A Survey on FPGA Virtualization

Anuj Vaishnav, Khoa Dang Pham and Dirk Koch
School of Computer Science, The University of Manchester, Manchester, UK
Email: {anuj.vaishnav, khoa.pham, dirk.koch}@manchester.ac.uk

***Abstract*—FPGA accelerators are being applied in various types of systems ranging from embedded systems to cloud computing for their high performance and energy efficiency. Given the scale of deployment, there is a need for efficient application development, resource management, and scalable systems, which make FPGA virtualization extremely important. Consequently, FPGA virtualization methods and hardware infrastructures have frequently been proposed in both academia and industry for addressing multi-tenancy execution, multi-FPGA acceleration, flexibility, resource management and security. In this survey, we identify and classify the various techniques and approaches into three main categories: 1) Resource level, 2) Node level, and 3) Multi-node level. In addition, we identify current trends and developments and highlight important future directions for FPGA virtualization which require further work.**

I. Introduction

FPGAs are being deployed in a wide variety of applications, which range from embedded applications to large-scale datacentres. This broad scope of requirements presents a challenge in terms of the efficient development of applications, management of resources, and scaling of systems, which calls for effective virtualization of FPGAs.

The term "Virtualization" has acquired various different meanings over time, with a common trait being the introduction of an abstract layer to simplify the interface and hide the complexity of the system. In particular, for FPGA virtualization, the definitions and techniques have changed over time due to the change in application requirements when compared to an earlier survey on FPGA virtualization in 2004 by Plessl and Platzner [1]. In that work, FPGA virtualization was classified into three categories: temporal partitioning, virtualized execution, and virtual machine.

Temporal partitioning is used to fit large designs on relatively smaller FPGAs by reconfiguring an FPGA to host a partition of the design at a time. This was the first FPGA virtualization approach when device capacity was often not sufficient to host the intended netlist sizes. Temporal partitioning is still used for certain applications but the majority of applications which require more FPGA resources than a single chip can offer also tend to require executing the application in parallel i.e. in the spatial domain with multiple FPGAs rather in than in time [2], [3]. Thus, temporal partitioning is nowadays usually used at task-level for large-scale datacentres where a task may span multiple FPGAs but may be swapped with another task in time.

Virtualized execution in the survey [1] was used to define the approach of splitting up applications into multiple communicating tasks (e.g., following a Petri-Net model) and using a run-time system to manage them. The aim of this was to support device independence within a device family. We will see in Section IV, how such run-time systems are deployed to support a wide range of application in recent years while decoupling the application development from the static logic requirements. It is now used not only for device independence within family but also for higher design productivity, isolation and resource management.

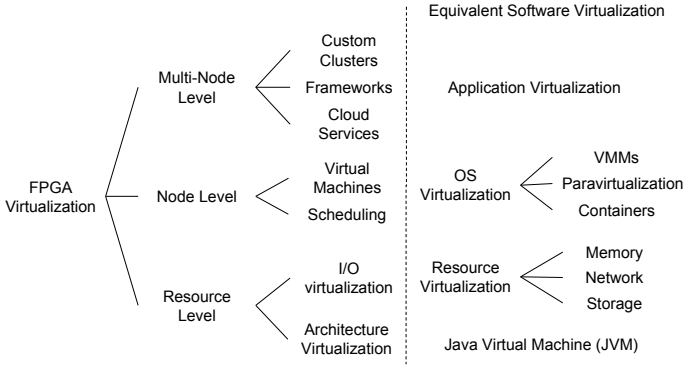

Fig. 1: Classification of FPGA virtualization techniques and their software equivalent.

Finally, Plessl and Platzner defined *virtual machines* to be systems which provide complete device independence by using an abstract architecture to describe applications. This architecture could be translated later into native architecture by a remapping tool or an interpreter. This approach, in particular, now falls under Overlays [4] (also called Intermediate architecture or fabric [5]) where the abstract architecture can be defined in many ways as discussed further in Section III-A. The term "virtual machine" is these days used for the static architecture which provides support for accelerators and is often referred to as a Shell [2] or Hypervisor for virtual FPGAs (vFPGAs) [6].

Nowadays, FPGA virtualization is starting to coincide with techniques for software virtualization at a conceptual level, with growing support for heterogeneous systems and concepts such as Acceleration as a Service (AaaS) [7]. The objectives of FPGA virtualization are similar to the core objectives that resulted in the development of virtualization used in traditional CPU/software systems. The main objectives are:

- **Multi-tenancy:** Ability to serve multiple different users using the same FPGA fabric.
- **Resource Management:** Providing an abstraction/driver layer to the FPGA fabric and means of scheduling tasks to the FPGA as well as monitoring its resource usage.
- **Flexibility:** Ability to support a wide range of acceleration workload i.e. from custom accelerators to framework specific accelerators designed in a High-Level Language (HLL) or a Domain Specific Language (DSL).
- **Isolation:** Providing the illusion of being a sole user of the FPGA resources for better security, fewer dependencies and correctness of the program execution.
- **Scalability:** The system/application can scale to multiple different FPGAs or can support multiple different users at relatively low overhead.
- **Performance:** The impact of virtualization should be minimal on performance achievable and FPGA resources

usable by the user application.

- **Security:** Ensuring information of other tenants is not leaked and for safekeeping the infrastructure from malicious users.
- **Resilience:** Ability to keep the system/service running despite failures.
- **Programmer's productivity:** Improving the time to market and reducing the complexity of deploying a design to an FPGA from its software description.

Despite sharing these objectives, virtualization techniques from the software domain cannot always be directly applied to FPGAs in a straightforward manner. This is mostly due to one fundamental difference between CPU/GPUs and FPGAs i.e. applications are hardware circuits rather than a set of commands in assembly. This leads to various differences which need to be considered when devising a solution for virtualization: a very high context switch penalty, space-time sharing rather than just time, different development tools, high development time and high heterogeneity in the system as each accelerator represents a distinct hardware module. Thus, many different FPGA virtualization approaches have been proposed as summarized in Figure 1.

