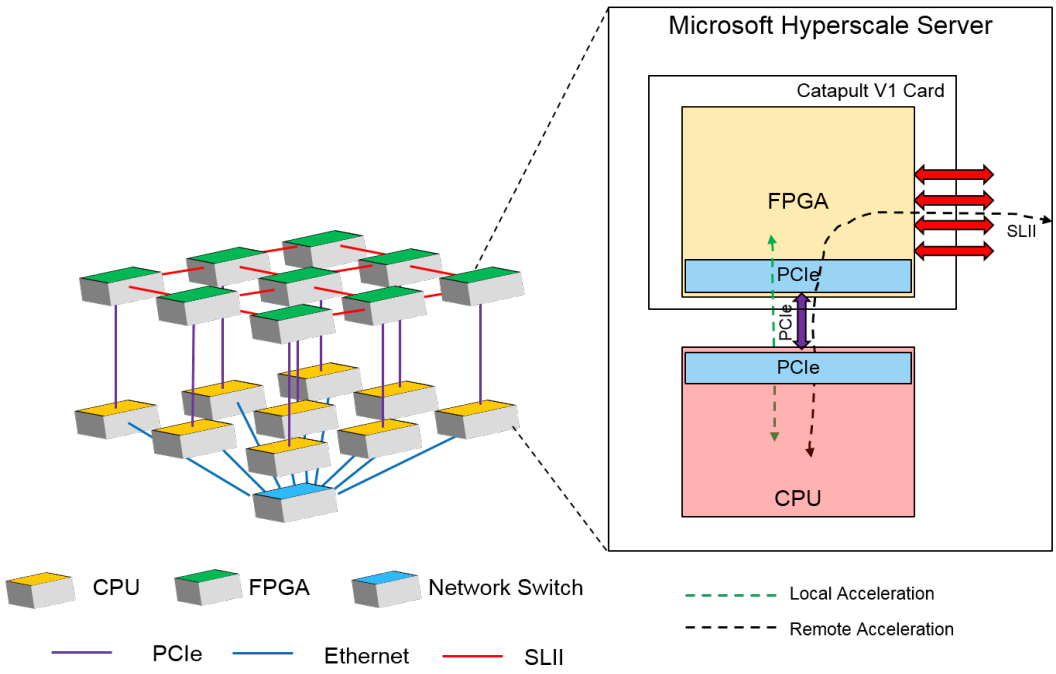


Figure 2.15: Microsoft Catapult V1

is kept constant it has reduced latency by 29%.

Catapult is a promising example which shows the potential of heterogeneous computing at large scale in cloud data centers. Even though catapult has shown good results, as a system deployed in a data center it has few drawbacks. First, each FPGA is node attached which makes 3 points of failures; server, FPGA and network, and this decreases the fault tolerance of the whole system. This makes the system inefficient because most of the resources in data centers are deployed under fail-in-place strategy, where even a fault occurs in a resource they are not repaired or replaced soon to reduce management overhead. Second, technology tailoring and scaling up and down of resources are inherently difficult with the node attached FPGAs. Third, dedicated inter-FPGA network breaks the homogeneity of the data center network and increases management overhead. Since the whole IT stack is managed and customized solely for a single application, this can be considered as an application specific platform or a data center. Even though, it is a significant management overhead, it can be traded off for the performance achieved. However, in the case of general purpose cloud data centers this is not practical. The heterogeneous devices are used in a diverse kind of applications. Therefore, the deployment must be flexible and scalable like traditional data center resources such as server and storage.

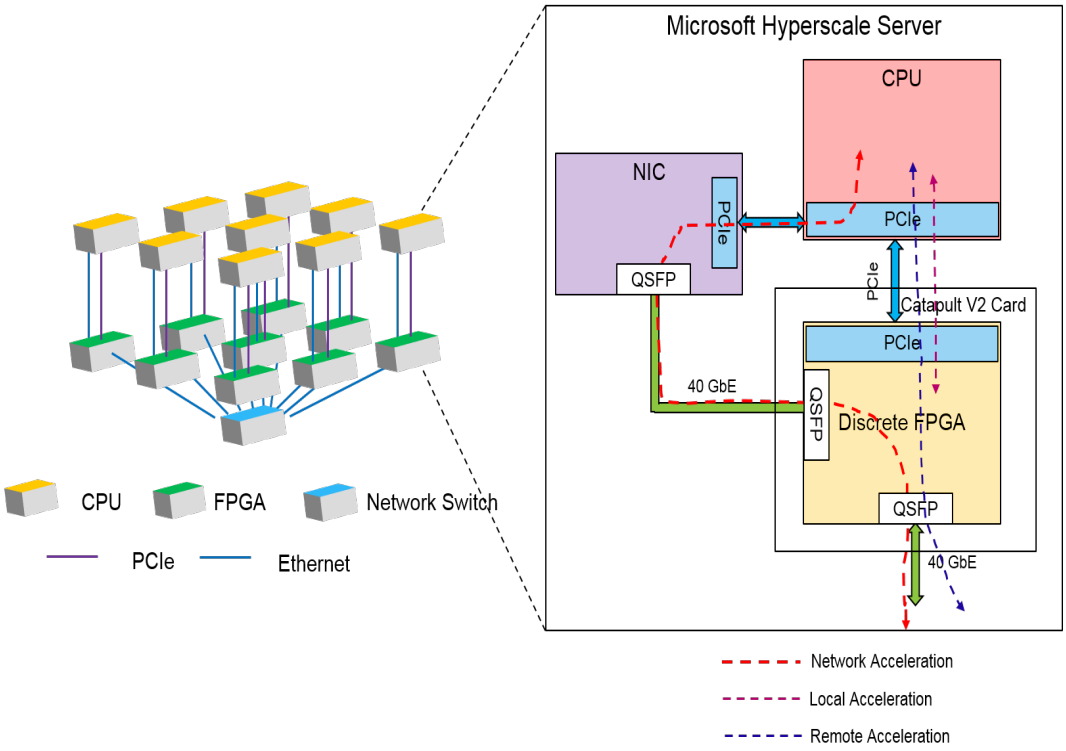


Figure 2.16: Microsoft Catapult V2

2.3.5.3 Microsoft Catapult V2

The shortcomings of the Catapult V1 [25] approach due to the tight coupling of FPGAs to the server and the dedicated network for interconnecting FPGAs have been solved to some extent in Catapult V2 [1] by connecting the PCIe-attached FPGAs to the DC network as shown in Figure 2.16. Unlike in V1 approach, in this approach as many FPGAs as needed can be used by a single server and those FPGAs can be connected in a flexible, user-defined topology. However, in the V2 approach, to use the FPGAs for application acceleration as well as for network acceleration, the server NIC is connected to the DC network through the FPGA. This makes all the incoming and outgoing traffic to and from the server pass through the FPGA. The FPGA is partitioned into two regions using partial reconfiguration. The dynamic region executes the application acceleration whereas the static region executes the network acceleration. This architectural aspect brings several disadvantages into V2 approach: (i) The full programming of the FPGA disrupts the network connection to the server, making it difficult to offer reliable infrastructure services in cloud DCs (ii) any change in the DC network affects the FPGA as well, which increase the system HW cost for network upgrade. One way to get around these two issues is to connect both the server and the FPGA independently to the DC network, as depicted in Figure 2.17 from [14]. However, this approach still

does not provide the scalability required from the FPGA infrastructure.

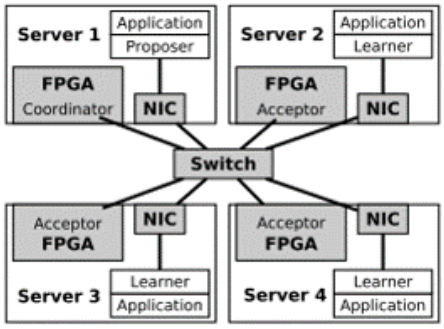


Figure 2.17: Server and PCIe-Attached FPGA Independently Connected to a Common DC Network [14]

2.3.5.4 Maxeler MPC-N Series

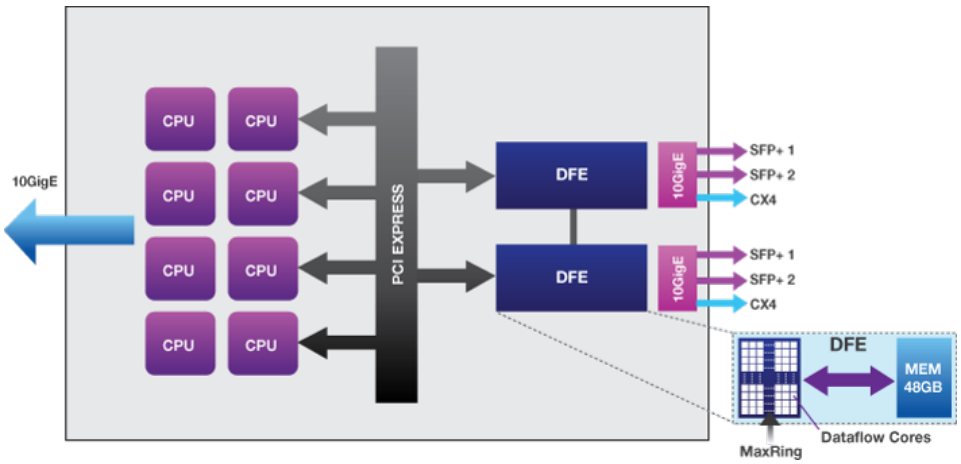


Figure 2.18: Maxeller MPC-N Series [7]

Maxeler dataflow engine [7] is a FPGA-based dataflow computer designed for data-compute-intensive applications like financial data processing. Maxeler MPC-N series is a heterogeneous compute appliance (Figure 2.18), which consists of an Intel Xeon CPU and two FPGAss. Each FPGA is attached to the CPU over PCIe and also to the DC network over SFP/SFP+. FPGAs are connected in a secondary link for direct communication between each other. Except for the secondary link between two FPGAs, the architecture is almost similar the one depicted in Figure 2.17.

2.3.5.5 Maxeler MPC-C Series

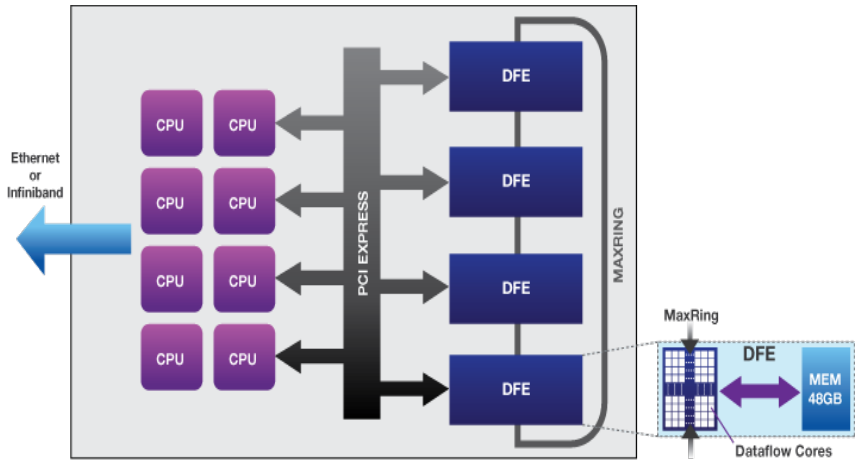


Figure 2.19: Maxeller MPC-C Series [15]

Maxeler MPC-C series [15] is an enhanced version of MPC-N series, where the number of FPGAs in the appliance is doubled to 4. Similar to MPC-N series, the FPGAs are directly connected forming a ring called MAX Ring. In this series, the direct 10 GbE connections out of the FPGAs are pruned. [107] has used MPC-C series to implement finite difference time domain method for solving Maxwell's equations.

2.3.5.6 Amazon F1 Instance

Amazon EC2 F1 [47] instance is a server node with up to eight FPGAs. F1 instances include the Xilinx UltraScale Plus FPGA with local 64 GB DDR4 ECC protected memory. FPGAs are PCIe-attached, and each FPGA is connected together over a dedicated 400 Gbps secondary network as shown Figure 2.20.

2.3.5.7 AXEL

Axel [108] is a heterogeneous cluster that targets compute-intensive applications such as N-body simulations. It comprises a cluster of heterogeneous nodes each consisting of a CPU, an FPGA, and a GPU connected via PCIe. The CPUs of each node communicate over Gigabit Ethernet, and the FPGAs of each node communicate over an infiniband network. The communication between tasks across different nodes is based on the OpenMPI framework through ethernet, while the communication between the tasks across different processing elements in the same node is based on the shared memory inter process communication. The experiment results show that for a single node, N-body simulation run time is 22.7 times faster in the Axel architecture compared to a CPU only implementation. When the number of nodes is increased to 16, the performance does not scale linearly and

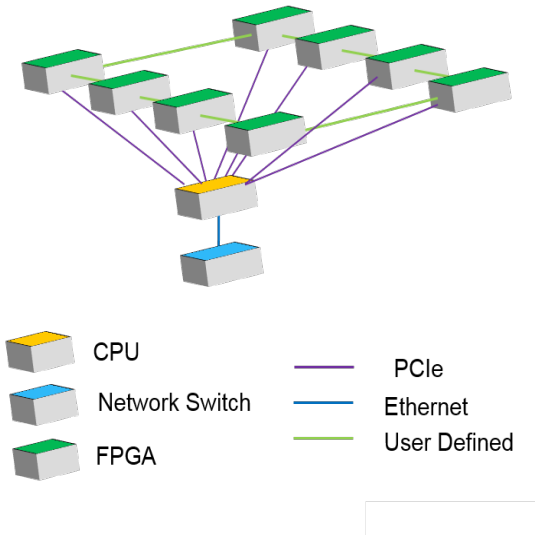


Figure 2.20: Amazon EC2 F1 Instance

the improvement is only 4.4 times. This is because of the communication overhead between nodes.

2.3.5.8 OpenPipe

OpenPipes [109] is an FPGA-based distributed application deployment framework based on PCIe-attached FPGAs. The inter-FPGA and SW-FPGA communication occurs via TCP over Ethernet network. In order to change the behavior of the distributed dynamically, software-defined networking is used.

2.3.6 Network-Attached

2.3.6.1 Maxeller MPC-X Series

In contrast to the MPC-N and MPC-C series, Maxeler MPC-X series [16] offers an FPGA only appliance with 8 FPGAs connected by MAXRing. The external connection from this appliance is over Infiniband and each FPGA is directly connected to an Infiniband switch fabric within the appliance (Figure 2.21). In this approach, even though FPGAs are decoupled from the CPUs, they are connected in a fixed topology, restricting the infrastructure to a certain set of applications.

2.3.6.2 BEE2

Berkeley Emulation Engine (BEE2) [95] developed in 2004 has five Xilinx Virtex-II Pro 70 FPGAs hosted on a single motherboard. Out of five, four computational FPGAs are connected in a ring. The central control FPGA is connected to other

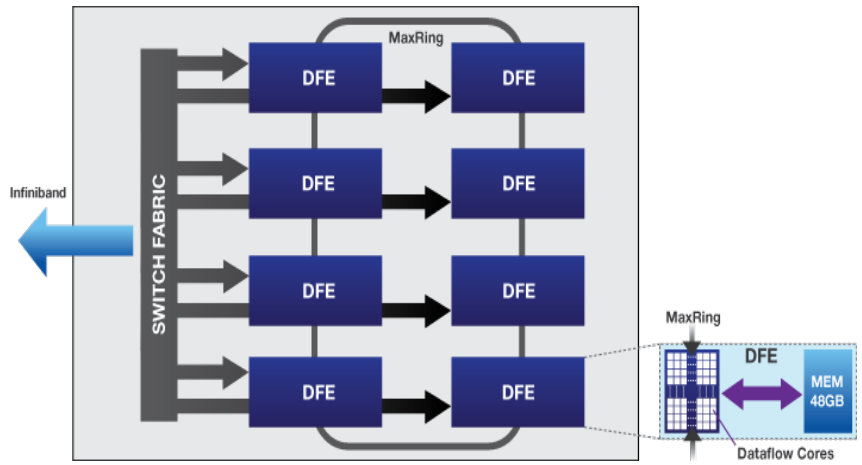


Figure 2.21: Maxeller MPC-X-Series Appliance [16]

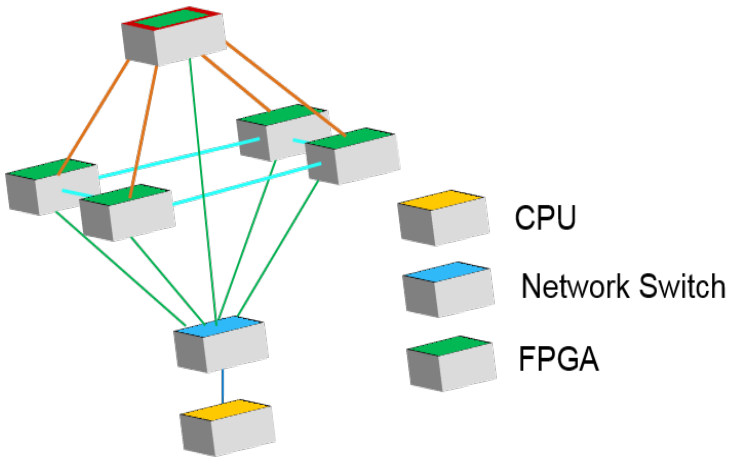


Figure 2.22: UC Berkeley BEE2

four FPGAs in a star topology. Compute-intensive tasks run on the outer ring while the control FPGA runs Linux managing configuration and off-board I/Os.

2.3.6.3 Copocobana/Rivyera

COPACOBANA 5000 [94] [17] consists of an 18-slot backplane equipped with 16 FPGA-cards and 2 controller cards. The latter connect the massively parallel FPGA-computer to an in-system off-the-shelf PC. Each of the FPGA-cards carry 8 high performance FPGAs interconnected in a one-dimensional array as shown in Figure 2.23. The interconnection between the individual FPGA-cards and between the FPGA-cards and the controller is organized as a systolic chain. There are fast point-to-point connections between every two neighbors in this chain. The first controller communicates with the first FPGA on the first card and the last FPGA

on the last card is connected to the second controller.

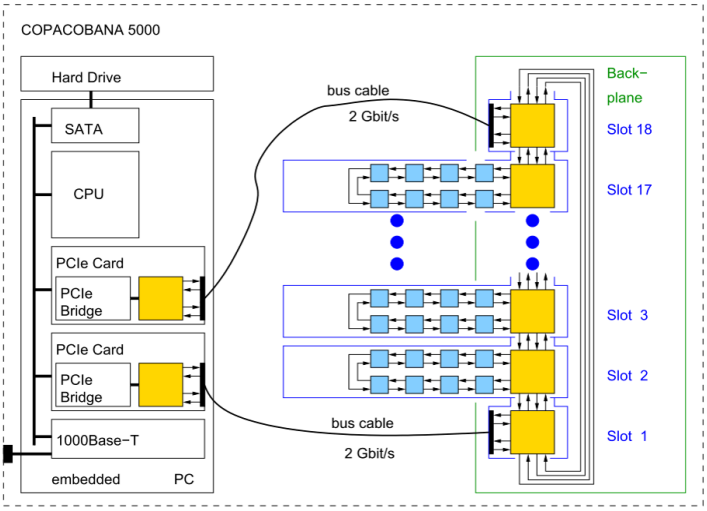


Figure 2.23: COPACOBANA 5000 System Architecture [17]

2.3.6.4 CUBE 512-FPGA Cluster

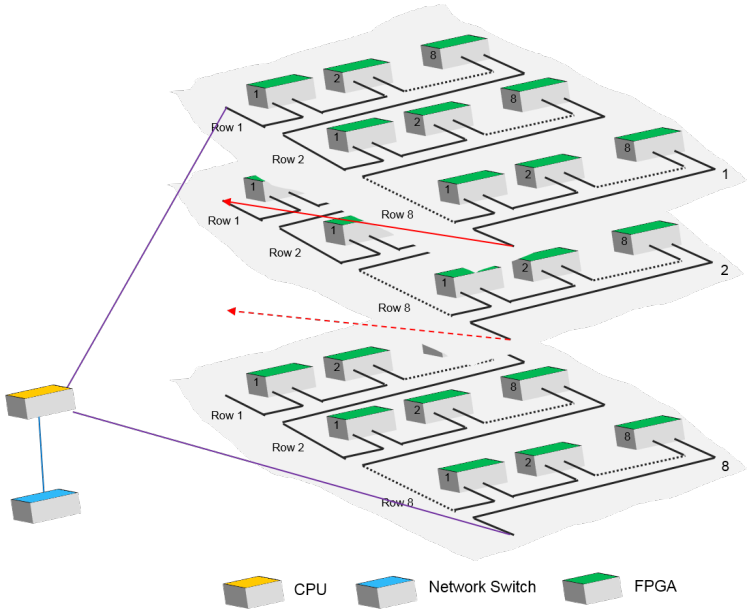


Figure 2.24: CUBE 512-FPGA Cluster

Cube [93] is a 512-FPGA systolic array system. The system is made from 8 boards

each containing 64 FPGA devices connected in a cube structure for a total of 512 FPGA devices. According to authors, RC4 key search engine implemented in cube can perform a full search on the 40-bit key space in 3 minutes, improving 359 times compared to a multi-threaded software implementation on a 2.4 GHz Intel Quad-Core Zeon processor. One drawback of this system is that the network topology is an 8x8x8 3-D Mesh, not optimized for latency. Therefore, communications latencies are very high for the applications that do not fit this paradigm.

2.3.6.5 RAMP

RAMP(Research Accelerator for Multiple Processors) [96] is a MicroBlaze processor based message-passing multi-core system capable of running scientific benchmarks. This system is built by replicating multiple BEE2 [95] boards, and it consists of 768-1008 MicroBlaze cores in 64-84 Virtex-II Pro 70 FPGAs on 16-21 BEE2 boards scaling to 1000 cores in a standard 42U rack. The board-to-board communication is over a 10 Gbs full-duplex link.

2.3.6.6 TMD

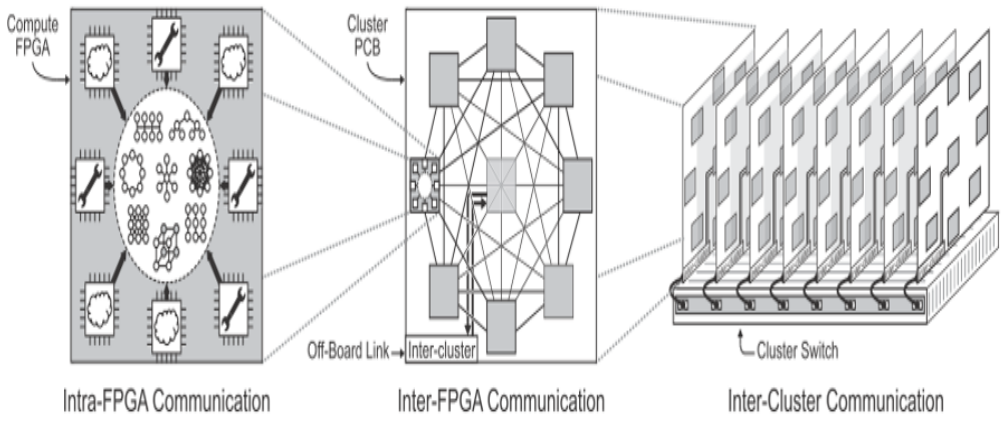


Figure 2.25: TMD FPGA Cluster [18]

TMD [18] is a system built by attaching multiple boards each with a cluster of fully connected 9 FPGAs. Within a single board all 9 FPGAs are connected to each other using 2.5 Gb/s serial transceivers and the boards are connected to the node via 10 Gb/s links. The authors focused on accelerating applications with high computation-to-communication ratios (e.g., molecular dynamics simulations), instead of data-intensive applications. The machine enables designers to implement

large-scale computing applications using a heterogeneous combination of hardware accelerators and embedded microprocessors spread across many FPGAs.

2.3.6.7 Nanostream

NanoStream [110] is an application specific heterogeneous architecture for stream analytics in data-intensive applications. The heterogeneous architecture is based on the Xilinx Zynq©-7000 System-on-Chips (SoC). NanoStream software stack runs on the ARM-based processing system (PS) and the accelerator cores runs on the FPGA-based programmable logic (PL) in the SoC. Multiple SoCs are connected over an Ethernet network for scalable system deployment.

2.3.6.8 NARC

NARC [19] is a standalone network-attached FPGA card designed for HPC and network applications. The custom board consists of a Xilinx FPGA and an ARM processor. The ARM processor and an external PHY are used as the network interface to connect the FPGA to the 1 G Ethernet-based network. Even though, the FPGA card is standalone and network-attached, the major drawback of this approach is that all the network packets have to pass through a general purpose ARM processor, which degrades overall application performance that can be delivered by the FPGA.

2.4 Summary

Custom HW accelerators have been used widely in stringently constrained computing environments, such as mobile computing. However, diminishing physical characteristics of CMOS (Moore's law and Dennard scaling) and lack of viable alternatives have attracted custom HW accelerators towards main-stream computing as well. This trend is becoming clear with the advent of FPGAs, GPUs and ASICs in the DCs in recent years.

ASICs are specialized HW that runs specialized applications. The time to market, the flexibility (in terms of the backward compatibility and modification of applications), and the up-front NRE cost make ASICs not suitable for dynamic DC infrastructures, where the applications/algorithms change frequently. GPUs are also specialized HW, but the range of applications that run on top of them are not highly specialized. However, the performance depends on the nature of the applications, and GPUs typically perform well for parallel floating-point computation. In contrast, FPGAs are general-purpose HW, which can be specialized in the field to run specific applications. FPGAs perform well for parallel applications as well as for streaming applications.

This thesis focuses on FPGAs for main-stream computing in cloud DCs. Figure 2.27 shows the summary of the state-of-the-art large-scale FPGA deployments

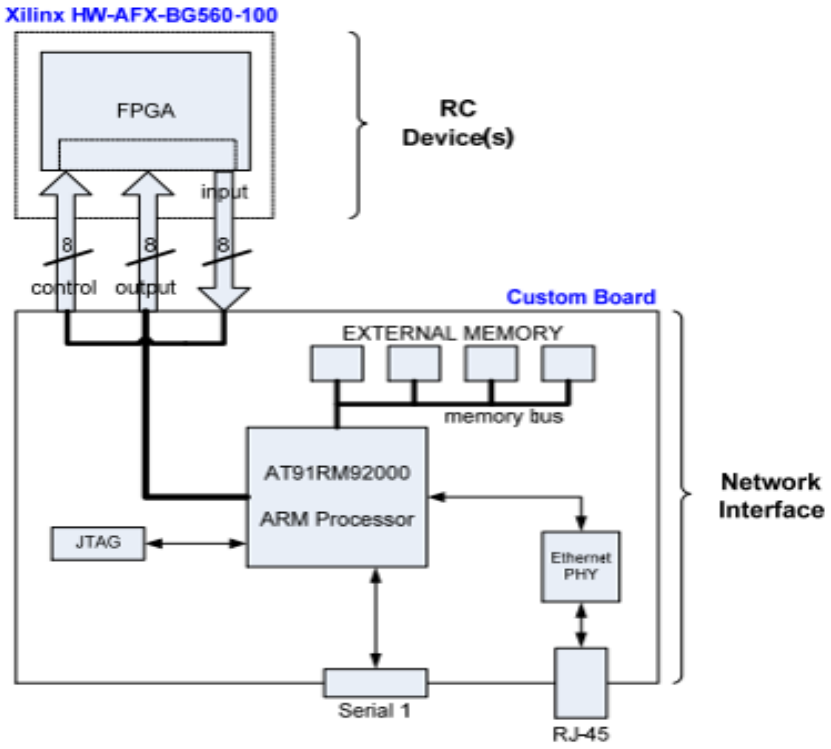


Figure 2.26: NARC FPGA Card Architecture [19]

reviewed in this chapter. Those deployments can be categorized into three types based on the CPU-FPGA attachment architecture: (i) bus-attached, (ii) hybrid-attached (bus- and network-attached) and (iii) network-attached. Tight coupling of FPGAs to servers over a peripheral bus limits the scalability of the FPGA infrastructure in terms of the number of FPGAs that can be deployed independently of the number of servers. Network-attached FPGAs are expected to solve this issue, but state-of-the-art network-attached deployments are connected on fixed topologies, limiting the scope of the applications that can run on them.

Table 2.1 shows the scalability of those deployments at rack scale. Some architectures are projected to rack-scale versions for the comparison. Among these deployments, RIVYERA and BEE2 are architectures specialized for a specific application, whereas HP and Nallatech are FPGA-accelerated commercial off-the-shelf servers. Microsoft is the only state-of-the-art DC deployment. According to the comparison, extensively customized dedicated FPGA racks (RIVYERA, Maxeller, BEE2) can be built at high density, scaling up to around 1300 FPGAs per rack, whereas state-of-the-art FPGA-accelerated heterogeneous compute racks (Microsoft, HP, Nallatech) scale up to only few hundreds of FPGAs per rack. However, both approaches are not suitable for deployment of FPGAs at large-scale in cloud DCs: (i) although specialized deployments offer high scalability, they do not have

Table 2.1: FPGA Rack Performance Comparison

Per Rack	RIVYERA	Maxeller	BEE2	Microsoft	HP	Nallatech
FPGA	Virtex 4	Virtex 6	Virtex-2	Stratix 5	Stratix 4	Arria 10
No of FPGAs	1280	256	160	96	378	248
LUTs (10^6)	39	76	15	33	160	106
FFs (10^6)	39	152	15	66	160	424
FPGA:Host	1:0	1:0	1:0	1:1	1:1	8:1
FPGA:DRAM	1:0	1:96	-	1:8	1:64	1:16
FPGA:NetBW	8:1	-	1:10	1:40	1:1	-

the flexibility, and (ii) heterogeneous compute racks provide neither scalability nor flexibility.

Along the evolution of heterogeneous computing (Figure 2.28), we envision that future main-stream computing systems would comprise more custom HW accelerators than CPUs (Figure 2.28-(c)). We have already seen this trend in the mobile-computing domain for exploiting power efficiency and space optimization, where some state-of-the-art SoCs [20] dedicate more than half of the chip space for accelerators (Figure 2.29 and 2.30). In DCs, the scale and the nature of applications change frequently. Hence, fixed allocation of FPGAs to an application hinders the exploitation of the maximum efficiency available from the compute infrastructure. Therefore, two key requirements expected from the FPGA infrastructure in cloud DC systems are (i) a flexible CPU:FPGA ratio and (ii) the ability to interconnect FPGAs in user-defined topologies. To satisfy these requirements, a novel architecture is required for deploying FPGAs at large scale in DCs. In the next chapter, a system architecture is proposed based on these requirements and on the knowledge gathered in this chapter.

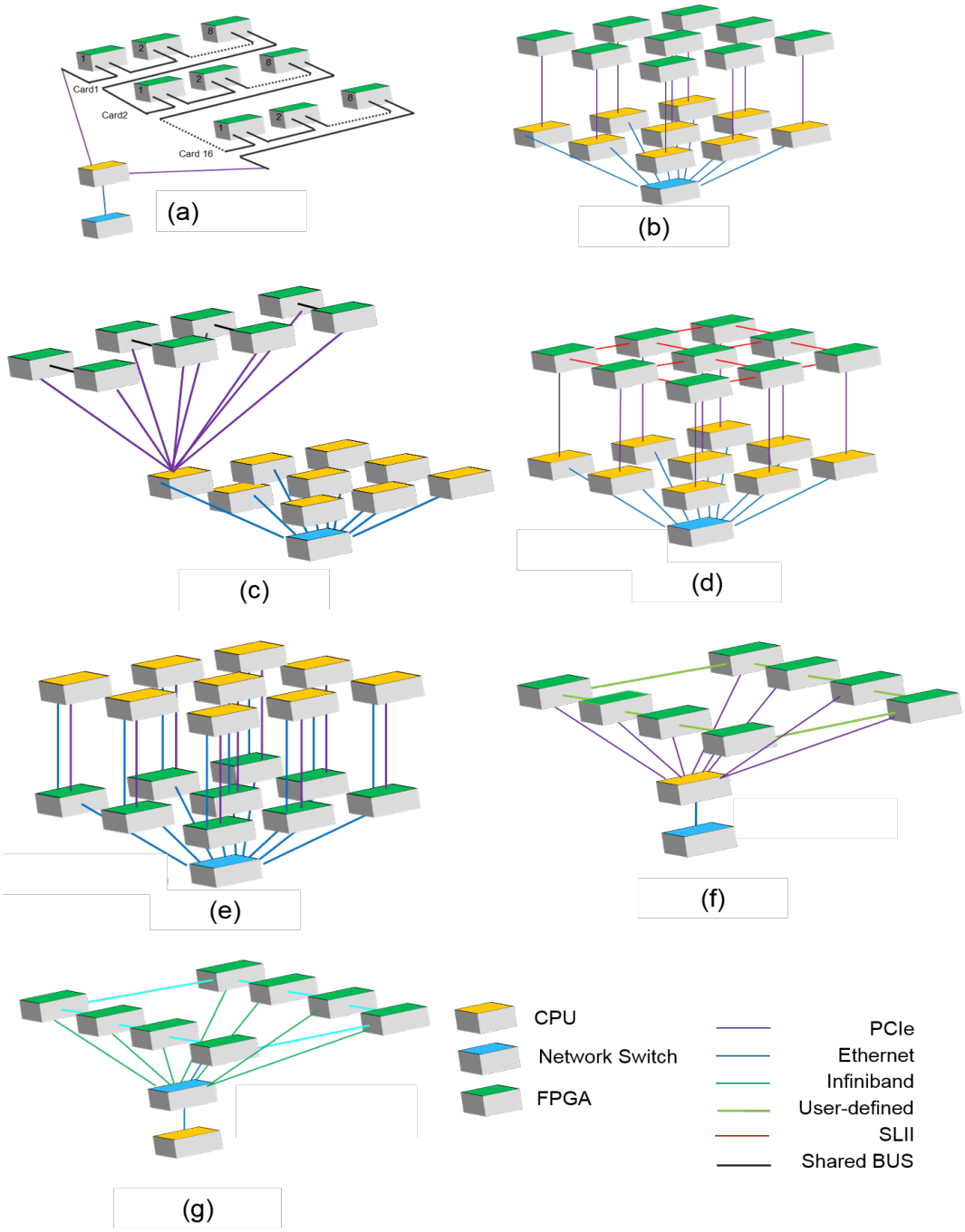


Figure 2.27: Large-scale FPGA Deployment Summary: (a) RIVYERA, (b) HP, IBM SuperVessel, (c) Nallatech, (d) Microsoft Catapult V1, (e) Microsoft Catapult V2, (f) Amazon EC2 F1 Instance, and (g) Maxeller MPC-X

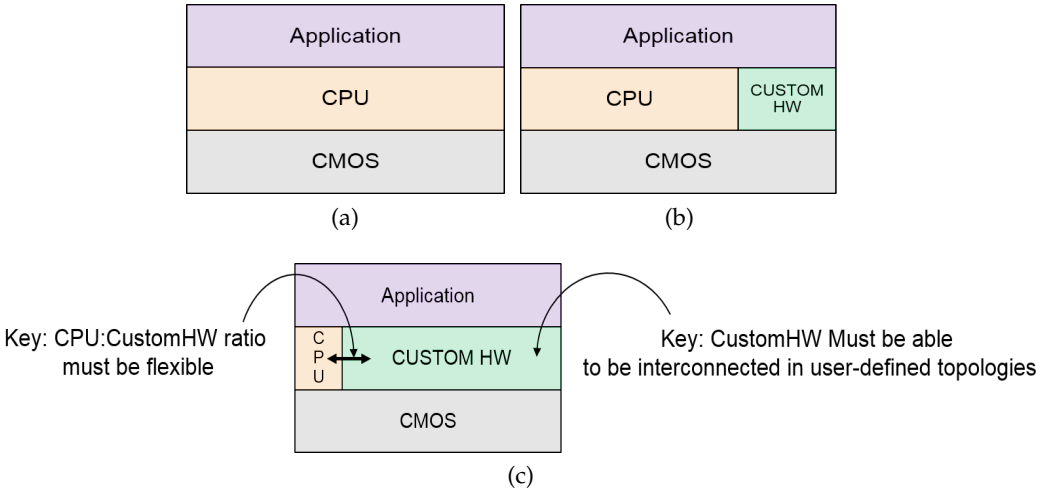


Figure 2.28: Emergence of Heterogeneous Computing: (a) Traditional Data-Processing Platform, (b) HW-Accelerated Data-Processing Platform, (c) Custom-HW-Dominated Future Data-Processing Platforms

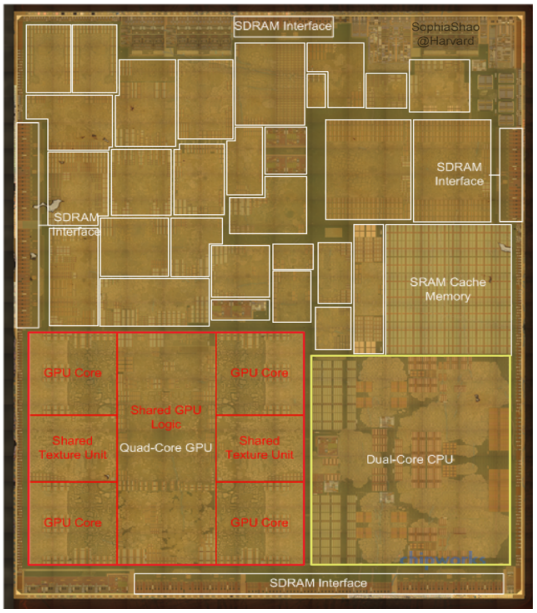


Figure 2.29: Apple A8 SoC Die Photo [20]

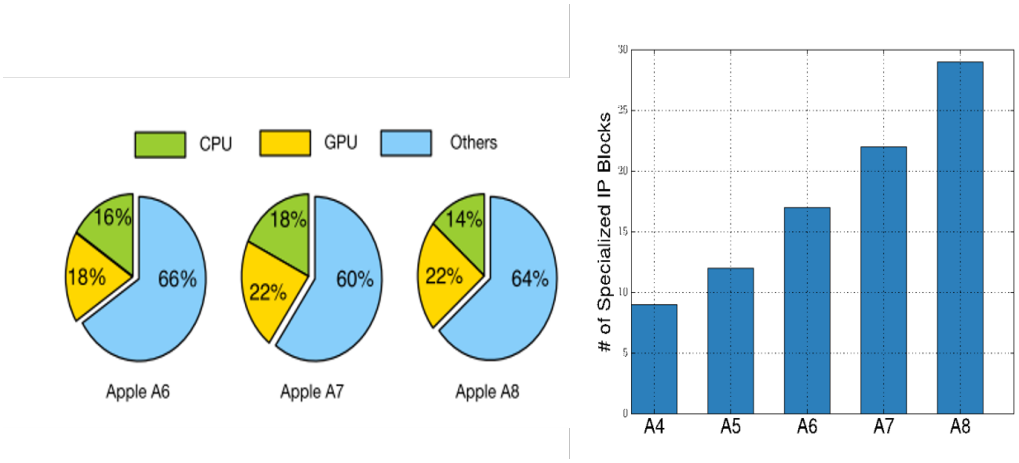


Figure 2.30: Apple SoC Constitution [20]

Chapter 3

System Architecture

This chapter proposes a system architecture for deploying FPGAs at large scale in DCs. The chapter is organized as follows: Section 3.1 analyzes infrastructure requirements for enabling FPGAs in DCs. Based on the identified infrastructure requirements, next three sections define the system architecture. Section 3.2 proposes to change the way FPGAs are deployed in DCs by introducing standalone disaggregated FPGAs, followed by Section 3.4 elaborates on the proposed DC infrastructure for deploying standalone disaggregated FPGAs. Finally, Section 3.5 explains how those FPGAs are managed and provisioned to the users in cloud DCs. The chapter is summarized in Section 3.6.

3.1 Infrastructure Requirements

FPGAs are widely used in storage and networking appliances in DCs. In the computing space, FPGAs have been used for specific applications in specific computing infrastructures, such as HPC [97] and mobile computing. However, at the time of writing this, FPGAs just started to make their way in to the DCs as general purpose compute resources [25] [1] [47]. Introducing FPGA as a compute resource to DCs, which runs general purpose applications, needs an analysis to figure out the requirements that should be satisfied in large-scale deployments. The infrastructure requirements we have identified are: (i) scalability, (ii) flexibility, (iii) reliability, (iv) homogeneity, (v) resource management (vi) system cost and (vii) power efficiency. These requirements are studied in the next sections. Along with the knowledge gathered in Chapter 2, the outcome of this requirement analysis is used to define the system architecture.

3.1.1 Scalability

One of the most influential benefits of using DC infrastructures is scalability. Scalability refers to being capable of seamlessly adding more DC resources, such as

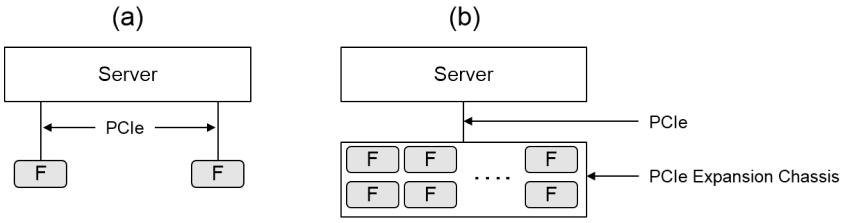


Figure 3.1: Attaching FPGAs to a CPU over PCIe: (a) Direct PCIe-attachment and (b) PCIe-expansion chassis based attachment

servers, storage, and network devices. The capability to scale in this manner, allows to provide as much resources as needed by the DC users. In other words, DCs provide their customers the possibility to scale their applications from a single VM to virtually any number of servers.