In this paper we survey the literature on FPGA virtualization and provide the following contributions:

- Survey of the techniques and architectures proposed for virtualizing FPGAs (Sections III–V).
- Classification (Section II) and qualitative analysis of the proposed virtualization models (Sections III–V).
- Highlight current trends and developments and their important future directions for FPGA Virtualization (Section VI).

II. Classification of FPGA Virtualization

Existing work on FPGA virtualization can be classified in many ways for example, based on the virtualization techniques, different use cases or execution models. However these classification criteria may change with time, for example with the advent of new applications or changes in requirements over time. Instead, we propose a classification scheme based on the abstraction levels of the standard computational systems that apply virtualization, such that the same classification can capture future changes in the field and categorize the work at following levels (as shown in Figure 1): Multi-node level, Node level, and Resource level:

- *Resource level*: A resource on an FPGA can be of two types: reconfigurable or non-reconfigurable. Hence, for this level, we consider architecture virtualization and I/O virtualization. Examples at this level include Overlays [8], [9] for architecture abstraction and transparent I/O sharing in a multi-tenant system [6], [10].
- *Node level*: A 'node' is defined as a single FPGA. Thus, infrastructure and resource management techniques are considered for this level. Examples include VMM support [11], [12], run-time systems [13] and Shells (also called FPGA OS and Hypervisor-vFPGA approach) which are used to serve multiple concurrent user accelerators [2], [11], [14]–[16].
- *Mutli-node level*: 'Multi-node' is defined as a cluster of two or more FPGAs. Hence for this level, we consider techniques and architectures used to connect multiple FPGAs for accelerating a job. Examples at this level include Leap [17], MapReduce [18]–[20] and Catapult [2].

III. Virtualization at Resource Level

In contrast to standard CPU/Software virtualization, FPGAs have to virtualize two distinct types of resources: reconfigurable and non-reconfigurable. These resources operate in a fundamentally different manner from each other. The reconfigurable resources are based on the FPGA architecture and is the fabric onto which the accelerators are mapped. This mapping process varies from architecture to architecture and, thus, the accelerators require re-synthesis when used across different systems, which can take anywhere from minutes to days depending on the complexity of the design. Thus, the virtualization at this level for reconfigurable resources aims to provide portability of accelerators and rapid compilation by mapping each accelerator to an abstract architecture and then using a remapping tool or interpreter.

Whereas, the non-reconfigurable resources mostly represent the I/O resources which are also found in the CPU/Software domain. Hence, these resources require similar abstraction and security measures to software but with hardware support on FPGAs. This can include hard IP such as embedded CPUs or memory controllers as well as soft-logic that is considered not to be changed at run-time. We discuss the virtualization techniques employed for both types of resources in the following subsections, in more detail.

A. Overlays

Overlays provide another level of programmability that is implemented on top of the low-level FPGA resources as shown in Figure 2. This is commonly done to improve productivity, allow run-time compilation, or to support portability of functionality across different systems (and even different FPGA architectures).

Overlays allow the compilation process to be split into two parts which considerably decreases the compilation time required for generating an accelerator when the CAD tool part can be omitted. Note, Java Virtual Machine's byte-code relates to native machine code as the overlay's application binary relates to the configuration bitstream of the overlay that is implemented on the FPGA fabric. Similarly, like byte-code translation to native machine code for achieving better performance, overlay application may be directly translated into configurations of the underlying FPGA architecture [21], [22].

Overlay architectures can range from multi-core systems to custom processing elements (PE) to fine-grained look-up table (LUT) types, depending on the application or productivity requirements.

Jain provided a comprehensive overview on coarse-grained FPGA overlays in his PhD thesis [23], which are programmable at the data-word/operator level. Concisely, coarse-grain FPGA overlays can be soft-core processors, either from academia [24] or from industrial vendors [25], [26], vector processors [27]–[31], and connected arrays of processing elements (PEs) [9], [32]–[35] in which programmable PEs and interconnects are provided. The motivation behind these soft-core processor approaches tends to be the provisioning of a more familiar programming environment for software developers, in contrast to the fine-tuned hardware accelerator implementations which target performance. In particular, vector processors based on multi-ALU parallelism have been shown to achieve a significant speed-up compared to soft-core

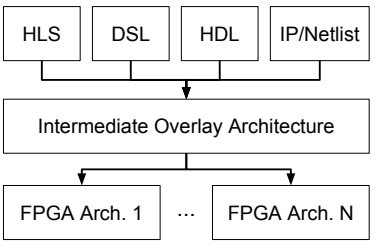

Fig. 2: Overlay design flow.

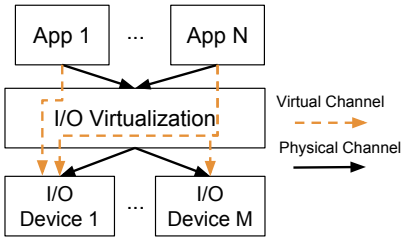

Fig. 3: I/O virtualization architecture, where an application can access multiple different I/O devices as if they were one or share the I/O devices transparently.

counterparts. Examples include DySER [35], Venice [28] and Vegas [29].

In connected arrays of PEs (also known as coarse-grained reconfigurable arrays CGRAs), a PE executes an arithmetic operation and data is transferred over an interconnect network amongst the PEs. In CGRAs, PEs and interconnect logic can be programmed cycle by cycle as in time-multiplexed overlays [4], [36]–[38], or kept static during PE execution as in spatially-configured overlays [32], [33]. Most common interconnect topologies for CGRAs include nearest-neighbour style [9], [32], and island-style [33]–[35]. Overall, CGRAs, use either static, dynamically changeable point-to-point connection [39], or network on chip (NoC) communication [40]–[44]. These network implementation methods provide a trade-off between implementation cost and flexibility (e.g., for providing more or less complex communication patterns). However, for virtualization, recent lightweight soft packet switched Network-on-Chip (NoC) (e.g., Hoplite [43]) and especially hard NoC [44] seem promising solutions to provide resource-efficient and high-speed interconnects for programmable overlays.