When new resources, such as FPGAs are introduced in DCs, the same illusion must be given to the users. To provide a scalable solution to the customers, the FPGA infrastructure must be designed in a way, such that it can pack a large number of FPGAs in a unit DC rack space. With the traditional PCIe-attached approach, there would be one or two FPGAs per server node (Figure 3.1-(a)) scaling only up to around 64 FPGAs per DC rack. In this approach, the total amount of FPGAs in a DC can be increased by increasing the number of servers. However, this type of scaling takes up a large amount of space and increases the overall infrastructure cost. One way to increase the number of FPGAs per rack is to use a dedicated chassis, such as a PCIe-expansion chassis (Figure 3.1-(b)). Using the state of the art PCIe-expansion chassis [22], the number of FPGAs per rack can be scaled up to around 108. In addition to the increased FPGA density, the other advantage of this approach is the cost reduction due to the high FPGA/Server ratio.

3.1.2 Flexibility

1. **CPU Independence:** DC workloads are heterogeneous and dynamic. Some workloads may require one server and a part of an FPGA and some other workloads may require one server and multiple FPGAs. Therefore, the FPGA infrastructure must be able to provision as many FPGAs as needed independent of the number of servers. Similar to renting virtual machines, users should be able to rent as many FPGAs as required, use them in heterogeneous applications and release them when not needed. If the FPGAs are PCIe-attached, as discussed in Section 3.1.1, only few devices can be hosted by a server and this number is fixed. For heterogeneous and rapidly changing workloads, sometimes this deployment becomes over provisioning and some other times it becomes under provisioning.

2. **Programmable Topologies:** As data center workloads are becoming increasingly heterogeneous and dynamic, software-defined infrastructures are highly desirable, so that resources can be dynamically assigned according to the workload requirements. Therefore, being able to connect multiple FPGAs in a software-defined manner according to the dynamic workloads requirements are highly desirable.
3. **Workload Migrations:** In DCs, workloads are migrated to different servers from time to time to meet diverse user and management requirements such as SLA, system optimization and disaster avoidance. The usage of FPGAs should be smooth over such migrations.

3.1.3 Reliability

In large scale DCs, failed compute resources are kept for months and years without repairing or replacing soon to reduce the management overhead. This is called the "fail-in-place" strategy. To follow this strategy, the DC infrastructure must be designed in a way such that failed resources do not make much impact on the overall system reliability. New resources in the DC, such as FPGAs, must follow the same strategy to maintain a higher reliability. To follow the fail-in-place strategy, FPGAs must be independent of other infrastructure resources such as server/CPU nodes.

When FPGAs are PCIe-attached and hosted by a server [25] [111], they rely on the server particularly for power and management. If the server fails, the FPGA fails as well. In some approaches the server NIC is connected to the DC network via the PCIe-attached FPGA [1]. In those approaches, the full reconfiguration of the FPGA disrupts the network connection to the server, hindering the provisioning of those servers to run applications, which require 100% service availability.

Malicious data can intercept the operation of infrastructure resources and eventually bring down the whole system. Therefore, it is required to monitor for malicious data communicated into and out of FPGAs to detect such incidents and react expeditiously. To protect the system from possible damages that can cause by FPGAs, in most approaches a shell is used [25] [27] [112], which is designed in way to monitor and protect the system from possible damages that may be caused by the FPGA and the application. This shell is provided by the FPGA provider and the user application runs inside the shell.

3.1.4 System Cost

Typically, an off-the-shelf FPGA card costs around $2000 to $3000 [113] [114], whereas the cost of a DC-class sever is around $5000. Attachment of an off-the-shelf FPGA to a server node increases the overall cost of the node by around 40%-60%. Large-scale commercial FPGA deployments shows that the addition of an FPGA card increases the cost of server node by 30% [25]. The price of FPGA chips

are decreasing with the continuous improvement in process technology. However, FPGA card manufacturers add a cost of around $1000 to $1500 on top of the FPGA chip to release an FPGA card to the market. The high cost involvement of the FPGA card manufacture hinders the reach of real cost advantage to the end users.

The cost optimization of a large-scale FPGA deployment can be achieved in two ways: (i) the way FPGAs are deployed must be changed so that an expensive server node is not compulsory to deploy FPGAs. (ii) instead of using off-the-shelf FPGA cards, the FPGA cards must be designed from the scratch and manufactured by pruning unwanted peripherals. Most of the large FPGA deployments follow the second approach and develop their own FPGA cards to suit their infrastructure [25] [111]. However, these large-scale FPGA deployments still rely on a server node to host the FPGA cards. Even though the cost of the FPGA unit is reduced by a large margin, having a server to host those optimized FPGA cards inhibits the gain in overall system cost.

3.1.5 Power Efficiency

As discussed in Chapter 2, FPGAs are highly energy efficient computing resources compared with CPUs and GPUs. However, FPGAs being tightly coupled to a server node hinders the exploitation of its power efficiency due to the higher power consumption of the server node. Large-scale commercial FPGA deployments [25] and research on FPGA-accelerated applications [21] (Figure 3.2) show that the addition of an FPGA card increases the power consumption of a server node by 10%. Even though FPGAs are highly energy efficient, [21] shows that the server node plus FPGA combination becomes less energy efficient than multi-threaded CPUs due to the high power consumptions of the overall server system. But, when only the power consumption of the FPGA itself is considered, FPGAs are more power efficient than CPUs and GPUs.

This observation guides us to two directions to increase the overall power efficiency: (i) increase the utilization of the server node. By adding more jobs to the server CPU, the overall resource utilization can be increased. But, adding more jobs to the same CPU could affect the performance of the individual applications in terms of the processing speed. Therefore, careful workload placement is needed to exploit both the server and the FPGA resource in an optimal way. (ii) another option is to add more than one FPGA to the server node and offload as much processing as possible. But, as discussed in Section 3.1, adding more FPGAs to the server node could result in resource over provisioning in dynamic environments like cloud DCs. (iii) the third option is to decouple the FPGA from the server node and deploy them as standalone disaggregated computing resources. The disaggregation makes the FPGA completely independent in terms of the power consumption.

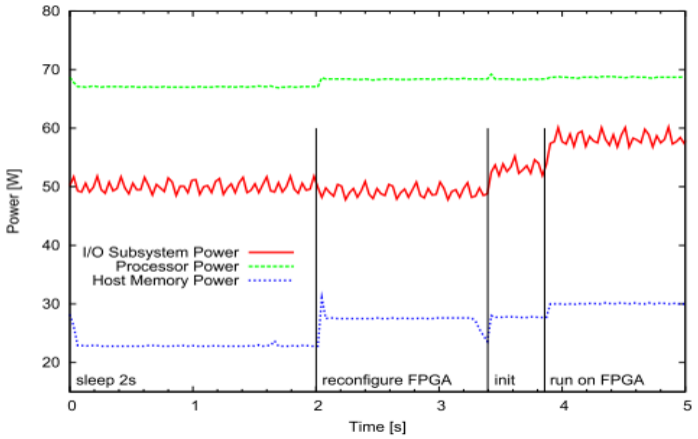


Figure 3.2: CPU, Memory, and I/O Power Consumption when a PCIe-attached FPGA is in Idle, Reconfiguration, and Computation Steps [21]

3.1.6 Homogeneity

Large-scale Internet companies host thousands or even millions of servers, storage, and network units in their DCs. Even a one second outage of these infrastructures cost them millions of dollars. Hence, maintaining those infrastructures in a high-available manner is important. However, maintaining such huge infrastructures are always costly, particularly in terms of the power and the labor. The infrastructure maintenance cost can be reduced by reducing the labor effort required and allowing more efficient problem detection and repair processes. The infrastructure being homogeneous helps to automate the maintenance work and also reduces specialized labor skills needed. Therefore, addressing these issues, DC operators design their infrastructure to be homogeneous, so that each main HW unit, server, storage, or network, across the DC can be maintained in a way requiring less labor effort.

When deploying FPGAs in the data centers, this homogeneity must not be broken. The obvious way to deploy FPGAs is by attaching them to servers via PCIe. As long as these PCIe attached FPGAs are available in each and every server, the server unit is homogeneous across the DC. However, at the DC level, having one FPGA per server prescribes a fixed ratio between FPGA and server resources, which is discussed in detail in Section 3.1.1. To allow a server to access more than one FPGA, either multiple FPGAs must be attached to a server through a PCIe switch using a PCIe-expansion chassis [22] or each PCIe-attached FPGA must be connected in a dedicated network [25]. In both these cases, a dedicated network is added to the DC, breaking the network homogeneity and in turn increasing the labor effort required to maintain that particular network. Further, addition of a new network increases the HW cost required for cabling and interfaces.

To maintain the network homogeneity, the FPGAs must be connected to the servers though the same networking mechanisms used for inter-server communications. TCP/IP/Ethernet is still the dominant networking protocols used in the DCs. Therefore, it is highly desirable for FPGAs to be connected to the commodity DC network.

3.1.7 Management

Once FPGAs are deployed in the DC, they must be managed similarly to other compute resources, such as servers, storage and network. Typically, these resources are virtualized and added to resource pools. According to the demand, the pooled resources are used to provision user requests. Once the resources are released by users, they go back to the respective pools until used again. In the DCs, the FPGAs must also be virtualized and pooled into resource pools, so that they can be used on demand by the user.

3.1.8 Summary

From the system requirements reviewed above, to satisfy flexibility, reliability, homogeneity and power efficiency, a new way for attaching FPGAs to CPUs is required. To satisfy scalability and system cost, we need a novel DC infrastructure to deploy FPGAs. To manage FPGAs in DCs, a resource management framework to manage and provision FPGAs to cloud users is needed. Based on this analysis, the next sections define the system architecture.

3.2 CPU-FPGA Attachment Interface

As shown in Figure 3.3, there are mainly three options to connect FPGA compute resources to the CPU-based server resources in the DC: (i) system bus-attached, (ii) PCIe-attached, and (iii) network-attached.

3.2.1 System Bus-Attached

One option is to place the FPGA on the same board as the CPU when a tight or coherent memory coupling between the two devices is desired (Figure 3.3-(a)). We do not expect such a close coupling to be generalized outside the scope of very specific applications. First, it breaks the homogeneity of the compute module in an environment where server homogeneity is sought to reduce the management overhead and provide flexibility across compatible hardware platforms. Second, in large DCs, failed resources can be kept in place for months and years without being repaired or replaced, in what is often referred to as a fail-in-place strategy. Therefore, an FPGA will become unusable and its resources wasted if its host CPU fails. Third, the footprint of the FPGA takes a significant real estate away from the compute module –the layout of a large FPGA on a printed circuit board is similar

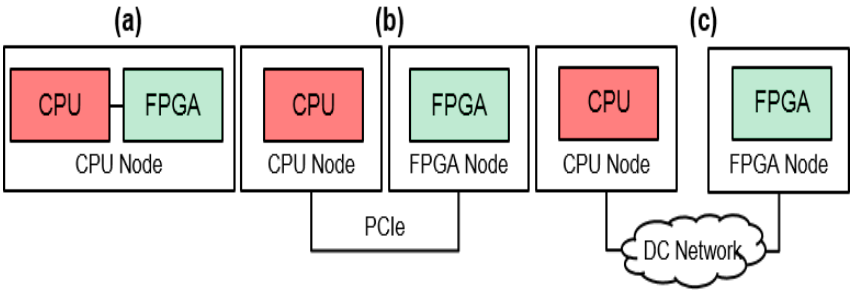


Figure 3.3: Options for Attaching an FPGA to a CPU

to the footprint of a DDR3 memory channel, i.e. 8-16GB–, which may require the size of the module to be increased (e.g., by doubling the height of the standard node board from 2U to 4U). Finally, the power consumption and power dissipation of such a duo may exceed the design capacity of a standard node board. The third and the final points explained above are particularly important when designing high-density DC infrastructures, which is explained in Section 3.4.3.

3.2.2 PCIe-Attached

The second and the most popular option in use today is to implement the FPGA on a daughter-card and communicate with the CPU over a high-speed point-to-point interconnect such as the PCIe-bus (Figure 3.3-(b)). This path provides a better balance of power and physical space and is already put to use by FPGAs [25] as well as graphics processing units (GPU) in current DCs. However, this type of interface comes with the following two drawbacks when used in a DC. First, the use of the FPGA(s) is tightly bonded to the workload of the CPU, and the fewer the PCIe-buses per CPU, the higher is the chance of under-provisioning the FPGA(s), and vice-versa. Catapult [25] uses one PCIe-attached FPGA per CPU and solves this inelastic issue by deploying a secondary inter-FPGA network at the price of additional cost, increased cabling and management complexity. Second, server applications are often migrated within DCs. The PCIe-attached FPGA(s) affected must then be detached from the bus before being migrated to a destination where an identical number and type of FPGAs must exist, thus hindering the entire migration process. Finally, despite the wide use of this attachment model in high-performance computing, we do not believe that is a way forward for the deployment of FPGAs in the cloud because it confines this type of programmable technology to the role of coarse accelerator in the service of a traditional CPU-centric platform.

3.2.3 Network-Attached

The third method for deploying FPGAs in DCs is to directly connect the FPGA to the DC network (Figure 3.3-(c)). The main implication of this scheme is that

the FPGA must be turned into a standalone DC resource, which can be connected and managed like any other compute node in the DC. From a practical point of view, this requires an FPGA module to be equipped with an FPGA, optional local memory, a management interface and a network controller interface (NIC). Adding a NIC to an FPGA enables that FPGA to communicate with other DC resources, such as servers, disks, I/O and other FPGA modules. Multiple such FPGA modules can then be deployed in DCs independently of the number of CPUs, thus overcoming the limitations of the two previous options.

The networking layer of such an FPGA module can be implemented with either a discrete or an integrated NIC. A discrete NIC (e.g., dual 10 GbE NIC) is a sizable application-specific integrated circuit (ASIC) typically featuring 500+ pins, 400+ mm^2 of packaging, and 5 to 15 W of power consumption. The footprint and power consumption of such an ASIC does not favor a shared-board implementation with the FPGA (see above discussion on sharing board space between an FPGA and a CPU). Inserting a discrete component also adds a point of failure in the system. Integrating the NIC into the reconfigurable fabric of the FPGA alleviates these issues and is becoming practical with the latest FPGA devices which can implement a 10 Gb/s network protocol stack in less than 5-10% of their total resources [115]. Normally, the Ethernet media access controller (MAC) of such a protocol stack connects to an external physical layer device (PHY) whose task is to perform encoding/decoding and serialization/deserialization functions, as well as a transceiver, such as the enhanced small form-factor pluggable transceiver (SFP+) whose task is to physically move the data bits over the media according to a specific physical layer standard.

Finally, the integrated version of the NIC provides the agility to implement a specific protocol stack on demand, such as Virtual Extensible LAN (VxLAN) [116], Internet Protocol version 4 (IPV4), version 6 (IPv6), Transmission Control Protocol (TCP), User Datagram Protocol (UDP) or Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE) [117]. Alternatively, it can also adapt to emerging new protocols, such as Generic Network Virtualization Encapsulation (Geneve) [118] and Network Virtualization Overlays (NVO3) [119].

3.2.4 Summary

In summary, we advocate a direct attachment of the FPGA to the DC network by means of an integrated NIC, and refer to such an FPGA as a standalone disaggregated FPGA. This paves the way towards using FPGAs in resource-centric DCs [120] [121] [122]. The combination of such standalone disaggregated FPGAs with emerging software-defined networking (SDN) technologies brings new technical perspectives and market value propositions, such as building large and programmable fabrics of FPGAs on the cloud.

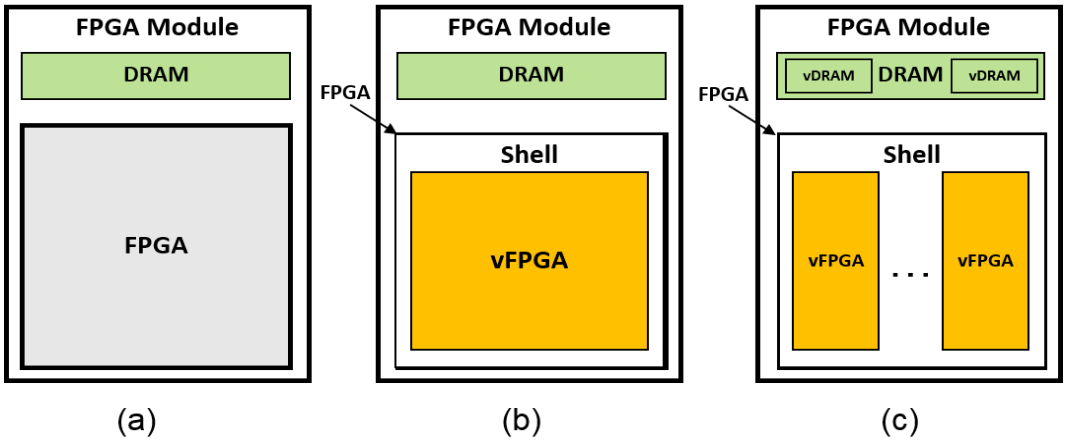


Figure 3.4: Different Ways of Resource Provisioning in the DC Based on Standalone Disaggregated FPGAs: (a) as a physical FPGA, (b) as a single virtual FPGA, (c) as multiple virtual FPGAs

3.3 FPGA Provisioning Methods

The reconfigurable logic and external memory resources of the standalone disaggregated FPGA can be partitioned and provisioned to the users in DCs in multiple different ways. The architectural aspects of the standalone disaggregated FPGA differs according to the way they are partitioned and provisioned. In this section, we explore the possible ways of provisioning FPGA resources and their architectural implications. As shown in Figure 3.4, the logic and memory resources of an FPGA card can be provisioned in 3 different ways: (i) as a physical FPGA, (ii) as a single virtual FPGA, and (iii) as multiple virtual FPGAs.

3.3.1 As a Physical FPGA

Provisioning as a physical FPGA (Figure 3.4-(a)) is the simplest approach. In this approach, the whole FPGA chip and the external memory is provisioned to the user. Once rented, the user has to implement his own communication layers to communicate with the FPGA and external memory over the DC network. While this approach offers the expert users the full flexibility to implement their applications, it would not be easy for the non-expert users to bring up the FPGA with the network and memory IO before placing his application. On the other hand, the infrastructure providers have to ensure that the DC infrastructure is not harmed by the possible malicious networks packets sent out by the FPGA.

This approach is good for low cost and low-density FPGAs, as FPGA would be used most probably for a single application. Hence, resource over provisioning would not occur. The choice to provision a physical FPGA also affects the management requirement. The user controls the complete FPGA and therefore, the

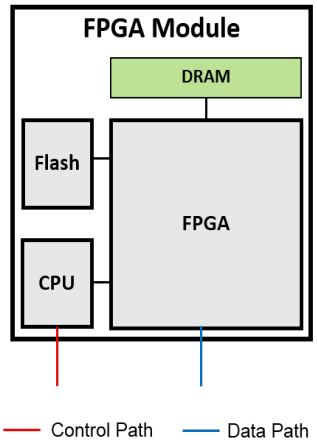


Figure 3.5: FPGA Module Architecture for Physical FPGA Provisioning

FPGA module needs to have a management agent outside of the FPGA chip to ensure that the FPGA module is powered up and can be programmed over the DC network. The path to this management agent (control path) must be fast enough so that FPGAs can be programmed and reprogrammed in a short period of time. Furthermore, the management agent must provide a rich set of features so that the FPGA module can be managed as a standard data center resource. This requires standard protocols such as TCP/IP to run on this management agent. Running such a protocol at an acceptable speed requires a CPU to be implemented on the FPGA module. Figure 3.5 shows a possible architecture for such an FPGA module.

3.3.2 As a Single Virtual FPGA

In this approach (Figure 3.4-(b)), the infrastructure vendor places a logic layer in the FPGA chip, which is called the SHELL. The rest of the logic resources are allocated for the ROLE or the user applications, which is also called the vFPGA. The shell provides the access to the DC network and to the memory resources for the vFPGA. The shell contains management agents as well, so that the FPGA module can be managed in the DC infrastructure, similarly to the other standard DC resources. The vFPGA is only offered to a single user at a moment.

This approach is also good for low-cost and low-density FPGA chips, as it would only be used for one application. The shell gives the control of the FPGA to the DC operator, which fulfills the reliability requirement by filtering and dropping malicious packets emerging from the ROLE. Having the shell within the FPGA eliminates the need for having a dedicated rich CPU in the FPGA module. Having a secondary control path is necessary to support the minimal control functions, such as to power up the FPGA and program the flash in case the default system image is crashed. The secondary control path does not need a high-speed network connection to the resource management software, as it is used only for control

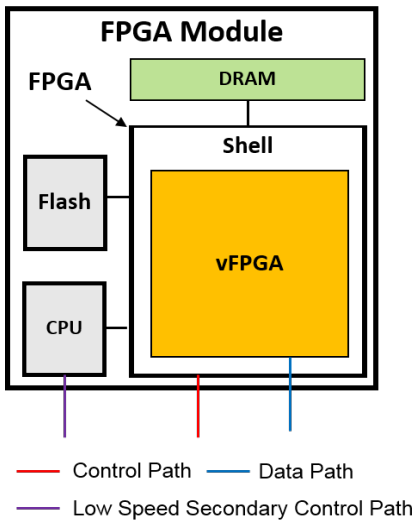


Figure 3.6: FPGA Module Architecture for Single vFPGA Provisioning

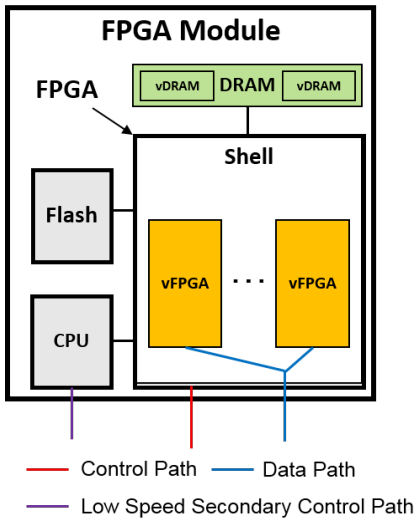


Figure 3.7: FPGA Module Architecture for Multiple vFPGA Provisioning

messages and rarely to send a bit stream. Hence to implement the secondary control path a low-power, low-cost micro controller can be used as shown in the Figure 3.6.

3.3.3 As Multiple Virtual FPGAs

In this approach (3.4-(c)), similarly to the approach explained in Section 3.3.2, the infrastructure vendor places a logic layer in the FPGA chip. The difference compared to the single virtual FPGA is that the remaining reconfigurable logic

resources are divided into multiple independent vFPGAs. This multi-vFPGA approach has been used in [100] [123]. In a multi-tenant environment, these multiple vFPGAs can be provisioned to multiple users. A vFPGA must always instantiate an interface to the network layer and the interface to the memory is optional. This approach is better for high cost and high-density FPGAs, as they have plenty of logic resources available. But, to ensure vFPGAs can be independently offered to multiple users, partial reconfiguration has to be used. The same external data path must be internally virtualized to provide the DC network access for the multiple vFPGAs as shown in Figure 3.7 and also the external DRAM must also be partitioned so that each vFPGA gets is own portion of the physical memory.

3.3.4 Summary

In summary, standalone disaggregated FPGAs can be partitioned and provisioned in multiple different ways and each approach has its advantages and disadvantages. For DC deployment, the single vFPGA provisioning is preferable as it (a) gives the infrastructure operator a better control of the FPGA module and (b) it better fits low-cost and low-density FPGAs, which are the desirable device option in the DCs. In the section 3.4, we define a system architecture for deploying disaggregated standalone FPGAs based on the single vFPGA approach.

3.4 Infrastructure for Deployment

In this section, at first the evolution of server and cloud data centers are reviewed. Next, different deployment options are compared with respect to the requirements laid out in Section 3.1. We also compare a deployment based on off-the-shelf HW to a density-optimized hyperscale DC infrastructure.

3.4.1 Evolution of Cloud Data Centers

The first-generation data centers were built using traditional, non-virtualized servers. These servers were not shared among multiple applications or large enough groups of users, which decreased the resource utilization. In addition to that, the DC resources were server-centric and tightly coupled to the server. For example, the storage could not be expanded beyond the physical limitations of the server, which hosted the respective disks. Lack of management tools and immaturity of virtualization techniques made on demand server provisioning impractical in the first generation DCs. It could take an enterprise months to deploy new applications requiring too much manual intervention. This forced IT organizations to overprovision resources to ensure that adequate spare computing capacity existed to satisfy demand spikes, provide disaster recovery and meet failover requirements further decreasing the resource utilization. Even though backup servers consume a fraction of the power, hundreds or thousands of idle servers collectively require a large amount of power and significant amount of space [124]. In the data centers

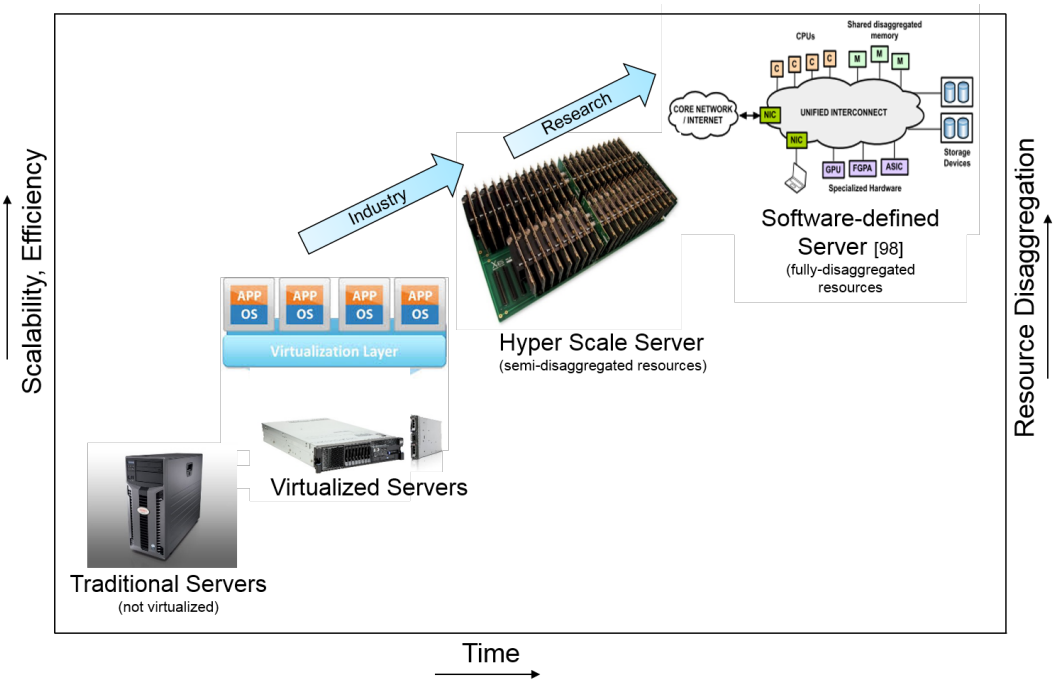


Figure 3.8: Server and Cloud Data Center Trends

built using these type of servers, the server utilization is typically in the range of 7-10%.

Between 2003 and 2010, virtualized servers, virtualized storage and software-defined networking (SDN) emerged. These virtualization technologies enabled the emergence of software-defined data centers (SDDC). SDDCs can pool the resources of the computing, network and storage to create a central, more flexible resource that can be reallocated based on needs. However, the physical infrastructure behind the SDDC did not change much. Also, the processors are not optimized for the diverse scale-out workloads resulting in suboptimal performance [125] [39] [126]. In addition to that, the virtualization overhead due to hypervisors started to become a performance issue [127]. Even though we did not see much change in the physical architecture and the resources were still mainly server-centric, resource disaggregation has slowly started [128] with storage and memory [129] being placed in the network decoupled from the servers. In the data centers built using virtualized servers, the server utilization is typically in the range of 7-18%.

Rapid development of management tools and overhead of server virtualization made the IT organizations to consider non-virtualized servers again [130]. Even though, at present, those servers can be better used for certain set of applications with rich management tools and programming models, the efficiency and the scalability is far below compared to the processing demands from the future applications. Meanwhile a new class of servers called ”hyperscale servers” are

being emerged. The hyper scale servers are built using low-cost, low-performance embedded processors [131], replacing server-class processors. Hyperscale processors consume 5 to 30 times less power than server-class counterparts and are 5 to 15 times more energy-efficient. Further, their absolute performance lags that of server-class processors by factors of 5 to 50 [110]. However, these hyperscale server processors are also limited in floating point performance and SIMD acceleration capabilities, making them ill-suited for real-time analytics with heavy computational components [110] [132]. Applications which ran on server-class processors have already been started to port into hyperscale servers [133]. But due to the lack of rich computational features in the processor, to run applications at server-class performance or even more, heterogeneous computing resources such as FPGAs can be used. When running such applications at scale, these heterogeneous resources can be disaggregated in addition to storage and memory for large scale deployment. The real advantage of hyperscale servers comes from the density. They take very small space compared to rack and blade servers which allows to densely pack thousands of servers in a single rack.

Coarse-grained resource disaggregation first started with storage. At present it has evolved up to memory and I/O [122]. Moving a step further, at the time of writing this, it is being discussed in the research community about the fourth generation data centers which will be built using fine-grained disaggregated resources [120]. The servers in these data centers are software-defined and they will be built dynamically restructuring the disaggregated resources [121] [134]. Here, fine-grained disaggregation means in addition to storage and memory, even the processor, I/O and Network interface card are decoupled from their original arrangements and organized into shared pools. These disaggregated resources are connected to each other using a high-speed interconnect [121] [135]. Based on fine-grained disaggregation, flexible run-time application deployment can be realized with optimized resource utilization. Moreover, fine-grain resource disaggregation gives in-detailed insights into the resources which helps further enhance data center optimization mechanisms such as improving PUE [136].

3.4.2 FPGA Cluster Built with Off-the-Shelf HW

Figure 3.9 and 3.10 show a potential FPGA cluster that can be built using off-the-shelf HW. The FPGA cluster consists of three main components: (i) FPGA chassis, (ii) Server Chassis, and (iii) top of rack Ethernet switch. FPGA chassis is based on a off-the-shelf PCIe-expansion chassis, such as from cyclone [22], which can host up to 18 PCIe devices. Each FPGA chassis has an associated server chassis. The FPGA chassis is built by deploying off-the-shelf PCIe FPGA cards on this chassis. For this design we assume a alphadata ADM-PCIE-KU3 FPGA card, featuring a Xilinx Kintex Ultrascale XCKU060 FPGA, two 10 GbE connections and 16 GB of DRAM. In this configuration, each FPGA is connected to the top of rack switch with 10 GbE connection (Figure 3.10). Each FPGA card has a USB connection to

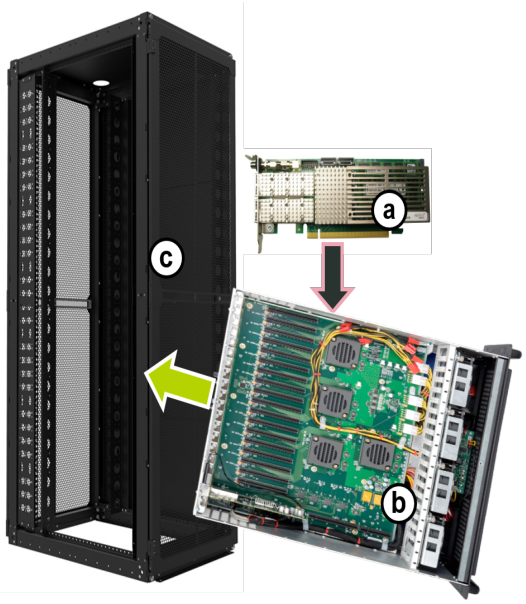


Figure 3.9: Building an FPGA Rack Using off-the-shelf HW: (a) FPGA module with 16 GB DRAM. (b) 2U Rack chassis [22] with 18 FPGA modules. (c) 42U Data Center Rack

program the FPGA, which is connected to the associated server chassis. A server chassis is used to manage the FPGAs in the FPGA chassis. The server chassis and FPGA chassis are interconnected using a PCIe-expansion cable. In the server chassis, all the FPGAs in the FPGA chassis are seen as directly connected to the server.

3.4.3 Hyperscale FPGA Cluster

The servers, which make up a cloud DC, are continuously shrinking in terms of the form factor. This leads to the emergence of a new class of hyperscale data centers (HSDC) based on small and dense server packaging. Miniaturized servers aim to leverage the advanced semiconductor manufacturing processes for the gates by integrating a complete server system on chip (SoC) for increased density and power efficiency. As a result, legacy memory controllers and high-speed I/Os are embedded on chip, thus eliminating the need for an external PCIe bus to support these I/Os. This miniaturization of the DC servers will transform the traditional way of deploying and operating an FPGA in a DC infrastructure. The shrinking DC form factor unit will enable the deployment of a large number of standalone disaggregated FPGAs, exceeding by far the scaling capacity of traditional PCIe bus attachments.

At the time of writing, there are several HSSs on the market [137] [138] [139] and at the research stage [3] [140]. Among these, the HSS of DOME [3] has the objective

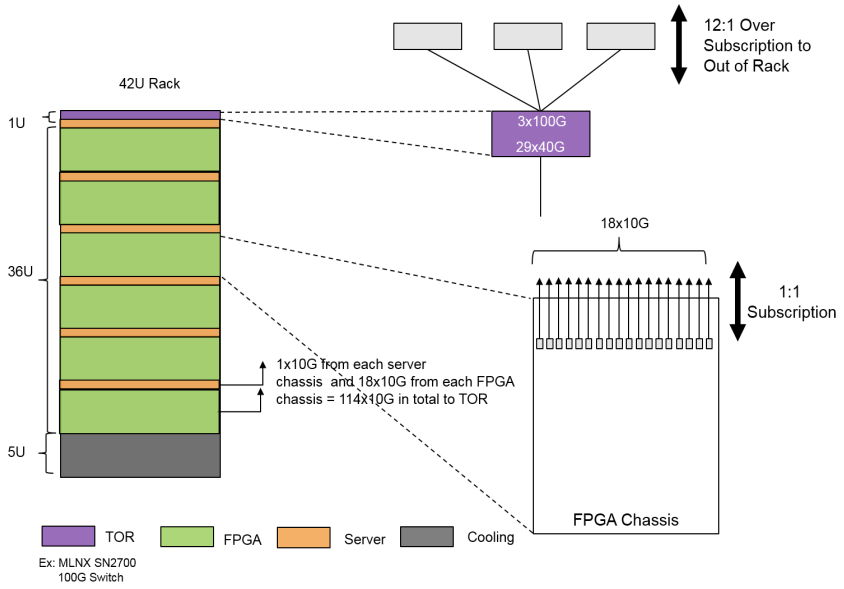


Figure 3.10: 108 FPGAs in a Rack Using off-the-shelf HW

of building the world's highest density and most energy efficient rack unit, and in this work we use that rack unit for deploying FPGAs.

3.4.3.1 FPGA Chassis

Figure 3.11 shows the packaging concept of the HSS rack chassis (19" by 2U[1]) proposed in [3]. This HSS is disaggregated [120] into multiple FPGA modules (Figure 3.11-(a)), each the size of a double-height dual in-line memory module (DIMM - 133mm x 55mm), which are densely plugged into a carrier base board (Figure 3.11-(b)).

Each FPGA module connects via a 10 GbE link to the south side of an Intel FM6000 Ethernet L2/L3/L4 switch, for a total of 320 Gb/s of aggregate bandwidth. The north side of the FM6000 switch connects to eight 40 GbE up-links, which expose the FPGA cluster to the DC network with another 320 Gb/s. This provides a uniform and balanced (no over-subscription) distribution between the north and south links of the Ethernet switch, which is desirable when building large and scalable fat-tree topologies (a.k.a. Folded Clos topology). The Ethernet switch provides the same aggregate throughput as a top-of rack switch (i.e. 640 Gb/s) and was shrunk down to the size of a smart phone (140 × 62 mm) in order to vertically fit in a 2U height chassis. A fully populated carrier board is referred to as a sled. Its various I/O voltage rails are generated by two shared power controllers

[1] 1U = one rack unit = 1.75 inches (44.45 mm)

Table 3.1: FPGA Rack Performance: Hyperscale vs Off-the-Shelf HW and State-of-the-art

Per Rack	Hyperscale	Off-the-Shelf HW	MS Catapult [1]
FPGA	XCKU060	XCKU060	Stratix V D5
FPGAs/Rack	1024	108	96
LUTs (10^6)/Rack	340	36	33
FFs (10^6)/Rack	680	72	66
FPGA:Host	1:0	18:1	1:1
FPGA:Memory(GB)	1:16	1:16	1:8
FPGA:Network BW(Gb/S)	1:10	1:10	1:10

(Figure 3.11-(b)), and the entire sled is managed by a 64-bit T4240 communication processor from Freescale running Fedora 23.

As shown in Figure 3.12-(a) and (b), two sleds fit a 19″ × 2U chassis, for a total of 64 FPGA modules. Figure 3.12-(c) shows the 10 GbE and 40 GbE interconnection network between the various connectors of such an assembly. The chassis implements two identical sleds, each consisting of the following interconnects: the red wiring within a sled corresponds to 10 GbE links connecting the 32 FPGA modules to the south side of the FM6000 Ethernet. The blue wiring within a sled corresponds to 40 GbE up-links connecting the north side of the same Ethernet switch to 8 Quad Small Form-factor Pluggable (QSFP) transceivers. The yellow wirings are 10 GbE links which provide a low-latency ring topology between every four neighboring FPGA modules of a given sled. The green wiring also consists of 10 GbE links that interconnect two sleds together for providing a redundant path to failover from the Ethernet switch of one sled to the switch of the neighbor sled. Finally, the black wiring between pairs of neighboring slots provides a PCIe x8 Gen3 interface.

The FPGA platform achieves its high packaging density by implementing a module every 7.6 mm. This very small stride does not allow for air-cooled heatsinks and fans. Instead, we deployed a combination of a passive cooling solution [141] at the FPGA module level that is coupled to an actively cooled element at the chassis level. Our implementation is done by replacing the FPGA cap with a custom made heat spreader that allows the transport of the thermal energy laterally from the chip away to the borders of the module board where the heat spreader is then coupled to an active hot-water cooled [142] heat sink. This passive heat sink is built using standard PCB lamination processes and materials.

Since the hyperscale FPGA chassis presented in section 3.4.3 is directly connected to the DC network as an FPGA appliance, it can be deployed in DCs in two different scenarios. First, Figure 3.13 shows an example configuration, where 1024 FPGAs are deployed in a 42U data center rack with 16 2U chassis. Out of total sixteen 40 GbE connections, eight are connected to the top of rack switch with 2:1 over-subscription at the chassis level. We assume Mellanox SN2700 as the top of

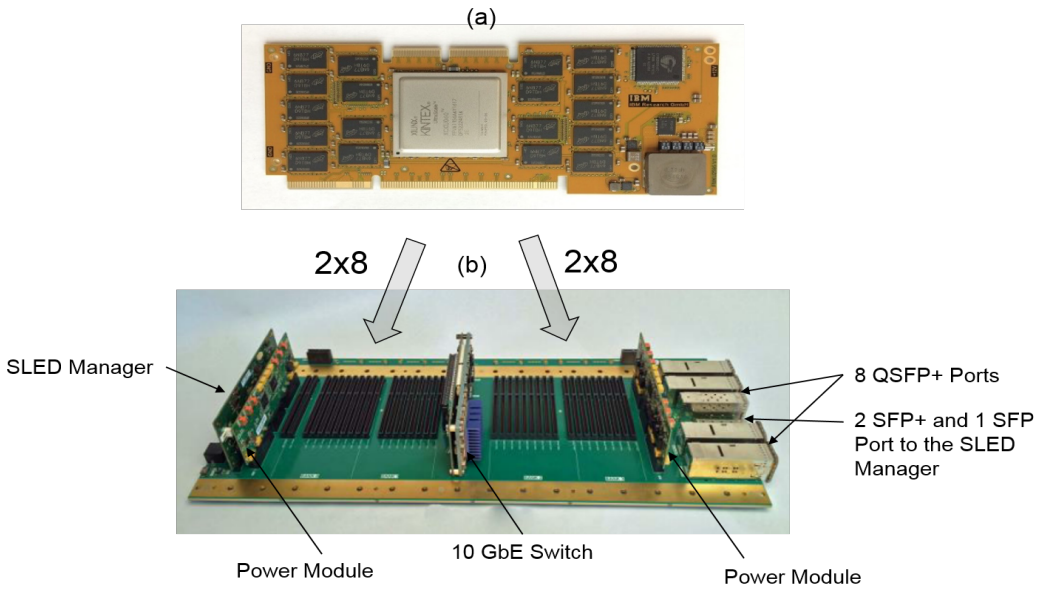


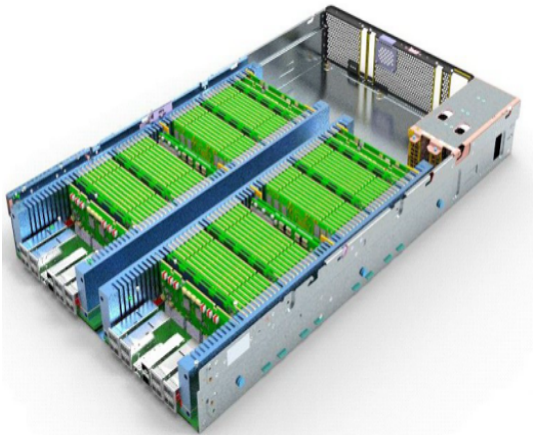
Figure 3.11: Hyperscale FPGA Packaging [23]: (a) FPGA Module (b) SLED that hosts 32 FPGA Modules

rack switch and there are 5 of them as shown in the Figure. With this configuration, all 1024 FPGAs are connected to out of the rack with 5:1 over-subscription. Second, Figure 3.14 shows another deployment scenario, where few FPGA appliances are deployed with multiple 2U server chassis in the same rack. Based on the application and the infrastructure, these two deployment scenarios can be used appropriately. However, the second deployment scenario is the ideal configuration for a general-purpose cloud deployment, as the first method needs some redesigning effort for the existing DC infrastructures.

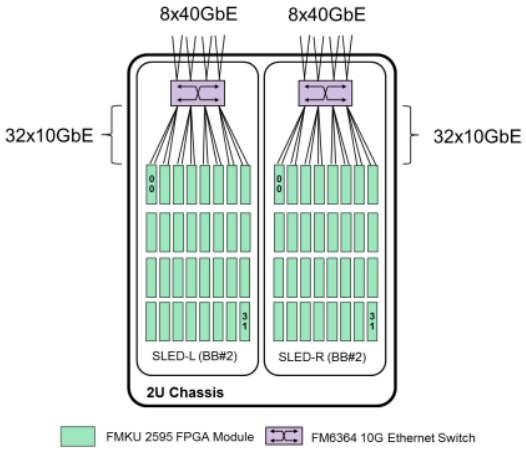
Table 3.1 compares the performance of an FPGA rack built by hyperscale FPGA, off-the-shelf HW and state-of-the-art Microsoft Catapult V2 [1] deployment. For the first two cases, Xilinx Kintex Ultrascale (XCKU060) FPGA is used, where as Catapult V2 uses Altera Stratix V D5 FPGA. In terms of rack FPGA density (FPGAs/Rack), hyperscale FPGA outperforms both off-the-shelf and Microsoft Catapult deployment by around 10 times.

3.5 Cloud Integration

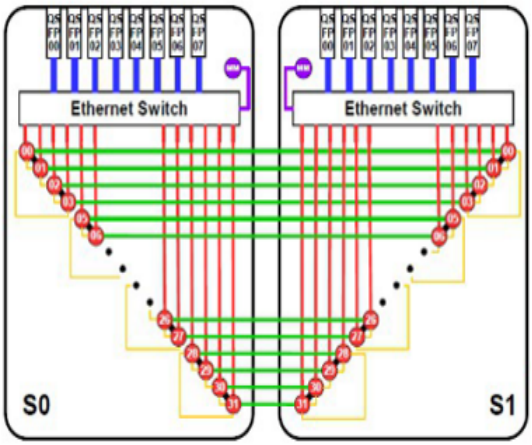
Cloud integration is the process of making DC resource like the standalone disaggregated FPGAs available in the cloud so that users can rent them. In this section, cloud computing is reviewed first. Then we present a framework for integrating FPGAs in the cloud (the cloud layer in Figure 3.17). We propose a new accelerator service for OpenStack [143], a way to integrate FPGAs into OpenStack, a way to



(a)



(b)



(c)

Figure 3.12: FPGA Chassis with two SLEDs [23]: (a) Physical View (b) Logical View (c) Network Wiring

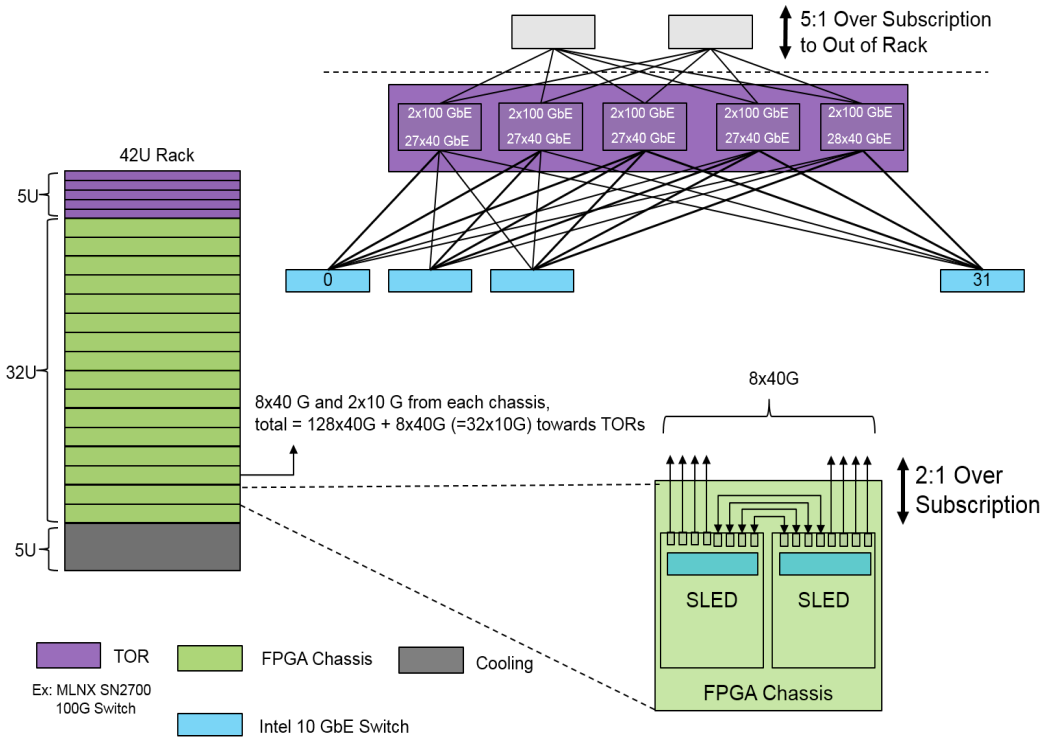


Figure 3.13: 1024 FPGAs in a Single Rack with 2:1 Over Subscription at Chassis Level and 5:1 Over Subscription at Rack Level

provision FPGAs on the cloud, and a way for the user to rent an FPGA on the cloud.

3.5.1 Cloud Computing

In traditional computing (3.15-(a)), applications are deployed on user owned computing infrastructures hosted on-premise. Users have to maintain the whole IT stack and pay for HW, power, cooling, space and SW licensing. Architecturally, the traditional IT stack is single-tenant limiting it to a single application and a single user group.

In contrast to traditional computing, computing is delivered over the network as a service in cloud computing. From the resource point of view, mainly three aspects are new in cloud computing [144]. First, the illusion of infinite computing resources available on demand eliminating the need for cloud users to plan ahead for future resource requirements. Second, the elimination of an up-front commitment by cloud users allowing the companies to start small and increase compute resources only when it is needed. Third, the ability to pay for use of computing resources on a short-term basis as needed and release them when not needed

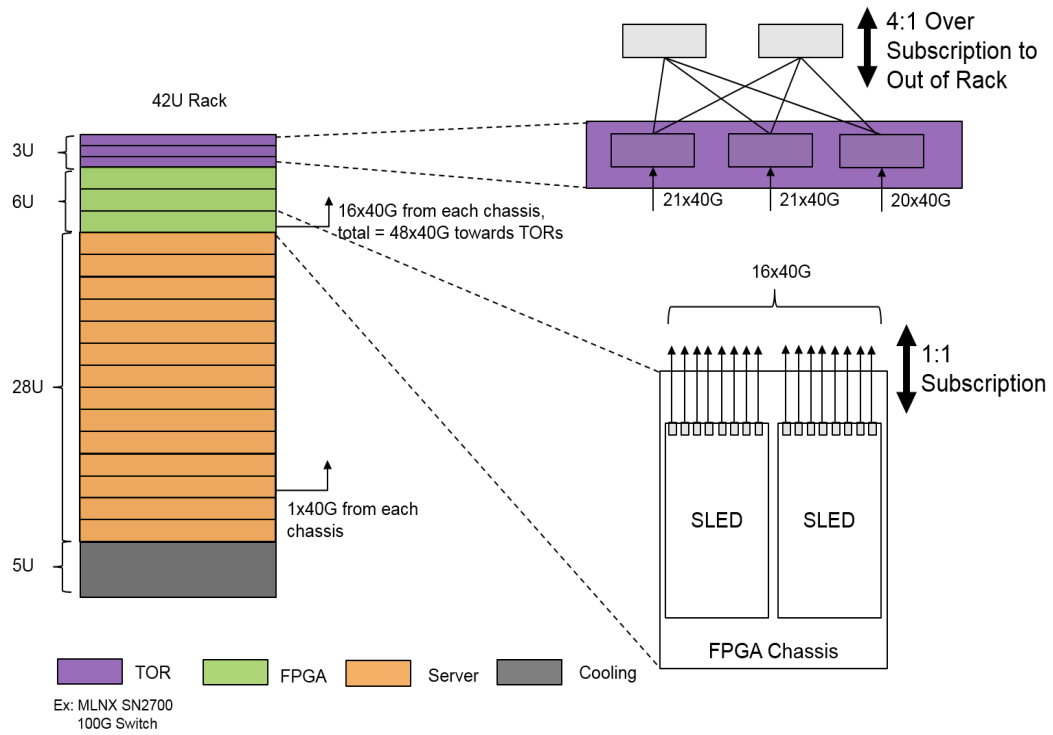


Figure 3.14: 192 FPGAs in a Single Rack with 1:1 Subscription at Chassis Level and 4:1 Over Subscription at Rack Level

allowing the computing resources to go when they are no longer useful.

Based on how the computing infrastructure is deployed and the service is delivered to the users, cloud computing can be classified into several service and deployment models.

3.5.1.1 Service Models

Service delivery models are defined according to the way computing is serviced to the users and the ownership of the different parts of the IT stack as shown in Figure 3.15.

1. **Infrastructure as a Service (IaaS)**

   In IaaS model (Figure 3.15-(b)), IT infrastructure including HW compute resources and operating system are provided as a service by the cloud vendors. Users deploy their application software on the rented infrastructure. Cloud providers typically bill IaaS services on a utility computing basis. The cost reflects the amount of resources allocated and consumed. In addition to general-purpose compute resources such as servers, storage and network,

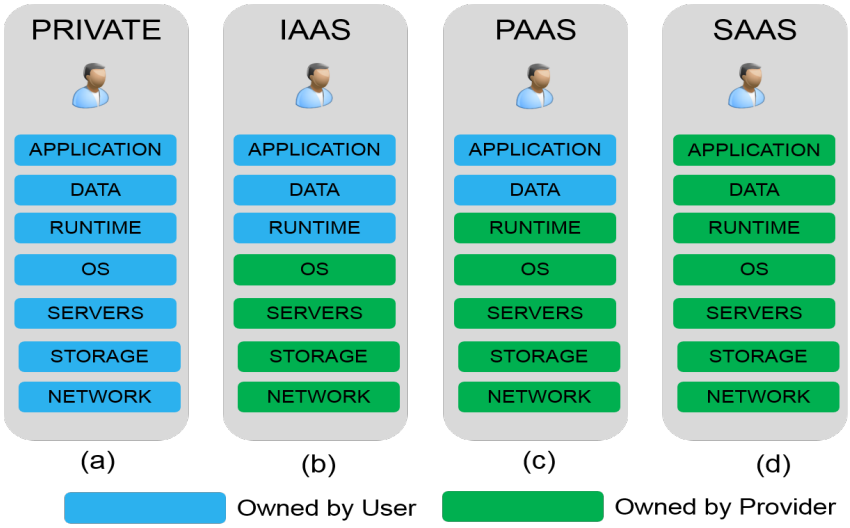


Figure 3.15: Cloud Service Delivery Models

heterogeneous compute devices such as GPUs and FPGAs can also be provisioned in this model [145] [111]. Amazon Web Services, Microsoft Azure, Google Compute Engine, RackSpace and IBM SoftLayer are few of the leading IaaS vendors.

2. **Platform as a Service (PaaS)**

   In PaaS model (Figure 3.15-(c)), IT infrastructure including HW compute resources, operating system and application runtime environment typically including programming language execution environment, databases, web servers and vendor specific tools are provided as a service by the cloud vendors. Application developers can develop and run their software applications on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software resources. Microsoft Azure, Google App Engine and IBM BlueMix are some popular PaaS solutions in the industry.

3. **Software as a Service (SaaS)**

   In SaaS model (Figure 3.15-(d)), users are provided the access to use application software over the network. Cloud providers manage the infrastructure HW, operating system and platforms and the application software. Search engines and email services from Google, Microsoft and Yahoo, social networking services such as Facebook, Twitter and Linkedin and customer service management (CRM) solutions from salesforce.com are some widely used SaaS solutions.

3.5.1.2 Deployment Models

Deployment models are defined, as shown in Figure 3.16, according to the existing place and the ownership of the computing infrastructure from the user's viewpoint.

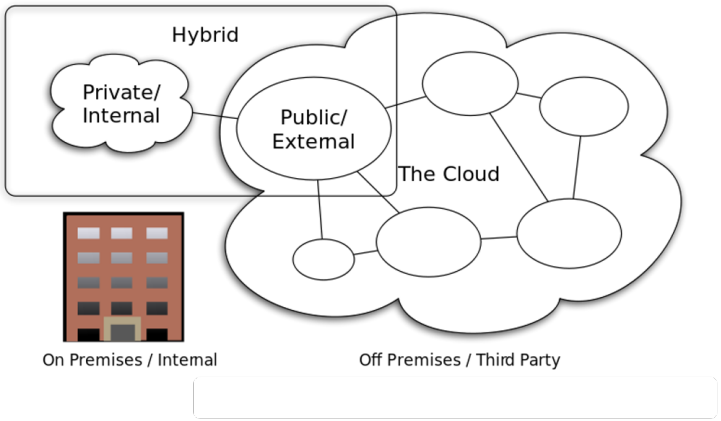


Figure 3.16: Cloud Deployment Models [24]

1. **Private Cloud**

   Private cloud is a cloud infrastructure operated solely for a single organization, whether managed and hosted internally or externally. From the perspective of security, private cloud is regarded as the best deployment model. In the private cloud, organizations can provide computing services internally, for example, to the users of their own institutes. This model is better for use cases where higher level of security can be traded off for expensive infrastructure budget.

2. **Public Cloud**

   In contrast to private cloud, public cloud infrastructure is open for public use. Technically, there may be little difference between public and private cloud architecture. However, security considerations are tight in public cloud because of the access by public over a non-trusted network such as Internet.

3. **Hybrid Cloud**

   Hybrid cloud is a composition of two or more clouds (private, public) that remain distinct entities but are virtually bound together, offering the benefits of multiple deployment models. Private cloud infrastructure can be used for data and applications which has higher security requirements while public cloud can be used for other applications.

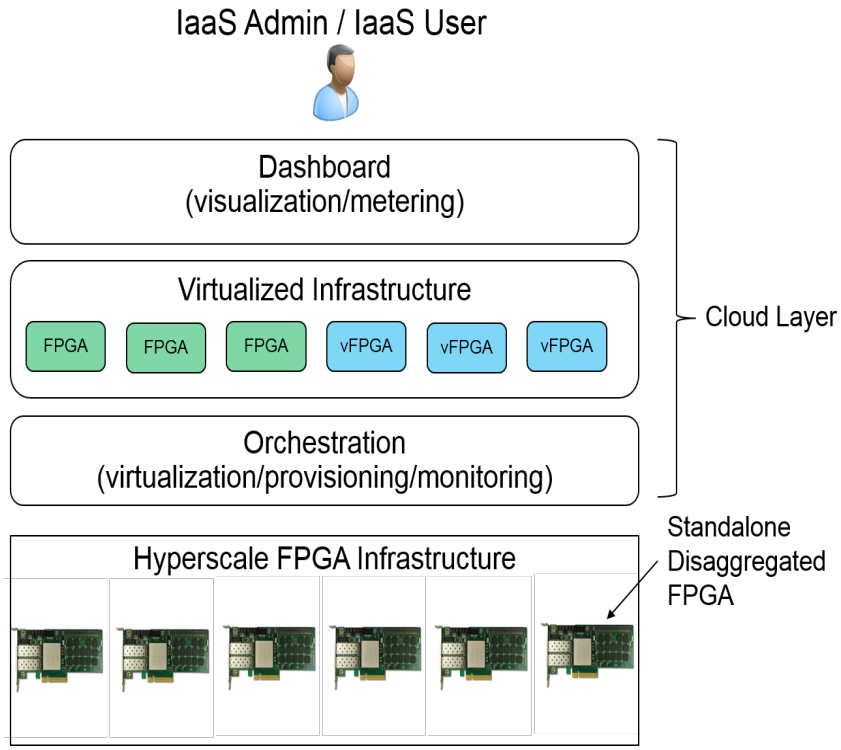


Figure 3.17: Conceptual View of Virtualized Disaggregated FPGA Infrastructure

3.5.2 Accelerator Service for OpenStack

We propose a new service for OpenStack to enable standalone disaggregated FPGAs in IaaS service delivery model in both public and private cloud. Figure 3.17 shows a conceptual view of the standalone disaggregated FPGAs in hyperscale DCs in a cloud setup. In previous research, FPGAs [123] [100] and GPUs [146] have been integrated into the cloud by using the *Nova* compute service in OpenStack. In those cases, heterogeneous devices are PCIe-attached and are usually requested as an option with virtual machines or as a single appliance, which requires a few simple operations to make the device ready for use.

In our deployment, in contrast, standalone FPGAs are requested independent of a host because we want to consider them as a new class of compute resource. Therefore, like *Nova*, *Cinder* and *Neutron* in OpenStack, which translate high-level service API calls into device-specific commands for compute, storage and network resources, we propose the accelerator service shown in Figure 3.18, to integrate and provision FPGAs in the cloud. In the figure, the parts in red show the new extensions we propose for OpenStack. To setup network connections with the standalone FPGAs we need to carry out management tasks. For that, we use an SDN stack connected to the *Neutron* network service, and we call it the network

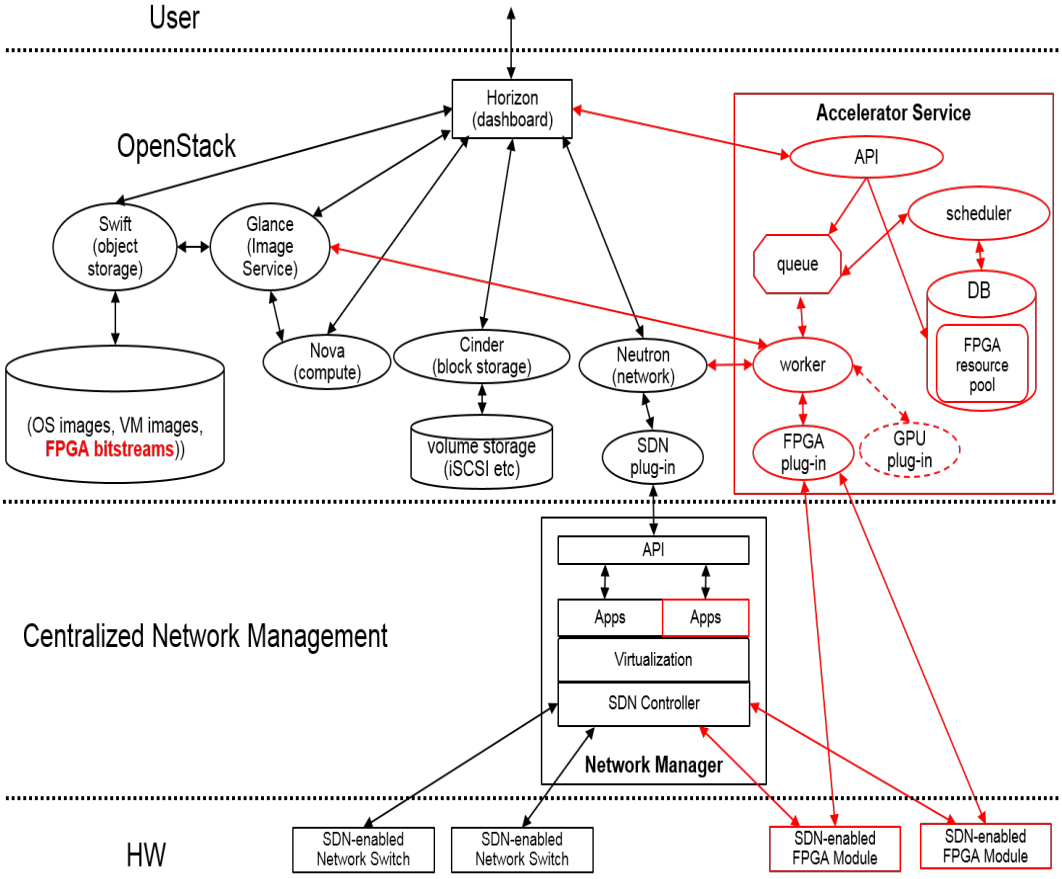


Figure 3.18: OpenStack Architecture with Standalone Disaggregated FPGAs

manager. Here we explain the high-level functionality of the accelerator-service and the network-manager components.

3.5.2.1 Accelerator Service

The accelerator service comprises an API front end, a scheduler, a queue, a data base of FPGA resources (DB), and a worker. The API front end receives the accelerator service calls from the users through the OpenStack dashboard or through a command line interface, and dispatches them to the relevant components in the accelerator service. The DB contains the information on pFPGA resources. The scheduler matches the user-requested vFPGA to the user logic of a pFPGA by searching the information in the DB and forwards the result to the worker. The worker executes four main tasks: *i*) registration of FPGA nodes in the DB; *ii*) retrieving vFPGA bit streams from the *Swift* object store; *iii*) forwarding service calls to FPGA plug-ins, and *iv*) forwarding network management tasks to the network manager through the *Neutron* service. The queue is just there to pass service calls

between the API front end, the scheduler and the worker. The FPGA plug-in translates the generic service calls received from the worker into device-specific commands and forwards them to the relevant FPGA devices. We foresee the need for one specific plug-in per FPGA vendor to be hooked to the worker. Other heterogeneous devices like GPUs and DSPs will be hooked to the worker in a similar manner.

3.5.2.2 Network Manager

The network manager is connected to the OpenStack *Neutron* service through a plug-in. The network manager has an API front end, a set of applications, a network topology discovery service, a virtualization layer, and an SDN controller. The API front end receives network service calls from the accelerator-worker through the *Neutron* and exposes applications running in the network manager. These applications include connection management, security and service level agreements (shown in red in the network manager in Figure 3.18). The virtualization layer provides a simplified view of the overall DC network, including FPGA devices, to the above applications. The SDN controller configures both the FPGAs and network switches according to the commands received by the applications through the virtualization layer.

3.5.2.3 Integrating FPGAs into OpenStack

In this sub section, the process of integrating FPGAs into OpenStack is outlined. The IaaS vendor executes this process as explained below.

When the IaaS vendor powers up an FPGA module, the ML of the FPGA starts up with a pre-configured IP address. This IP address is called the management IP. The accelerator service and the network manager use this management IP to communicate with the ML for executing management tasks. Second, the standalone disaggregated FPGA module is registered in the accelerator-DB in the OpenStack accelerator service. This is achieved by triggering the registration process after entering the management IP into the accelerator service. Then the accelerator service acquires the FPGA module information automatically from the ML over the network and stores them in the FPGA resource pool in the accelerator-DB. Third, a few special files, as explained in Section 4.3.3, is needed for vFPGA bitstream generation are uploaded to the OpenStack *Swift* object store.

3.5.2.4 Provisioning an FPGA on the Cloud

From the IaaS vendors' perspective, let's now look at the process of provisioning a single vFPGA. When a request for renting a vFPGA arrives, the accelerator-scheduler searches the FPGA pool to find a user logic resource that matches the vFPGA request. Once matched, the tenant ID and an IP address are configured for

the vFPGA in the associated pFPGA. After that, the vFPGA is offered to the user with a few special files which are used to generate a bitstream for user application.

3.5.2.5 Renting an FPGA on the Cloud

From the user's perspective, the process of renting a single vFPGA on the cloud and configuring a bitstream to it is as follows. First, the user specifies the resources that it wants to rent by using a GUI provided by the IaaS vendor. This includes FPGA-internal resources, such as logic cells, DSP slices and Block RAM as well as module resources, such as DC network bandwidth and memory capacity. The IaaS vendor uses this specification to provision a vFPGA as explained above.

Upon success, a reference to the provisioned vFPGA is returned to the user with a vFPGAID, an IP address and the files needed to compile a design for that vFPGA. Second, the user compiles his design to a bitstream and uploads it to the OpenStack *Swift* object store through the *Glance* image service. Finally, the user associates the uploaded bitstream with the returned vFPGAID and requests the accelerator service to boot that vFPGA. At the successful conclusion of the renting process, the vFPGA and its associated memory are accessible over the DC network.

3.5.2.6 Multi-FPGA Fabrics on the Cloud

Motivated by the success of large-scale SW-based distributed applications, such as those based on MapReduce and deep learning [40], we want to give the users a possibility to distribute their applications on a large number of FPGAs. This sub section describes a framework for interconnecting such a large number of FPGAs in the cloud that offers the potential for FPGAs to be used in large-scale distributed applications.

We refer to a multiple number of FPGAs connected in a particular topology as a multi-FPGA fabric. When the interconnects of this fabric are reconfigurable, we refer to it as a programmable fabric. Users can define their programmable fabrics of vFPGAs on the cloud and rent them using the proposed framework. Figure 3.19 shows such two fabrics in which vFPGAs with different sizes are shown in different patterns. These two fabrics are used to build two different types of applications. As an example, a fabric of FPGAs arranged in a pipeline, shown in Figure 3.19-(a), is used in Catapult [25] for accelerating page-ranking algorithms, which we discussed in prior art. Figure 3.19-(b)) shows a high-level view of a fabric that can be used for map-reduce type of operations.

3.5.2.7 Renting a multi-FPGA Fabric on the Cloud

The renting and provisioning steps of such a fabric in the cloud are as follows. First, user decides on the required number of vFPGAs and customizes them as mentioned above in the case of a single vFPGA. Second, the user defines its fabric

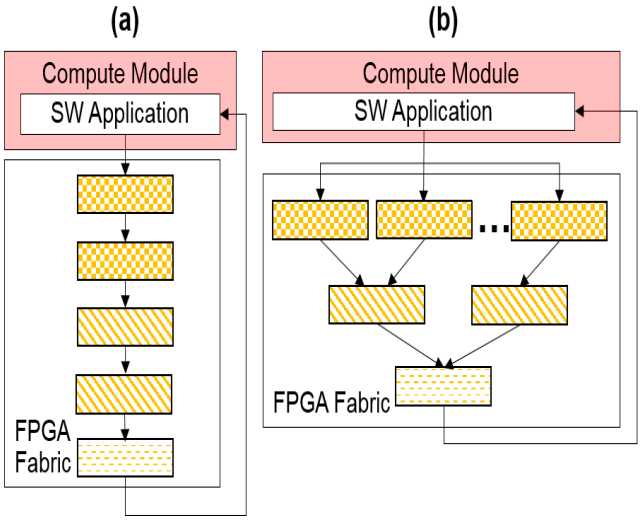


Figure 3.19: Two Examples of multi-FPGA Fabrics

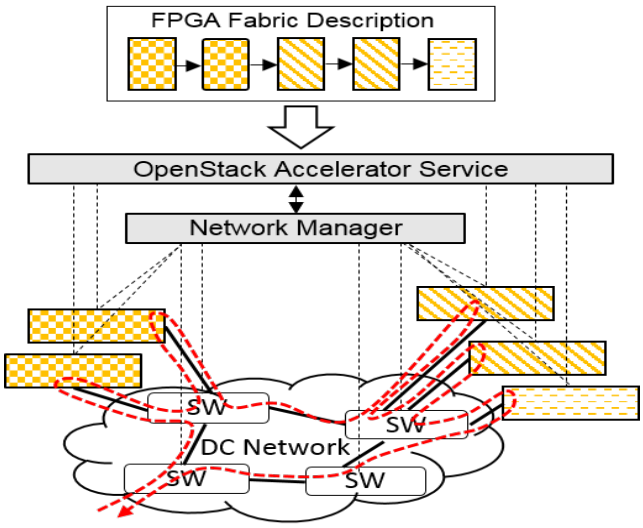


Figure 3.20: FPGA Fabric Deployment; SW: Network Switch

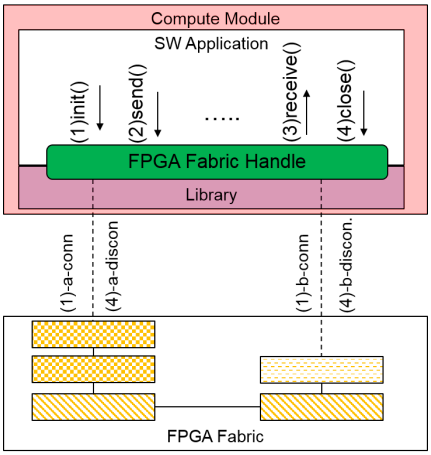


Figure 3.21: Multi-FPGA Fabric Programming Model

topology by connecting those customized vFPGAs on a GUI or with a script. We call this fabric a vFPGA Fabric (vFF). In vFF, the number of network connections between two vFPGAs can be selected. If a network connection is required between a vFPGA and the SW applications that uses the vFF (explained in the next sub section), it is also configured in this step. Third, the user rents the defined vFF from the IaaS vendor. At this step, the user-defined fabric description is passed to the OpenStack accelerator service. Then, similar to a single vFPGA explained earlier, the accelerator service matches the vFF to the hardware infrastructure as shown in Figure 3.20. In addition to the steps followed when matching a single vFPGA, the scheduler considers the proximity of vFPGAs and optimal resource utilization when matching a vFF to the hardware infrastructure. After that, the accelerator service requests the network manager to configure the NSL of assoicated pFPGAs and intermediate network switches to form the fabric in HW infrastructure. Fourth, the user associates a bitstream with each vFPGA of the vFF and requests to boot the fabric. Finally, on successful provisioning, an ID representing the fabric (vFFID) is returned to the users that is used in the programming phase to access the vFF.

3.5.2.8 Using a multi-FPGA Fabric from SW Applications

The way a vFF is used from a SW application is explained here. We consider the pipeline-based vFF shown in Figure 3.19-(a) as an example and show how it can be used from a SW application. We assume this fabric runs an application based on data-flow computing. The text-analytics acceleration engine explained in [147] is an example of such an application. Also, we assume that the L4 protocol used is a connection-oriented protocol such as TCP.

To make the applications agnostic to the network protocols and to facilitate the programming, we propose a library and an API to use both the vFPGAs and vFFs.

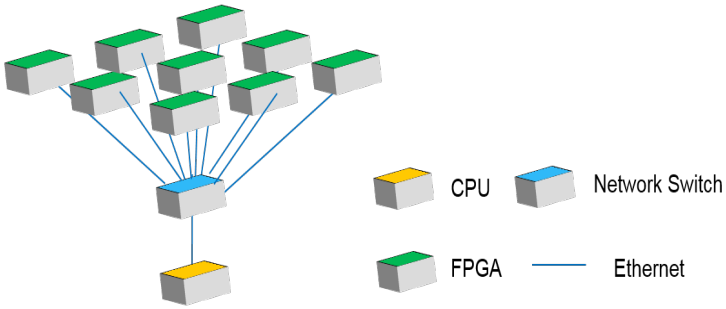


Figure 3.22: Standalone Disaggregated FPGA Deployment Architecture

The vFFID returned at the end of the fabric-deployment phase is used to access the vFF from the SW applications by means of the vFFID. Below are the steps for accessing a vFF. First, the vFF is initialized from the SW application. This initiates a connection for sending data to the vFF as shown by (1)-a-conn in Figure 3.21. The immediate SDN-enabled switch triggers this connection-request packet and forwards it to the network manager. On behalf of the first vFPGA in the pipeline, the network manager establishes the connection and updates the relevant FDB entries in the associated pFPGA. For receiving data, the library starts a local listener and tells the network manager to initiate a connection on behalf of the last vFPGA in the pipeline ((1)-b-conn). Then, the SW application can start sending data and receiving the result by calling send() and receive() on the vFFH, respectively. If configured by the user at the vFF definition stage, connections are created for sending back intermediate results from the vFPGA to the SW application. When close() is called on the vFFH, the connections established are closed detaching the fabric from the SW application. The connections for accessing memory associated with each vFPGA are also established in a similar manner through the network manager in the fabric initialization phase. The SW applications can write to and read from the memory using the vFPGAID.

3.6 Summary

To enable large-scale deployment of FPGAs in DCs, we advocate for a change of paradigm in the CPU-FPGA and FPGA-FPGA interfaces. We propose an architecture that sets the FPGA free from the CPU and its PCIe-bus by connecting the FPGA directly to the DC network as a standalone disaggregated resource (Figure 3.22). Cloud vendors can then provision these FPGA resources in a similar manner as the CPU, memory and storage resources.

Meanwhile, the servers, which make up a cloud DC, are continuously shrinking in terms of the form factor. This leads to the emergence of a new class of hyperscale data centers (HSDC) based on small and dense server packaging. This miniaturization of the DC servers is a game-changing requirement that will transform the

traditional way of instantiating and operating an FPGA in a DC infrastructure. Shrinking of the DC form-factor unit enables the deployment of a large number of disaggregated FPGAs, exceeding the scaling capacity of traditional PCIe bus attachments. Compared with state-of-the-art FPGA deployments, the hyperscale approach proposed in this thesis increases the FPGA rack density by 10 times to 1024 FPGAs per rack.

Once such disaggregated FPGAs become available on a large scale in DCs, vendors can rent them out on the cloud. However, as existing server-provisioning mechanisms are not suitable for this purpose, we propose a new resource-provisioning service in OpenStack for integrating such standalone disaggregated FPGAs into the cloud. Also, as cloud users will be able to request multiple FPGAs from the cloud, we provide them with the possibility to implement a programmable interconnection network of FPGAs in a cost-effective, scalable and flexible manner on the cloud. Therefore, we also propose a framework to interconnect multiple FPGAs in a user-defined topology, and for the cloud vendor to deploy such a topology in its infrastructure. We expect this software-defined approach of FPGA networking to offer new technical perspectives and solutions for processing large and heterogeneous data sets in the cloud.

Chapter 4

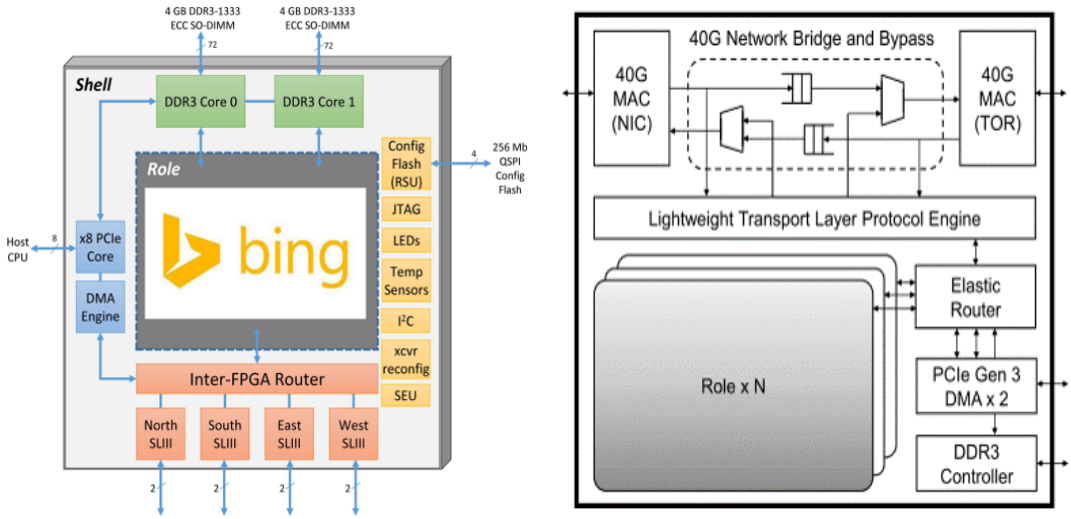
Standalone Disaggregated FPGA

In this chapter, an architecture and its prototype implementation is presented for the standalone disaggregated FPGA. This chapter is organized as follows: The section 4.1 reviews state-of-the-art shell-role architectures, which abstracts FPGA I/O. The section 4.2 elaborates on the architecture of the proposed standalone disaggregated FPGA and section 4.3 explains the implementation of the prototype. The implementation of the simulation environment for the standalone disaggregated FPGAs is described in section 4.4, followed by the experimental results in section 4.5. Section 4.6 discusses the experiment results and the chapter is summarized in section 4.7.

4.1 Abstracting FPGA I/O with Shell-Role Architectures

In typical FPGA programming environments, for example, an off-the-shelf FPGA card deployed in a private compute cluster, the user is often responsible for developing not only the application itself but also building and integrating system functions required for memory access, host-to-FPGA as well as inter-FPGA communication. When offering FPGAs as a service this approach is not feasible as: (i) infrastructure vendor must keep the control of the device for better management and security, (ii) bringing up of system functions is an additional burden for the users, and (iii) system functions are not portable from device to device, particularly when I/Os and FPGA architecture changes.

FPGA shells allow for faster coding of applications by removing the need to develop system-related (I/O) FPGA hardware by the application developers. FPGA I/O is abstracted and provided using pre-configured I/O components, allowing FPGA developers to focus on their applications. Furthermore, the shell allows to keep the control of the FPGA HW in the hands of infrastructure vendor. This approach is common for all the FPGA-based solution vendors. Microsoft catapult [25], IBM CAPI [26], Xilinx streaming interface, and Amazon F1 instance [27] provide these shells. Shell characteristics are tightly coupled with how the FPGA



(a) (b)

Figure 4.1: Microsoft Shell Architecture: (a) Catapult V1 Shell [25], (b) Catapult V2 Shell [1]

is attached to the CPU. The common feature in all these implementations is that the shell has a PCIe interface to the host CPU and shell provides the access to one or more multiple resource types of PCIe, DRAM, and network. In all these implementations, FPGA is managed via PCIe or JTAG interface.

4.1.1 Microsoft Catapult Shell

Microsoft Catapult V1 shell (Figure 4.1-(a)) contains 6 main components, consuming 23% (Table 4.1) of ALM resources of an Altera Stratix V FPGA [25]: (i) two DRAM controllers, which can be operated independently or as a unified interface. Dual-rank DIMMs operate at 667 MHz, whereas single-rank DIMMs (or only using one of the two ranks of a dual-rank DIMM) operates at 800 MHz, (ii) four high-speed serial links running SerialLite III (SL3), a lightweight protocol for communicating with neighboring FPGAs. It supports FIFO semantics, Xon/Xoff flow control, and ECC, (iii) router logic to manage traffic arriving from PCIe, the role, or the SL3 cores, (iv) reconfiguration logic, based on a modified remote Status Update (RSU) unit, to read/write the configuration Flash, (v) the PCIe core, with the extensions to support DMA, (vi) Single-event upset (SEU) logic, which periodically scrubs the FPGA configuration state to reduce system or application errors caused by soft errors.

The Catapult V2 shell (Figure 4.1-(b)) contains 6 components, consuming 44% of ALM resources of an Altera Stratix V D5 FPGA [1]: (i) two 40 G network controllers, one facing top-of-rack switch and the other facing the sever NIC, (ii) a

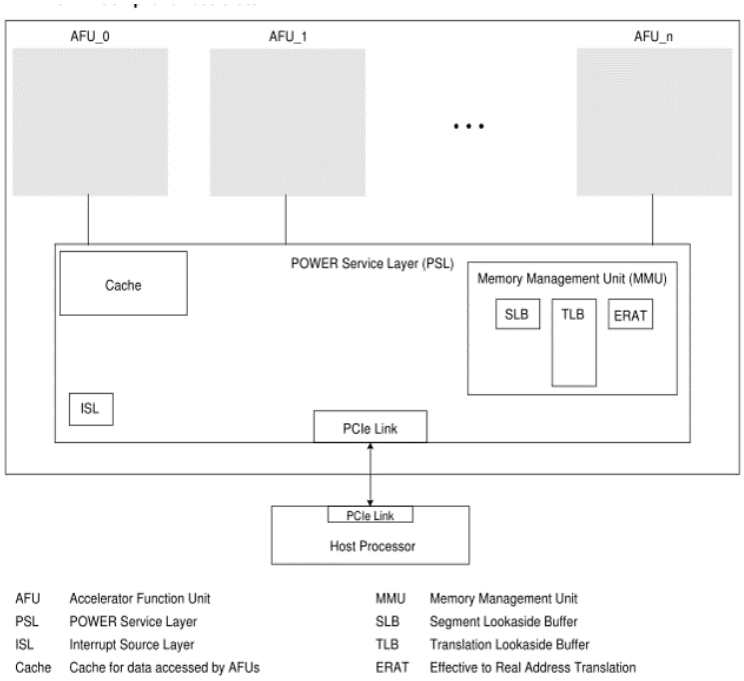


Figure 4.2: IBM Power Service Layer Shell [26]

40 G network bridge, (iii) a DDR3 memory controller, (iv) an intra-FPGA message router (elastic router) with virtual channel support for allowing multiple roles to access to the network, (v) LTL (Lightweight Transport Layer) protocol engine based on UDP for reliable inter-FPGA communication and (vi) PCIe controller with DMA engine.

4.1.2 IBM Power Service Layer Shell

IBM power service layer (PSL) shell (Figure 4.2), which runs at 250 MHz, contains four components, consuming around 25% (Table 4.1) of Altera Stratix V FPGA [112]: (i) PCIe controller for host-FPGA interaction, (ii) 256 B cache for accelerator function units (AFU), (iii) an interrupt source layer (ISL) for sending interrupts from the AFUs to the host applications, and (iv) memory management unit for virtual-to-physical memory translations, offering user the same virtual address space as in the host application.