The Firm-core project by Lysecky et al. [45] is an early example of a fine-grained overlay where all programming was carried out by user logic implemented on top of the FPGA. ZUMA [8] reduced implementation cost by mapping overlay LUTs and overlay multiplexers into LUT-configuration of the hosting FPGA fabric. Koch et al. [21] improved the implementation cost further by mapping the overlay routing directly onto the underlying FPGA routing fabric.

Overall, the extra level of programmability through an overlay comes at a substantial cost that needs to be justified. An example of this is the DRAGEN chip for DNA processing [46] where the overlay abstraction allows domain experts (who are not FPGA experts) to benefit from FPGAs. Another example where overlays can be better than traditional HLS or RTL accelerators are the situations that require rapidly changing functionality (at a speed which cannot be met using partial reconfiguration). For instance, VectorBox commercializes vector processor overlays where the same compute substrate is used for various concurrently running tasks under real-time constraints [27].

B. I/O Virtualization

I/O virtualization aims at managing I/O resources such that multiple applications can share the same resource or access multiple different resources with the same interface. This is shown in Figure 3, where a virtual channel is established between the device and the user which does not necessarily correspond to the physical channels available for the resources. The virtualization layer in the middle can be used to enforce security measures (e.g., in memory virtualization [6], [15]), hide complexity of the I/O interface [6], monitor resource usage and enforce QoS (e.g., in cloud systems) [11], as well as optimizing access time (e.g., providing buffers for memory load and stores).

Fundamentally, the virtualization support for I/O is the same as in CPU/software systems, with the main difference being the implementation technology. For FPGAs, the control logic can be either implemented in a software domain (e.g., by using a soft-core [15] or a host CPU [16], [47]) or a special hardware module [6], [10], [14] using some reconfigurable resources. The software approach tends to be used for high flexibility or to save more reconfigurable resources for application logic. Whereas, the hardware approach tends to be used for high performance I/O access and management.

Moreover, the I/O virtualization layer can also be used to assist a CPU for higher performance I/O access rather than just serving the accelerators on the chip. Abbani et al. [48] has shown how FPGAs can be used to accelerate storage up to 6× performance improvement for data-intensive applications running on a distributed reconfigurable active solid-state drive (SSD) platform. While Chalamalasetti et al. [49] and Lavasani et al. [50] showed an FPGA accelerator based solutions for Memcached. Further, Microsoft uses FPGAs to reduce network traffic to CPU by delegating the majority of Network-Interface-Card (NIC) requests directly to FPGAs [2]. These are just a few examples which have been applied to accelerate I/O by offloading compute-intensive work to an FPGA as middleware.

IV. Virtualization at Node Level

Virtualization support at the node level represents the infrastructure (in both hardware and software) required to manage the resources related to a single FPGA. It can be split into three different categories: Virtual Machines Monitors (VMMs), Shells, and Scheduling, as described in the following subsections.

A. Virtual Machine Monitors (VMMs)

VMMs are a standard way of performing virtualization in the CPU world and tackle various challenges which apply to FPGA virtualization. Thus, it is only natural to extend them to support FPGAs. One such attempt was by Wang et al. [12], where the FPGA accelerator was integrated at the lower device driver level in the Xen VMM. Their experiments show that VMMs can be implemented with close-to-zero overhead compared to accessing FPGA accelerator without a VMM layer and, at the same time, share the accelerator among multiple operating systems. This approach is a good fit for applications which require fixed accelerator support and tight coupling of VMMs on host CPU with an FPGA

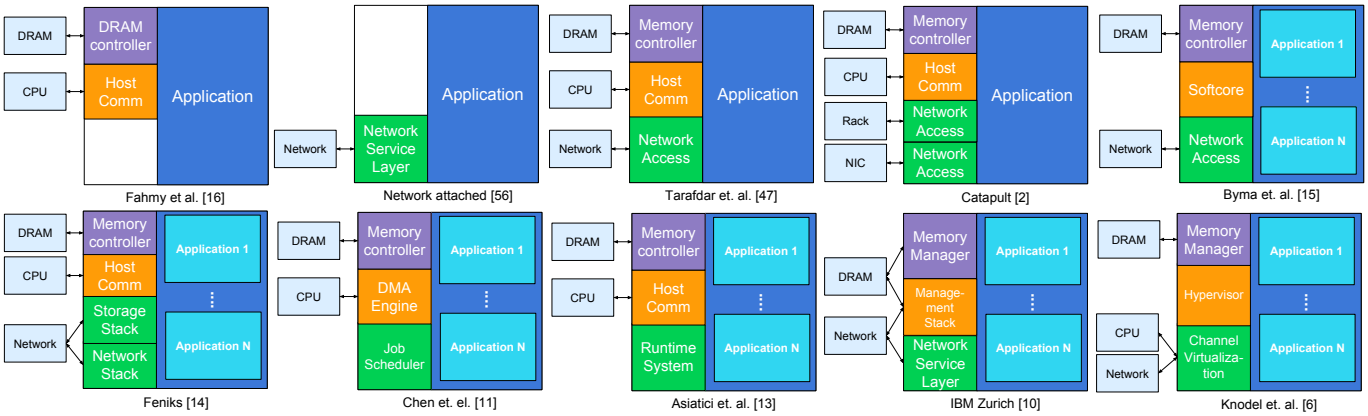

Fig. 4: Examples of proposed FPGA infrastructure and shells for hosting one for more reconfigurable applications with virtualization.

accelerator. The micro-kernel approach by Xia et al. [51] takes this one step further and by providing multiple partial regions and resource sharing mechanisms for VMMs, while Jain et al. [52] use a micro-kernel with overlay virtualization to provide accelerator portability. Another similar effort is presented by Chen et al. [11] for integrating Xilinx FPGAs into Linux-KVM with OpenStack support. Their approach suggests decoupling the static logic (OS functionality) from dynamic logic (applications) which is beneficial for various reasons (as detailed in Section IV-B).

Overall, all these prototypes provide isolation between multiple processes or virtual machines, with some support for resource allocation, and they aim at fulfilling the following virtualization objectives: Multi-tenancy, Resource Management, Isolation, and Security and Resilience. The dependency between virtual machines and FPGAs can be limiting to certain applications but this can be mitigated by co-scheduling a set of applications which require the same accelerator functionality on the same FPGA (via an accelerator library); this is discussed further in Section IV-C.

Moreover, FPGA Virtualization with VMMs integration tends to adopt the conventional model of treating CPU as a first-class citizen and FPGA as a special peripheral. This, makes the incorporation of FPGAs in software domains a little easier for the software programmers since accessing an FPGA accelerator tends to become a simple library call with a similar interface as a GPU. Hence, applications can be scaled to use multiple FPGAs by employing standard software frameworks which are used for GPUs, with minor modifications.

B. Shells

Shell is a recent term used to describe the static part of the FPGA system/bitstream and is often referred to as *FPGA OS* or *Hypervisor for vFPGA*s. It essentially provides the common infrastructure required for different applications i.e. all of the I/O virtualization support, resource management and the drivers required to program the applications. A shell without VMM support can be considered as the hardware equivalent of the Container virtualization [53] where the virtualization of application execution is done at the process level rather than at OS level. It achieves this by providing isolation, resource monitoring, and management along with I/O and driver dependency resolution.

There are many ways in which this can be achieved. Figure 4 shows some of the infrastructures proposed in recent works which target virtualization in particular. While Table I lists the various works in this direction where the platform provides decoupling of application logic from the OS logic based on the execution models they support.

At this level of virtualization, the execution models are the prime drivers for deciding which functionality is required to be a part of shells and these can be categorized into four types:

1) *SFSA*: Single FPGA Single Application
2) *SFMA*: Single FPGA Multiple Applications
3) *MFSA*: Multiple FPGAs Single Application
4) *MFMA*: Multiple FPGAs Multiple Applications

In most cases, multiple applications on a single FPGA are achieved using multiple partial regions. These regions can be asymmetric as well as symmetric in shape. The asymmetric approach is usually taken to support multiple different sizes of modules without having to reconfigure the entire FPGA [54]. The symmetric approach uses similar or identically sized partial regions (also called 'tiled regions' or 'resource slots'). A good example of this is the Erlangen Slot Machine [55] which was designed with flexibility in mind. Further with symmetric approach an application can occupy one or more adjacent slots which provides more flexibility for the resource allocation to reduce internal fragmentation (as shown in RC2F [6]).

Each execution model requires some form of connectivity and these can be split into three different classes: i) Host connectivity, ii) Independent connectivity, and iii) Hybrid connectivity. Shells which support only host connectivity, let a host CPU handle the resource management of the FPGA and use the majority of reconfigurable resources for the applications [16]. With the availability of embedded CPU cores providing a rich set of peripherals (e.g., ZYNQ Ultrascale+ FPGAs), this model is now feasible as a single chip solution. However, for data-centre/cloud environments it can lead to under-utilization of resources as the FPGA allocation is tightly coupled with the CPU allocation. Thus, there are proposals which recommend independent connectivity for these scenarios, which allow sharing of FPGA resources among multiple different CPUs [10], [56] or even standalone FPGAs for the application [2], [10], [15], [57]. This flexibility often requires FPGA support for the network layer and other I/O resources (e.g. Ethernet and memory controllers), which can occupy a

System	Applications: Single	Applications: Multiple
Single FPGA	[16], [59], [47], [2] [60], [54], [61], [56] [63], [64], [65], [66] [67], [68], [3]	[15], [6], [14], [11] [55], [62], [60], [13] [54], [61],
Multi-FPGA	[56], [47], [2], [63] [65], [67], [68], [3]	[15], [14], [10], [6]

TABLE I: Hardware platforms with static and partial regions. Note that the certain platforms may be able to scale multiple FPGAs via CPU, but without the network support or mention in the paper, we refrained from including them into the multi-FPGA category.

considerable amount of the available FPGA resources in some cases (as shown in Table II). In contrast to host connectivity and independent connectivity, the hybrid approach aims at supporting both forms of connectivity to benefit from offloading control intensive or complex resource management tasks to CPUs but also use the special hardware on an FPGA to accelerate I/O accesses if required (as in Microsoft's Catapult platform [2]).

Additionally, it is important to acknowledge that, when using a shell, there are performance overheads due to the layout constraints imposed by the partially reconfigurable regions. For instance, Yazdanshenas and Betz [58] have shown that constraining the logic placement to a particular area on the chip for the user application can lead to high contention for wiring during Place and Route which, in turn, leads to longer wires, resulting in slower modules. Consequently it can be observed that increasing the number of partial regions to support higher multi-tenancy is likely to lead to comparatively slower modules and static logic. Thus, finding the optimal size and number of partial regions for virtualization support is an interesting open direction of research.

C. Scheduling

Scheduling, in general, is a well-established topic in software systems, specifically for multi-tenancy and for improving utilization of resources. The conventional techniques of scheduling are preemptive, non-preemptive, and co-operative scheduling which can be used to share the FPGA in the *time domain*.