4.1.3 Amazon EC2 F1 Shell

Amazon EC2 F1 shell (Figure 4.3) contains two components [27]. The resource consumption of the shell was not available publicly at the time of this writing. The two components are: (i) PCIe controller for host-FPGA interaction, (ii) a memory controller for DDR4-based DRAM access.

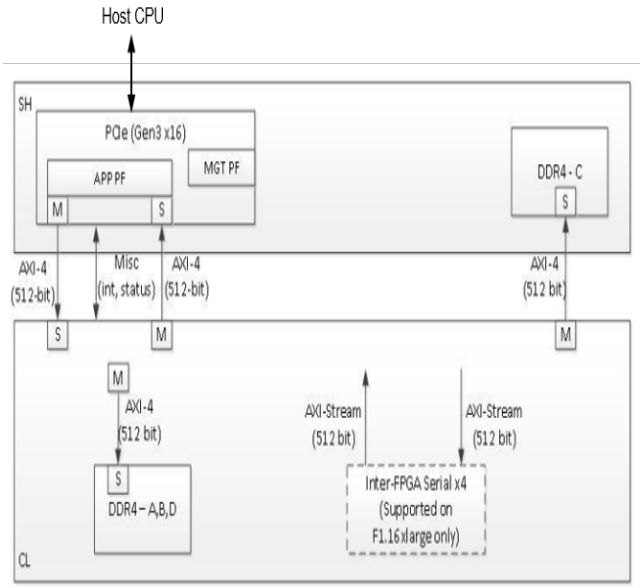


Figure 4.3: Amazon EC2 F1 Instance Shell [27]

4.1.4 Xilinx Donut Shell

Xilinx donut shell (Figure 4.4) contains two main components. The resource consumption of the shell was not available publicly at the time of this writing. The two components are: (i) PCIe controller for host-FPGA interaction, (ii) a performance counter.

4.1.5 NetFPGA SDN Shell

NetFPGA is a platform for research and education in the networking domain. There are many research projects around it and hence there is no dedicated shell architecture for it. But, SDN is the main area of research around NetFPGA. Figure 4.5 shows an example shell for such a SDN implementation [28]. In this particular example, the shell contains four 1 GbE controllers and a PCIe controller to interact with the host.

4.1.6 Summary

The common feature of all the shell-role architectures reviewed above is the PCIe-block, which is used to attach the FPGA to the CPU. The other components except for the PCIe block mostly depends on the scope of the target applications of each shell-role implementation. Typically, all these shells consume more than 20% of the resources of an FPGA (Table 4.1). The next section elaborates on the design of a shell-role architecture for standalone disaggregated FPGAs.

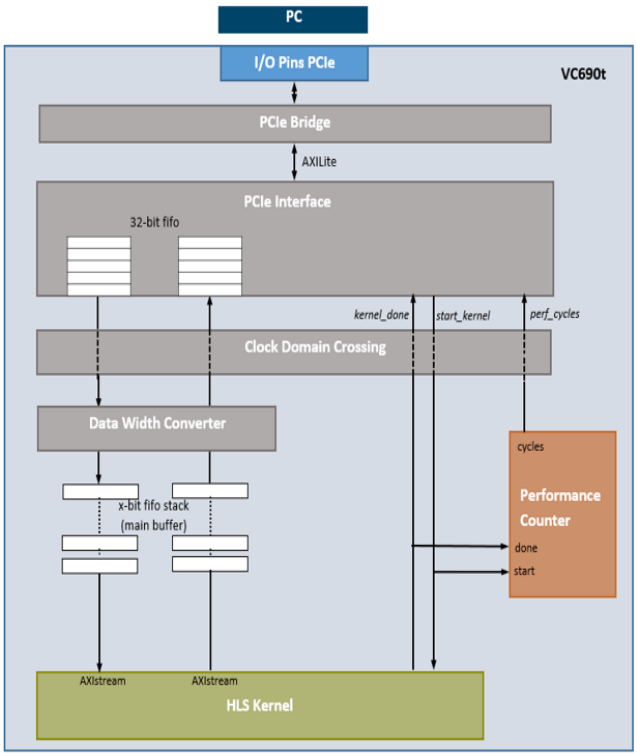


Figure 4.4: Xilinx Donut Shell

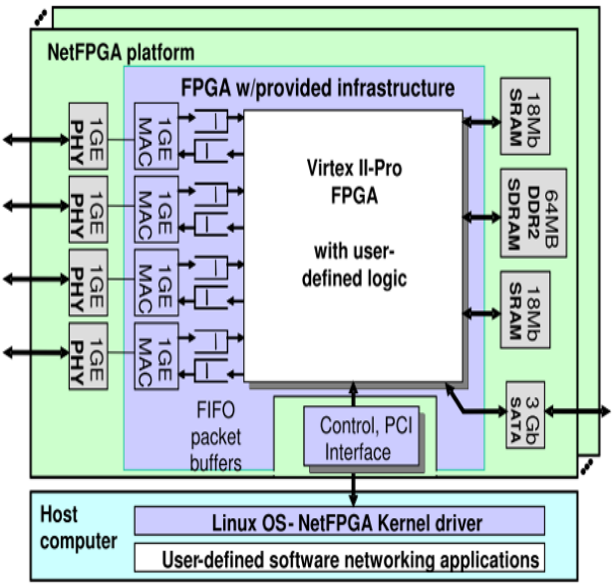


Figure 4.5: NetFPGA SDN Shell [28]

Table 4.1: Resource Consumption of State-of-the-art Shell Architectures

Shell	FPGA	ALM[1]	LUT	FF	BRAM
Microsoft Catapult V1	Altera Stratix V D5	39698	-	-	-
Microsoft Catapult V2	Altera Stratix V D5	76010	-	-	-
IBM PSL	Xilinx Kintex Ultrascale 60	-	54945	75661	281
Amazon F1	Xilinx Virtex Ultrascale+	-	-	-	-
Xilinx Donut	Xilinx Virtex 7	-	48147	323049	880
NetFPGA SDN	Xilinx Virtex-II Pro 50	-	-	-	-

[1] Adaptive Logic Module.

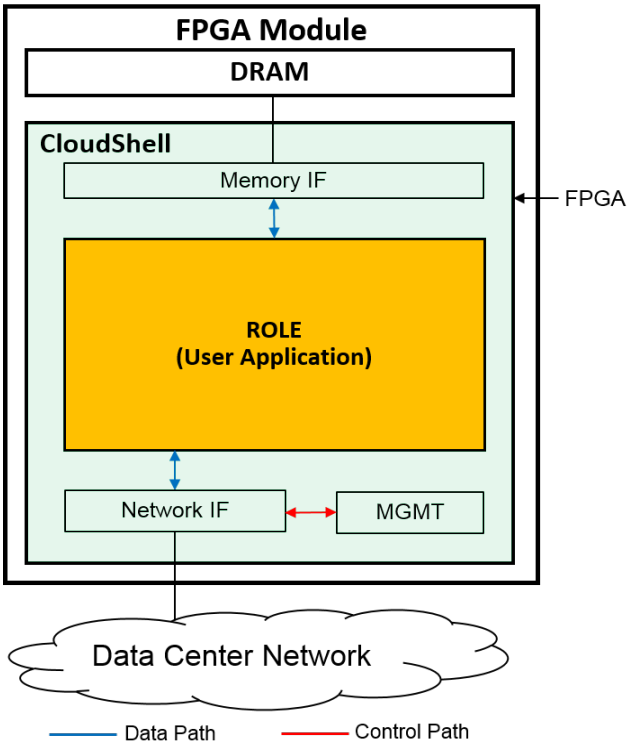


Figure 4.6: Standalone Disaggregated FPGA Architecture

4.2 Standalone Disaggregated FPGA Architecture

Figure 4.6 illustrates the high-level system architecture of the proposed standalone disaggregated FPGA. We define a shell-role architecture in the floorplan of the FPGA. The *shell*, which is called the cloud shell from here on, is persistent as long as the FPGA is powered up. It implements the minimum functions required for the FPGA to boot and to connect to the DC network. The *role* region is shown in yellow in Figure 4.6. It represents the larger part of the reconfigurable logic

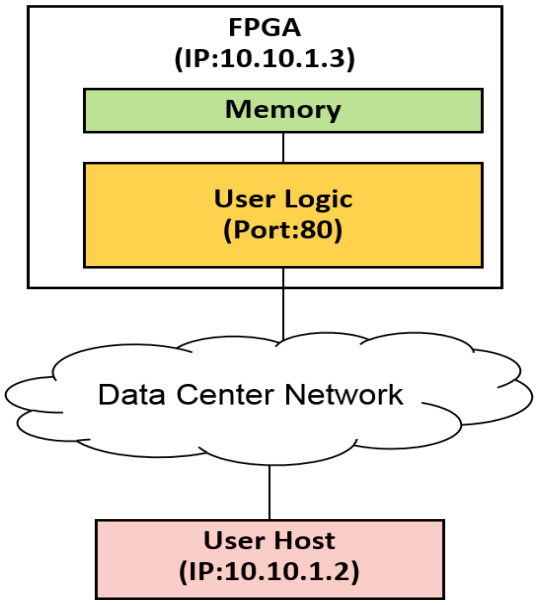


Figure 4.7: The Abstraction Offered Over the Network

and is dynamically allocated to the user's applications by a resource-provisioning service, which control the programming and erasing of this region. Figure 4.6 also illustrates the multiplexing and de-multiplexing performed by the network IF of the cloud shell between the user data traffic and the management traffic. The Figure 4.7 shows the abstraction offered to the DC user based on this architecture, where once the FPGA is rented and the user application is deployed, the user logic is accessible over a particular IP/port pair.

4.2.1 User Application (vFPGA)

The user application gets the most of the reconfigurable logic resources from the FPGA chip. The cloud shell provides the necessary infrastructure to run an application in the user region. This include multiple I/O channels to the DC network and to the external memory along with the clock.

4.2.1.1 I/O Channels

I/O channels to the user application are FIFO-based and they are mainly divided into two: (i) network I/O and (ii) memory I/O.

4.2.1.1.1 Network I/O There are one or more I/O channels to the user application. Each I/O channel to the network is distinguished by a 4-tuple, which consists of a source IP address, destination IP address, source port and destination port. The number of I/O channels to the user application depends on the application

requirement, and it can be specified when user requests/rents the FPGA. For example, the I/O channels to the DC network can be in the form of UDP/IP, TCP/IP, RoCE or any other protocol which is supported by the surrounding cloud shell.

Initialization of the network ports can be done in two ways: (i) the user application initialize before starts communication and (ii) the cloud shell initializes the ports and only provides data I/O to the user application. If the user application initializes the ports, similarly to a typical SW application running in Linux, at run time it has the option to decide and connect or listen to a desired server or client by specifying the 4-tuples. If the surrounding cloud shell initializes the network ports for the user application, the user has to specify the desired network connections when renting the FPGA and then the cloud shell along with the FPGA management software has to bring up the requested connections for the user application. Both options have its advantages and disadvantages. The first option is the simplest to implement for the cloud shell, but it adds some extra work for the user, while the second option gives much more control for the cloud shell in terms of underlying network infrastructure. For example, knowing the details of network connections in advance, gives the cloud shell and the FPGA management SW to implement advance networking features, such as QoS.

Network I/O channels can be used to communicate application data as well as the application status for monitoring purposes and to send back intermediate results to the original application. The data width can be the same as of cloud shell or data width converter blocks can be used from FPGA vendor tools.

4.2.1.1.2 Memory I/O The number of I/O channels to the memory depends on the total available memory channels in the FPGA module and the number available channels out of the total channels to the user application. If the cloud shell does not use some memory channels to implement its own logic, all the available total memory channels are offered to the user application.

4.2.1.2 Logic Resources

After the cloud shell has been implemented, the user application gets rest of the FPGA's reconfigurable logic resources. The bigger the FPGA chip, the higher the logic resources available for the user application, as the cloud shell usually requires a constant amount of logic resources independent of the FPGA chip. From the evolution of the FPGA technology, we observe that as the functions get matured they are increasingly implemented as hard IP cores in the FPGAs. If the soft IP cores used in the cloud shell, such as the network controller and memory controller are available in the FPGAs as hard IP cores, that will make more reconfigurable logic resources available to the user application. For example, Xilinx Ultrascale devices have integrated 100 GbE MAC and PHY subsystems [148].

4.2.1.3 Frequency

The user application can use the same clock frequency as the cloud shell or can use a different frequency. When using a different frequency either cloud shell should provide this clock or the application has to convert the cloud shell provided clock to its desired clock.

4.2.2 Cloud Shell

The cloud shell implements 3 functions, which include a memory management layer (MEM) for interfacing with external memories, a network layer (NET) for interfacing with the DC network, and a management unit (MGMT) for monitoring and controlling the standalone disaggregated FPGA resource. These three functions are represented with white boxes in Figure 4.6.

4.2.2.1 MEM IF

MEM IF provides the access to the memory for the user application. MEM IF may consists one or more memory controllers according to the available memory channels in the FPGA module.

4.2.2.2 NET IF

NET IF provides the access to the DC network for the user applications. The predominant L2 network in the DCs is Ethernet, hence to be compatible with existing DC network infrastructures the L2 network of the NET IF is based on Ethernet. At the time of writing this thesis, the typical DC Ethernet speed is 10G, but moving rapidly towards 40 G and 100 G. The FPGAs today are also rapidly moving beyond 10 G towards 40 G [149] and 100 G [148]. Therefore, at first, the NET IF must at least support 10 G and then move towards higher bandwidths. When moving towards 40 G and 100 G usually much more reconfigurable logic resources are needed, but the FPGAs now being shipped by major vendors have moved the PHY and MAC to hard IP cores, leaving more logic resources for the actual application logic.

4.2.2.2.1 FPGA Integrated MAC and PHY The networking layer of such an FPGA module can be implemented with either a discrete or an integrated NIC. A discrete NIC (e.g., dual 10 GbE NIC) is a sizable application-specific integrated circuit (ASIC) typically featuring 500+ pins, 400+ mm^2 of packaging, and 5 to 15 W of power consumption. The footprint and power consumption of such an ASIC do not favour a shared-board implementation with the FPGA (see above discussion on sharing board space between an FPGA and a CPU). Inserting a discrete component also adds a point of failure in the system. Integrating the NIC into the reconfigurable fabric of the FPGA alleviates these issues and is becoming

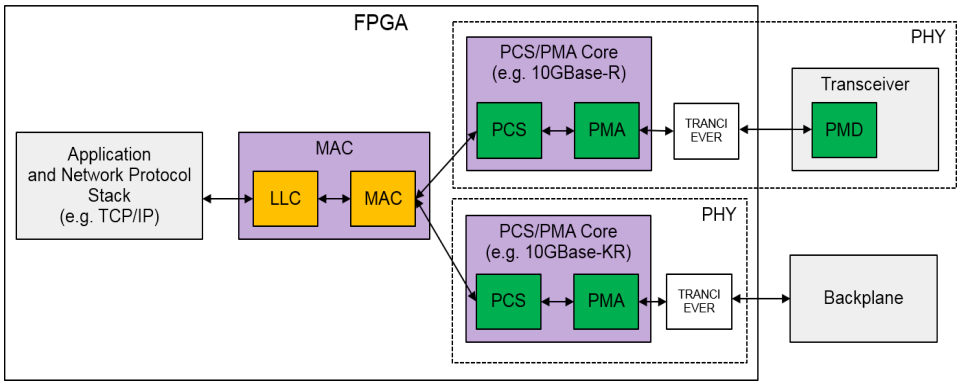


Figure 4.8: Two ways of using integrated PHYs in FPGA: (i) Base-R with External Transceiver and (ii) Base-KR Connecting to Backplane without External Transceiver

practical with the latest FPGA devices which can implement a 10 Gb/s network protocol stack in less than 5-10% of their total resources [115].

Normally, the Ethernet media access controller (MAC) of such a protocol stack connects to an external physical layer device (PHY) whose task is to perform encoding/decoding and serialization/deserialization functions, as well as a transceiver, such as, the enhanced small form-factor pluggable transceiver (SFP+) whose task is to physically move the data bits over the media according to a specific physical layer standard. However, the need for an external PHY and an external transceiver can be skipped by selecting the appropriate FPGA device from a family. Second, all mid- and high-end networking-oriented FPGAs offer integrated high-speed transceivers that already support most of the popular PHYs (Figure 4.8). These integrated transceivers operate at line rates up to 32 Gb/s, and they commonly support the 10 GBASE-KR (10 Gb/s) and 40 GBASE-KR4 (40 Gb/s) Ethernet standards, which we seek for interconnecting our modules over a distance up to 1 meter of copper printed circuit board and two connectors. This removal of an external PHY and transceiver is a key contributor in the overall power, latency, cost and area savings.

4.2.2.2.2 FPGA MAC Address Each FPGA module must have a unique MAC address. Basically, there are two options to assign a mac address to the FPGA module. First, the MAC address can be hard coded in the cloud shell logic. However, in a DC environment, there might be thousands of FPGA modules deployed. Having a unique MAC address in each FPGA means a separate bit stream has to be generated for each of the FPGA module in the DC. This is not practical in a hyperscale, dynamic DC environment.

Second, the MAC address can be statically bonded to the FPGA module, independently of the FPGA chip. In this approach, the MAC address of the device may be

stored in a small read only memory of the FPGA module and it is written into the memory during manufacture of the module. When the FPGA module is booted for the first time, it reads the MAC address from the read-only memory and store in the cloud shell as long as the FPGA is powered. We use this method in our implementation.

4.2.2.2.3 MAC Address Resolution When FPGA communicates with other network-attached DC resources over Ethernet, it needs to resolve the MAC addresses of those devices. The MAC address can be resolved in two ways. First, MAC address is configured in the cloud shell as well as in the network switches in the DC by a centralized FPGA manager. This approach saves reconfigurable logic resources required to implement a protocol in resolving MAC addresses, but the system complexity is increased because of the MAC address management by the centralized FPGA manager. Further, centralized FPGA manager must keep track of all the network switches in the DC that would possibly be crossed by the packets originated by the FPGA or packets destined to the FPGA.

Second and the practical approach is to use a standard address resolution protocol, such as ARP over Ethernet. In this approach, the ARP protocol must be implemented in the cloud shell and the resolved MAC addresses must be stored for communication with the other network-attached resources. Section 4.3 explains the evaluation of both these approaches.

4.2.2.2.4 FPGA IP Address Similarly to having a MAC address, the FPGA module must also have an IP address to communicate with other resources. The FPGA module IP address can be assigned in two ways. First, by hard coding the IP address in the cloud shell. As discussed in Section 4.2.2.2.2, hard coding a unique value in the FPGA requires to generate a unique bit stream for each FPGA available in the DC. At the DC scale, this is not practical. Second, by using a standard protocol to retrieve the IP address dynamically. In this approach, a protocol, such as DHCP (dynamic host configuration protocol) must be implemented in the cloud shell, so that the FPGA module's IP address can be retrieved dynamically based on its MAC address.

4.2.2.2.5 Health Monitoring In a large-scale DC environment, keeping track of all the resources by status monitoring is required to ensure the overall system reliability. The minimum system-level monitoring method used by network-attached DC resources is the ICMP-based request and reply messaging or ping. Therefore, the standalone disaggregated FPGA must support such a standard protocol, so that basic device status monitoring can be done.

4.2.2.2.6 Network Stack Virtualization The network protocol stack, which consists of a typical TCP/IP stack, can be implemented in two ways. First, dedicated protocol stacks are implemented for user region and vendor region, as shown in

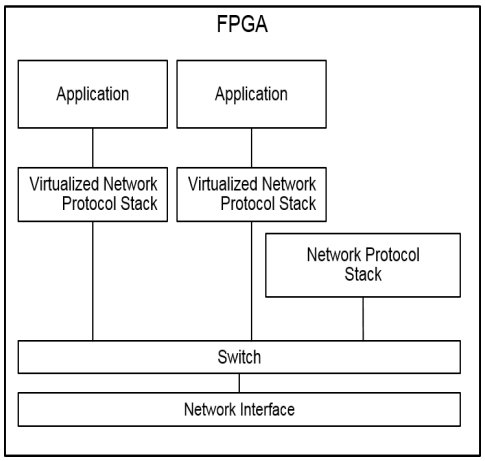


Figure 4.9: FPGA Network Stack Virtualization

Figure 4.9. We do not expect the user region to have dedicated network protocol stack unless there are multiple vFPGAs owned by different tenants. Further, this approach consumes a large amount of compute resources that can otherwise be used for the vFPGA. Second and preferred method is to have a single protocol stack shared by the cloud shell and vFPGA.

4.2.2.3 MGMT

MGMT includes one or more agents that listen on a pre-defined TCP port for commands and management data from an external SW service, which we call FPGA management utility (FMU). The communication link to the MGL is called the control path of the standalone disaggregated FPGA. The agents may include functions, such as configuring a new vFPGA using partial reconfiguration, vFPGA monitoring, and executing other utility functions. In this implementation, the management layer runs an agent that can execute TCP listen, connect, and close commands on the underlying FPGA network protocol stack to connect multiple FPGAs together in a software-defined manner.

4.3 HW Prototype Implementation

To realize the standalone disaggregated FPGA architecture presented in Section IV, we implemented a prototype on a Xilinx Virtex7 XC7VX690T FPGA. This section covers the implementation of the prototype, which is shown in Figure 4.10.

4.3.1 User Application (vFPGA)

The vFPGA (or ROLE in the shell-role architecture shown in Figure 4.10) hosts the application. The vFPGA has one or more communication links through the NSL

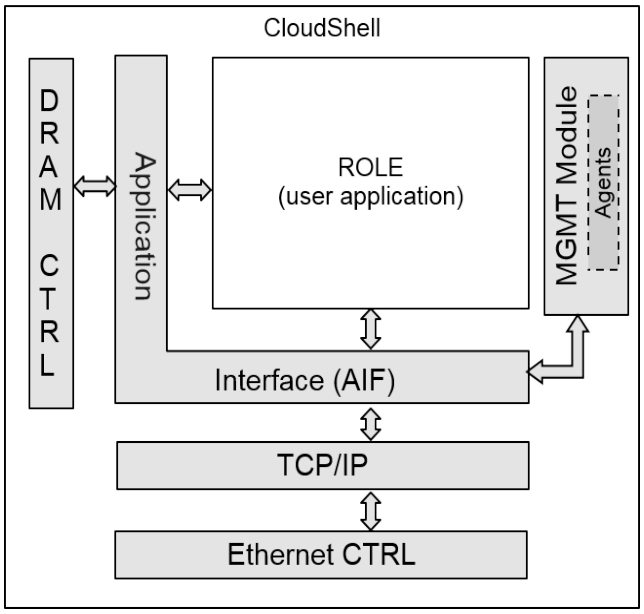


Figure 4.10: Standalone Disaggregated FPGA Prototype

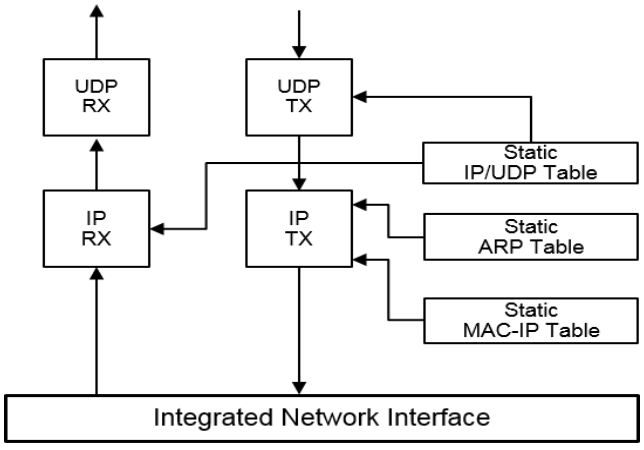


Figure 4.11: UDP Only with Centralized Control Plane Approach

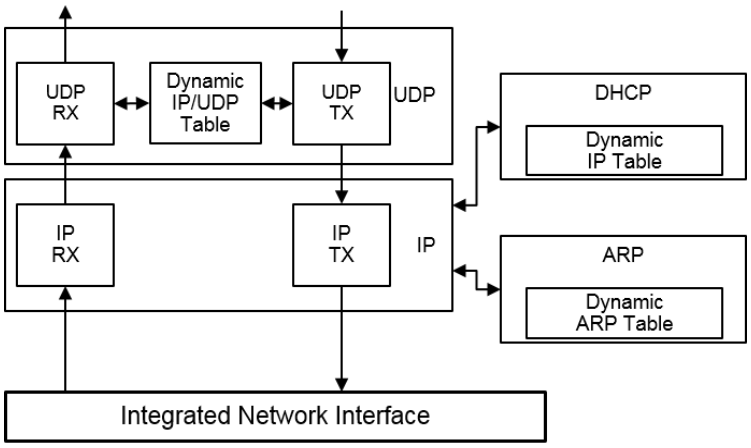


Figure 4.12: UDP Only with Distributed Control Plane Approach

to the servers and to other vFPGAs over the DC network. As shown in Figure 4.10, we call these communication links the data path of the standalone disaggregated FPGA. The links can be reliable connection-oriented, such as TCP, or unreliable connection-less, such as UDP. These communication links offered to the vFPGAs are FIFOs, relieving the user (application writer) from complex network programming tasks. The data fed to the TX FIFOs is wrapped in network packets and forwarded to the relevant destination by the underlying NSL. Similarly, the network packets received by the NSL are first unwrapped, and then the payload is fed to the RX FIFO to be used by the application. The user distinguishes the various communication links to the vFPGA from each other using the FIFO index. When application-specific protocols are needed, they can be built within the vFPGA atop the FIFO-based TCP or UDP links. The use of the vFPGA in a real application is discussed in the Chapter 6.

4.3.2 Cloud Shell

4.3.2.1 Network Service Layer

4.3.2.2 Network Controller (NET CTRL)

The network controller implements the data link layer (L2) of the Ethernet standard, for which we used a Xilinx 10GbE MAC IP core. A Xilinx 10GbE PHY IP core connects the MAC to the external PHY over the integrated GTX transceivers of the FPGA.

4.3.2.3 Network and Transport Stack (NTS)

The NTS consists of a L3 (IP) and L4(TCP and UDP) protocol stack. For implementing the NTS, we first investigated two approaches: (i) Centralized Control

Plane and (ii) Distributed Control Plane. These two approaches were evaluated by implementing the NTS only with IP and UDP protocols.

We implemented IP and UDP layers with a 64b bus at 156.25 MHz. The minimum size of the Ethernet frame that travels in the physical layer is 84 bytes, and the UDP payloads of up to 18B make this smallest Ethernet frame on the wire (Preamble(7B) + Start of Frame(1B) + MAC Header(14B) + IP Header(20B) + UDP Header(8B) + Payload(18B) + FCS(4B) + Inter-Frame Gap(12B) = 84B). This requires that to achieve the line rate, each module in the data path must process the packets of up to 18B of UDP payload in lesser than 10.5 clock cycles. We designed each module of the data path to satisfy this criterion. We implemented the architecture explained above in C++ using Xilinx Vivado HLS.

4.3.2.3.1 UDP/IP with Centralized Control Plane In the centralized control plane approach (Figure 4.11), we do not implement control path protocols such as ARP (Address Resolution Protocol) and DHCP. In this approach, instead of implementing a distributed control plane by having the ARP functionality in each FPGA, a centralized control plane can be used, from where the ARP tables are programed using SDN. In this implementation, we use static ARP and UDP/IP tables. The UDP/IP table consist of five tuples: (a) source port, (2) destination port, (3) source IP, (4) destination IP, and (5) buffer ID. The same approach based on SDN can be used to program the five tuples belong to open ports of each vFPGA. The IP/UDP table along with vFPGA-MAC-IP table are updated when new vFPGAs are started on the physical FPGA.

4.3.2.3.2 UDP/IP with Distributed Control Plane In this approach, ARP and DHCP functionality is implemented in the NTS and it can operate without any centralized intelligence (Figure 4.12). The tables are updated dynamically, based on the ARP, DHCP, and UDP functionality. The comparison of the two approaches (Table 4.2) shows that the difference in resource consumption are in the range of 0.8% to 1.3% for FFs and LUTs, whereas the value is around 10% for BRAMS. These observations show that the centralized control plane approach does not bring much benefit in terms of resource consumption, particularly when the overhead of centralized controller is concerned.

4.3.2.3.3 TCP/IP There are multiple implementations of TCP/IP stacks in FPGAs [150]. When choosing a stack for the standalone disaggregated FPGA three requirements were considered: (i) being open source, so that it can be adapted to our infrastructure, (ii) 10 GbE support, as we run on 10 GbE DC network, and (iii) high-level synthesis, for faster development. Satisfying these requirements, the most suitable stack for us was the TCP/IP stack developed by Xilinx [151] [152]. Although this TCP/IP stack is designed specifically for a mem-cached application, the open source nature allows us to modify it according to our cloud DC approach.

Table 4.2: Resource Consumption: UDP/IP with Centralized vs Distributed Control Plane

Approach	Module	LUT	FF	BRAM
Centralized	IP	1118	1123	4
	UDP	2990	3122	7
	IP/UDP Table	129	145	0
	ARP Table	139	129	0
	Total	4376	4519	11
	% of XC7VX690T	1.01	0.52	0.75
Distributed	IP	1789	2063	28
	UDP	3548	3957	22
	ARP	3304	4270	46
	DHCP	1354	1315	66
	Total	9995	11605	162
	% of XC7VX690T	2.31	1.34	11.02

4.3.2.4 Application Interface (AI)

The AI has two main functions: (1) It consists of four memory access modules corresponding to each vFPGA to enable access to the memory that belongs to each vFPGA over the network by external hosts. (2) It consists of a switch that multiplexes and de-multiplexes incoming and outgoing packets to and from vFPGAs and memory access modules. In the receive path, the incoming payload received by the AI is forwarded to the relevant vFPGAs or memory access modules according to the accompanying buffer ID. In the transmit path, the buffers belong to vFPGAs and memory access modules are scanned in a round-robin fashion, and the payload that belongs to the first non-empty buffer is forwarded to the NTS along with the buffer ID.

4.3.2.5 Management Layer (MGMT)

The MGL includes one or more agents that listen on a pre-defined TCP port for commands and management data from an external SW service, which we call FPGA management utility (FMU). The communication link to the MGL is called the control path of the standalone disaggregated FPGA. The agents may include functions, such as configuring a new vFPGA using partial reconfiguration, vFPGA monitoring, and executing other utility functions. In this implementation, the management layer runs an agent that can execute TCP listen, connect, and close commands on the underlying FPGA network protocol stack to connect multiple FPGAs together in a software-defined manner.

4.3.2.6 Memory Controller (MEM CTRL)

For the memory controller, we used a Xilinx MIG (Memory Interface Generator) dual-channel memory controller with an AXI (AMBA eXtensible Interface) interface.

4.3.3 Flow of Building Application

To enable users to build their applications for the standalone disaggregated FPGA, a set of files are provided by the vendor (Figure 4.13). These files include: (i) top level file that wraps the whole design including the user application and the shell, (ii) the shell as a design checkpoint (.dcp), and (iii) a constraint file which maps the design to the FPGA pins. The user application can be built in two ways: (i) using HLS design flow, which starts with a high-level C/C++-based design and (ii) using HDL (Hardware Description Language, such as Verilog or VHDL) design flow. HDL-based user application is then integrated with the vendor-provided placeholder for the user application. Finally, the usual FPGA design flow is executed, starting from synthesis until bit-stream generation. In this step, partial reconfiguration can also be used, so that the cloud shell and the user application can be configured into the FPGA separately. In the work covered in this thesis, partial reconfiguration is not considered.

Once the user-generated bit-streams is configured to the standalone disaggregated FPGA, first, the cloud shell is initialized. In this initialization process, an IP address for the FPGA is assigned by DHCP. Once initialized, an application running on a sever can connect to the user application in the FPGA over the DC network. For this connection standard TCP/IP is used. In the case of partial reconfiguration, the cloud shell and the role of the FPGA are configured in two steps as shown in the flow diagram of Figure 4.14. First, the FPGA is booted from a default initial system setup image provided by the DC infrastructure provider. This image includes the cloud shell, which is required for the board to boot and signal its presence on the DC network by requesting the assignment of a dynamic IP address. Second, a partial reconfigurable bitstream containing only the customized logic of the user's application (role) is sent over the DC network to the management layer of the FPGA, which stores a copy of that bitstream at a specific offset into the flash device. Next, the management layer triggers the partial reconfiguration controller of the FPGA to reconfigure the new user-logic part into the FPGA.

4.4 Simulation Environment

This section explains the development of the simulation environments for both the cloud shell (Figure 4.15) and the role (Figure 4.16). The cloud shell simulation is required for the infrastructure vendor who develops and enhances the functionality the shell to offer increasingly rich services to the role. The simulation of role

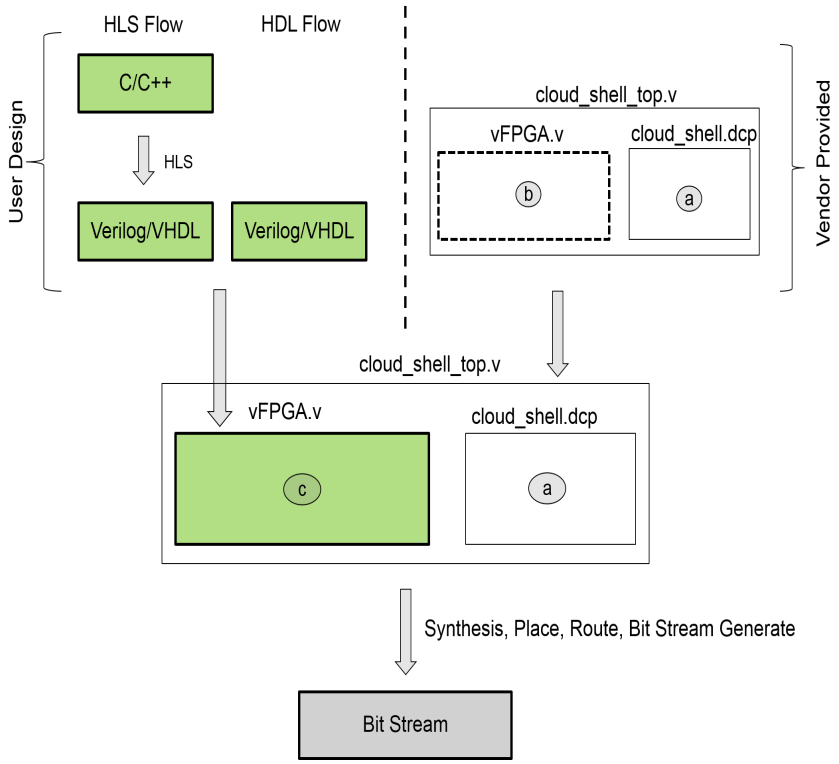


Figure 4.13: Flow of Building Applications: (a) cloud shell (b) black box (placeholder for user application) (c) user application

is used by the application developers, before placing their HW application in the real FPGA HW.

4.4.1 Cloud Shell Simulation

Xilinx Vivado HLS C simulator allows to run standard Linux C library functions in the test bench. This allows us to integrate standard TCP/IP software socket code in the simulator test bench code. Using the TCP/IP user space library functions, incoming packets can be read at different levels in the packet structure. For example, only the payload of the packet, payload with the IP header, or payload with both IP header and Ethernet header can be received by the application. This flexibility allows the simulation platform to split the cloud shell at different stages for simulating a particular piece of logic.

To feed the HLS-based TCP/IP stack, the incoming data must be received with the packet header, including the MAC header, IP header and the TCP or UDP header. To receive the packet header with the incoming data, the applications uses raw sockets. In this implementation, a simple loopback application is running in the HLS TCP/IP stack to ensure that the stack is working properly in the simulation

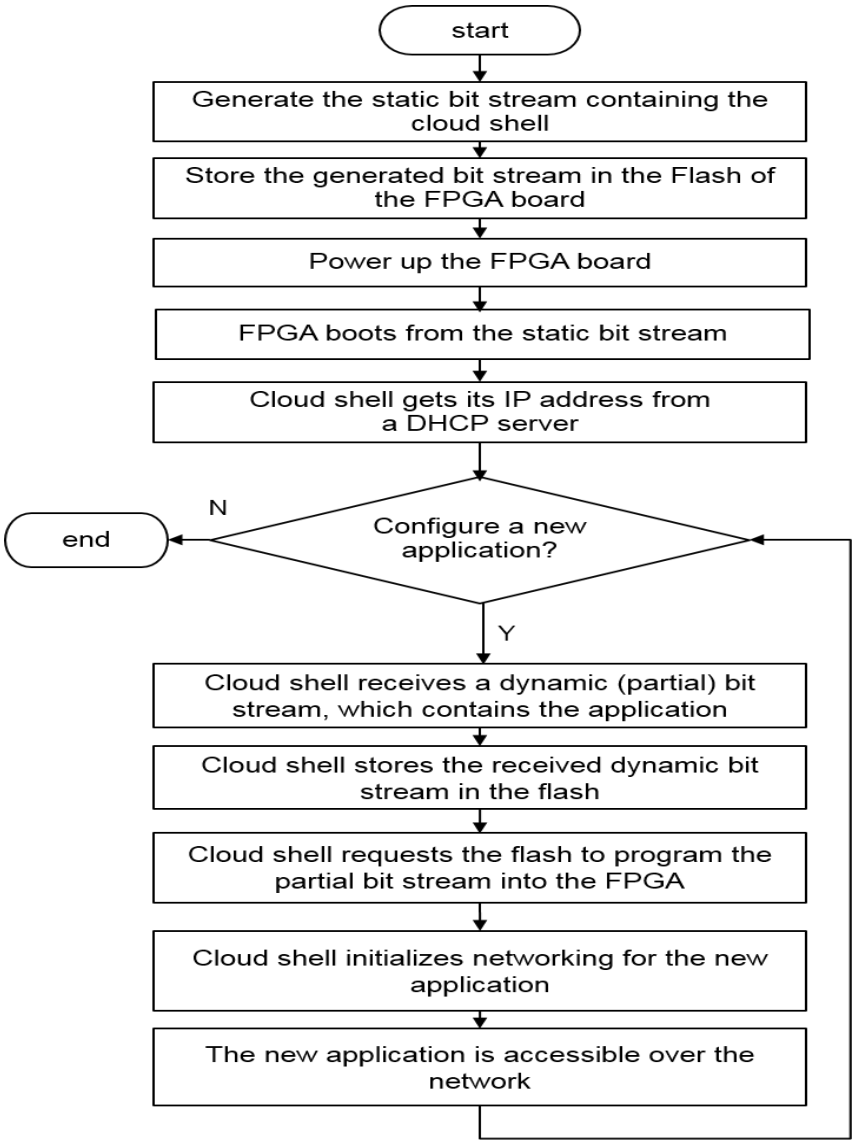


Figure 4.14: Flow of Configuring a Standalone Disaggregated FPGA

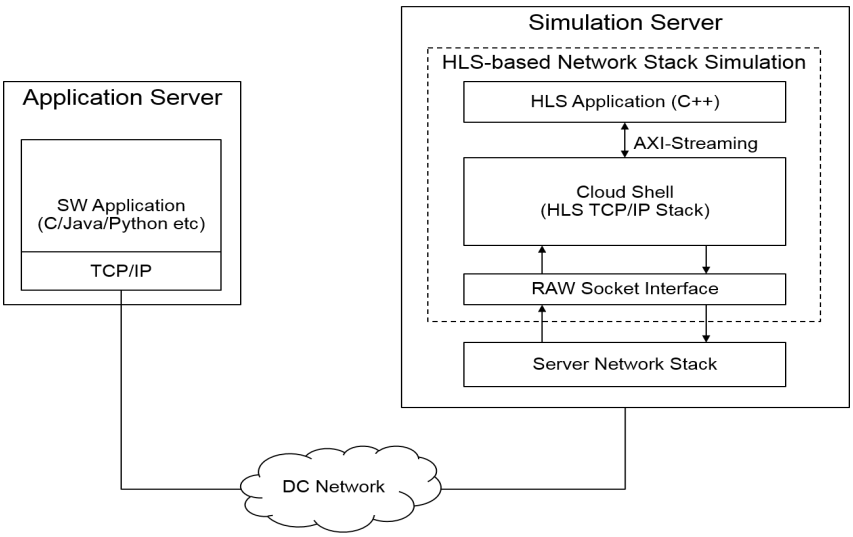


Figure 4.15: Cloud Shell Simulation Platform

environment.

4.4.1.1 RX Path

When using connection-oriented protocols, such as TCP/IP, the incoming connection requests must not be forwarded to the host network stack, for example the Linux kernel network stack. Instead, the raw packet must be forwarded to the simulation platform. For filtering the packets, standard Linux firewall tool, iptables, is used. The raw socket interface in the simulation platform reads the packet and forwards it to the HLS TCP/IP stack. Before feeding the data to the HLS TCP/IP stack, incoming data has to be deserialized and adjusted according to the width of the data path of the stack.

4.4.1.2 TX Path

In the TX path, the data going out from the HLS TCP/IP stack has to be serialized and organized in a buffer before sending through the standard TCP/IP stack. Before sending the data over the raw socket, the complete Ethernet frame including the data and the protocol headers must be constructed appropriately.

4.4.2 User Application Simulation

For the simulation of the user application, we modeled the Standalone Disaggregated FPGA in C. The user applications run on top of this model. In this work, the user application considered is written in C. For C-to-Verilog communication, there are several interfaces are available, such as PLI (Programming Language Interface), DPI (Direct Programming Interface) and VPI (Verilog Procedural Interface) [153].