However, these techniques cannot be blindly applied for all types of FPGA accelerators as the state information which needs to be saved for an FPGA accelerator is in general non-trivial if a context switch is performed at an arbitrary point in time. This is because the state may be spread out across Flip-Flops, Logic Cells and BRAMs. Saving and restoring the complete state for such a case can easily take anywhere from microseconds to milliseconds [69] in addition to partial reconfiguration latency. Further, the hardware support required for a preemptable hardware module tends to be rather restrictive and very target specific. One possible way to perform this without special hardware restriction is proposed by Rupnow et al. [70], where a hardware task is either blocked, dropped or rolled back to CPU if a context switch needs to be performed. Using preemption on FPGAs entirely transparently is either time consuming (for accessing the static information within the configuration data) or expensive if the state access is provided through an interface (e.g., through a scan chain). This may be solved in future FPGA architectures or by design technologies that reduce the cost of context switching. For example, Bourge et al. [71] proposed an HLS extension that includes a scan chain only for a subset of the available registers by performing live-variable analysis for finding states where the number of live registers to be stored is small.

The non-preemptive approach simplifies the design and can be implemented at relatively low-cost as the accelerators run to completion. While co-operative scheduling operates in a conservative manner and offers context switching when an application reaches an execution checkpoint for minimal overhead [51].

Hardware threads have been proposed in ReconOS [72] and HThreads [73] to integrate scheduling for FPGAs in standard software operating systems to bring FPGAs to the software programming world by utilizing standard software HDLs. The technique requires a tight CPU-FPGA coupling and can be directly applied on MPSoC platforms, as High-Level Synthesis (HLS) is now reaching a mature state.

Most of the scheduling approaches for FPGAs are non-preemptive and focus on optimizing in the time domain. Only very recent research is investigating FPGA virtualization in the *spatial domain*. Asiatici et al. [13] proposed dynamic scheduling which can take advantage of the free slots available at run-time to improve utilization and thus the performance. This scheduling approach maximizes the number of pipeline instances when a task is created and keeps this allocation untouched until task completion. A dynamic scheduling technique which can potentially increase or decrease accelerator's resource consumption according to the workload requirements at run-time could be particularly desirable moving forward in multi-tenant systems.

OS infrastructure	FPGA Area	Multi-tenancy	Scalability	Flexibility	Programmer Productivity	Isolation	Utilization
Byma et al. [15]	74% (BRAM)	✓	Medium	Medium	Medium	Low	Medium
Chen et al. [11]	41% (Logic)*	✓	Medium	Medium	High	Medium	Medium
IBM Zurich [10]	33% (Logic)*	✓	High	Medium	Medium	High	Medium
Fahmy et al. [16]	7% (Total)	✗	Low	Low	Medium	-	Low
Catapult [2]	76% (Total)	✗	High	High	Low	-	Low
Network-attached [56]	32% (BRAM)	✗	Medium	Medium	Medium	-	Low
Tarafdar et al. [47]	19.56% (BRAM)	✗	High	Medium	High	-	Low
Feniks [14]	13% (Logic)	✓	High	High	High	High	Medium
Knodel et al. [6]	42% (Logic)	✓	High	High	Medium	High	Medium
Asiatici et al. [13]	-	✓	Medium	Low	Medium	Medium	High

TABLE II: Comparison of infrastructure present at the node level. For Scalability we consider the support of connectivity in the shell, for Flexibility support for range of accelerator design is considered i.e. custom accelerator to HLL to DSL usage, for programmer productivity development time is considered which would depend on component level support and accelerator generation requirements, for isolation we consider how advance support is provided by component managers and finally for utilization we consider the amount of region which would be in use at run-time this includes support for multiple accelerators on chip as well as level of scheduling techniques employed.
*The only reported area metric.

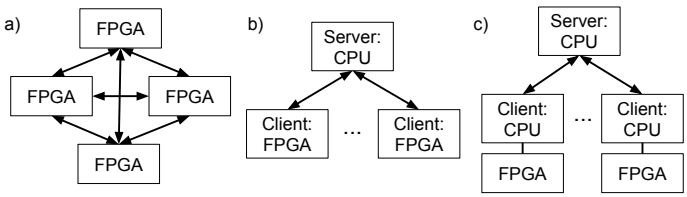

Fig. 5: Set of architectures used to scale the accelerator across multiple FPGAs. Where a) represents FPGA to FPGA communication, b) shows a server-client architecture where FPGA directly communicate with the server and c) denotes the server-client architecture where FPGAs are special peripherals to the client CPU.

Moreover, to target FPGA as a Service (FaaS) and Acceleration as a Service (AaaS) models, specific scheduling approaches have been proposed. One such approach is mapping multiple users which require the same accelerator functionality on the same FPGA such that partial reconfiguration is not required to serve another user [74], [75]. Another approach in this direction has been implemented for VineTalk by Mavridis et al. [76], to provide support for sharing an FPGA within a native server, a virtual machine, or a container in a heterogeneous data-centre. The user can select the appropriate accelerator type (e.g., GPU or FPGA) based on workload or algorithm specifics through this API. With the adoption of OpenCL as a de-facto standard for heterogeneous computing and proposals like SparkCL [77] for bridging the gap between Java and OpenCL, these techniques seem to be leading the way for the heterogeneous computing paradigm. The automation of appropriate accelerator selection based on these methods represents an interesting new research area.

V. Virtualization at Multi-Node level

The aim of virtualization at multi-node level is to map an acceleration job across multiple FPGAs in a transparent manner. To provide this abstraction, a virtualization model must connect multiple FPGAs and abstract the details of how they are connected from the user. Applications at this level would often run on a data-parallel piece of the problem where each FPGA computes the whole job or alternatively where multiple FPGAs accelerate a problem in conjugation with each other.

Multiple FPGAs can commonly be connected in three different ways as shown in Figure 5: i) FPGA-to-FPGA architecture where FPGAs directly communicate with other FPGAs; ii) a server-client architecture where a server is a remote CPU issuing work to the independent FPGAs; and iii) the traditional server-client architecture where both server and clients are CPUs and an FPGA is a special hardware peripheral attached to the client CPUs. Note that systems may combine these models to form a *hybrid architecture* based on the application requirements.

These architectures are the underlying base for the three virtualization models: Custom Clusters, Frameworks, and Cloud services. We discuss each model in further detail below.