Among them, the DPI is the state of the art, and it has replaced the PLI and VPI by directly allowing the verilog code to call standard C library functions, instead of using a user-defined wrapper. In the simulation, we used DPIC [154], which is widely used to interface C code with verilog [155]. For VHDL, FLI [156] can be used.

This section explains the simulation environment for application development for the standalone disaggregated FPGA. As shown in the Figure 4.16, the application development set up consists of two servers, one to run the SW application, which is called the application server, and one to run the simulation platform, which is called the simulation server. The motivation behind building this application development environment is to completely simulate the application before placing it in a real standalone disaggregated FPGA.

In the setup, the SW application can be in any language that can use TCP/IP sockets to communicate with a remote server. In the simulation server, the SW driver, which is written in C language, performs the task of the cloud shell in the real disaggregated FPGA. TCP/IP is used for the communication between the SW application in the application server and the SW driver in the simulation server. For the communication between SW driver and the RTL application, which is the device under test in this case, System Verilog Direct Programming Interface (DPI-C) is used. By using DPI-C, the verilog-based RTL application can call functions written in C from the SW driver. By placing the input and output variable appropriately in the SW driver, the two-way communication can be ensured between the RTL application and the SW driver. On top of DPI-C, AXI-streaming semantics are used for the communication between SW driver and the RTL application, resembling the same infrastructure in the real disaggregated FPGA.

4.5 Evaluation

We evaluated our architecture in terms of network latency, throughput, application predictability, and resource consumption. The standalone disaggregated FPGA architecture presented was implemented and validated on a Alpha Data PCIe card featuring a Xilinx Virtex7 XC7VX690T FPGA. The design uses one 10 GbE network interface and two 1333 MHz DDR3 SODIMMs with 8 GB each. In the experimental setup, the Alpha Data FPGA cards are plugged into a PCIe expansion chassis [22], from which only power is taken for the cards. The FPGA cards are connected via a top of rack switch [157] to two servers. Each server consists of a 4-core (with 2 CPU threads per each core) Intel i7-3820 clocked at 3.6 GHz and has 32 GB of main memory. The servers run Linux (Fedora 22, Kernel 4.0) and are equipped with a DC-class Mellanox ConnectX-3Pro 10 GbE Controller on PCIe Gen3. The max payload size and the max read request size of the PCIe configuration are 256 B and 4096 B respectively. The VMs run the same Linux configuration explained above on two KVM (Kernel-based Virtual Machine) hosts. Each VM has a single

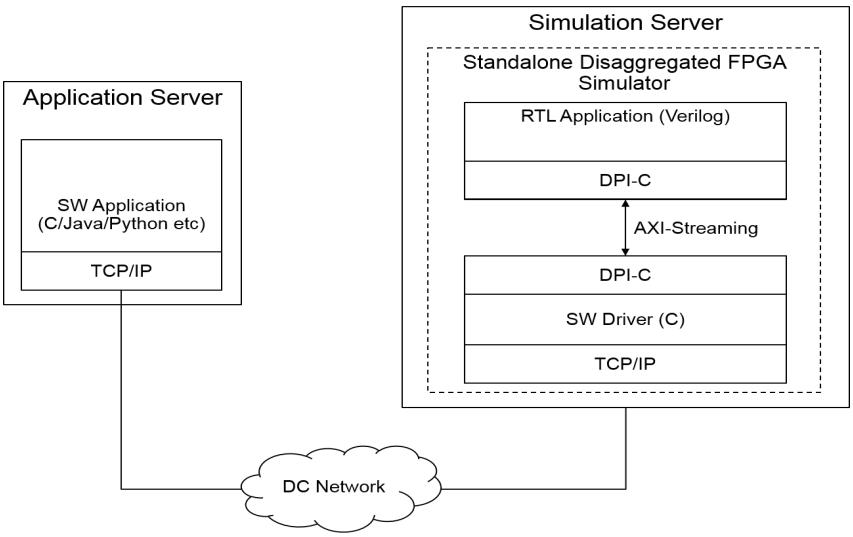


Figure 4.16: Simulation Platform of Role (User Application)

core running at 3.6 GHz and 4 GB of memory. For the CTs, we used standard Linux application containers.

Although bare-metal server, VM, and container network performance has been reported in [127], comparison of those compute resources has not been done against FPGAs. In this work, we compared the network latency, throughput, and variance of response time of multiple SW and FPGA configurations using six experimental cases: (a) VM-VM, (b) VMDIO-VMDIO, (c) CT-CT, (d) Native-Native, (e) Native-FPGA, and (f) FPGA-FPGA. In each configuration, the first member acts as the client and the second as the server. The network stack configurations of the experimental cases are shown in Figure 4.17. In the VM configuration, the VM network stack is connected to the host NIC through virtio and vhost drivers over a Linux bridge, whereas in the VMDIO case, the VM network stack is directly connected to the host NIC. The CT network stack is connected to the host NIC through two virtual ethernet interfaces (VETH PAIR) over a Linux bridge.

4.5.1 Latency

To measure the latency, we used a client-server application where the client sends a message to the server, and the server sends back the same message to the client. FPGA does not send the messages to the external memory, whereas the NICs in servers DMA copy the messages to the DRAM. For each message size ranging from 1 B to 1472 B, the average time of one million such round trips is taken as the final RTT.

As shown in Figure 4.19, moving from VM-VM to FPGA-FPGA, the RTT becomes incrementally better. We observe that FPGA-FPGA achieves an impressive RTT

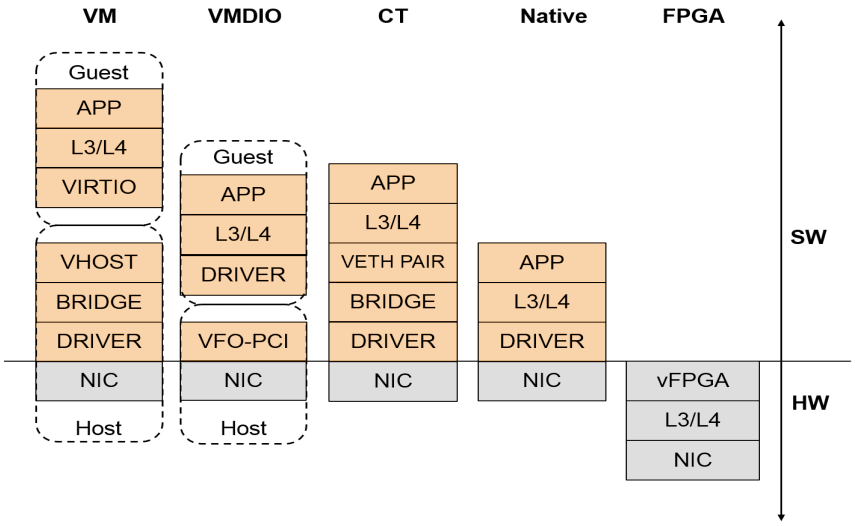


Figure 4.17: Network Stack Configurations of The Experimental Cases

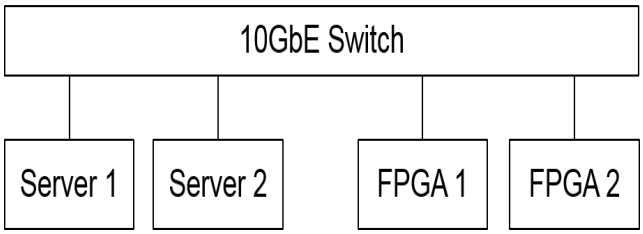


Figure 4.18: Experimental Setup

of 2.8 μs and 12.1 μs for 1 B and 1472 B messages, respectively. The FPGAs decrease the node-to-node RTT by a factor of 10x to 35x compared with the VMs and by a factor of 5x to 16x compared with VMDIO. FPGA-FPGA node latency is 3x to 12x better than Native and CTs and 2x to 7x better than native-FPGA. Also, we observe that the FPGA-FPGA RTT of 3.1 μs for 64 B messages is better than RDMA and other kernel bypass networking solutions on native servers. Mellanox VMA (Mellanox Messaging Accelerator) achieves an RTT of 2.86 μs for 64 B messages in a back-to-back configuration without crossing a network switch [158], and iWARP (internet Wide Area RDMA Protocol) achieves an RTT of 9.6 μs for 8B messages crossing a network switch [159]. The network switch we used adds 0.8 μs of latency to the RTT of a message. The native-FPGA RTT performance shows that even if an FPGA is used on one side of the communication channel, the performance cannot be significantly improved because SW is involved on the other side.

We also observed that when the message size is gradually increased from 1 B to 1472 B, the latency gap between FPGA-FPGA and other SW cases becomes smaller.

The overall packet transmit latency consists of two components: (i) the control path latency and (ii) the data path latency. The control path latency is independent of the data size for the packets under MTU size. From the results, it is clear that for SW cases there is almost no change in the latency when the message size is increased from 1 B to 1472 B. This is because the control path latency dominates the overall latency. But when the message size is increased, the data path latency gradually becomes the dominant contributor to the overall latency. In the case of FPGA, there is no control path and the push from the application to the data path drives the packet out of the FPGA. From these observations, it is clear that for large message sizes, such as 1 MB, there would not be a clear gap in the latency between FPGA-FPGA and other SW cases. But, low latency is particularly important for control messages. Usually, the control messages are very small and fit in a MTU size message. Hence, even if it is for small message sizes, the low latency results produced by the FPGA is beneficial for the distributed applications in DCs.

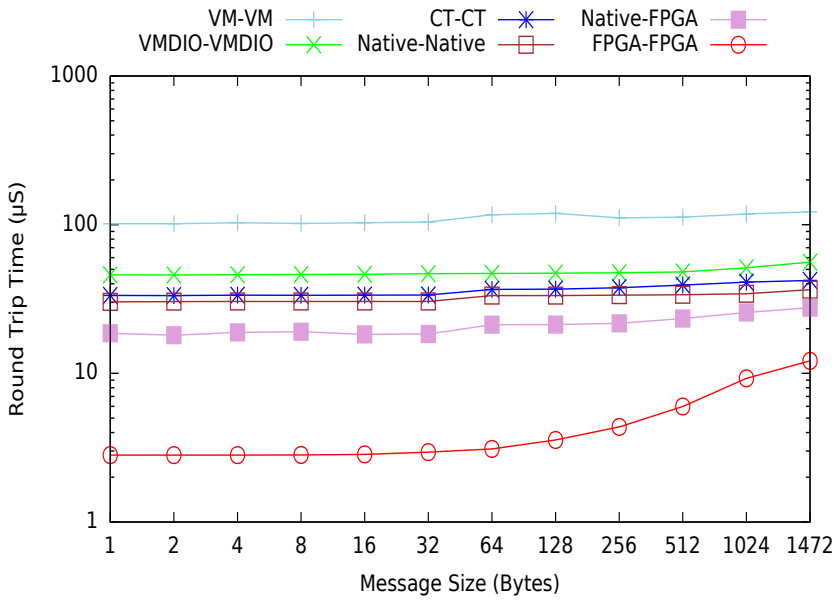


Figure 4.19: Latency Comparison

4.5.2 Throughput

To measure the throughput, we used another client-server application where the server sends a stream of packets and the measurement is taken at the client. In a single iteration, one million packets are received from the server, and the average of ten such measurements are taken as the final result. In this experiment, we defined the maximum theoretical throughput as follows, considering the overhead added in each layer from L1-L4: Maximum theoretical throughput = Line Rate *

(UDP Payload/(UDP Payload + UDP Header (8 B) + IP Header (20 B) + MAC Header (14 B) + FCS (4 B) + Preamble (7 B) + Start of Frame (1 B) + Inter-Frame Gap (12 B))).

As shown in Figure 4.20, FPGAs can achieve the maximum theoretical throughput at all message sizes ranging from 1 B to 1472 B. Irrespective of the experimental case, the throughput performance of SW is poor, particularly for the smaller message sizes. For 1B messages, the native servers, CTs and VMs can achieve only 0.32, 0.28 and 0.2 million messages per second respectively, whereas the FPGA can achieve 14.88 million messages per second, with an impressive improvement of up to 73x. Even though VMDIO's RTT is better than that of VMs, the throughput is extremely poor for small message sizes, and thus we excluded VMDIO from the comparison. The literature shows that the native servers can achieve 3.64 million messages per second for 1 B messages using kernel-bypass networking solutions [158], which is still far below than what FPGAs can achieve.

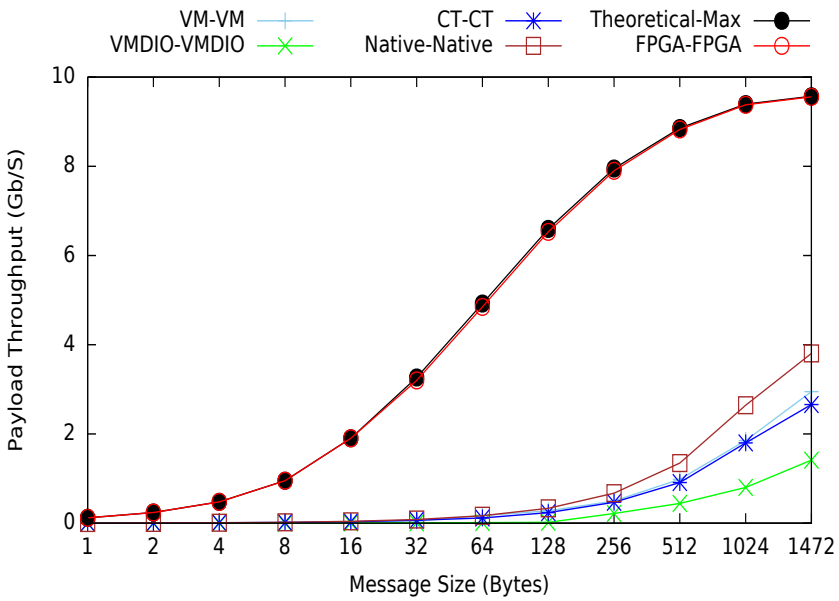


Figure 4.20: Throughput Performance

4.5.3 Latency Variation

We evaluated the latency variation by considering the RTT (Round Trip Time) of one million iterations for each payload size. The standard deviation of the 99th percentile of the RTT distribution is shown in Figure 4.21. The standard deviation of the FPGA-FPGA latency ranges between very low vales of 0.027 and 0.043, whereas the standard deviation for the VM-VM ranges between 16.731 and 22.154. For CT-CT, native-native, and native-FPGA, we observe a value ranging

between 1.1 and 2.3, whereas the standard deviation of VMDIO-VMDIO gradually increases up to 7.8 with the message size.

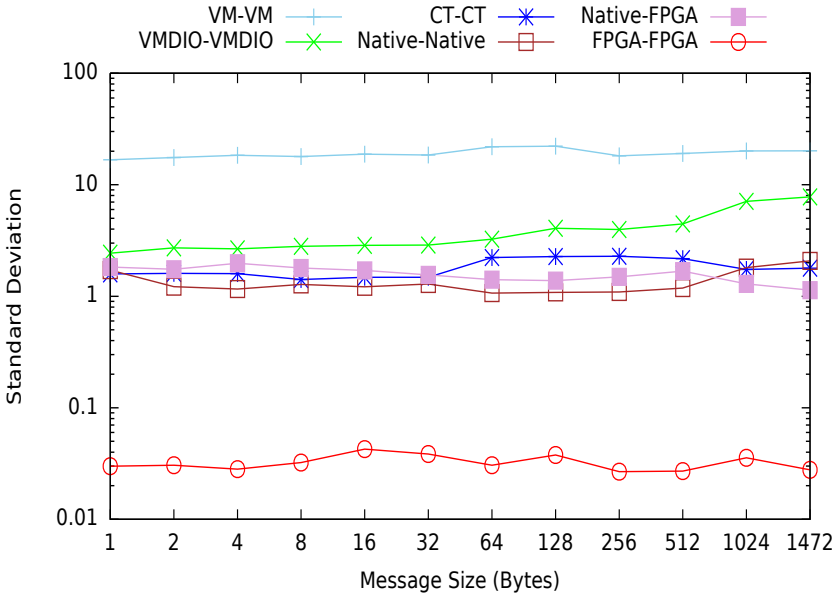


Figure 4.21: Variation of Response Time (99th Percentile)

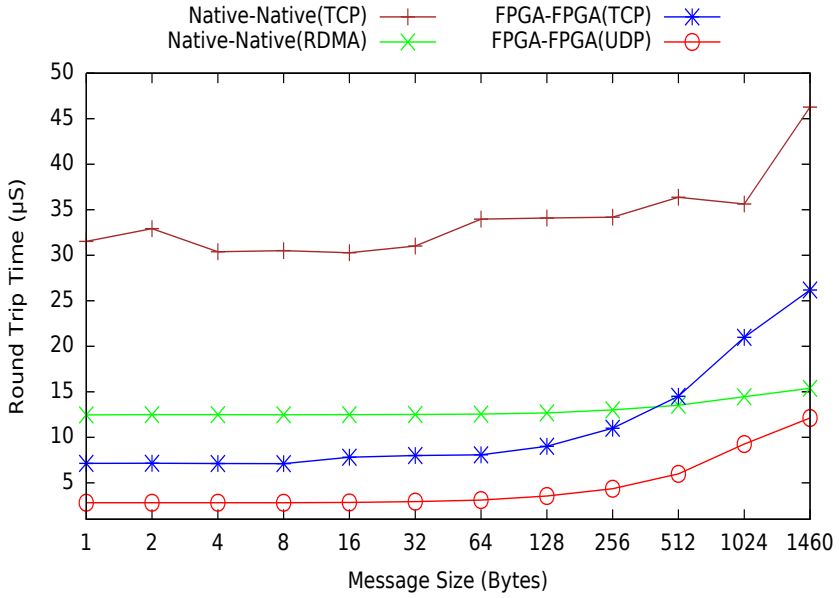


Figure 4.22: TCP Latency Comparison

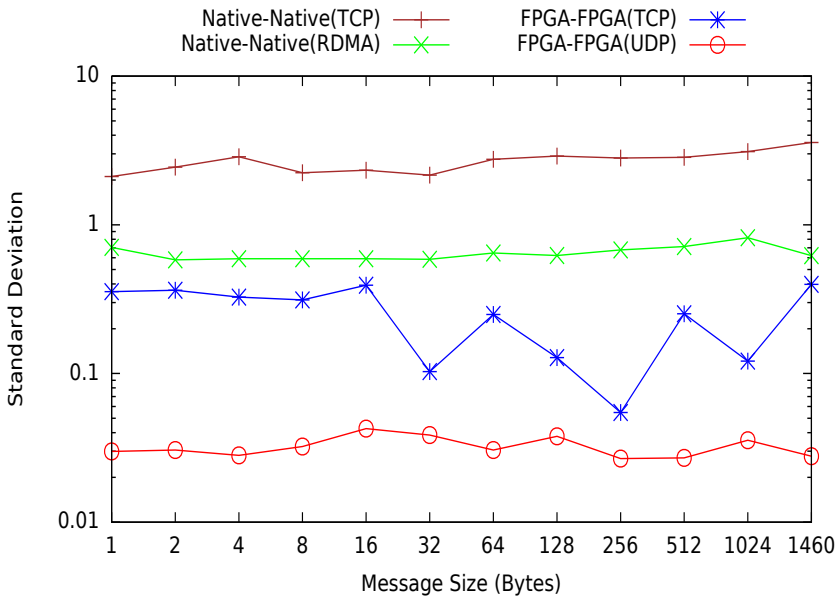


Figure 4.23: TCP Latency Variation

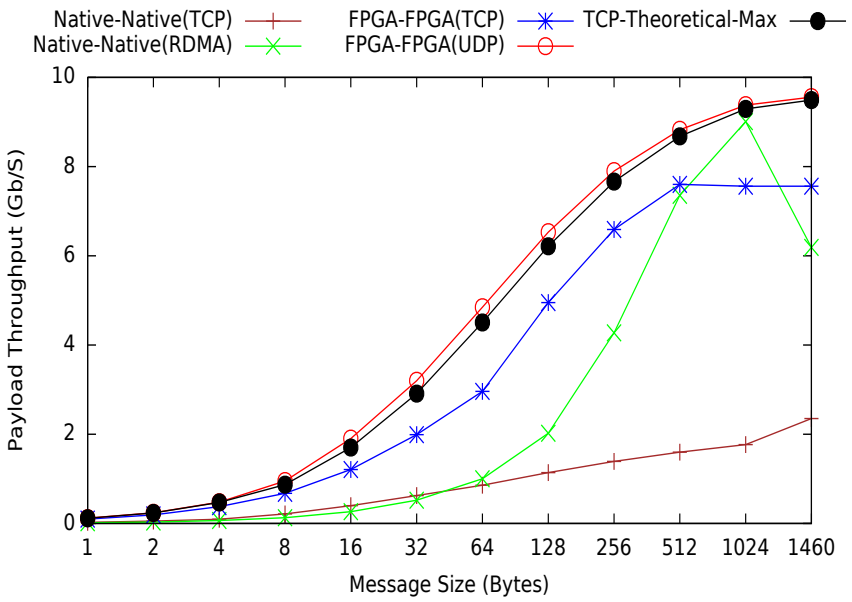


Figure 4.24: TCP Throughput Comparison

4.5.4 FPGA Resources

Table 4.3 and 4.4 show the resource consumption of the shell based on UDP/IP and TCP/IP, respectively. The NET CTRL (MAC/PHY) and the MEM CTRL are Xilinx IP cores. In the case of UDP/IP, the design use around 62 K LUTs and

59 K Flip-Flops, which is equal to 14% and 7% of the overall available resources. Out of the total resources, the network-related modules (NET CTRL, IP/UDP, AI) consume less than 4% of the LUT, FF and BRAM resources. In the case of TCP/P, the design use around 88 K LUTs and 100 K FFs, which is equal to 20% and 12% of the overall resources.

Table 4.3: Resource Consumption of UDP/IP Based Shell

Module	LUT	FF	BRAM
IP	1118	1123	4
UDP	2990	3122	7
NET CTRL	4581	5101	11
MEM CTRL	39373	34172	38
MV	5382	4991	66
Application Interface	4142	4316	24
Top Level	4541	5705	34
Total	62289	59160	193
% of XC7VX690T	14	7	13

Table 4.4: Resource Consumption of TCP/IP Based Shell

Module	LUT	FF	BRAM
IP	1789	2063	28
TCP	15159	16460	270
NET CTRL	4581	5101	11
MEM CTRL	39373	34172	38
Application Interface	1913	1768	15
Other (ARP/DHCP/Top Level)	25721	40576	28
Total	88536	100140	390
% of XC7VX690T	20	12	27

4.6 Discussion

4.6.1 Performance

Overall, the experimental results show that FPGAs outperform general-purpose servers in network performance by a large margin. We found that the server network throughput is significantly degraded for smaller message sizes, whereas FPGAs achieve the line rate irrespective of the message size. Even though modern discrete NICs support advanced features, such as segmentation offload, small messages do not benefit from them. The impressive FPGA-FPGA latency and

throughput open the path for the deployment of standalone disaggregated FPGAs in DCs. Also, this eliminates the need to implement RDMA and other kernel-bypass networking mechanisms [158], which are known to provide better latency and throughput than traditional networks, when deploying distributed applications in DCs.

When running multiple applications, time-sharing is used in the CPUs, where the processor is divided in its computing time among multiple tasks. This increases the contention for the resources when adding more applications on the same CPU. On the other hand, space-sharing is used in the FPGAs, where a dedicated physical space of the silicon chip is reserved for each application. This fundamental architectural difference in the FPGA solves the scheduling issues when running multiple applications on the same compute resource, significantly improving the application response time variation.

In the implementation of TCP/UDP/IP on CPU, the CPU involves in both the control and the data path of the network packets (Figure 4.25-(a)), whereas in the case of RDMA on CPU, the CPU only involves in the control path (Figure 4.25-(b)). In the FPGA implementation of TCP/UDP/IP, there is no control path and only the data path exists (Figure 4.25-(c)), as FPGA applications are typically implemented in a data-flow architecture instead of using a control-flow architecture, like in CPUs.

Furthermore, in the case of FPGA, as all the components including the network controller, the network stack, and the application are tightly integrated within the FPGA HW, the communication link between two applications that run on two standalone disaggregated FPGAs resembles a physical wire connection. On the other hand, in the case of CPU, a large number of SW functions executes multiple processing steps, particularly in the control path, when moving data from application to application over the network, which significantly degrades the latency and throughput performance. Even if RDMA improves the network performance compared to TCP/UDP/IP on CPU by bypassing the kernel on the data path (Figure 4.25-(b)), it fails to match the performance of TCP/UDP/IP implemented on FPGA, as the RDMA control path still runs on SW.

4.6.2 FPGA Resource Consumption

We find that it takes less than 4% of the resources of a Xilinx Virtex7 (XC7VX690T) FPGA to hook it up to UDP/IP over an Ethernet network, and that the overall architecture consumes only 14% of the total resources. Even if management interfaces are used to connect the device to a centralized management software, we expect the overall resource utilization to be around 20%, which is a comparable amount considering the 23% resource utilization in the Catapult FPGA fabric [25].

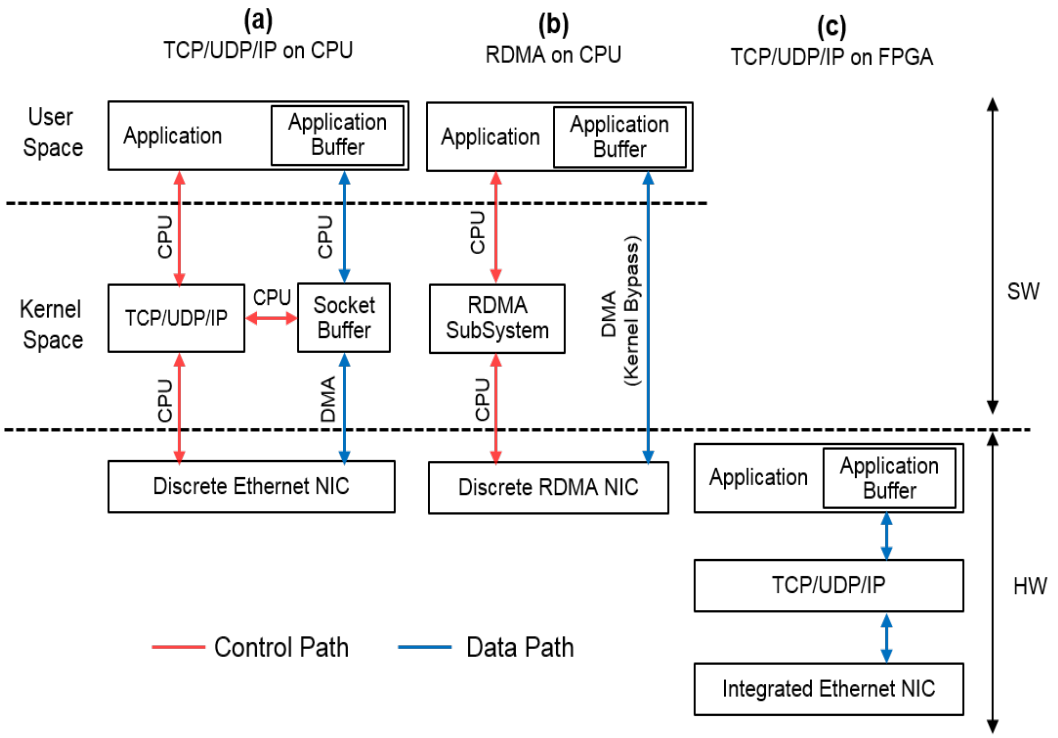


Figure 4.25: TCP/IP and RDMA on CPU vs TCP/IP on FPGA

4.6.3 Impact on Applications

Even if the available compute power is sufficient, inefficient networks hamper the scalability of CPU-based applications. For example, when the number of servers is increased beyond a certain threshold, the training of a distributed deep neural network becomes slow, as the network overhead starts to dominate [40]. Similarly, the scalability of Hadoop TeraSort [42] is hampered by the network throughput performance, particularly in the data-shuffling phase, in which an increased amount of DC network traffic is generated.

Moreover, on-line data-intensive applications, such as high-frequency trading (HFT) and web searches, need to obey tight latency constraints. In synchronous cluster applications, a typical cause of degraded performance is variance in processing times across different servers, leading to many servers waiting for the single slowest server to finish a given computation phase [40]. This variance in processing times is caused by unpredictable scheduling of CPU and IO resources.

According to our results, as FPGAs perform far better than SW in throughput, latency, and variation in response time, they largely resolve the network bottlenecks in CPU-based distributed compute fabrics.

4.6.4 Network Protocol

We also found that FPGA UDP throughput reaches the theoretical maximum throughput and its latency stays at very low values, whereas the TCP throughput and latencies are better than those of RDMA for small massage sizes. In contrast to UDP, the TCP throughput stays well below the theoretical maximum throughput for all message sizes. Our UDP implementation used only one fifth of the resources of the TCP stack we evaluated, and moreover the TCP stack required a few hundred connections to achieve the maximum throughput, whereas UDP needed only one.

The insights gained from the results call for a custom protocol that performs like UDP and has the minimum features for reliable inter-FPGA communication, using fewer resources than TCP. Preferably, this protocol must be able to saturate the line rate using a few connections, because large number of connections entails the issue of how to distribute the traffic between connections to achieve the maximum rate.

Even though we have implemented unreliable UDP protocol in the prototype, we want to implement a reliable protocol for inter-FPGA communication. Traditional Ethernet does not guarantee lossless frame reception. Instead, packets are dropped whenever a receive buffer reaches its maximum capacity. The modern CEE (Converged Enhanced Ethernet) networks are designed to prevent these frame losses by using a link-level flow-control mechanism called PFC (Priority Flow Control). As our FPGA network stack has been designed to be lossless, we can build an end-to-end lossless DC network for inter-FPGA communication by deploying FPGAs on modern CEE networks. In those infrastructures, we envision that the level of reliability that must be provided by the L4 protocols, such as TCP, can be further simplified, leading to a wide use of simpler L4 protocols, such as UDP. This reliable protocol can be implemented on top of UDP, like UDT (UDP-based Data Transfer Protocol) [160] and QUIC (Quick UDP Internet Connections) [161] does.

4.7 Summary

In this chapter, an architecture for a standalone disaggregated FPGA is proposed. The standalone disaggregated FPGA module consists of an FPGA to serve as generic and programmable hardware accelerator, a nonvolatile memory device, such as a serial or parallel flash memory, to store the FPGA's configuration information, and the external memory in the form of DRAM. Because a standalone disaggregated FPGA resource is decoupled from any host, it must operate in a self-contained manner by executing tasks that were previously under the control of a host server. To achieve this, a shell-role architecture is used: the shell abstracts FPGA I/O, while the role hosts the user applications. The state-of-the-art FPGA shells interact with CPUs over a PCIe bus, but in the approach proposed here TCP/IP over 10 GbE used for CPU-FPGA as well as for inter-FPGA commu-

nication. The shell provides the access to the network, memory, and management functions.

To verify the proposed architecture, a prototype was built in a commercial FPGA. The prototype built was evaluated in terms of network latency, latency variation, throughput, and resource consumption. As these FPGAs are deployed in DCs, we compare these metrics with other DC compute resources, such as bare-metal severs, VMs, and CTs. The results show that standalone disaggregated FPGAs outperform them in terms of network latency and throughput by a factor of up to 35x and 73x, respectively. We also observed that the proposed architecture consumes 23% of the total FPGA resources (Xilinx Virtex7 (XC7VX690T)). Finally, we also evaluated a shell based on only UDP/IP, which consumes only 14% of the FPGA resources.

Chapter 5

Software-Defined Multi-FPGA Fabrics

In this chapter, software-defined multi-FPGA fabrics are introduced, which are built using standalone disaggregated FPGAs (explained in chapter 4) in a software-defined manner. This chapter is organized as follows: Section 5.1 reviews state-of-the-art multi-FPGA systems and their applications. Section 5.2 reviews requirements of multi-FPGA systems in cloud DCs and proposes software-defined multi-FPGA fabrics. Section 5.3 summarizes the chapter.

5.1 Multi-FPGA Systems

When a single server is not sufficient to run an application producing desired results, usually the application is split and distributed on to multiple servers on a compute cluster. Similarly, when the capacity of a single FPGA is not enough to handle a desired functionality in a particular application, multiple FPGAs are used in the application, splitting and distributing the functionality among those FPGA chips. One of the main features that distinguishes different kinds of such multi-FPGA architectures is the topology, meant as the way the different FPGAs are connected to each other [29]. There are mainly two types of topologies based on the way multiple FPGAs are interconnected: (i) fixed topologies and (ii) programmable topologies.

5.1.1 Fixed Topologies

In fixed topologies, multiple FPGAs are interconnected over hard-wired connections linking specific I/O pins of FPGAs over a multi-FPGA board [95] [94] [93] [96] [18] or over a dedicated point-to-point network [7] [15] [16] [25] [47]. The simplest fixed topology is the linear array (Figure 5.1-(a)), where multiple FPGAs are chained together and the application data usually flows in one direction [94] [93]. By closing the loop in a linear array arrangement, FPGA rings (Figure 5.1-(b)) are made [15] [16] [47] [162]. The most widely adopted fixed topology is the mesh and it is the popular choice for multi-FPGA board deployment. In a mesh, FPGAs

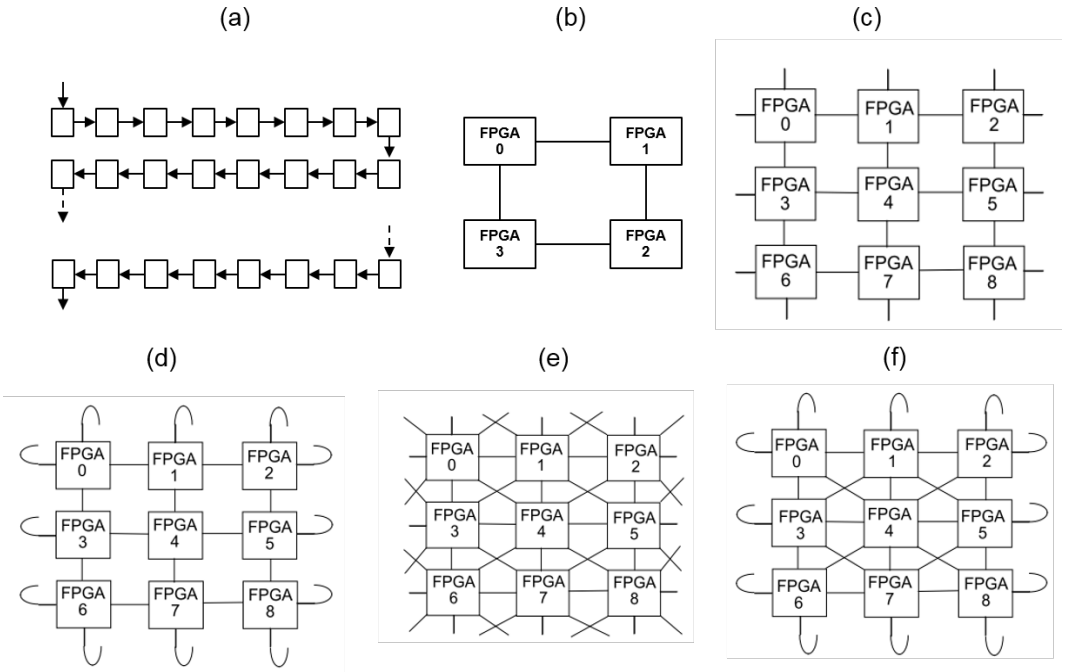


Figure 5.1: Fixed-Topology Multi-FPGA Systems: (a) Linear Array (b) Ring (c) 4-Way Mesh (d) 4-Way Torus (e) 8-Way Mesh (f) 8-Way Torus

are arranged on a grid and are connected in a nearest-neighbor pattern. In 4-way meshes [95](Figure 5.1-(c)), the wires connect only horizontal and vertical neighbors, whereas in 8-way meshes [18](Figure 5.1-(e)), each FPGA is connected also to its diagonal neighbors. When the FPGAs on two opposite boundaries of the grid are connected in a circular fashion, that topology is called torus [25] [97](Figure 5.1-(d)(f)).

The advantage of the mesh topology is the inherent expandability of the architecture due to the use of local connections. Adding an FPGA to an existing architecture means creating some local connections without any other constraint. The disadvantages are due to the fact that there is no fixed-length path between every pair of FPGAs, which causes different delays in signal transmission and the need to use some area to implement communication logic in intermediate chips. Several topologies adopt hard-wired connections. In a complete-graph topology, each FPGA is connected to each other. Despite this topology offers a direct connection between any pair of FPGAs, as the number of chips increases, the width of each connection decreases due to the fixed number of available pins in each FPGA. Moreover, as the number of FPGAs increase, it is difficult to practically realize at the circuit level due to the large number of connections between chips [30] [29] [163].

5.1.2 Programmable Topologies

Programmable topologies consist of wires connected to components, which are also reconfigurable logic or network devices that can switch traffic based on network packet information [2] [1]. Re-programmable components can be programmed to implement a particular connection between the wires. The most used topology using programmable connections is the crossbar. In this topology, chips are divided in to two classes based on their functionality. Logic bearing FPGAs contain the logic functions and perform computations (the lower ones in Figure 5.2-(a)(b)), while routing chips provide the connections between logic chips (the upper ones in Figure 5.2-(a)(b)). The idea is that communication between any pair of logic FPGAs requires exactly one extra routing hop, such that communication delays are all equal. When only one chip is used to provide the interconnections, the crossbar is said to be total. When several chips are used, the topology is named partial crossbar. Due to the cost of producing a big routing chip, partial crossbar is usually preferred. The routing chips can also be standard FPGAs. The literature [30] shows about cheaper re-programmable interconnect devices with large number of pins called FPIDs (Field Programmable Interconnection Devices), or FPICs (Field Programmable Interconnections Chips). However, those chips cannot be found on today's market. Crossbar architectures have the drawback that they are not expandable, since the connections are implemented over a global communication infrastructure [30] [29] [163].

Virtual Wires [164] is a technology for sharing physical wires among multiple logical ports within the FPGA. The programmability comes from the matching of logical ports to the physical wires. When two virtual wire technology-enabled FPGAs are connected, end-to-end virtual wires between FPGAs can be created. This is somewhat similar to running VLANs atop Ethernet.

Re-programmable components can also be network switches (Figure 5.2-(c)(d)), as in traditional computer networks. In this case, FPGAs are directly attached to the network. Based on the network protocols used, Ethernet or Infiniband, topologies can be made programmable. The disadvantage of this approach compared to multi-FPGA board approach is the high latency, as the network packets for inter-FPGA communication have to cross through a network switch.

5.1.3 Applications

One of the most successful uses for FPGA-based computation is in ASIC logic emulation. The designers of a custom ASIC need to make sure that the circuit they designed correctly implements the desired functionality. Software simulation can perform these checks but does so quite slowly. In logic emulation, the circuit to be tested is instead mapped onto a multi-FPGA system, yielding a solution several orders of magnitude faster than software simulation [29] [165] [166].

The execution of identified compute-intensive kernels that has a high tendency to

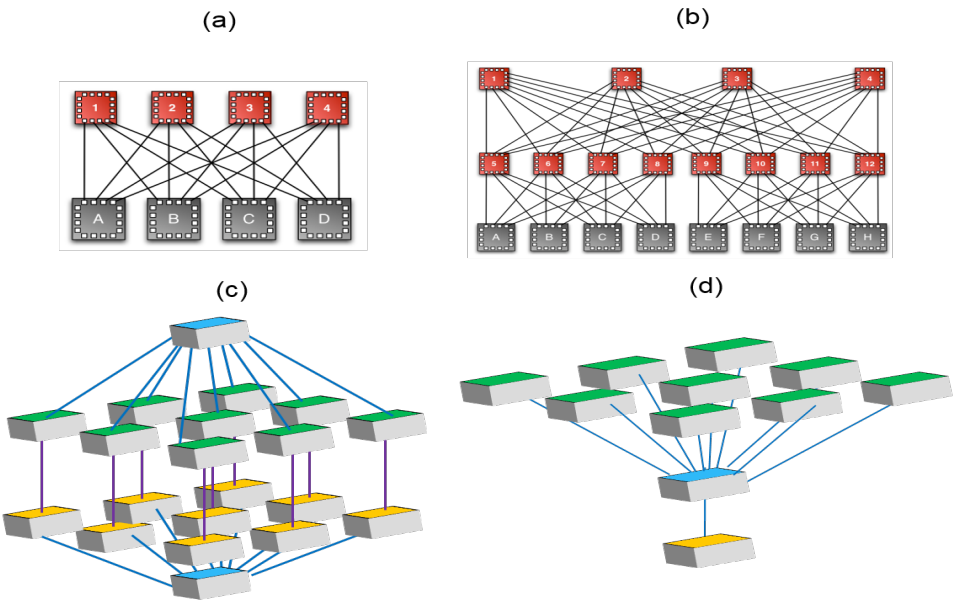


Figure 5.2: Programmable-Topology Multi-FPGA Systems: (a) Crossbar (b) Hierarchical Crossbar [29] [30] (c)(d) Star

occur can be accelerated using FPGAs, as custom HW circuits can be built tailored to the application. HPC systems [97] [93] [167] take advantage of this and get the application processing improved by order of magnitudes by fully and partially offloading compute-intensive processioning to multi-FPGA platforms.