A. Custom Clusters

In the Custom Cluster approach, the computation is split among multiple FPGAs in terms of stages where the data is passed after processing from one FPGA to another by FIFO/File/Network semantics. The approach often follows a systolic array model where each Processing Element (PE) is an FPGA. Examples of this FPGA-to-FPGA architecture are Leap [17], JetStream [68] and Maxeler MPC-C and MPC-X series [3]. Leap requires the user to annotate the code with special pragmas to create FIFOs which are latency insensitive to communicate with a different FPGAs. These FIFOs send data using high-speed serial transceivers and expose the content again in FIFO buffers on the other side of the channel. A similar approach is taken by Maxeler when connecting multiple FPGAs using a proprietary point-to-point connection called MaxRing. As an alternative, JetStream connects multiple FPGAs through direct PCIe links.

Accelerators can be designed to communicate directly with other nodes by explicitly using the network connections, such that data movements and compute is tailored to the target application. An example in this category is Levenshtein distance implementation for CUBE (a one-dimensional cluster of 512 FPGAs) which exploits a systolic architecture for stream processing across multiple FPGAs [78].

B. Frameworks

The Framework model for virtualization takes a similar path to the software world and adopts the server-client architecture for deploying accelerators. A server CPU is responsible for configuring the FPGAs and managing application data, while FPGAs are responsible for performing the actual computation. Various data management techniques have been extensively studied before for distributed systems using CPUs and the same techniques can potentially be reused for FPGAs. In particular, frameworks like MapReduce can be supported with FPGAs by implementing the map and reduce functions as FPGA accelerators, where the data can be distributed to FPGAs and collected after the computation in the standard MapReduce manner [18], [19], [79]. Furthermore, frameworks can also be used to bridge the heterogeneity between devices, as the programmer writes the application with a fixed interface. An example of this is Axel, where MapReduce is used to form heterogeneous clusters providing FPGAs and GPUs [20]. However, to implement this, a JVM-based framework needs to be extended to support FPGAs in an efficient manner. It was shown by Chen et al. [80] that JVM-to-FPGA communication overhead can be significant and requires a careful handling when using frameworks such as Apache Spark.

The support for an industry standard language for heterogeneous computing such as OpenCL has been investigated to virtualize FPGAs at the application source code level. For instance, by using the SDAccel framework for OpenCL from the vendor Xilinx, Tarafdar et al. [47] proposes using an MMU layer to map data across multiple FPGAs (i.e. allocating data across multiple FPGA and using a directory to fetch/send the data transparently from/to the remote node). Iordache et al. [75] proposed the concept of FPGA Groups to share one or more physical FPGAs which are configured with the same accelerator. Here the granularity of resource allocation is a whole FPGA which may not lead to high utilization due to fragmentation. However, an auto-scaling algorithm allows variation of the number of FPGAs in a group dynamically. A similar model is proposed by Huang et al. [74] for their Blaze run-time which aims at implementing FPGA as a Services (FaaS). Blaze not only extends Hadoop YARN (a cluster management system) with accelerator sharing among multiple computing tasks but also reduces the required pro-

gramming effort considerably for systems like Apache Spark and YARN.

C. Cloud Services

The cloud service model completely abstracts the details of where the computation is taking place from a user. The user is only guaranteed QoS and correctness of the output thus, in the background, an FPGA can be employed to perform computing instead of a CPU. This approach is different from providing FPGA as a Service (e.g., EC2-F1 service by Amazon [81]), as it does not relate to provisioning of FPGAs but provisioning of the application/web-service itself. This approach was taken by Microsoft for accelerating the Bing web search ranking algorithm [2] to achieve 95% higher performance at the cost of 10% higher power consumption. Similar results have been reported by Baidu for accelerating Deep Neural Networks (DNN) [82]. Moreover, the run-time system can utilize FPGAs as co-processors to accelerate compute-intensive kernels in a high-performance computing environment. An example of this is the work done by El-Araby et al. for Cray XD1 [83] where FPGAs were used as co-processors for the Single Program Multiple Data paradigm.

There are also hybrid architectures which can support multiple forms of connectivity based on the FPGA infrastructure available at the node level. FPGAs with network access in specific can support all forms of connectivity as a CPU can be provided as a soft-core or an embedded SoC on an FPGA. The most common technique to manage an FPGA is to use OpenStack for provisioning of the FPGAs and letting the user connect and program the FPGA using an IP or MAC address of the FPGA or vFPGA [15], [47], [56]. This gives the user the flexibility to connect to the FPGA using remote procedure calls or socket connections. Moreover, this complexity can be further abstracted away by using frameworks and libraries for common applications. A good example of a hybrid architecture is Microsoft's Catapult project [2] which allows the acceleration of tasks as a special peripheral connected to a host CPU core but can also communicate among FPGAs directly in a standalone fashion.

VI. Trends and future directions

Current major directions for FPGA virtualization and OS infrastructure include: i) multi-tenant support to share the same compute substrate among multiple users/applications and ii) scaling towards large networks of FPGAs in data-centres in order to accelerate large-scale problems.

With the rise of multi-tenant support in FPGAs, the OS infrastructure will require more support for resource management which could not only help setting up the FPGA for user access but also perform scheduling to meet QoS requirements and to maximize resource utilization. Such resource managers have been shown to be vital in the software world. An example of this is the Borg cluster management system which Google uses internally to manage most of its clusters [84]. It highlighted that carefully matching different applications to be executed on the same CPU can allow fitting the same workload with 20% fewer machines. Note that there are currently no reasons to believe that such benefits cannot be gained for FPGAs as well.