Cryptanalysis of modern cryptographic algorithms require massive computational effort, often between 2^56 to 2^80 operations. A characteristic of many cryptanalytical algorithms is that they can run in a highly parallel fashion with very little interprocess communication. Such applications map naturally to a hardware based design, requiring repetitive mapping of the basic block, and can be easily extended by adding more chips as required [94].

Multi-FPGA systems are also better in parallel computing where multiple kernels can run independently because of the nature of spatial computing, which makes them suitable for deep learning [168] [169] [170] and big-data applications [171] [172]. Sequence alignment is one of the most popular application areas in bioinformatics. Nowadays, the exponential growth of biological sequence data becomes a severe problem if processed on standard general-purpose PCs, because space and energy requirements introduce significant costs [173] [17].

5.1.4 Summary

In summary, multi-FPGA systems are mainly characterized by the FPGA interconnect topology. Fixed topologies are the widely used choice, as almost all of the

multi-FPGA deployments address specific problems. The system infrastructures are highly customized to suit those specific applications. Programmable topologies have been introduced to increase the application flexibility. Programmable topologies vary from infrastructures with dedicated FPGAs for inter-FPGA routing to FPGAs directly attached to data center networks. Although the flexibility is improved by programmable topologies, interconnecting applications which runs on FPGAs still need the FPGAs to be reprogrammed. From the perspective of applications, a wide scope of applications including logic simulation, HPC, cryptography, deep learning, bio-informatics, big data and application-specific cloud infrastructures are successfully using multi-FPGA systems.

5.2 Multi-FPGA Systems in Cloud Data Centers

All the multi-FPGA deployments and applications reviewed in Section 5.1 are based on application-specific HW deployed on private infrastructures. Although [25] and [1] are based on public cloud, they operate at SaaS layer, where the HW infrastructure is designed to be application specific. Motivated by the success of large-scale SW-based distributed applications such as those based on MapReduce and deep learning [40], and the promising results of off-cloud multi-FPGA systems, we want to give the users a possibility to distribute their applications on a large number of FPGAs in the cloud. However, as the applications running on general-purpose cloud data centers are inherently dynamic and diverse, the traditional deployment of FPGAs explained above are not flexible enough to be deployed in general-purpose DCs.

5.2.1 Software-Defined Multi-FPGA Fabrics

In general-purpose DCs, the infrastructure must be able to provide as many FPGAs as needed independent of the number of servers based on the application requirements. It might be one server and a part of an FPGA or one server and thousands of FPGAs as shown in Figure 5.3. Those multiple FPGAs must be able to be connected in flexible topologies (Figure 5.4) on demand and to be released when no longer needed. For example, some applications need to implement load balancing to offer continuous delivery of services at certain SLAs (service level agreement). To implement load balancing, dynamic scaling of compute resources is needed. In such a situation, FPGA-centric application must be able to add/remove FPGAs and connect/disconnect them dynamically. However, traditionally, interconnecting multiple FPGAs require a new bit stream to be generated and the FPGA to be reprogrammed. As this is a long process, the operation of the distributed applications are disrupted.

Addressing above issue, this thesis introduces software-defined multi-FPGA fabric framework. Using this framework, multiple FPGAs can be interconnected on demand in user-defined topologies over the DC network as shown in Figure 5.5. The

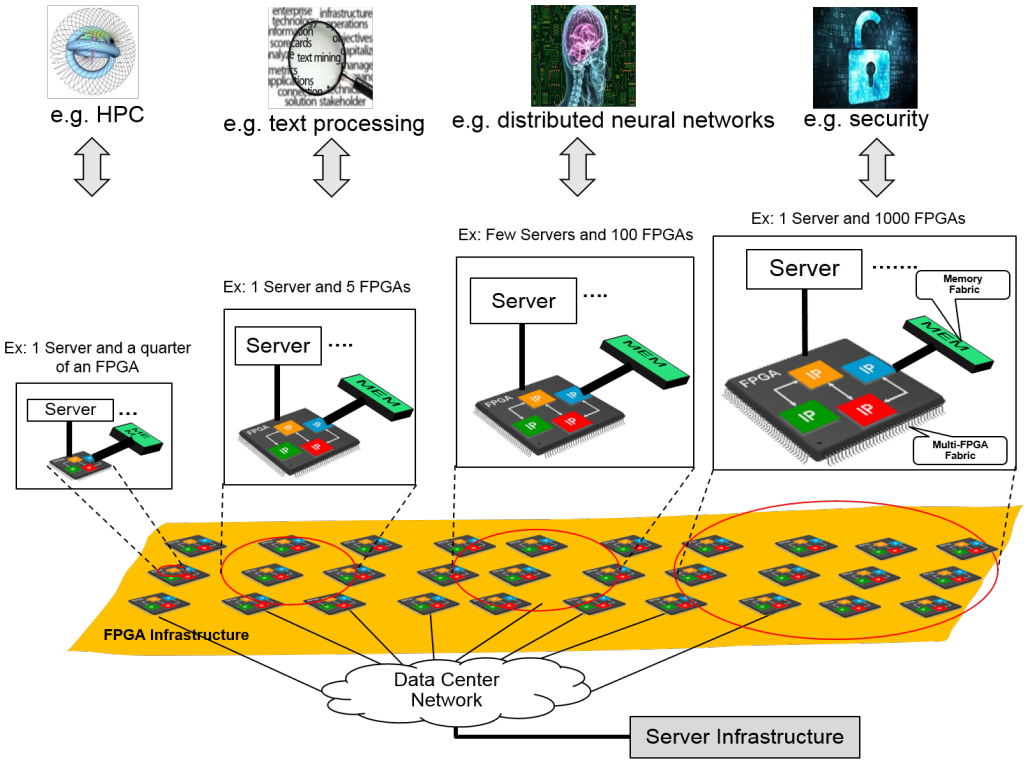


Figure 5.3: Use Cases for Scalable Allocation of Reconfigurable Resources in Cloud DCs

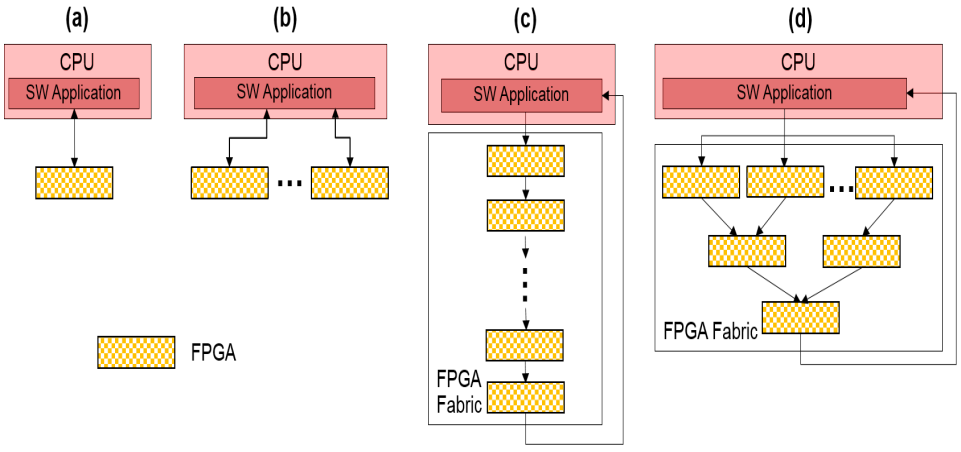


Figure 5.4: Flexible Arrangement of Standalone Disaggregated FPGAs for Diverse Use Cases

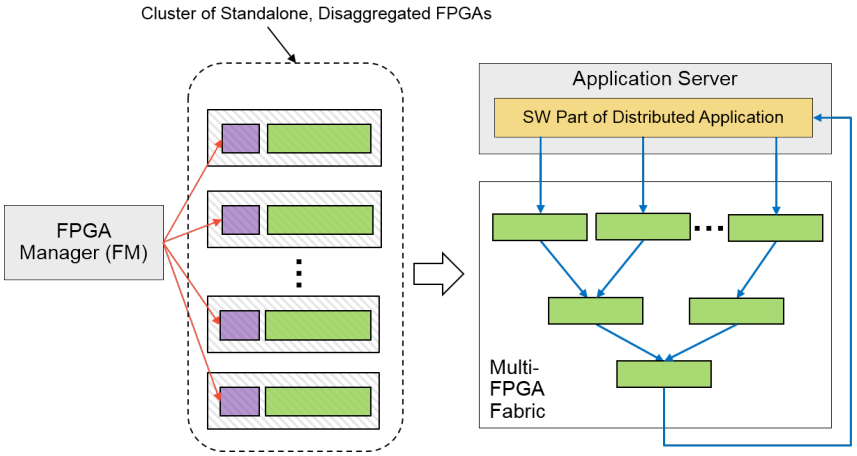


Figure 5.5: Software-Defined multi-FPGA Fabric: On Demand Formation of multi-FPGA Fabrics

basic idea of SDMFF is to connect the vFPGAs in multiple FPGAs dynamically in arbitrary topologies (Figure 5.6).

For making the vFPGA topology programmable, meta data provided by the underlying TCP/IP stack is used. The meta data is in the form of 16-bit integer, which collectively represents the network packet's source IP, destination IP, source port and destination port. The SDMFFs are formed by dynamically making network connections between two FPGAs and associating the meta-data of those connections with the vFPGA's FIFO ID in the corresponding FPGAs.

In a SDN-enabled switch, OpenFlow [174] forwards network packets from a particular port to another port based on the protocol header information of the corresponding packet. Based on the defined rules, while forwarding packet's protocol header information may also get changed. Compared to SDN, SDMFF works at the application layer with the network packet's layer 4 payload. Virtual Wires [164] was a technology to increase the inter-chip communication BW in FPGA-based logic emulators. Limited inter-chip communication bandwidth results in low gate utilization (10 to 20 percent of usable gates). This resource imbalance increases the number of chips needed to emulate a particular logic design and thereby decreases emulation speed, since signals must cross more chip boundaries. Typically, emulators only use a fraction of potential communication bandwidth because they dedicate each FPGA pin (physical wire) to a single emulated signal (logical wire). These logical wires are not active simultaneously and are only switched at emulation clocking speeds. Virtual Wires overcome pin limitations by intelligently multiplexing each physical wire among multiple logical wires and pipelining these connections at the maximum clocking frequency of the FPGA. A virtual wire connects a logical output of one FPGA to a logical input on another FPGA.

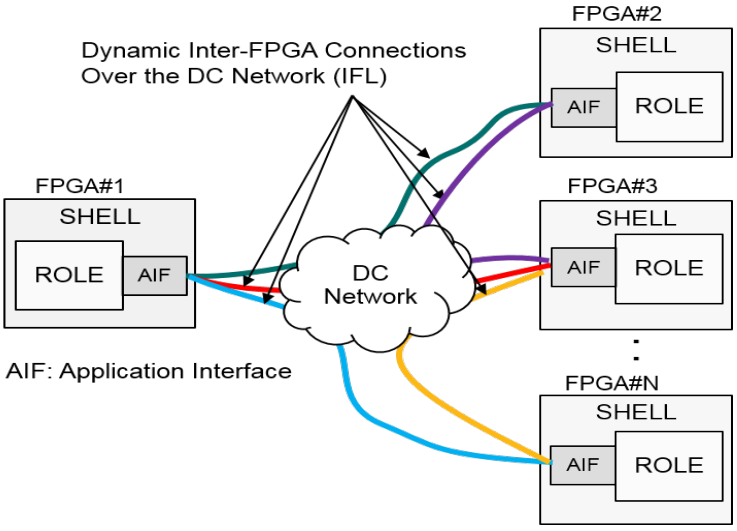


Figure 5.6: Dynamic Inter-FPGA Connections Over the DC Network

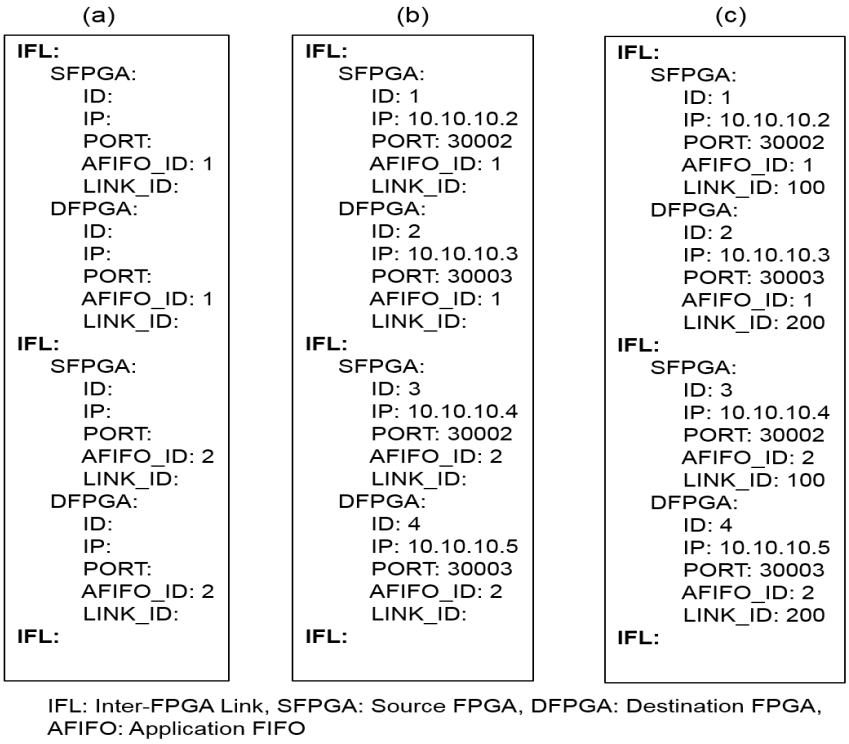


Figure 5.7: SDMFF Topology Definition: (a) user-defined configuration, (b) configuration after resource allocation and (c) configuration after fabric is formed

5.2.2 Fabric Topology Definition

SDMFF topology is defined in a configuration file in plain text. The structure of such a file is shown in Figure 5.7. Each inter-FPGA link (IFL) represents a network connection (such as TCP) between two FPGAs. Two FPGAs in an IFL, which consists of the source FPGA (SFPGA) and the destination FPGA (DFPGA), has 4 attributes: (i) FPGA IP address, (ii) port number, (iii) application (vFPGA or ROLE in shell-role notation) fifo ID, and (iv) connection ID. The fabric topology configuration is updated in 3 steps: (i) at user definition of the topology, (ii) at resource allocation, (iii) and at fabric formation.

At first step, user defines the multi-FPGA fabric topology by adding IFLs in the configuration file and by updating only the application FIFO ID. Application FIFO IDs in the same IFL on two FPGAs represent the two endpoints of an IFL with respect to application data. By configuring those application fifo IDs in the configuration file, user expects whatever the data written at one end of the IFL to be ended up in the other end in the destination FPGA. Once the user defined topology is handed over to the resource allocation service, the configuration file is further updated with the FPGA ID, FPGA IP address and the ports (Figure 5.7-(b)). The information up to this step is used by the FPGA manager at the fabric formation step to build the multi-FPGA fabric. At the fabric formation step, IFLs are formed over the DC network based on the configuration information. In the FPGA, each connection is uniquely identified by an identifier, which is updated in the configuration as the connection ID (Figure 5.7-(c)). This connection ID is the 16 b integer based meta-data explained in section 5.2.1.

5.2.3 FPGA Manager

Each FPGA chassis runs an agent, which reports the information of the FPGA to the FM. FM (Figure 5.9) keeps record of all the FPGAs in the data center in its DB and allocates resources according to the user's multi-FPGA fabric creation request. MFFC (Multi-FPGA Fabric Controller) is a service in FM, which exposes the functionality to create multi-FPGA fabrics to the users. MFFC contains three main components: (i) task decoder, (ii) fabric topology decoder and (iii) fabric builder. The task decoder identifies whether the request is to create a new fabric, to delete an existing fabric or to modify an existing fabric. Based on the identified task, the topology decoder decodes the received topology and allocate or deallocate resources using the FM resource allocator. Then the fabric builder sends appropriate commands to each FPGA associated with the multi-FPGA fabric, which is defined by the topology configuration.

5.2.4 Multi-FPGA Fabric Agent

Standalone Disaggregated FPGA presented in Chapter 4 is enhanced to build SDMFFs. The two enhancements done are (i) the modification of the application

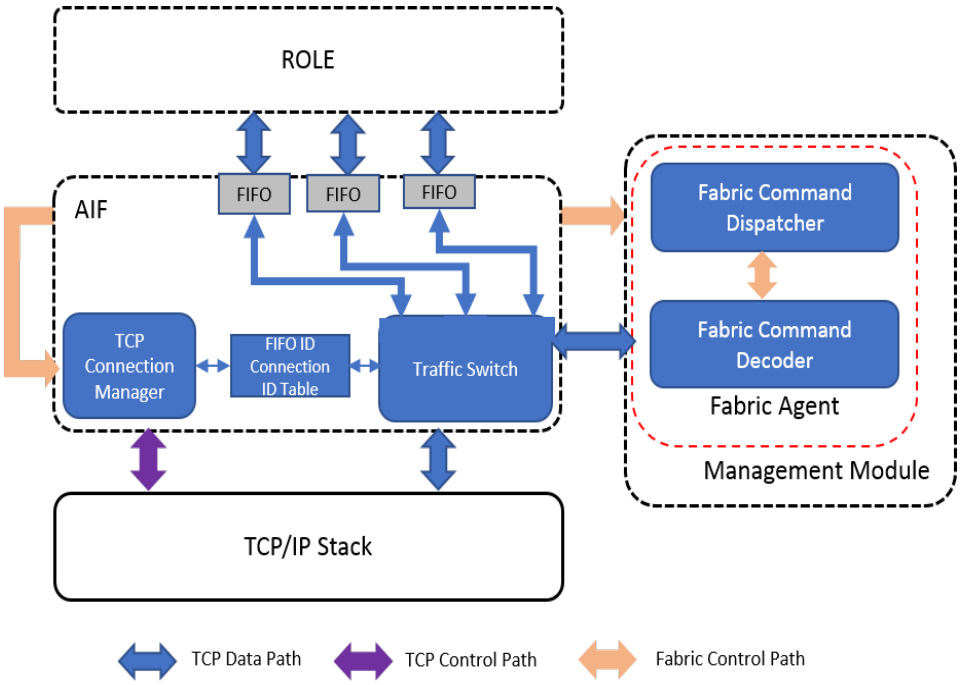


Figure 5.8: Application Interface and Fabric Agent Architecture

switch and (ii) the addition of a new agent, which is called multi-FPGA fabric agent (FA), to the management layer as shown in Figure 5.8. A key feature of these enhancements is the minimal dependency on the underlying network protocol stack. The meta-data provided by the underlying TCP stack is used together with the meta-data of application FIFOs from the application interface to switch data at payload-level.

Fabric agent listens on a predetermined TCP port when the FPGA is configured with the SHELL. The FPGA manager connects to this port over TCP and the MFFC sends fabric commands (Table 5.2) to the FA based on the SDMFF protocol (Section 5.2.5). After decoding the received commands by the fabric command decoder, the fabric command dispatcher sends the relevant TCP-based commands to the TCP connection manager. The connection manager executes the commands on the underlying TCP stack. The returned value from the TCP stack is the connection ID, which is a unique value differentiating each TCP connection. The link ID along with the vFPGA's FIFO ID are sent to the application switch, which in turn programs the two-column table, consisting of link IDs and vFPGA's FIFO IDs. This information is later used by the application switch to perform switching of the network packet payload to the relevant FIFO in the application layer.

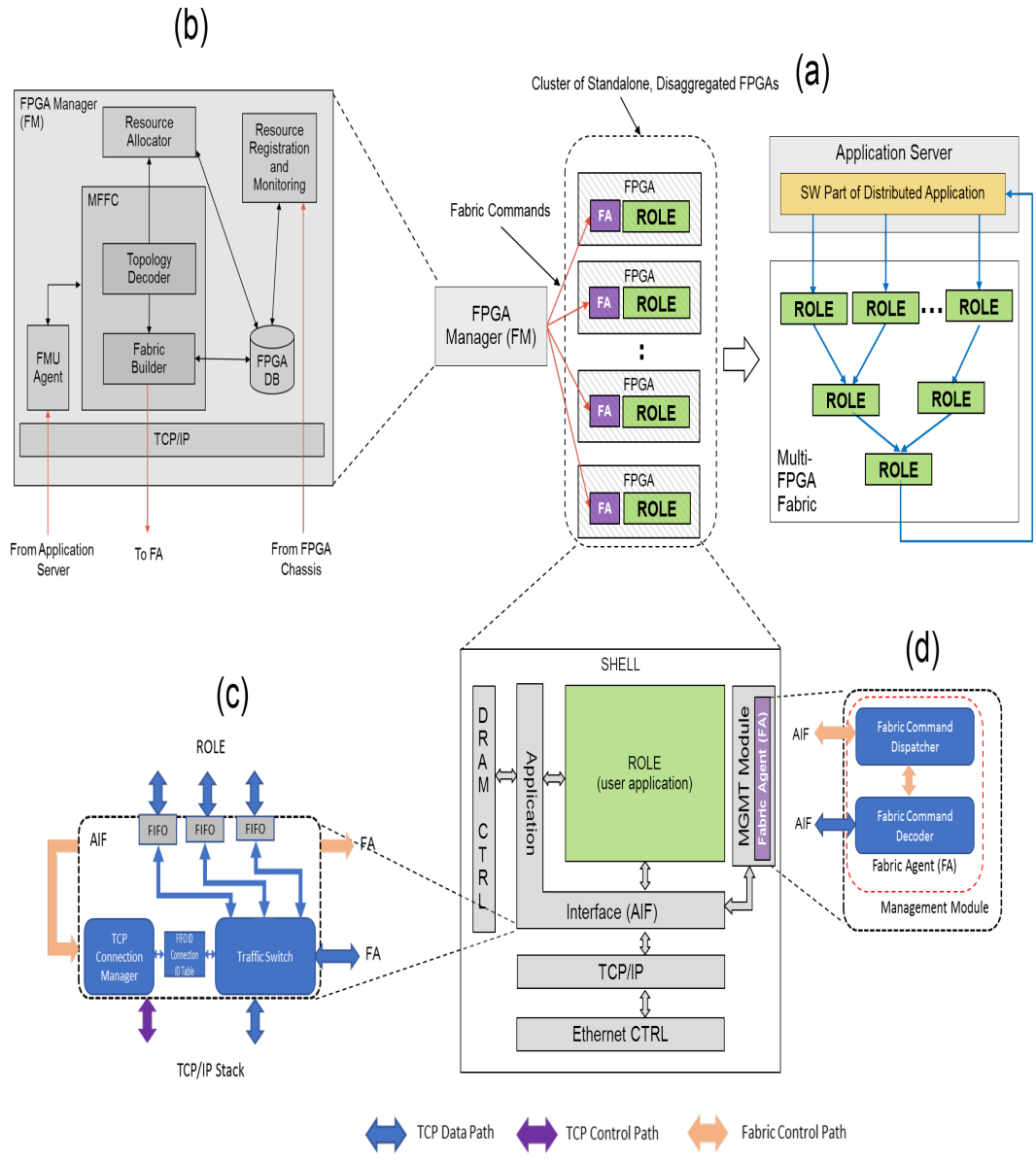


Figure 5.9: SDMFF Framework: (a) An Example SDMFF Interconnect (b) FPGA Manager (c) Programmable Application Interface (d) Multi-FPGA Fabric Agent

```
struct sdmff_protocol_header {    //software-defined multi-fpga fabric protocol header
    uint16 agent;          //1:FA, else=drop
    uint16 cmd;            //1:link add, 2:link update, 3:link remove, 4:link status
    uint16 sub_cmd;        //1:connect, 2:listen
    uint32 src_ip;         // source ip
    uint32 dst_ip;         // destination ip
    uint16 src_ip_port;    // source port
    uint16 dst_ip_port;    // destination port
    uint16 src_bid;        // source buffer id
    uint16 dst_bid;        // destination buffer id
    uint16 cid;            //connection id
}__attribute__((packed));
```

Figure 5.10: SDMFF Protocol Header Defined in C

5.2.5 SDMFF Protocol

SDMFF protocol specifies how the FPGA manager and the fabric agent communicates over TCP/IP. The protocol header is encapsulated in TCP payload and it contains 9 fields to facilitate the commands required to perform fabric creation, fabric modification and fabric delete operations. The protocol header defined in C language is shown in Figure 5.10 and the details of the header fields are given in Table 5.1.

Table 5.1: SDMFF Protocol Header Details

Field	Width	Details
agent	16	Agent ID. ID of the fabric agent is 1.
cmd	16	Command sent to the agent specified in the 'agent' field. 1: link add, 2: link update, 3:link remove, 4:link status
sub_cmd	16	Sub command associated with the command specified in 'cmd' field. 1:connect, 2: listen
src_ip	32	Source IP address of the IFL.
dst_ip	32	Destination IP address of the IFL.
src_ip_port	16	Source port of the IFL.
dst_ip_port	16	Destination port of the IFL.
src_bid	16	Source buffer ID of the IFL.
dst_bid	16	Destination buffer ID of the IFL.
cid	16	Connection ID of the IFL.

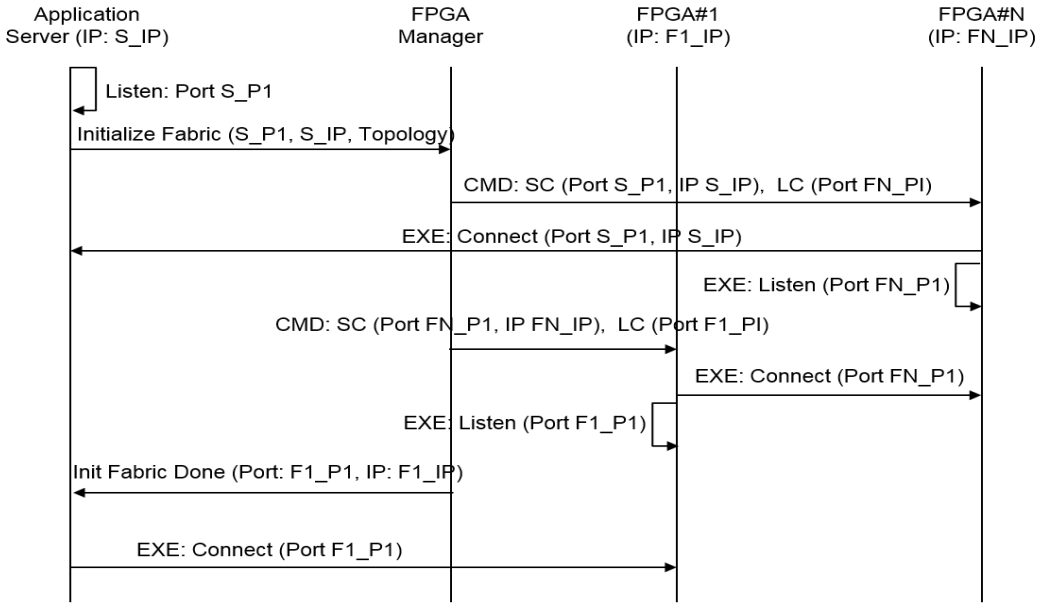


Figure 5.11: The flow of forming a multi-FPGA fabric

5.2.6 Flow of Building SDMFF

Figure 5.5 shows the flow of forming a SDMFF in a pipelined manner. The server running the SW part of the distributed application first starts listening on a particular port. Then it sends a fabric creation request to the FM with its listening port (S_P1) and IP address (S_IP). Upon receiving the request from the server, the FM decode the fabric topology as explained in Section 5.2.3, and sends appropriate connect (SC) and listen (LC) commands (CMD), shown in Table 5.2, to each FPGA as shown in Figure 5.11. Accordingly, fabric agent of each FPGA executes (EXE) the commands received by the FM to connect with the server application and the other FPGAs. Once all the FPGAs are connected, the FM sends a reply to the server with the listening port (F1_P1) and the IP address (F1_IP) of the first FPGA in the pipeline. Once this information has been received, the server starts a connection to that listening port (F1_P1). With that step completed, the SDMFF has been formed and the distributed application is ready to start execution.

5.2.7 Evaluation

Table 5.3 shows the reconfigurable resources consumed by the SDMFF extensions in the FPGA. Fabric agent consumed 621 LUTs, 458 FFs and 9 BRAMs, whereas the enhanced application switch consumed extra 551 LUTs, 760 FFs and 23 BRAMs. In the case of SDMFFs, as FPGA reconfiguration is not needed to build multi-FPGA fabrics, the fabric formation time (control path latency) is significantly reduced. We compared the SDMFF control path latency with the traditional ap-

Table 5.2: SDMFF Commands

CMD	Arg1	Arg2	Arg3	Arg4	Return Val
SC[1]	Destination IP	Destination Port	FIFO ID	–	Connection ID
LC[2]	Destination IP	Destination Port	Source Port	FIFO ID	Connection ID
CC[3]	FIFO ID	Connection ID	–	–	–

[1] Start Connection.
[2] Listen for Connection.
[3] Close Connection.

Table 5.3: Resource Consumption of TCP/IP Based Shell with MFFA

Module	LUT	FF	BRAM
IP	1789	2063	28
TCP	15159	16460	270
NET CTRL	4581	5101	11
MEM CTRL	39373	34172	38
Application Interface	2464	2528	38
Fabric Agent	621	458	9
Other (ARP/DHCP/Top Level)	26200	41526	42
Total	90187	102308	436
% of XC7VX690T	21	12	30

proach of new bit-stream generation and FPGA reconfiguration. Table 5.4 shows the time required to form a multi-FPGA fabric, which consists of two FPGAs, by the two methods considered: (i) by new bit-stream, and (ii) by SDMFF. The first approach took 29 minutes in our development platform to from the multi-FPGA fabric, whereas SDMFF took only 0.754 ms. Our development platform runs Xilinx Vivado 2016.4 version on RedHat Linux-based (RHEL 7.0) Intel machine (Single socket i7-3820 CPU @ 3.60GHz, 4 Cores, 8 Threads, 32 KB L1 Cache, 256 KB L2 Cache, 10 MB L3 Cache, 32 GB DRAM). Vivado is configured to use all the HW threads to fully optimize the compile time.

Table 5.4: Time for Multi-FPGA Fabric Formation with Two FPGAs

Fabric Formation Method	Control Path Latency
New bit stream generation + FPGA reconfiguration[1]	29 minutes
SDMFF	0.754 ms

[1] FPGA reconfiguration is performed remotely over the DC network, and it took 9 s out of 29 minutes.

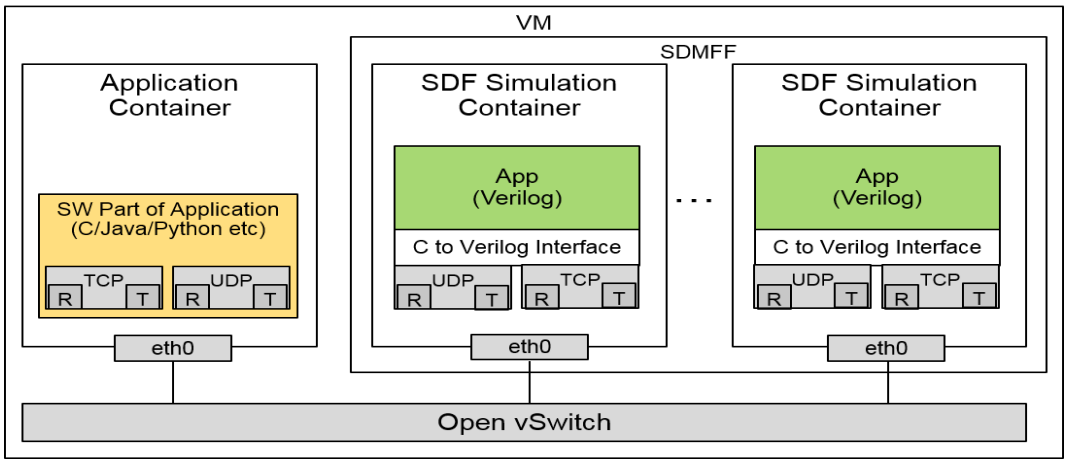


Figure 5.12: SDMFF Simulation Platform

5.2.8 Simulation Environment

The simulation environment for SDMFF is built by extending the simulation environment shown in chapter 4 for standalone disaggregated FPGAs. As shown in Figure 5.12, the SDMFF simulation environment is built in a single virtual machine, which consists of multiple Linux containers. All the containers are connected over an instance of a software-based openvswitch. One container runs the SW application while the rest of the containers runs the standalone disaggregated FPGA simulator.

5.3 Summary

Multi-FPGA systems are usually built targeting a specific application in mind. Hence, they are organized in a certain topology, tailored to exploit the maximum efficiency for the application considered. However, the applications that run on general-purpose cloud DCs change frequently. Moreover, heterogeneous computing has evolved to a level where more custom HW accelerators are preferred over more CPUs for maximum performance and energy efficiency. The amount of CPUs and custom HW required depends on the nature of the application. Further, interconnecting multiple FPGAs over the DC network requires the reprogramming of each FPGA, which in turn requires a considerable effort from building the bitstreams to reconfiguration. Even with that approach, changing the FPGA topology while the application is running disrupts the application.

Hence, a framework to interconnect multiple standalone disaggregated FPGAs on demand is proposed in this chapter. The architecture consists of an extension to the standalone disaggregated FPGA presented in Chapter 4 and a software framework to expose the HW extensions to the DC resource management. The HW extension includes a multi-FPGA fabric agent in the management layer and a pro-

grammable table, which consists of application FIFO IDs and network connection IDs, in the application switch. By interacting with the fabric agent, an external FPGA manager builds multi-FPGA fabrics in a software-defined manner on demand. This approach (i) allows the data path of distributed applications to be changed dynamically without FPGA reconfiguration and (ii) reduces the control path latency (time taken to form a fabric) for building multi-FPGA fabrics from 29 minutes to 0.754 ms, compared with the traditional approach of new bit-stream generation and FPGA reconfiguration.

Chapter 6

Experimental Validation by Applications

This chapter shows two applications deployed on top of standalone disaggregated FPGAs and SDMFF. The chapter is organized as follows:

In the first part (Section 6.1), a RESTful web service application on a standalone disaggregated FPGA is demonstrated. The RESTful IP block is elaborated in Section 6.1.1 and the web service application on top of the RESTful IP block is explained in Section 6.1.2. The experiments and results are discussed in Section 6.1.3.

In the second part (Section 6.2), a distributed text analytics application that runs atop SDMFF-accelerated UIMA distributed computing framework is demonstrated. First, text analytics is explained in general, followed by the explanation of UIMA distributed computing framework in Section 6.2.1. The enhancements done to the UIMA framework with PCIe-attached FPGAs and SDMFF is explained in Section 6.2.2. Next, porting of the text analytic application to standard UIMA is explained in Section 6.2.3 and the same application in enhanced UIMA is elaborated in Section 6.2.4. Experiments and results are shown in Section 6.2.5 and 6.2.6 respectively. The results observed, and the perspectives gained are discussed in Section 6.2.7. Section 6.3 summarizes the chapter.

6.1 RESTful Web Services

In today's cloud environments, many services can be accessed via RESTful APIs [175]. The REST (representational state transfer) provides interoperability between computer platforms and programming languages. A RESTful API uses the HTTP verbs (GET, POST, etc.) together with the uniform resource identifier (URI) to trigger an operation which will return a response in a pre-defined format, most commonly XML or JSON. Some services for example are natural language processing (NLP) tasks such as speech-to-text, text-to-speech (sentiment analysis) or image analysis.

Traditionally, these web services are provided via application servers that implement the HTTP protocol and then communicate via the common gateway inter-

face (CGI) or similar protocols with the application that implements the service function. Such an application can in turn access an FPGA to utilize accelerated functions and serve requests faster.

We present a configurable intellectual property (IP) block that is implemented on the standalone disaggregated FPGA prototype to allow these accelerated functions to be exposed to the HTTP client. The IP block implements the basic functionality of the HTTP protocol and decodes the requests according to the developers specification in OpenAPI format [176]. The block manages the connections for timeouts and inspects the HTTP header for all required fields and their respective values. The accelerated function remains in charge of processing the payload and generating a response payload.

6.1.1 REST IP Block

Any FPGA application that wants to expose its functionality as a RESTful web service must implement the HTTP protocol. Because it is a standard protocol many aspects of the communication can be implemented in an IP block that can be shared by all applications. A common FPGA design technique is to configure IP blocks which are then used by the application. Our REST IP block can be configured using an OpenAPI specification where a subset of features is currently supported.

6.1.1.1 OpenAPI Configuration

In order to configure the REST IP block, we leverage the use of an OpenAPI specification (OAS). It defines machine-readable interface files for describing and documenting RESTful web services. It defines the various URI paths and HTTP verbs that are available for the API. Furthermore, it defines required parameters for each API method and specifies what MIME type each method consumes and produces.

An FPGA developer can write this specification according to the services that the application supports. The specification is then consumed by our generator tool which creates a set of customized Verilog files which implement the specification. The customized IP block can then be instantiated in the FPGA design and connected with the actual application. The generator tool will in turn create a mapping between URIs and binary command words that are used to communicate with the application. Figure 6.1 outlines the design flow for creating the REST IP block for a specific application

6.1.1.2 Architecture

The architecture of the REST IP block (Figure 6.2) is designed for use with the shell-role architecture of the standalone disaggregated FPGA. The REST IP block resides in the ROLE and interfaces with the cloud shell over two FIFO-based AXI4 stream interfaces, which is an on-chip interconnect standard by ARM. One input

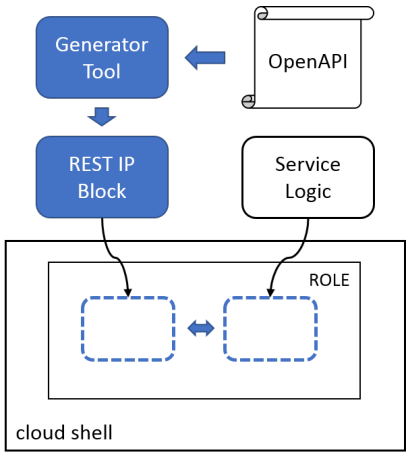


Figure 6.1: Design Flow for Designing with the REST IP Block.

stream and one output stream, which only carry the application layer data. The AXI4 stream standard defines an ID field which is used to indicate a connection ID and differentiate between different clients.

The first stage of the REST IP block decodes the HTTP header of an incoming request. Because the TCP protocol transmits at a segment level, the REST IP block needs to re-assemble these segments to a complete HTTP request message. While decoding the HTTP header, the IP block collects and stores the segments in DDR memory on the standalone disaggregated FPGA. It uses the *Content-Length* field to determine the overall length of the message. If this field is missing in the HTTP header, the request is flushed, and an error response is sent to the client (411 - Length required). Also, a limit can be set on the payload length. If that limit is exceeded, the request is flushed and the error response 413 payload to large is sent to the client.

The decoding stage uses a set of finite-state machines that can consume the input stream at wire speed (10 Gbps). Each state machine is responsible for a specific header field and translates the decoded element into a binary integer identifier, which is later used by the top-level state machine. If only a segment of the message is received the current state of the state machines are saved to a connection buffer on the FPGA. When the next segment of a connection arrives, the states are restored and the decoding continues. This context swap requires two clock cycles on the FPGA which is roughly 15 ns for this implementation.

The request's type and URI are decoded using the OpenAPI specification provided by the developer. The specification was used by our generator tool to create a custom state machine which maps the various API methods to a binary instruction word and is supplied to the actual application block. These commands are sent via an AXI4 stream interface once the complete payload has been received. The payload data is supplied via a second AXI4 stream interface to the application

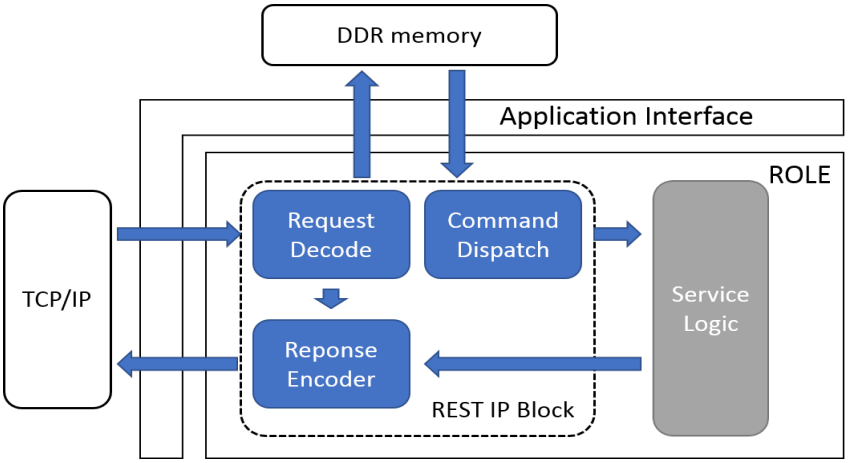


Figure 6.2: Main Modules of the REST IP Block in the ROLE of the Cloud Shell

module. Any mismatching URIs or MIME types are immediately flushed and an error response is sent to the client without the application being involved.

Responses are generated by the response encoder of the REST IP block. It consists of two finite-state machines. One responsible for the overall response and one for generating the header information. A response can be triggered from two sources: the application or the internal request decoding logic. All internal responses are error codes or the informational *Continue* response and do not have a payload. Applications in turn may trigger the responses *OK* or *Internal Server Error* which both can have a payload. A third alternative for applications is the *No Content*, which indicates successful execution of the API call but the response requires no payload.

If the application provides a payload it has two options: either it knows the length of the payload at the time it submits the response command. Or the payload length is unknown which is encoded as a zero-length payload with a generic OK response command. In this scenario, the response encoder logic will send an HTTP response header with the transfer encoding set to *chunked*. In this mode, the payload is sent via HTTP in chunks which are generated by the response encoder. Although this introduces some overhead to the processing performance of the FPGA, it is a nice feature of the HTTP protocol because it avoids storing data on the local DDR memory. The AXI4 conformant *TLAST* signal indicates the last piece of data to be sent in this mode from the application and concludes the response payload.

6.1.2 Web Service

As an example service, we choose a natural language processing (NLP) application which scans scientific documents for relevant entities. The service accepts plain

text documents and returns a set of annotations which is returned as a JSON object. The application logic is implemented using the annotation query language (AQL) FPGA compiler framework from [177]. The logic is capable of processing the document in a single pass by evaluating it one byte per clock cycle, which is an eighth of the line rate. Using multiple instances of the processing logic, this performance could be increased, but for the initial evaluation this has not been implemented.

The RESTful API specification defines which annotations should be returned to the client. Examples are */annotate/* where are types of annotations are returned or */annotate/ValueUnit* returning only annotations of the *ValueUnit* type. The text analytics application core generates all of these annotations in parallel, and we filter the results at the output of the core. The filtered results are then forwarded to the REST IP block where they are sent in *chunked* mode to the client.

6.1.3 Evaluation

To evaluate our example web service application presented in Section 6.1.1, we compare its performance against a server-based implementation. The server version has been evaluated as a pure software implementation and as an accelerated service using an FPGA.

6.1.3.1 Setup

All components are connected via a 10 Gigabit Ethernet network. The application client runs on an x86-based server at 2.6 GHz and 32 GB of memory. The server machine is an IBM POWER8-based server at 2.92 GHz and 512 GB of memory. The POWER8 processor has 20 physical cores and can run up to 160 hardware threads simultaneously. The server hosts a commercial-off-the-shelf (COTS) FPGA accelerator board based on a Xilinx Kintex UltraScale FPGA, which is connected via the CAPI interface to the processor. The standalone disaggregated FPGA is also implemented on the same FPGA card, which is powered by a PCIe extension chassis. The card is connected via SFP+ connectors to the 10 GbE network. Figure 6.3 illustrates the experimental setup.

We use Apache JMeter [178] as the client application to generate the API requests from the client. We use HTTP 1.1 as the communication protocol which defines all connections to be persistent. This means that the TCP connection is kept alive and multiple request/response pairs can be sent. This reduces the overhead of re-establishing a TCP connection for each call. JMeter uses multiple threads to create the requests and collects information about the processing performance.

On the server machine we run a nginx HTTP server [179] with four worker processes. The actual web service is a Python application hosts by the uWSGI [180] application server running 20 processes. The two servers communicate via the Python standard web server gateway interface (WSGI). The web service applica-

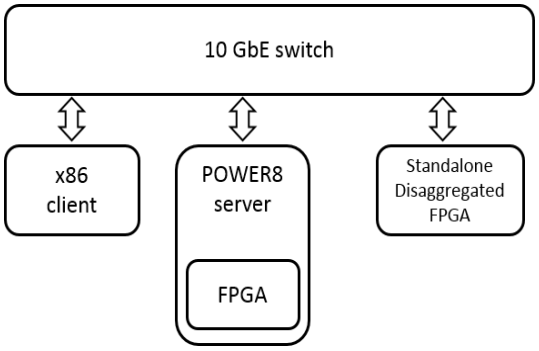


Figure 6.3: Experimental Setup.

tion can access the accelerated functions on the CAPI-attached FPGA via a Python extension and can run in accelerated or in standard mode.

To evaluate the impact of the individual components we ran the tests in five scenarios:

- **nginx:** nginx only serving a static file
- **nginx+uwsgi:** nginx making a call to an WSGI app that immediately returns an empty result
- **nginx+uwsgi+app:** An actual API call to the application with all processing performed in software
- **nginx+uwsgi+app+fpga:** An actual API call where the application utilizes the accelerated function on the FPGA
- **standalone disaggregated FPGA:** An actual API call where all processing is done on the standalone disaggregated FPGA

6.1.4 Results

Apache JMeter reports the overall processing throughput in requests per second as well as the minimum, maximum and mean processing time of the individual API calls. Table 6.1 summarizes the results of all five test case scenarios when there are 100 simultaneous requests in flight.

The pure nginx baseline performance is at 37,389 requests/s which is in line with numbers that can be found on the web. As there is no application processing involved at all, this is the highest rate at which the service could be operated on the POWER8 server. When adding a WSGI call to the picture the throughput performance drops by nearly a factor of four but the jitter for the individual processing time is relatively small.

When running the full application API call, the performance significantly drops. The maximum throughput observed was 540 requests per second which results in

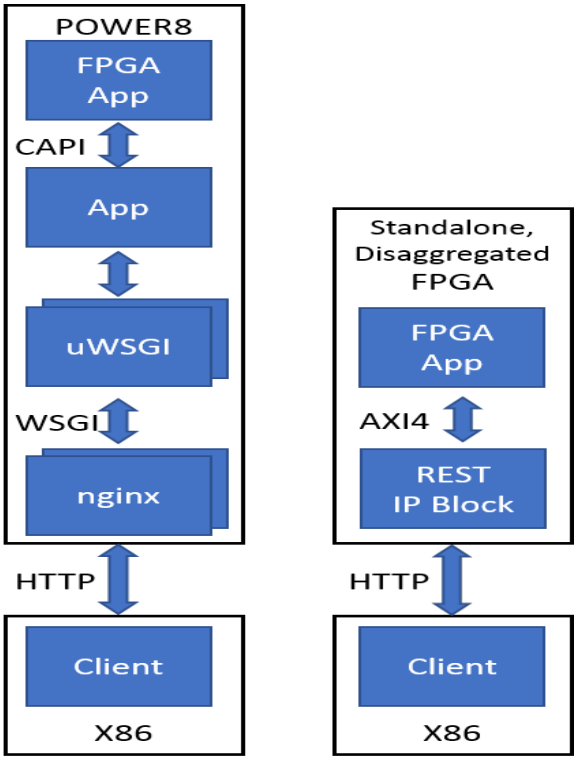


Figure 6.4: Involved Processes, Modules and Communication Protocols on the POWER8 Server and the Standalone Disaggregated FPGA Scenarios.

a mean processing time of 175 ms for the client. Also, the variation of the processing time becomes much wider. While the fastest call required 31 ms the slowest took six times longer, 197 ms. When the application uses the CAPI-attached FPGA the performance numbers are getting better again. The throughput increases to 7,917 requests/s and the processing times do not exceed 9 ms for the client.

With the proposed architecture the entire service is running on the standalone disaggregated FPGA. The client makes a direct RESTful API call to the FPGA, thus the entire communication and application processing occurs on a single chip. Our architecture was able to provide a processing throughput of 166,093 requests/s which is more than 20 times higher than the accelerated version of the service on a high-end server node. All processing calls required a maximum of 1 ms to complete. In a real-life environment, this number would depend on the actual distance in the network from the client to the server. The results show that application throughput is increased by 15x with the acceleration of only application logic, whereas the throughput is increased by 308x with the acceleration of whole stack, including TCP, HTTP, REST, and the application. Therefore, compared to the traditional approach of acceleration using PCIe-attached FPGAs, standalone disaggregated FPGA performs 20x better.

Table 6.1: Results for the performance measurements running with 4 nginx processes and 20 uWSGI processes.

Case	Requests/S	Minimum (ms)	Mean (ms)	Maximum (ms)
nginx	37,389	1	1	2
nginx+uwsgi	15,930	2	6	8
nginx+uwsgi+app	540	31	175	197
nginx+uwsgi+app+fpga	7,917	1	3	9
standalone disaggregated FPGA	166,093	0	0	1

Another interesting effect was observed with the number of concurrent requests. The standalone disaggregated FPGA performs significantly better than the server implementation in situations where there is little or very high load. Figure 6.5 shows the number of requests per second over the number of concurrent requests that are in flight. If there is only one request in flight the performance is mainly latency driven, therefore the pure hardware implementation on the standalone disaggregated FPGA is much faster. With an increasing number of simultaneous requests the performance depends more on the overall processing power of the server. The actual web service gains more from processing more connections at the same time as the baseline nginx measurement or the standalone disaggregated FPGA. But while these measurements remain stable with their performance for 1,000 requests, they uWSGI-based measurements drop again.

6.1.4.1 Power Consumption

Power efficiency always has been a stronghold for FPGAs. With the standalone disaggregated FPGA, this efficiency can be fully exploited in a cloud environment. While the standalone disaggregated FPGA requires at maximum 25 W to operate the web service, the fully equipped POWER8 server requires 340 W in idle mode and more than 360 W when running the web service. The POWER-based servers can be measured using the on-system sensors of the server [181]. This is more than 13 times compared to the standalone disaggregated FPGA.

6.1.4.2 System Cost

Adding an FPGA to each server in a DC environment significantly increases the cost of a server unit. Microsoft catapult implementation has increased this cost by 30% [25] . For FPGA-centric applications the server that hosts the FPGA might not be efficiently used. Hence, by deploying standalone disaggregated FPGAs, the cost of the server can be omitted from the total cost of the FPGA infrastructure. As an example, considering the application used in this paper, the system cost can be

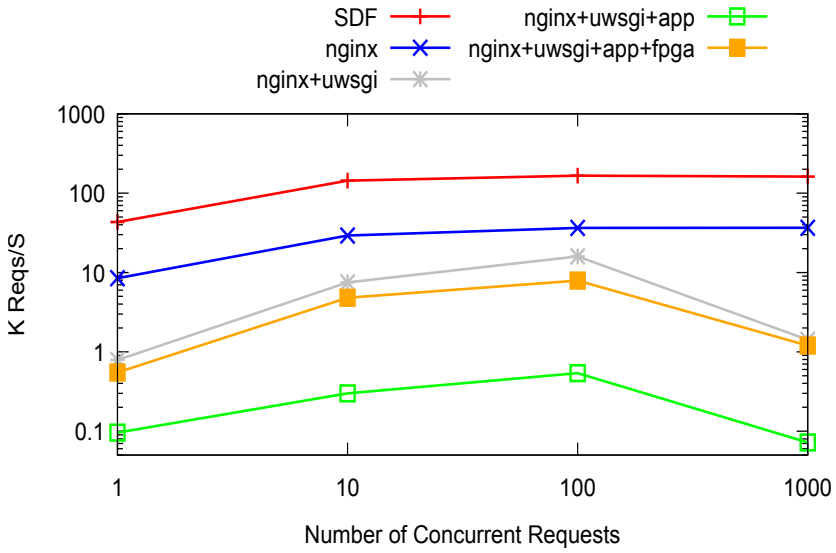


Figure 6.5: Number of Requests Served Over Different Number of Concurrent Requests on Log Scale (SDF: Standalone Disaggregated FPGA).

cut down by around $5000 (according to price on web), by completely eliminating a server to host the web application.

6.1.4.3 Resource Consumption

When developing on FPGAs the individual functional modules require resources like configurable logic blocks (CLBs) or internal memory elements (BlockRAM). Therefore, it is important to keep the resource consumption low for the interface logic to allow the actual application to utilize the remainder. Together with the cloudFPGA's network service layer, the REST IP Block requires about 20 % of the overall logic resources of the FPGA and 30 % of the internal memory blocks. This is comparable with the resources required by the POWER service layer to implement the CAPI protocol. With newer generations of FPGAs, more resources will be available for the application module.

6.2 Distributed Text Analytics

Text analytics refers to the task of information extraction from documents containing natural-language text. The main aim is to transform the unstructured information contained in these documents into a structured form, i.e. tables. This is an important processing step in many of today's big data analytics applications ranging from social media analysis to compliance check and data-center log surveillance. The document-level analysis needs to be run before any higher-level algorithms can perform further analyses.

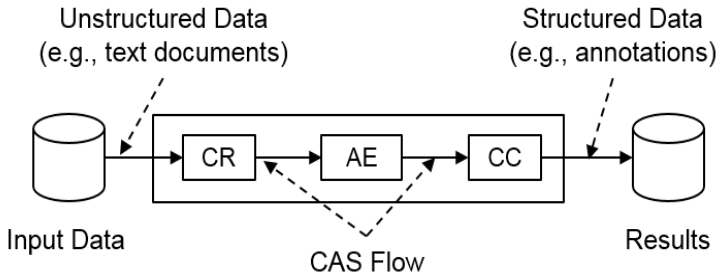


Figure 6.6: UIMA Pipeline

Text analytics involves several steps from the natural-language processing (NLP) domain, such as tokenization or named-entity recognition (NER). Each step may be performed by either a rule-based or a machine-learning-based implementation. While machine-learning-based approaches are well established, rule-based approaches are often maintained in the enterprise domain to achieve fast and deterministic results [182].

Tokenization is usually the first step when analyzing a document, which breaks up the input text into individual words and characters. For many Western languages, whitespace tokenization is often sufficient to produce a useful set of tokens by splitting the text on whitespace and punctuation characters. The tokens are then used by subsequent processing steps, such as, dictionary pattern matching or distance checks (how many tokens are two entities apart).

Named-entity recognition (NER) identifies words or patterns in the document text and assigns them to categories. Words or word sequences can be determined to be a person's name or a geographic location, whereas character sequences may be identified as telephone numbers or date/time stamps. Thus, NER involves pattern matching in the form of dictionary matching, which requires a pattern to match on token boundaries, and regular expressions, which operate on the document text without token definitions. Both operations are essential for text analytics and have shown significant performance benefits when run on FPGAs [183] [184] [185].

6.2.1 UIMA

Several libraries exist to perform natural-language processing tasks. Every library uses its own type system to create and exchange information between different processing steps. This makes the integration of several libraries into a single application difficult. The Unstructured Information Management Applications (UIMA) [35] specification is an OASIS (Organization for the Advancement of Structured Information) standard that defines how various components can define a common type system and how data is exchanged between them. The UIMA framework also manages the execution of multiple components and can be configured

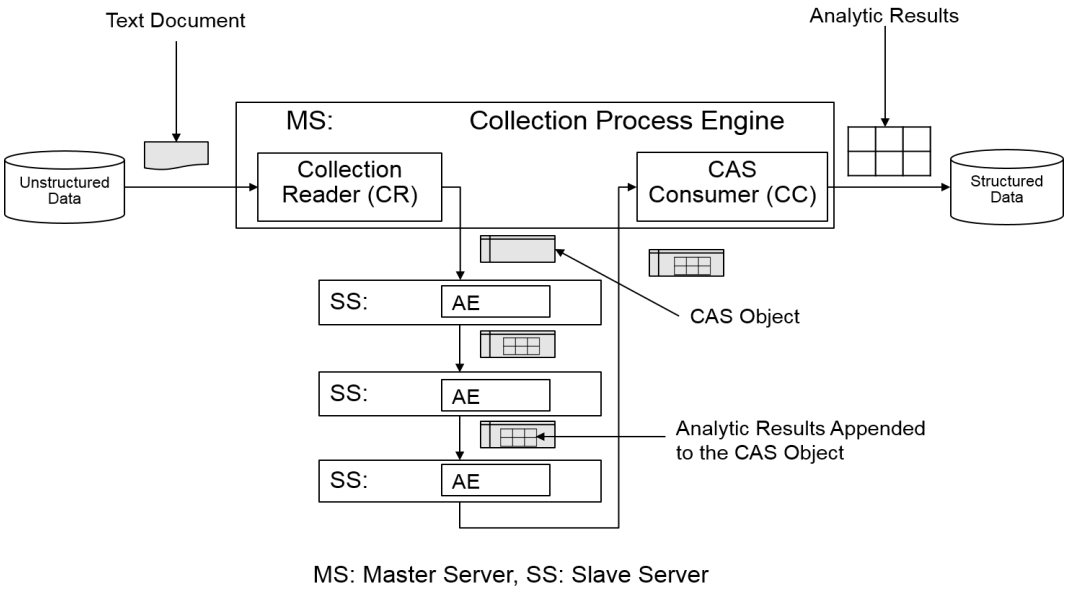


Figure 6.7: Standard UIMA Pipeline with Multiple Hosts

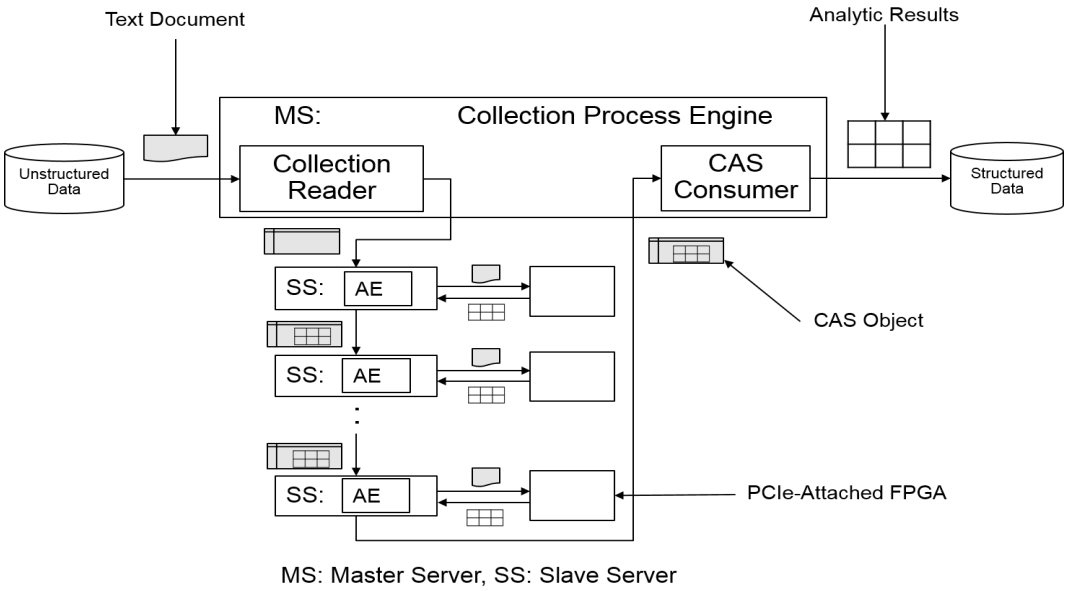


Figure 6.8: Standard UIMA Pipeline with Multiple Hosts Enhanced with PCIe-Attached FPGAs

to create a processing pipeline [31].

UIMA defines multiple component types and the Common Analysis Structure (CAS) data structure. The CAS object is the central data structure that is created for every document. Every analysis step can retrieve information from the CAS or create results on it. As shown in Figure 6.6, the three main components to create a UIMA processing pipeline are (i) Collection Reader (CR), (ii) Analysis Engine (AE), and (iii) CAS Consumer (CC).

The CR is the input component and retrieves the documents from an input source, i.e., filesystem or database. It creates the initial CAS object and passes it to the actual processing pipeline. The processing pipeline is a so-called aggregated analysis engine (AE) consisting of multiple primitive AEs. The UIMA framework will call specific routines from each of the primitive AEs to process every document. When processing completes, the CS will receive the CAS object and store the relevant results to an output destination.

To create large-scale high-throughput and low-latency applications, UIMA offers an asynchronous scale out (UIMA-AS) [186] version. Instead of calling the processing routines of the AEs synchronously, the individual AEs receive their input CAS from a queue that is assigned to each AE. This allows every AE to work on its own input queue whenever data becomes available. It also simplifies multi-threading as multiple individual threads that run the same AE share the same input queue.

To scale out the analysis application to multiple machines in a server cluster (Figure 6.7), UIMA-AS uses the Java Message Service (JMS) and a messaging broker to manage the queues. The CAS object is communicated via the individual nodes as a serialized XML object, and connections are made through TCP or HTTP. If the CAS object remains on a node, it is kept as a binary object. On each node, one or multiple AEs can be deployed together with their corresponding queues. The deployment descriptor of UIMA-AS specifies how many threads should be used to run a specific AE and additional parameters. The CAS object is then sent around between the individual nodes for processing before returning to the master server node, where the application resides.

6.2.2 Enhanced UIMA

To use FPGAs to accelerate the applications on UIMA framework, we enhanced the UIMA framework by incorporating FPGAs. To compare the benefits of using disaggregated standalone FPGAs versus PCIe-attached FPGAs, we enhanced the UIMA framework by using both the PCIe-attached FPGAs and the SDMFF platform. The next two subsections explain the enhancement done to the UIMA framework.

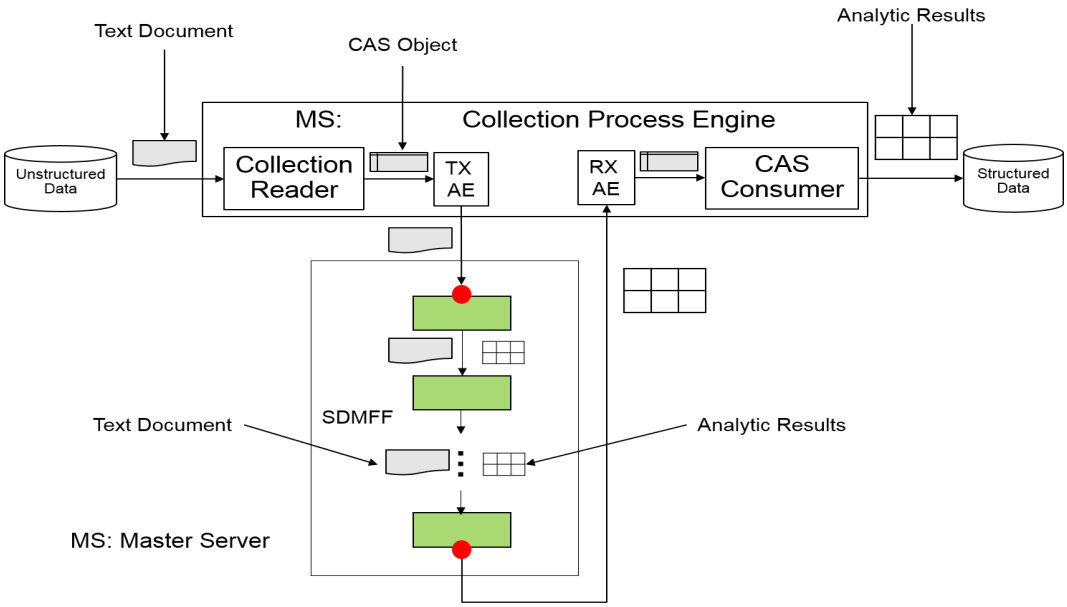


Figure 6.9: SDMFF-Enhanced UIMA Pipeline

6.2.2.1 PCIe-Attached-FPGA-Enhanced

In the standard UIMA framework, each of the slave server node that runs the Analytic Engines is enhanced by attaching an FPGA over the PCIe bus (Figure 6.8). The slave servers we used are based on IBM power 8 servers, hence it enables to use CAPI for CPU-FPGA communication. When the FPGA is attached via PCIe to each slave server node, there is no change in the UIMA communication scheme, but the AE's code that uses the FPGA is adapted, so that the incoming embedded document in CAS object is retrieved and fed to the FPGA. After processing is completed, the results returned by the FPGA is in turn attached to the CAS object and forwarded to the next slave server node in the UIMA pipeline.

6.2.2.2 SDMFF-Enhanced

For using SDMFFs, we alter the communication structure by adding a sending (TX AE) and a receiving (RX AE) primitive AE to the processing pipeline that remain on the master server node (Figure 6.9). The descriptor of the TX AE contains all information necessary to set up the multi-FPGA fabric pipeline before starting the processing step. In this setup, the FMU is also running in the master node. Once the fabric is formed, every FPGA knows where it receives data from and where it has to send its results. When processing, the TX AE will send only the text document embedded in the CAS object to the first FPGA in the processing pipeline and forward the CAS to the RX AE. As this is a local operation, no network communication is involved. The RX AE will wait for the FPGA processing pipeline

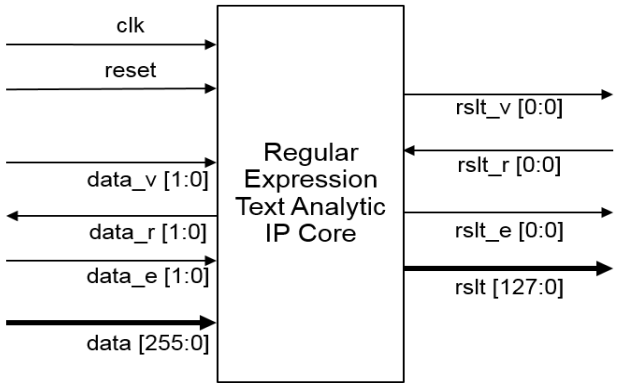


Figure 6.10: Regular Expression Text Analytic IP Core

to complete processing the text document, which was embedded in the CAS, and then add the results to the actual CAS object.

This setup allows a flexible and efficient communication without implementing a complex JMS stack on the individual FPGAs. The UIMA-AS framework enables the TX and RX AEs to continuously send and receive data while maintaining UIMA compliance.

6.2.3 Text Analytics on Standard UIMA

Regular expressions can be implemented in Java using the built-in java.util.regex package. This enables a straight forward software reference implementation for a UIMA processing pipeline.

6.2.4 Text Analytics on Enhanced UIMA

To run a text-analytics task on the FPGA, we use IP cores generated by the compilation framework presented by Polig *et al.* [187]. In this work, we use a regular expression IP core as the application (Figure 6.10). The framework compiles queries written in the Annotation Query Language (AQL) to a hardware description (verilog) that can be synthesized to FPGA logic. The top-level module generated uses AXI-Streaming-like interfaces to accept an input document and produce output results.

The input document stream is synchronized with an optional token stream, which defines the token boundaries. If no token definitions are available, this stream must indicate the document size for the input logic to consume the document stream. The results are annotations in the form of four integers: two defining the begin and end the position of the annotation, and one identifier value to determine the type of annotation. To employ generally available libraries, we use the compilation framework only to compile regular expressions.

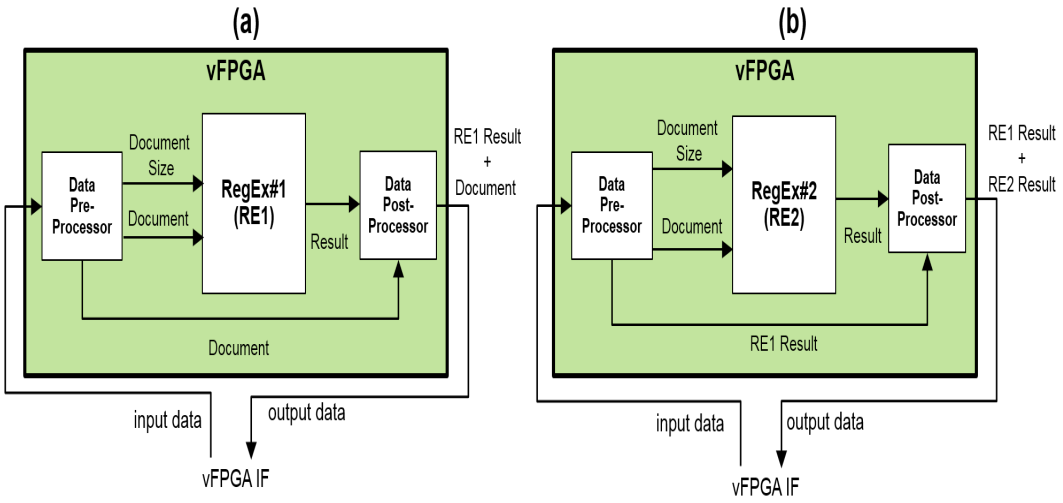


Figure 6.11: Implementation of vFPGA for the application: (a) Standalone Disaggregated FPGA1 and (b) Standalone Disaggregated FPGA2

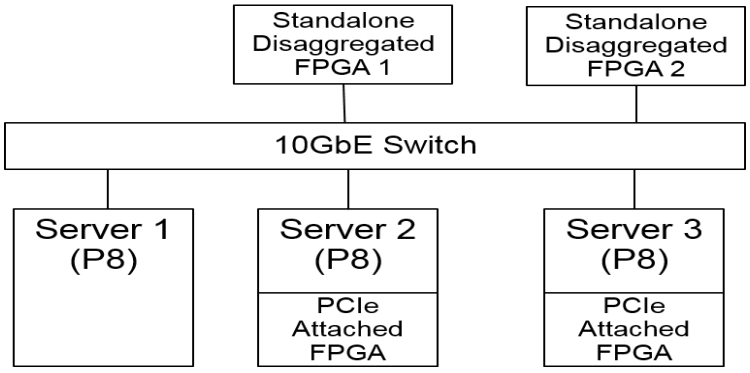


Figure 6.12: Experimental setup

$$Throughput = ClockFrequency \times DataWidth$$

The regular expression core process one byte in each clock cycle. If the clock frequency considered is 156.25 MHz, the maximum throughput of the core is 1.25 Gbps.

In this work, we use a regular expression IP core as the application, and an SDMFF that consists of two standalone disaggregated FPGAs. This application is hosted by the vFPGA as shown in Figure 6.11. The vFPGA has three main components: (a) a data pre-processor, (b) a data post-processor, and (c) the regex core. The data pre-processor and the post-processor are collectively called the vFPGA application wrapper. The data pre-processor executes two functions: First, the regex core must be fed with each document and its size. When documents are sent over

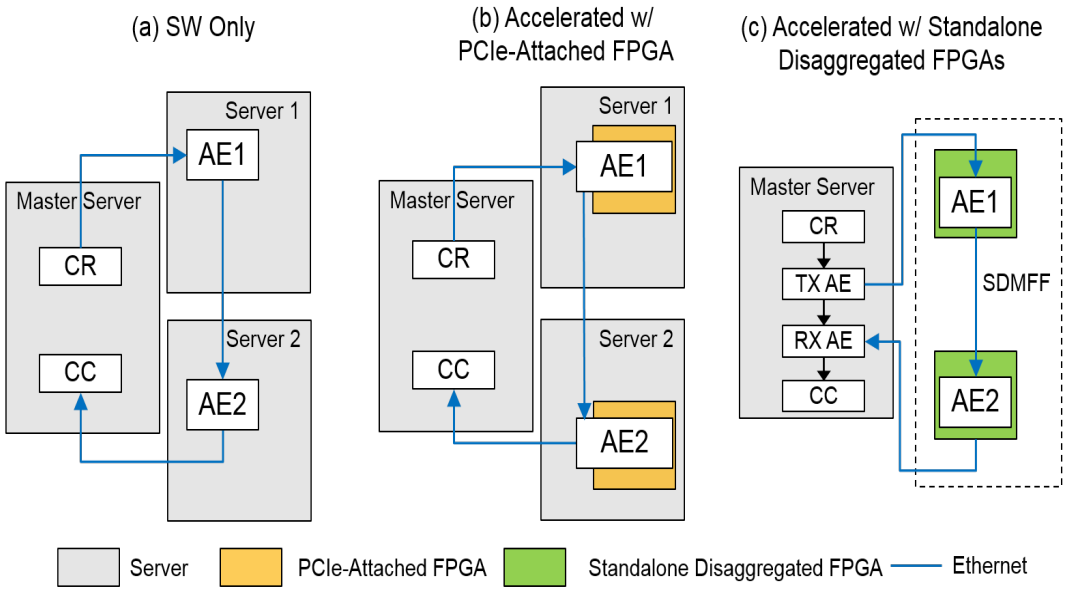


Figure 6.13: UIMA Pipeline-based Experimental Cases

TCP, the document boundaries are lost. Hence, the document boundaries must be recovered before feeding data to the application. Therefore, a messaging layer atop TCP is implemented in the data pre-processor. The server sends each document with its size prepended in 8 bytes. According to the document size, the messaging layer stacks TCP data belongs to each document into a single AXI stream message and forwards it to the application. Second, as the data bus width of the regex core is 128 bit, but the bus width of the vFPGA is 64 b, we convert the bus width from 64 b to 128 b. For the conversion, we used Xilinx AXI stream data-width converters.

The data post-processor first converts the data width of the results generated by the application from 128 bit to 64 bit, and then combines the converted result with the data coming directly from the data pre-processor before forwarding everything to the vFGPA interface.

As shown in Figure 6.11, the data pre-processor of vFPGA in standalone disaggregated FPGA1 forwards the document as it is to the data post-processor, whereas in the vFPGA of standalone disaggregated FPGA2, the data pre-processor discards the document after feeding the application with the document data and its size. After that, the data pre-processor forwards the result received from the vFPGA of standalone disaggregated vFPGA1 to the data post-processor. The data post-processor of the vFPGA in standalone disaggregated FPGA2 combines the results of standalone disaggregated FPGA 1 and 2, and forwards it to the vFPGA IF, which eventually ends up in the RX AE in the UIMA framework.

6.2.5 Evaluation

We evaluated our architecture in terms of network latency, throughput, and lantecy variation. The standalone disaggregated FPGA architecture presented was implemented and validated on a Alpha Data PCIe card featuring a Xilinx Virtex7 XC7VX690T FPGA. The design uses one 10 GbE network interface.

As shown in Figure 6.12, the experimental setup consists of up to three server nodes, each equipped with two IBM POWER8 processors running at 3.5 GHz. Two servers contain commercial off-the-shelf FPGA accelerator cards using an Altera Stratix V A7 FPGA and are connected to the processor via CAPI. All servers run Linux Kernel 3.10 and use IBM Java version 8 and UIMA 2.8.1. The standalone disaggregated FPGA cards are plugged into a PCIe expansion chassis [22], from which only power is taken for the cards. The FPGA cards and the three servers are connected to a 10 GbE top-of-rack switch.

We considered three experimental cases: (i) a SW-only implementation (Figure 6.13-(a)), (ii) an implementation accelerated with PCIe-attached FPGAs (Figure 6.13-(b)), and an implementation accelerated with standalone disaggregated FPGAs (Figure 6.13-(c)). One server is considered the master server and is responsible for the running the collection reader (CR) and the CAS consumer (CC). It also runs the aggregate analysis engine, which coordinates the order in which the primitive analysis engines (AEs) are executed. The primitive AEs are run by the other two server nodes. These server nodes run either the software-only variants of the AEs implemented in pure Java or the FPGA-accelerated version.

For the standalone disaggregated scenario, the two servers running the primitive AEs are replaced by the standalone disaggregated FPGAs. The master server remains in charge of reading the documents and collecting the results, but now also sets up the multi-FPGA fabric using the FPGA management utility described in Section 5.2.3 and sends and receives data to and from the FPGAs. Internally the CAS object moves from the sending AE (TX AE) to the receiving AE (RX AE), which involves no data movement or processing.

As explained earlier, both of our AEs perform regular expression matching. AE1 identifies several date formats, while AE2 reports credit card numbers in the document. The SW case uses the standard Java regular expression class, whereas the FPGA version is compiled using [187].

6.2.6 Results

The latency and the throughput measurements are done by using standard UIMA tooling. By running *runRemoteAsyncAE*, the number of processed documents and characters are reported together with the required run time. These numbers represent an end-to-end measurement for processing a document collection. The scaleout deployment has been tuned by using available knobs from the UIMA-AS

framework, such as the number of threads running an AE or taking care of the (de-)serialization of the CAS, CAS pool size and memory requirements.

6.2.6.1 Latency

Figure 6.15 shows the latency results in milliseconds for different document sizes. The SW-only implementation took 6, 8, and 20 ms for the document sizes of 512, 1024, and 2055 B, respectively, whereas the implementation accelerated with PCIe-attached FPGAs took 6, 7, and 18 ms. Regardless of the document size, the standalone disaggregated FPGA implementation took 0.5 ms. When moving from SW-only version to PCIe-FPGA version, we observed a minor improvement in latency. In contrast, for standalone disaggregated FPGAs, the latency improved by a factor of 40 compared with SW-only version.

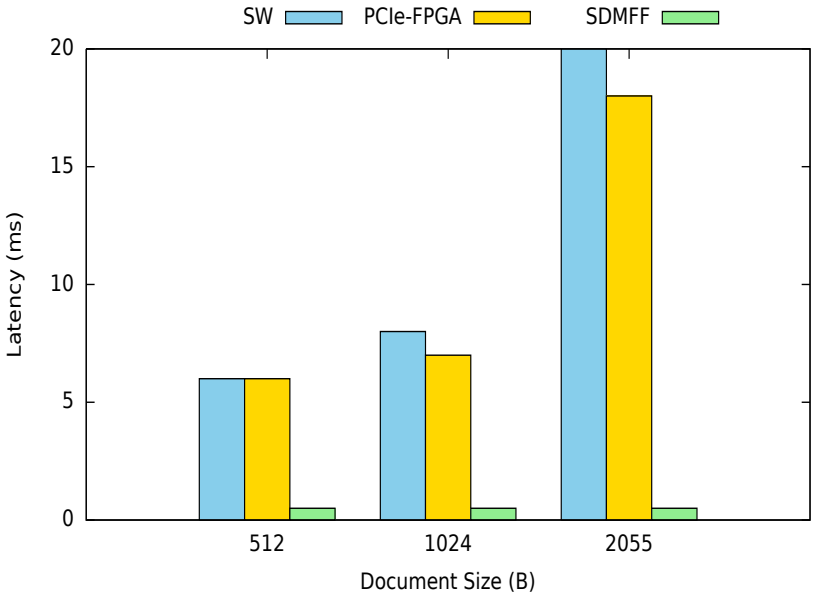


Figure 6.14: Text Analytics on UIMA: Latency

6.2.6.2 Throughput

Figure 6.16 shows the throughput results in number of characters per second for different document sizes. Similarly to the latency results, when comparing the throughput between the SW case and the PCIe-FPGA version, we observe a minor improvement of 30%. In contrast, for network-attached FPGAs, the throughput increases by a factor of 14 for the two smaller document sizes and by a factor of 18 for the 2 kB case.

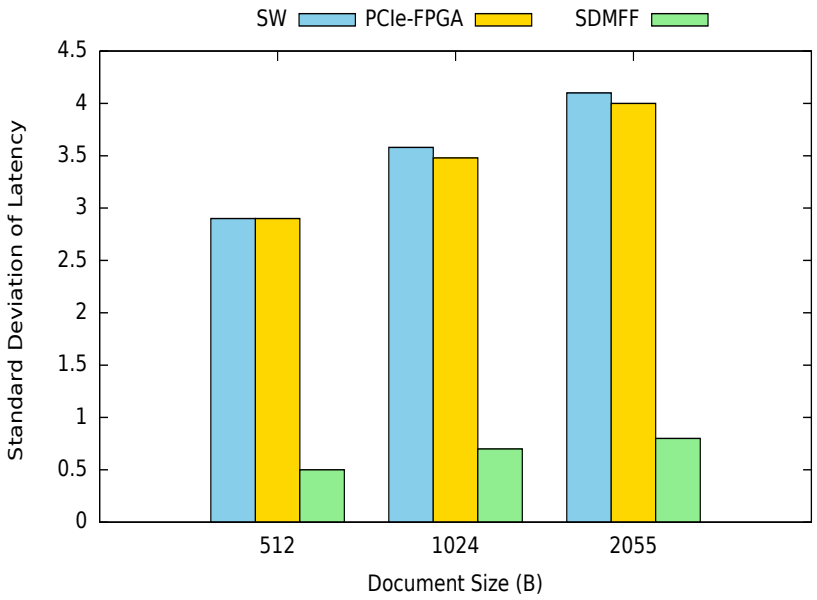


Figure 6.15: Text Analytics on UIMA: Latency Variation

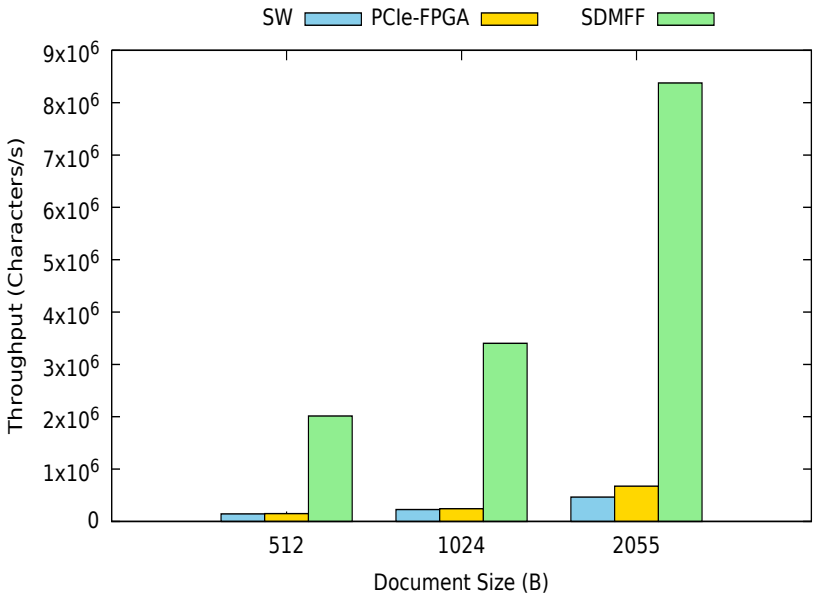


Figure 6.16: Text Analytics on UIMA: Throughput

6.2.6.3 Latency Variation

We evaluated the latency variation by considering the latency of one million iterations for each document size. The standard deviation of the latency distribution is shown in Figure 6.15. The standard deviation of the latency ranges between 2.9

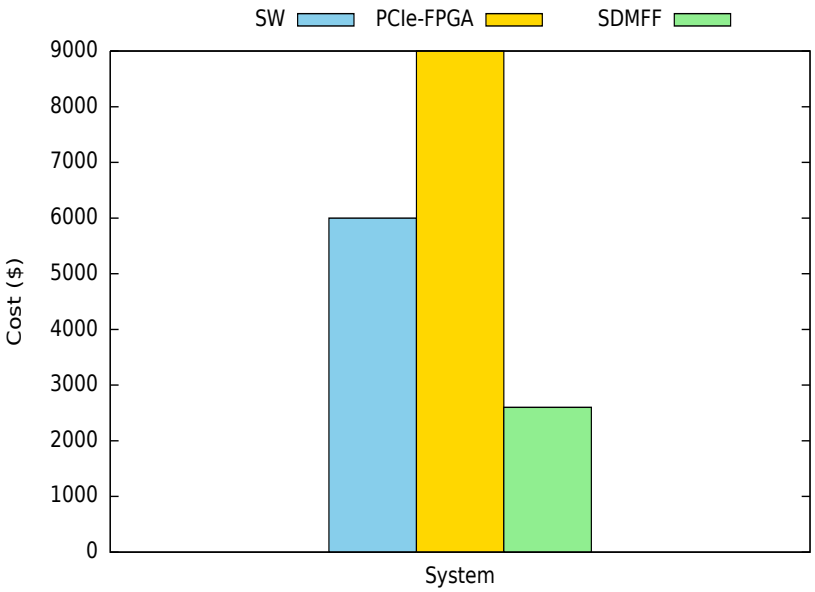


Figure 6.17: Cost Comparison of Three Text Analytics Systems

and 4.1 in the case of SW-only and PCIe-attached FPGA implementations, where as for the standalone disaggregated FPGAs it ranges between 0.5 and 0.8. Our preliminary experiments for inter-FPGA communication with raw TCP data showed that the standard deviation varies between 0.2 to 0.4, but in these experiments the master server, which generates and receive data contributes to the increment of the variation.