At data-centre scale, elasticity has been one of the main drivers for many users when using cloud resources and as more data-centre workloads are currently moving to cloud (e.g., [82] and [2]), a similar feature is needed for FPGAs. Currently, the elasticity in FPGAs is commonly employed at the whole FPGA device level with a constraint of running the same accelerator for all applications running on an FPGA. However, moving into the future with a large variety of applications for FPGAs and the availability of larger FPGAs, these constraints may be too limiting to optimize resource utilization in a cloud context. Thus, providing elasticity at a more fine-grained level (e.g., partial regions) may become a more suitable and interesting area of work.

The trend to use a heterogeneous computing using substrates such as CPUs, GPGPUs and FPGAs will continue to grow in order to maximize performance and energy efficiency and FPGA virtualization will have to incorporate this. Hence, moving forward into the future, heterogeneous computing with FPGAs is likely to present a challenging and interesting direction of research.

Moreover, virtualization techniques for FPGAs need to provide security not only for network and memory access but also for the compute substrate itself, as it opens up to new types of attacks. For example, a malicious bitstream can potentially damage an FPGA [85] or crash the system until power cycle reset [86] or perform a side channel attack [87], [88]. To mitigate these types of attacks, FPGA specific solutions need to be designed (e.g., a configuration controller to identify and block malicious bitstreams). Thus a special security layer needs to be designed in parallel of virtualization moving forward.

VII. Conclusion

We have provided a survey on FPGA virtualization that, in particular, covers recent trends of deploying FPGAs in cloud environments. This spans virtualization within an FPGA device, virtualizing complete FPGA devices as well as virtualizing networks of FPGA. With this survey, we intend to summarize state-of-the-art work and highlight important directions of research that are, for example, needed to use FPGAs in multi-tenancy scenario and cloud environments in the same way as CPUs are currently used.

VIII. Acknowledgements

This work is supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632). We would like to thank James Garside from the University of Manchester for his valuable suggestions during manuscript editing.

References

[1] C. Plessl and M. Platzner, "Virtualization of Hardware-Introduction and Survey," in *ERSA*, 2004.
[2] Putnam et al., "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *ISCA*, 2014.
[3] O. Pell et al., "Maximum Performance Computing with Dataflow Engines," in *High-Performance Computing Using FPGAs*, 2013.
[4] H. K.-H. So and C. Liu, "FPGA Overlays," in *FPGAs for Software Programmers*, 2016.
[5] J. Coole et al., "Intermediate fabrics: Virtual Architectures for Circuit Portability and Fast Placement and Routing," in *CODES+ISSS*, 2010.
[6] O. Knodel et al., "Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures," *RECATA*, vol. 2, 2017.
[7] S. A. Fahmy and K. Vipin, "A Case for FPGA Accelerators in the Cloud," *SoCC*, 2014.
[8] A. Brant and G. G. F. Lemieux, "ZUMA: An Open FPGA Overlay Architecture," in *FCCM*, 2012.
[9] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A Fully Pipelined and Dynamically Composable Architecture of CGRA," in *FCCM*, 2014.