6.2.6.4 System Cost

The system cost of three experimental scenarios are considered assuming that the cost of a server is 2000$, the cost of an off-the-shelf FPGA card is 1500$ and the cost of a custom-built FPGA is around 500$. As shown in Figure 6.17, the cost of PCIe-attached version is 1.5 times higher than that of the pure SW system and the cost of the system accelerated by standalone disaggregated FPGAs is 2 times less than the pure SW system.

6.2.7 Discussion

6.2.7.1 Performance

The minor performance improvement observed when moving from SW only to PCIe-FPGA reflects the overhead of the scaleout framework. Because the actual processing time for running the regular expression in the FPGA is much shorter than in SW. The regular expression used for AEs requires 700μs on average in SW, while only taking 18μs in the FPGA. This is a 38-fold improvement, but the

communication overhead of the framework still dominates the end result of latency, throughput, and latency variation. Compared with the SW-only version and the acceleration of the application with PCIe-attached FPGAs, the tight coupling of network packet processing and computation within the FPGA, and the elimination of the overhead of the framework when sending and receiving CAS objects (CAS-object creation, the serialization and de-serialization) lead to superior latency, throughput, and latency variation results for standalone disaggregated FPGAs. For standalone disaggregated FPGAs, the application throughput is limited by the throughput of a single TCP connection in the FPGA TCP/IP stack. Currently, we are working on improving this per-connection throughput.

An alternative way to setup the PCIe-attached FPGA experimental case is to add a sending AE and a receiving AE to the master server, similarly in the case of standalone disaggregated FPGAs. Even if we do this, the servers that host the PCIe-attached FPGAs have to execute the network packet processing, which significantly degrades the overall application performance, particularly when the processing pipeline gets longer. Furthermore, a DC-class server consumes around 200 W of power, where as an FPGA device consumes around 25 W. Hence, moving from PCIe-attached FPGA implementation to standalone disaggregated FPGAs, we achieve an order of magnitude improvement in power consumption.

Table 6.2: Resource Consumption of TCP/IP-Based Shell with Application

Module	LUT	FF	BRAM
IP	1789	2063	28
TCP	15159	16460	270
NET CTRL	4581	5101	11
MEM CTRL	39373	34172	38
Application Interface	2464	2528	38
MFFA	621	458	9
vFPGA APP Wrapper	2761	1650	14
APP(RegEx)	5548	6807	2
Other (ARP/DHCP/Top Level)	26760	42566	59
Total	99056	111805	469
% of XC7VX690T	23	13	32

6.2.7.2 Resource Consumption

Table 6.2 shows the resource usage of the standalone disaggregated FPGA prototype with the application. The design uses around 99K LUTs, 111K Flip-Flops, and 469 BRAMS, which is equal to 23%, 13%, and 32% of the overall available resources, respectively. Out of the total resources, the NSL consumes around 20% of the LUT, 11% of the FF and 29% of BRAM resources, contributing the most for

the resource consumption. In this work, the application we use consumes only a very low amount of resources, but in our future work we want to experiment with applications that fully use the FPGA resources.

6.3 Summary

In this chapter, the standalone disaggregated FPGA and SDMFF prototypes presented in the Chapters 4 and 5 were demonstrated with real-world use cases.

In the first use case, an HTTP-based RESTful layer was implemented in the standalone disaggregated. A natural language processing application was ported on to this RESTful web service layer. This implementation was compared with a pure SW implementation and an accelerated version of it using PCIe-attached FPGAs. The results show that the standalone disaggregated FPGA increases the application throughput by 300x and 20x compared to the pure SW implementation and the accelerated version, respectively.

The second use case was built by porting a distributed text-analytic application onto an UIMA distributed computing framework. Text analytics refers to the extraction and transformation of information from unstructured data in text documents to a structured from. This is an important step of big-data analytics applications to gain insights from the data. When a single node is not sufficient to execute these tasks efficiently, multiple nodes are used. To distribute tasks onto multiple nodes, distributed computing frameworks, such as UIMA, are used. In this work, UIMA is used to make a processing pipeline for the multi-node execution of text-analytics applications.

The UIMA-based distributed text-analytics application is built in three flavors: (i) pure SW-based, (ii) accelerated with PCIe-attached FPGAs and (iii) accelerated with SDMFF. In the first case, text analytics is executed on SW, where as in latter two cases text analytics processing is accelerated by FPGAs. In case II, UIMA-based processing nodes are enhanced with PCIe-attached FPGAs, and in case III, server-based processing nodes are replaced by a SDMFF. The comparison shows that text analytics on the SDMFF-accelerated UIMA framework outperforms both other implementations by large margins, and improves the latency, the throughput, and the latency variation by a factor of 40, 18, and 5, respectively. The insights gained from these results open the way for running large-scale applications on standalone disaggregated FPGAs in DCs.

Chapter 7

Conclusion and Directions for Further Research

7.1 Conclusion

The performance requirements of modern big-data applications are exceeding the performance offered by general-purpose servers in current cloud data centers. Further performance improvements from CPUs are limited because of the physical constraints of CMOS and the lack of viable alternatives to the CMOS technology. These trends have increasingly attracted interest for HW accelerators in mainstream computing over the recent years. HW accelerators can have many forms: FPGAs, GPUs, ASICs, DSPs etc. This thesis focused on FPGAs by investigating a system architecture to efficiently deploy and use FPGAs at large scale in cloud data centers.

The provisioning of FPGAs as standalone resources with direct connections to the data-center network is one of the key enablers for a large-scale deployment of FPGAs in data centers. This is a profound change of paradigm in the CPU-FPGA and inter-FPGA communication. The standalone disaggregated FPGA promotes the FPGA to the rank of a peer processor in the data center. Data centers must take this paradigm shift into account to host FPGAs and other similar heterogeneous computing resources on a large scale in the future. This thesis proposes an architecture to deploy standalone disaggregated FPGAs in a density-optimized manner. Compared to FPGA clusters built by off-the-shelf HW and state-of-the art large-scale FPGA deployments, the proposed system increases the rack FPGA density by 2x.

The applications that run on data centers change frequently. Hence, two key requirements that must be satisfied by the FPGA infrastructure are (i) flexible allocation of CPUs and FPGAs to an application on demand and (ii) the ability to connect those FPGAs in user-defined topologies. To achieve this, this thesis proposes software-defined multi-FPGA fabrics (SDMFF) by extending the functional-

ity of standalone disaggregated FPGAs. SDMFFs are multi-FPGA fabrics built by interconnecting multiple standalone disaggregated FPGAs in arbitrary topologies over the DC network in a software-defined manner. The feasibility of SDMFFs is shown by building a prototype using commercial FPGAs in a real data-center environment. This thesis also compares the latency, throughput and predictability of multi-FPGA fabrics compared with those of other data-center compute resources, such as bare-metal servers, virtual machines and containers.

Finally, a SDMFF prototype is evaluated in a real-world data-center application. The application executes text analytics on an UIMA distributed-computing framework. We compared three versions of the distributed text-analytics application: (i) based on pure SW, (ii) UIMA enhanced with PCIe-attached FPGAs, and (iii) UIMA enhanced with SDMFF. In the latter two cases, text analytics processing is accelerated by the FPGAs. The comparison shows that standalone disaggregated FPGAs outperform both other implementations by large margins, and improves the latency, the throughput, and the latency variation by a factor of 40, 18, and 5, respectively.

The insights gained from the work presented in this thesis open the way for (i) cloud vendors to deploy standalone disaggregated FPGAs at large scale in data centers, and (ii) cloud users to improve the performance of their applications by using the scalable and flexible FPGA infrastructure provided by the cloud vendors.

7.2 Directions for Future Work

Deploying FPGAs as standalone disaggregated resources in cloud data centers brings many opportunities, but also raises many challenges. This thesis built the foundation for the deployment and use of standalone disaggregated FPGAs at large scale in cloud data centers by making a proof of concept. The next paragraphs explain the potential future work to make the concept much stronger.

Network Protocols for SDF: The standalone disaggregated FPGA protoype built in this thesis used TCP/IP as the network protocol atop Ethernet. TCP is a complex protocol that uses a significant amount of reconfigurable logic resources in the FPGA. Instead of using TCP, light-weight network protocols, such as those based on UDP, can be implemented for reliable data transmission over IP. Light-weight protocols gain significant importance, particularly when moving towards 40/100 GbE, because implementing complex protocols while maintaining line-rate becomes increasingly difficult at higher frequencies.

End-to-End Lossless Networking: The traditional network stacks implemented on servers are lossy, and packets are dropped when buffers overflow. The L3 and L4 stacks of the FPGA have been designed to be lossless. By configuring the vendor-provided L2 layer (MAC IP core) as lossless, the entire FPGA network

stack can be made lossless. By attaching such lossless FPGAs to a lossless DC network (for example, Converged Enhanced Ethernet (CEE)) for inter-FPGA communication, an end-to-end lossless compute fabric can be built.

Resource Efficiency: Our results show that the cloud shell of standalone disaggregated FPGA consumes around 20% of the reconfigurable resources of a state-of-the-art FPGA. But once the functions of the cloud shell get matured, those parts can be hardened so that more reconfigurable resources are available for applications. When hardening the matured functionality, interfaces can be exposed to the infrastructure vendor so that parts of the harden logic (for example management functions) can still be programmable over the network.

End-to-End Predictable Compute Fabrics: The observations in this thesis show that far more deterministic results can be obtained from FPGAs even with distributed computing compared to CPUs. However, when the networks get congested, the degree of predictability may change. Therefore, making the network predictable by using emerging technologies such as SDN, end-to-end predictable compute fabrics can be built.

Application Partitioning for SDMFF: Mapping a distributed application to an SDMFF requires partitioning the application such that resources are used efficiently, which helps use only a minimal number of FPGAs for the application. Therefore, having an application-partitioning framework for SDMFFs is a must. For partitioning the application, the number of LUTs, FFs, the network BW, the amount of external memory and the memory BW can be considered.

Self-Organizing SDMFFs: This thesis presented multi-FPGA fabrics built using a centralized controller. The intelligence of each SDF is contained in this centralized controller, instead of having it in each FPGA. Another way to build multi-FPGA fabrics is to have the intelligence of each FPGA in a master FPGA in the fabric. In this way, multi-FPGA fabric can be built within the FPGA cluster, without each FPGA having to connect with the centralized controller.

Multiple Virtual FPGAs: FPGAs are becoming increasingly rich in terms of logic density, I/O functions, internal memory capacity, external memory BW, etc. Therefore, offering a high-end FPGA to a single user/application might not be efficient in terms of resource utilization. Hence, dividing the FPGA into multiple user partitions and offering them to multiple tenants is needed.

Cloud Enablement: IBM SuperVessel [99] and Fabric [188] are two non-commercial solutions that offer FPGAs in the cloud for research and education. Meanwhile, over the past year, commercial solutions started to emerge [47] [189]. Integrating standalone disaggregated FPGAs and their value additions, such as multi-FPGA

fabrics, into the cloud and offering them as a service is another area for potential future work.

Multi-Tenancy Support: When offering a single FPGA to multiple tenants, the FPGA must be securely partitioned. Partitioning must take into account both the physical space partitioning and the network partitioning. For network partitioning, state-of-the-art network virtualization methods, such as VXLAN [116] and GENEVE [118], must be implemented in the FPGA.

Security: Multi-tenancy requires secure isolation of FPGA resources. Moreover, widely used isolation techniques, such as VLANs and overlay virtual networks (OVN) must be used in order to co-exist with other infrastructure resources. Further, isolation even within the tenants is required for applications that require a higher level of security.

Application-Aware Networking with Multi-FPGA Fabrics: Understanding the application behavior helps use the network infrastructure efficiently for improved performance. As standalone disaggregated FPGAs improve the end-node network performance significantly compared to CPUs, the information on the applications running in multi-FPGA fabrics can be exploited for taking dynamic decisions in network traffic management in the DC more efficiently for overall improvement of the system performance [190] [191] [192].

Publications and Patents

Publications

1. R. Polig, J. Weerasinghe, C. Hagleitner "RESTful Web Services on Standalone Disaggregated FPGAs," in 9th International Conference on Cloud Computing Technology and Science (CloudCom 2017), Hongkong, Dec 2017, pp. 114-121.

2. F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, S. Paredes, "An FPGA Platform for Hyperscalers," in 25th International Symposium on High Performance Interconnects (HotI 2017), Santa Clara, California, Aug 2017, pp. 29-32.

3. J. Weerasinghe, R. Polig, F. Abel and C. Hagleitner, "Network-Attached FPGAs for Data Center Applications," in 15th IEEE International Conference on Field-Programmable Technology (FPT 2016), Xian China, Dec 2016, pp. 36-43.

4. J. Weerasinghe, F. Abel, C. Hagleitner and A. Herkersdorf, "Disaggregated FPGAs: Network Performance Comparison against Bare-Metal Servers, Virtual Machines and Linux Containers (*Best Student Paper*)," in 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2016), Luxembourg, Dec 2016, pp. 9-17.

5. J. Weerasinghe, F. Abel, C. Hagleitner and A. Herkersdorf, "Enabling FPGAs in Hyperscale Data Centers," in 2015 IEEE International Conference on Big Data and Cloud Computing (CBDCom 2015), Beijing China, Aug 2015, pp. 1078-1086.

6. J. Weerasinghe and F. Abel, "On the Cost of Tunnel Endpoint Processing in Overlay Virtual Networks," 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC 2014), London UK, Dec 2014, pp. 756–761.

Patents (Pending)

1. J. Weerasinghe, F. Abel, C. Hagleitner , "Communication Channel for Reconfigurable Computing Device, Filed: April, 2017
2. J. Weerasinghe, F. Abel, C. Hagleitner , "Network Attached Reconfigurable Computing Device, Filed: August, 2016

Bibliography

[1] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[2] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *2015 IEEE International Conference on Big Data and Cloud Computing (CBDCom)*, August 2015, pp. 1078–1086.

[3] R. Luijten and A. Doering, "The DOME embedded 64 bit microserver demonstrator," in *2013 International Conference on IC Design Technology (ICICDT)*, May 2013, pp. 203–206.

[4] G. Andrews, "What is openpower?" March 2015. [Online]. Available: https://www.ibm.com/developerworks/community/

[5] N. Zhang and R. Brodersen, "The cost of flexibility in systems on a chip design for signal processing applications," 2002. [Online]. Available: http://bwrc.eecs.berkeley.edu/Classes/EE225C/Papers/arch_design.doc

[6] J. Deaton, "Accelerating computing of the future." [Online]. Available: https://channels.theinnovationenterprise.com/articles/9781-accelerating-computing-of-the-future

[7] Maxeler, "Maxeler dataflow computing." [Online]. Available: www.maxeler.com

[8] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, March 2015.

[9] P. Sundararajan, "High performance computing using fpgas," sept 2010.

[10] Xilinx, "Zynq-7000 all programmable soc." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[11] N. Hemsoth, "Intel marrying fpga, beefy broadwell for open compute future," 2016. [Online]. Available: https://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/

[12] Convey Computer, "Hybrid-core: The "Big Data" computing architecture," xxx 20xx. [Online]. Available: www.conveycomputer.com

[13] W. Paper, "Fpga accelerated compute node." [Online]. Available: http://www.nallatech.com/wp-content/uploads/Nallatech-FPGA-Accelerated-Compute-Node.pdf

[14] H. T. Dang, P. Bressana, H. Wang, K. Lee, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "Network hardware-accelerated consensus," *CoRR*, vol. abs/1605.05619, 2016. [Online]. Available: http://arxiv.org/abs/1605.05619

[15] Maxeler, "Maxeler MPC-C series." [Online]. Available: https://www.maxeler.com/products/mpc-cseries/

[16] ——, "Maxeler MPC-X series." [Online]. Available: https://www.maxeler.com/products/mpc-xseries/

[17] G. Pfeiffer, S. Baumgart, J. Schröder, and M. Schimmler, *A Massively Parallel Architecture for Bioinformatics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 994–1003.

[18] A. Patel *et al.*, "A scalable FPGA-based multiprocessor," in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, April 2006, pp. 111–120.

[19] C. Conger *et al.*, "Narc: Network network–attached reconfigurable computing for attached reconfigurable computing for high high–performance, network performance, network–based applications based applications," Sept 2005.

[20] H. University, "Die photo analysis." [Online]. Available: http://vlsiarch.eecs.harvard.edu/accelerators/die-photo-analysis

[21] H. Giefers *et al.*, "Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid cpu/fpga system," in *2014 IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, June 2014, pp. 92–99.

[22] Cyclone, "Pcie2-2711 - PCIe Gen2 eight slot expansion system." [Online]. Available: http://cyclone.com/pdf/600_2711%20datasheet.pdf

[23] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes, "An fpga platform for hyperscalers," in *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug 2017, pp. 29–32.

[24] "Cloud deployment models." [Online]. Available: https://en.wikipedia.org/wiki/Cloud_computing

[25] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24.

[26] IBM, "Coherent accelerator processor interface user's manual," January 2015.

[27] D. Pellerin, "Announcing amazon ec2 f1 instances with custom fpgas hardware-accelerated computing on aws," December 2016. [Online]. Available: https://www.slideshare.net/AmazonWebServices/announcing-amazon-ec2-f1-instances-with-custom-fpgas

[28] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. New York, NY, USA: ACM, 2008, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477944

[29] A. Panella, *Design Methodologies for Dynamic Reconfigurable Multi-FPGA Systems*, 2008.

[30] S. Hauck, "Multi-fpga systems, phd thesis," 1995. [Online]. Available: https://usastore.alpha-data.com/store/

[31] E. A. Epstein *et al.*, "Making watson fast," *IBM Journal of Research and Development*, vol. 56, no. 3.4, pp. 15–1, 2012.

[32] , "Apache hadoop." [Online]. Available: http://hadoop.apache.org/

[33] M. Zaharia *et al.*, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[34] M. Isard *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007, pp. 59–72.

[35] D. Ferrucci and A. Lally, "UIMA: an architectural approach to unstructured information processing in the corporate research environment," *Natural Language Engineering*, vol. 10, no. 3-4, pp. 327–348, 2004.

[36] J. Hamilton, "Cloud computing is driving infrastructure innovation," May 2011. [Online]. Available: http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_WesternDigitalBoardMeeting.pdf

[37] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, March 2003.

[38] C. L. Belady, "In the data center, power and cooling costs more than the it equipment it supports," February 2007. [Online]. Available: https://www.electronics-cooling.com/category/volume-13/page/5/

[39] B. Grot *et al.*, "Optimizing data-center TCO with scale-out processors," *Micro, IEEE*, vol. 32, no. 5, pp. 52–63, Sept 2012.

[40] J. Dean *et al.*, "Large scale distributed deep networks," in *Neural Information Processing Systems, NIPS 2012*.

[41] M. Ebbers *et al.*, "Implementing IBM InfoSphere BigInsights on IBM System X," pp. 105–107. [Online]. Available: http://www.redbooks.ibm.com/

[42] Y. Guo *et al.*, "iShuffle: Improving Hadoop performance with shuffle-on-write," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, 2013, pp. 107–117.

[43] J. Weerasinghe and F. Abel, "On the cost of tunnel endpoint processing in overlay virtual networks," in *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, Dec 2014, pp. 756–761.

[44] J. Sacha, J. Napper, S. Mullender, and J. McKie, "Osprey: Operating system for predictable clouds," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, June 2012, pp. 1–6.

[45] K. Jang *et al.*, "Silo: Predictable message latency in the cloud," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 435–448.

[46] R. Kapoor *et al.*, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 9:1–9:14.

[47] Amazon, "Amazon f1 instance," 2017. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[48] D. Firestone, "Smartnic:accelerating azure's network with fpgas on ocs servers," Aug 2017. [Online]. Available: http://files.opencompute.org/oc/public.php?service=files&t=5803e581b55e90e51669410559b91169&download&path=/SmartNIC%20OCP%202016.pdf

[49] R. Bittner and E. Ruf, "Direct gpu/fpga communication via pci express," in *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ser. ICPPW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 135–139. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2012.20

[50] Y. Thoma, A. Dassatti, and D. Molla, "Fpga2: An open source framework for fpga-gpu pcie communication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013, pp. 1–6.

[51] PCI-SIG, "Single root i/o virtualization and sharing specification revision 1.0," Sept 2007. [Online]. Available: https://members.pcisig.com/wg/PCI-SIG/document/download/8272

[52] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Disaggregated fpgas: Network performance comparison against bare-metal servers, virtual machines and linux containers," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016, pp. 9–17.

[53] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached fpgas for data center applications," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 36–43.

[54] R. Polig, J. Weerasinghe, and C. Hagleitner, "Restful web services on standalone disaggregated fpgas," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 00, Dec. 2017, pp. 114–121. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CloudCom.2017.24

[55] S. Higginbotham, "Google takes unconventional route with homegrown machine learning chips." [Online]. Available: https://www.nextplatform.com/2016/05/19/google-takes-unconventional-route-homegrown-machine-learning-chips/

[56] J. O. nd others, "Sda: Software-defined accelerator for large-scale dnn systems," in *Hot Chips(Vol. 26)*, Aug 2014.

[57] M. Blott *et al.*, "Achieving 10Gbps line-rate key-value stores with FPGAs," in *the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.

[58] J. Lockwood *et al.*, "A low-latency library in FPGA hardware for high-frequency trading (HFT)," in *2012 IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI)*, Aug 2012, pp. 9–16.

[59] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA memcached appliance," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 245–254. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435306

[60] Nvidia, "Nvidia tesla p100 gpu accelerator," Oct 2016. [Online]. Available: http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-datasheet.pdf

[61] P. H. Jin, Q. Yuan, F. N. Iandola, and K. Keutzer, "How to scale distributed deep learning?" *CoRR*, vol. abs/1611.04581, 2016. [Online]. Available: http://arxiv.org/abs/1611.04581

[62] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. New York, NY, USA: ACM, 2009, pp. 873–880. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553486

[63] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 27–40.

[64] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the GPU – past, present and future," *Medical Image Analysis*, vol. 17, no. 8, pp. 1073 – 1094, 2013.

[65] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, Nov 2004, pp. 47–47.

[66] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–8.

[67] Intel, "Intel xeon processor e7-8800/4800 v4 product families," 2017. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e7-8800-4800-v4-product-families-brief.pdf

[68] M. J. Flynn, "High-performance computing using fpgas," in *High-Performance Computing Using FPGAs*. Springer New York, 2013.

[69] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "Asic clouds: Specializing the datacenter," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 178–190. [Online]. Available: https://doi.org/10.1109/ISCA.2016.25

[70] J. Gonzalez and R. C. Núñez, "Lapackrc: Fast linear algebra kernels/solvers for fpga accelerators," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012042, 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012042

[71] D. Yang, G. D. Peterson, and H. Li, "Compressed sensing and cholesky decomposition on fpgas and gpus," *Parallel Comput.*, vol. 38, no. 8, pp. 421–437, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2012.03.001

[72] Altera, "Radar processing: Fpgas or gpus?" May 2013. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01197-radar-fpga-or-gpu.pdf

[73] C. Brugger, L. Dal'Aqua, J. A. Varela, C. D. Schryver, M. Sadri, N. Wehn, M. Klein, and M. Siegrist, "A quantitative cross-architecture study of morphological image processing on cpus, gpus, and fpgas," in *2015 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*, April 2015, pp. 201–206.

[74] V. Venugopalan, "Evaluating latency and throughput bound acceleration of fpgas and gpus for adaptive optics algorithms," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[75] A. Rafique, G. A. Constantinides, and N. Kapre, "Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 24–34, Jan 2015.

[76] D. Chen and D. Singh, "Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2013, pp. 297–304.

[77] Berten, "Gpu vs fpga performance comparison," 2016. [Online]. Available: http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf

[78] B. Falsafi, B. Dally, D. Singh, D. Chiou, J. J. Yi, and R. Sendag, "Fpgas versus gpus in data centers," *IEEE Micro*, vol. 37, no. 1, pp. 60–72, Jan 2017.

[79] S. Kim *et al.*, "GPUnet: Networking abstractions for gpu programs," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 201–216.

[80] A. Younge *et al.*, "Evaluating GPU passthrough in xen for high performance cloud computing," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 852–859.

[81] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203–215, Feb 2007.

[82] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[83] Xilinx, "Vivado design suite user guide high-level synthesis," June 2016. [Online]. Available: https://www.xilinx.com/support/.../xilinx2016_2/ug902-vivado-high-level-synthesis.pdf

[84] Altera, "Implementing fpga design with the opencl standard," November 2013. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf

[85] "Intel socs: When architecture matters." [Online]. Available: https://www.altera.com/products/soc/overview.html

[86] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, "A reconfigurable computing system based on a cache-coherent fabric," in *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*, ser. RECONFIG '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 80–85. [Online]. Available: http://dx.doi.org/10.1109/ReConFig.2011.4

[87] P. Gupta, "Xeon+fpga platform for the data center," sept 2015. [Online]. Available: https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[88] IBM, "IBM coherent accelerator processor interface (CAPI) for POWER8 systems," Sept 2014. [Online]. Available: www.ibm.com

[89] C. Steffen and G. Genest, "Nallatech in-socket fpga front-side bus accelerator," *Computing in Science Engineering*, vol. 12, no. 2, pp. 78–83, March 2010.

[90] "Hypertransport consortium." [Online]. Available: http://www.hypertransport.org

[91] F. Chen, H. Cheng, X. Yang, and R. Liu, "Design and implementation of an effective hypertransport core in fpga," in *2008 IEEE International Conference on Cluster Computing*, Sept 2008, pp. 437–443.

[92] H. Litz, H. Froening, and U. Bruening, *A HyperTransport 3 Physical Layer Interface for FPGAs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 4–14. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00641-8_4

[93] O. Mencer *et al.*, "Cube: A 512-FPGA cluster," in *Programmable Logic, 2009. SPL. 5th Southern Conference on*, April 2009, pp. 51–57.

[94] T. Güneysu *et al.*, "Cryptanalysis with copacobana," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 57, no. 11, pp. 1498–1513, 2008.

[95] C. Chang *et al.*, "BEE2: a high-end reconfigurable computing system," *Design Test of Computers, IEEE*, vol. 22, no. 2, pp. 114–125, March 2005.

[96] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, "RAMP Blue: A message-passing manycore system in FPGAs," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 54–61.

[97] F. Belletti, M. Cotallo, A. Cruz, L. A. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, A. Muñoz-Sudupe, D. Navarro, G. Parisi, S. Perez-Gaviro, M. Rossi, J. J. Ruiz-Lorenzo, S. F. Schifano, D. Sciretti, A. Tarancon, R. Tripiccione, J. L. Velasco, D. Yllanes, and G. Zanier, "Janus: An fpga-based system for high-performance scientific computing," *Computing in Science Engineering*, vol. 11, no. 1, pp. 48–58, Jan 2009.

[98] R. Baxter *et al.*, "Maxwell - a 64 FPGA supercomputer," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, Aug 2007, pp. 287–294.

[99] IBM, "Supervessel cloud for power/openpower." [Online]. Available: www.ptopenlab.com

[100] F. Chen *et al.*, "Enabling FPGAs in the cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:10.

[101] , "Opencapi." [Online]. Available: http://www.opencapi.org/

[102] T. P. Morgan, "OpenCAPI," Oct 2016. [Online]. Available: http://www.nextplatform.com/2016/10/17/opening-server-bus-coherent-acceleration/

[103] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sept 2014.

[104] J. W. Lockwood *et al.*, "Netfpga–an open platform for gigabit-rate network switching and routing," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, June 2007, pp. 160–161.

[105] M. Blott *et al.*, ""fpga research design platform fuels network advances," in *Xilinx Xcell Journal*, Sept 2010.

[106] Altera, "SerialLite iii streaming megacore function user guide," Dec 2014. [Online]. Available: www.altera.com

[107] H. Giefers *et al.*, "Accelerating finite difference time domain simulations with reconfigurable dataflow computers," *SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 65–70, Jun. 2014.

[108] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with FPGAs and GPUs," pp. 115–124, 2010. [Online]. Available: http://doi.acm.org/10.1145/1723112.1723134

[109] G. Gibb and N. McKeown, "OpenPipes: Making distributed hardware systems easier," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 381–384.

[110] nanostreams, "A hardware and software stack for real-time analytics on fast data streams," sept 2013. [Online]. Available: http://www.nanostreams.eu/

[111] Amazon, "Amazon elastic compute cloud," xxx 20xx. [Online]. Available: www.amazon.com

[112] B. Wile, "Coherent accelerator processor interface (capi) for power8 systems decision guide and development process," October 2014.

[113] A. Data, "Alpha data fpga card price." [Online]. Available: https://usastore.alpha-data.com/store/

[114] Nallatech, "Nallatech fpga card price." [Online]. Available: http://www.nallatech.com/store/

[115] Mle, "TCP/UDP/IP Network Protocol Accelerator." [Online]. Available: http://www.missinglinkelectronics.com

[116] M. Mahalingam *et al.*, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks," August 2014. [Online]. Available: https://tools.ietf.org/html/rfc7348

[117] I. T. Association, "RoCE – RDMA over converged ethernet," Apr 2010. [Online]. Available: http://www.infinibandta.org/

[118] I. G. J. Gross and J. Sridhar, "Geneve: Generic network virtualization encapsulation," March 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-nvo3-geneve-04

[119] D. Black *et al.*, "An architecture for data center network virtualization overlays (nvo3)," September 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-nvo3-arch-08

[120] S. Han *et al.*, "Network support for resource disaggregation in next-generation datacenters," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, ser. HotNets-XII. New York, NY, USA: ACM, 2013, pp. 10:1–10:7.

[121] Huawei, "High throughput computing data center architecture," Jun. 2014. [Online]. Available: www.huawei.com

[122] J. Waxman, "Architecting cloud infrastructure for the future," Jun. 2014. [Online]. Available: www.intel.com

[123] S. Byma *et al.*, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '14, 2014, pp. 109–116.

[124] EMA, "Data center management: The key ingredient for reducing server power while increasing data center capacity," June 2010. [Online]. Available: https://www.cisco.com

[125] M. Ferdman *et al.*, "A case for specialized processors for scale-out workloads," *Micro, IEEE*, vol. 34, no. 3, pp. 31–42, May 2014.

[126] K. Sudan *et al.*, "A novel system architecture for web scale applications using lightweight cpus and virtualized I/O," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 167–178. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2013.6522316

[127] W. Felter *et al.*, "An updated performance comparison of virtual machines and linux containers," July 2014. [Online]. Available: domino.research.ibm.com

[128] M. I. . Stratergy, "Intel's disaggregated server rack," Aug 2013. [Online]. Available: www.moorinsightsstrategy.com

[129] K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 267–278, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1555815.1555789

[130] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," 2009.

[131] Freescale, "QorlQ T4240, T4160 and T4080 advanced multicore processors." [Online]. Available: http://www.freescale.com

[132] V. Salapura *et al.*, "Accelerating business analytics applications," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, Feb 2012, pp. 1–10.

[133] A. Ali *et al.*, "On the use of microservers in supporting hadoop applications," in *Proceedings of the IEEE International Conference on Cluster Computing (Cluster), Madrid, Spain*, Sept 2014.

[134] OpenCompute, "Open compute project." [Online]. Available: www.opencompute.org

[135] J. Weiss *et al.*, "Optical interconnects for disaggregated resources in future datacenters," in *Optical Communication (ECOC), 2014 European Conference on*, Sept 2014, pp. 1–3.

[136] J. Gao, "Machine learning applications for data center optimisation," 2014.

[137] Hewlett-Packard, "HP Moonshot: An accelerator for hyperscale workloads," 2013. [Online]. Available: www.hp.com

[138] SeaMicro, "Seamicro SM15000 fabric compute systems." [Online]. Available: http://www.seamicro.com/

[139] Dell, "Dell poweredge C5220 microserver." [Online]. Available: http://www.dell.com/

[140] Hewlett Packard, "The machine: A new kind of computer." [Online]. Available: http://www.hpl.hp.com/

[141] R. Luijten *et al.*, "Dual function heat-spreading and performance of the IBM/ASTRON DOME 64-bit µserver demonstrator," in *2014 IEEE International Conference on IC Design Technology (ICICDT),*, May 2014, pp. 1–4.

[142] M. K. Tiwari, S. Zimmermann, C. S. Sharma, F. Alfieri, A. Renfer, T. Brunschwiler, I. Meijer, B. Michel, and D. Poulikakos, "Waste heat recovery in supercomputers and 3d integrated liquid cooled electronics," in *13th Inter-Society Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, May 2012, pp. 545–551.

[143] OpenStack, "OpenStack cloud management software," xxx 20xx. [Online]. Available: www.openstack.org

[144] M. Armbrust *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing," 2009.

[145] A. Stanik *et al.*, "Hardware as a service (HaaS): The completion of the cloud stack," in *Computing Technology and Information Management (ICCM), 2012 8th International Conference on*, vol. 2, April 2012, pp. 830–835.

[146] S. Crago *et al.*, "Heterogeneous cloud computing," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2011, pp. 378–385.

[147] R. Polig *et al.*, "Giving text analytics a boost," *IEEE Micro*, vol. 34, no. 4, pp. 6–14, July 2014.

[148] Xilinx, "Ultrascale integrated 100g ethernet subsystem." [Online]. Available: https://www.xilinx.com/products/intellectual-property/cmac.html#overview

[149] ——, "40g/100g ethernet core." [Online]. Available: https://www.xilinx.com/products/intellectual-property/40_100g_ethernet.html

[150] A. Dollas *et al.*, "An open TCP/IP core for reconfigurable logic." in *FCCM*. IEEE Computer Society, 2005, pp. 297–298.

[151] Xilinx, "TCP/IP Stack." [Online]. Available: https://github.com/Xilinx/HLx_Examples/tree/master/Acceleration/tcp_ip

[152] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2015, pp. 36–43.

[153] S. Meyer, "Verilog plus c language modeling with pli 2.0: The next generation simulation language," in *Proceedings International Verilog HDL Conference and VHDL International Users Forum*, Mar 1998, pp. 98–105.

[154] S. Sutherland, "Integrating systemc models with verilog and system verilog models using the system verilog direct programming interface," 2004. [Online]. Available: https://github.com/ibm-capi/pslse

[155] IBM, "Power service layer simulation engine," 2008. [Online]. Available: https://github.com/ibm-capi/pslse

[156] A. Pool, "Using modelsim foreign language interface for c – vhdl co-simulation and for simulator control on linux x86 platform," 2008. [Online]. Available: https://github.com/andrepool/fli

[157] ADI Engineering, "Seacliff trail: Intel FM6000." [Online]. Available: www.adiengineering.com

[158] HP and Mellanox, "HP Mellanox low latency benchmark report 2012," July 2012. [Online]. Available: www.mellanox.com

[159] A. Trivedi *et al.*, "Rstore: A direct-access dram-based data store," in *2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS)*, June 2015, pp. 674–685.

[160] Y. Gu and R. L. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Comput. Netw.*, vol. 51, no. 7, pp. 1777–1799, May 2007.

[161] R. Hamilton *et al.*, "QUIC: A UDP-based secure and reliable transport for HTTP/2." [Online]. Available: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-01

[162] O. Knodel, A. Georgi, P. Lehmann, W. E. Nagel, and R. G. Spallek, "Integration of a highly scalable, multi-fpga-based hardware accelerator in common cluster infrastructures," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 893–900.

[163] M. A. S. Khalid, "Routing architecture and layout synthesis for multi-fpga systems," *Ph.D. dissertation, Dept. of ECE, Univ. Toronto*, 1999.

[164] J. Babb, "Virtual wires: Overcoming pin limitations in fpga-based logic emulation," Cambridge, MA, USA, Tech. Rep., 1993.

[165] Synopsys, "Zebu server-3: Industry's fastest emulation system," 2016. [Online]. Available: https://www.synopsys.com/verification/emulation/zebu-server.html

[166] P. Electronic, "Chipit platinum edition: Asic emulation and rapid prototyping system handbook," 2008.

[167] M. Nüssle, B. Geib, H. Fröning, and U. Brüning, "An fpga-based custom high performance interconnection network," in *2009 International Conference on Reconfigurable Computing and FPGAs*, Dec 2009, pp. 113–118.

[168] J. Wang, S. Yang, B. Deng, X. Wei, and H. Yu, "Multi-fpga implementation of feedforward network and its performance analysis," in *2015 34th Chinese Control Conference (CCC)*, July 2015, pp. 3457–3461.

[169] D. L. Ly and P. Chow, "A multi-fpga architecture for stochastic restricted boltzmann machines," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 168–173.

[170] J. J. Martínez-Alvarez, J. Toledo-Moreo, J. Garrigós-Guerrero, and J. M. Ferrandez-Vicente, "A multi-fpga distributed embedded system for the emulation of multi-layer cnns in real time video applications," in *2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010)*, Feb 2010, pp. 1–5.

[171] S.-W. Jun *et al.*, "BlueDBM: An Appliance for Big Data Analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 1–13.

[172] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485945

[173] L. Wienbrandt, *The FPGA-Based High-Performance Computer RIVYERA for Applications in Bioinformatics*. Springer International Publishing, 2014, pp. 383–392.

[174] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[175] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[176] "OpenAPI Initiative," https://www.openapis.org/, accessed: 2017-06-29.

[177] R. Polig, K. Atasu, H. Giefers, C. Hagleitner, L. Chiticariu, F. Reiss, H. Zhu, and P. Hofstee, "A hardware compilation framework for text analytics queries," *Journal of Parallel and Distributed Computing*, 2017.

[178] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[179] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[180] "The uWSGI project," https://uwsgi-docs.readthedocs.io/, accessed: 2017-06-14.

[181] H. Y. McCreary, M. A. Broyles, M. Floyd, A. Geissler, S. P. Hartman, F. Rawson, T. Rosedahl, J. Rubio, and M. Ware, "EnergyScale for IBM POWER6 microprocessor-based systems," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 775–786, Nov 2007.

[182] L. Chiticariu, Y. Li, and F. R. Reiss, "Rule-based information extraction is dead! long live rule-based information extraction systems!" in *EMNLP*, 2013, pp. 827–832.

[183] R. Polig, K. Atasu, and C. Hagleitner, "Token-based dictionary pattern matching for text analytics," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–6.

[184] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss, "Hardware-accelerated regular expression matching for high-throughput text analytics," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–7.

[185] R. Polig, "Text analytics on reconfigurable platforms," Ph.D. dissertation, München, Technische Universität München, Diss., 2015.

[186] E. A. Epstein, M. I. Schor, B. Iyer, A. Lally, E. W. Brown, and J. Cwiklik, "Making watson fast," *IBM Journal of Research and Development*, vol. 56, no. 3.4, pp. 15–1, 2012.

[187] R. Polig *et al.*, "Compiling text analytics queries to FPGAs," in *24th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6.

[188] University of Texas, "Fabric (fpga research infrastructure cloud)." [Online]. Available: https://www.tacc.utexas.edu/systems/fabric

[189] Accelize, "Enabling fpga-acceleration-as-a-service." [Online]. Available: https://www.accelize.com/

[190] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, "Sdn-based application-aware networking on the example of youtube video streaming," in *Proceedings of the 2013 Second European Workshop on Software Defined Networks*, ser. EWSDN '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 87–92. [Online]. Available: http://dx.doi.org/10.1109/EWSDN.2013.21

[191] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 103–108. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342462

[192] W. Paper, "Application-aware networking at a glance," 2013. [Online]. Available: http://mrv.com/wp-content/uploads/2016/06/MRV-Application-Aware-Networking-WP.pdf

Curriculum Vitae

Education

2014–2018	PhD in Electrical and Computer Engineering, Technical University of Munich, Germany.
2008–2010	MSc in Artificial Intelligence, Kyushu Institute of Technology, Japan.
2003–2007	BSc in Electrical and Electronics Engineering, University of Peradeniya, Sri Lanka.

Work Experience

2014–2018	PreDoctoral Researcher, IBM Research, Switzerland.
2013–2014	Visiting Scientist, IBM Research, Switzerland.
2010–2013	Software Engineer, Fujitsu Ltd, Japan.