[10] J. Weerasinghe et al., "Enabling FPGAs in Hyperscale Data Centers," in *15th IEEE UIC-ATC-ScalCom)*, Aug 2015.
[11] F. Chen et al., "Enabling FPGAs in the Cloud," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14, 2014.
[12] W. Wang, "pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment," in *CODES+ ISSS*. IEEE, 2013.
[13] M. Asiatici et al., "Virtualized Execution Runtime for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, 2017.
[14] J. Zhang, et al., "The Feniks FPGA Operating System for Cloud Computing," in *8th APSys*, 2017.
[15] S. Byma et al., "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack," in *FCCM*, May 2014.
[16] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA Accelerators for Efficient Cloud Computing," in *CloudCom*, Nov 2015.
[17] K. Fleming and M. Adler, "The LEAP FPGA Operating System," in *FPGAs for Software Programmers*, 2016.
[18] Z. Wang et al., "Melia: A MapReduce Framework on OpenCL-Based FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, Dec. 2016.
[19] J. H. C. Yeung, "Map-reduce as a Programming Model for Custom Computing Machines," in *FCCM*, April 2008.
[20] K. H. Tsoi and W. Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," in *18th FPGA*. ACM, 2010.
[21] D. Koch et al., "An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions," in *FPL*, Sept 2013.
[22] A. K. Jain et al., "Efficient Overlay Architecture Based on DSP Blocks," in *FCCM*, May 2015.
[23] A. K. Jain, "Architecture Centric Coarse-Grained FPGA Overlays," Ph.D. dissertation, Nanyang Technological University, 2017.
[24] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP block based FPGA Soft Processor," in *FPT*, 2012.
[25] Xilinx, "MicroBlaze Soft Processor Core." [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html
[26] Altera, "Nios II Processor." [Online]. Available: https://www.altera.com/products/processors/overview.html
[27] A. Severance et al., "Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor," in *CODES+ISSS*, 2013.
[28] A. Severance and G. Lemieux, "VENICE: A Compact Vector Processor for FPGA Applications," in *HCS*, 2011.
[29] C. H. Chou et al., "VEGAS: Soft Vector Processor with Scratchpad Memory," in *FPGA*, 2011.
[30] P. Yiannacouras, J. G. Steffan, and J. Rose, "VESPA: Portable, Scalable, and Flexible FPGA-based Vector Processors," in *CASES*, 2008.
[31] J. Yu, G. Lemieux, and C. Eagleston, "Vector Processing as a Soft-core CPU Accelerator," in *FPGA*, 2008.
[32] S. Shukla et.al., "QUKU: A FPGA Based Flexible Coarse Grain Architecture Design Paradigm using Process Networks," in *IPDPS*, 2007.
[33] A. Landy and G. Stitt, "A Low-overhead Interconnect Architecture for Virtual Reconfigurable Fabrics," in *CASES*, 2012.
[34] J. Coole and G. Stitt, "Fast, Flexible High-Level Synthesis from OpenCL using Reconfiguration Contexts," *IEEE Micro*, vol. 34, no. 1, 2014.
[35] V. Govindaraju et al., "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Micro*, 2012.
[36] A. D. Brant, "Coarse and Fine Grain Programmable Overlay Architectures for FPGAs," *MSc Thesis, University Of British Columbia*, 2012.
[37] R. Rashid et al., "Comparing Performance, Productivity and Scalability of the TILT Overlay Processor to OpenCL HLS," in *FPT*, 2014.
[38] K. Paul, C. Dash, and M. S. Moghaddam, "reMORPH: A Runtime Reconfigurable Architecture," in *ECDSD*, 2012.
[39] C. Hilton et al., "PNoC: a flexible circuit-switched NoC for FPGA-based systems," *IEEE Proc. - Computers and Digital Techniques*, 2006.
[40] N. Kapre et al., "Packet Switched vs. Time Multiplexed FPGA Overlay Networks," in *FCCM*, 2006.
[41] M. K. Papamichael et al., "CONNECT: Re-examining Conventional Wisdom for Designing Nocs in the Context of FPGAs," in *FPGA*, 2012.
[42] Y. Huan and A. DeHon, "FPGA Optimized Packet-switched NoC using Split and Merge Primitives," in *FPT*, 2012.
[43] N. Kapre and J. Gray, "Hoplite: Building Austere Overlay NoCs for FPGAs," in *FPL*, 2015.
[44] S. Yazdanshenas and V. Betz, "Interconnect Solutions for Virtualized Field-Programmable Gate Arrays," *IEEE Access*, vol. 6, 2018.
[45] R. L. Lysecky, K. Miller, F. Vahid, and K. A. Vissers, "Firm-Core Virtual FPGA for Just-in-Time FPGA Compilation," in *FPGA*, 2005.
[46] Edico Genome, "DRAGEN Bio-IT Platform," Accessed: 6 April 2018. [Online]. Available: http://edicogenome.com/dragen-bioit-platform/
[47] N. Tarafdar et al., "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center," in *FPGA '17*. ACM, 2017.
[48] N. Abbani et al., "A Distributed Reconfigurable Active SSD Platform for Data Intensive Applications," in *HPCC*, Sept 2011.
[49] S. R. Chalamalasetti et al., "An FPGA Memcached Appliance," in *13th FPGA*, 2013.
[50] M. Lavasani et al., "An FPGA-based In-Line Accelerator for Memcached," *IEEE Computer Architecture Letters*, vol. 13, no. 2, July 2014.
[51] T. Xia et al., "Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators," in *FPT*, Dec 2016.
[52] A. K. Jain, "Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform," *JSPS*, vol. 77, no. 1, Oct 2014.
[53] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, May 2015.
[54] Q. ZHAO, "Enabling FPGA-as-a-Service in the Cloud with hCODE Platform," *IEICE Transactions on Information and Systems*, 2018.
[55] C. Bobda et al., "The Erlangen Slot Machine: Increasing Flexibility in FPGA-based Reconfigurable Platforms," in *FPT*, Dec 2005.
[56] J. Weerasinghe et al., "Network-Attached FPGAs for Data Center Applications," in *FPT*, Dec 2016.
[57] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow, "Designing for FPGAs in the Cloud," *IEEE Design and Test*, 2017.
[58] S. Yazdanshenas and V. Betz, "Quantifying and Mitigating the Costs of FPGA Virtualization," in *27th FPL*, Sept 2017.
[59] K. Vipin and S. A. Fahmy, "DyRACT: A Partial Reconfiguration Enabled Accelerator and Test Platform," in *24th FPL*, Sept 2014.
[60] L. Wirbel, "Xilinx SDAccel: A Unified Development Environment for Tomorrows Data Center," *The Linley Group Inc*, 2014.
[61] Intel FPGA, "SDK for OpenCL," *Programming Guide. UG-OCL002*, vol. 31, 2016.
[62] M. Vesper, D. Koch, and K. Pham, "PCIeHLS: an OpenCL HLS framework," in *FSP 2017*, Sept 2017.
[63] K. Vipin et al., "System-level FPGA Device Driver with High-Level Synthesis Support," in *FPT*, Dec 2013.
[64] M. Jacobsen et al., "RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators," *ACM TRETS*, vol. 8, no. 4, Sep. 2015.
[65] K. Eguro, "SIRC: An Extensible Reconfigurable Computing Communication API," in *18th FCCM*, May 2010.
[66] J. Kulp, S. Siegel, and J. Miller, "Open Component Portability Infrastructure (OPENCPI)," Mercury Federal Systems Inc Arlington VA, Tech. Rep., 2013.
[67] D. Theodoropoulos et al., "Multi-FPGA Evaluation Platform for Disaggregated Computing," in *25th FCCM*, April 2017.
[68] M. Vesper et al., "JetStream: An Open-source High-performance PCI Express 3 Streaming Library for FPGA-to-Host and FPGA-to-FPGA Communication," in *26th FPL*, Aug 2016.
[69] M. Happe et al., "Preemptive Hardware Multitasking in ReconOS," in *Applied Reconfigurable Computing*. Springer Inter. Publishing, 2015.
[70] K. Rupnow et al., "Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking," in *17th FCCM*, April 2009.
[71] A. Bourge et al., "Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow," *ACM TRETS*, Dec. 2016.
[72] E. Lubbers and M. Platzner, "ReconOS: An RTOS Supporting Hard-and Software Threads," in *FPL*, Aug 2007.
[73] W. Peck et al., "Hthreads: A Computational Model for Reconfigurable Devices," in *FPL*, Aug 2006.
[74] M. Huang, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," in *SoCC '16*, 2016.
[75] A. Iordache et al., "High Performance in the Cloud with FPGA Groups," in *9th UCC*, Dec 2016.
[76] S. Mavridis et al., "VineTalk: Simplifying Software Access and Sharing of FPGAs in Datacenters," in *27th FPL*, Sept 2017.
[77] O. Segal et al., "SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters," *arXiv:1505.01120*, 2015.
[78] M. Yoshimi et al., "A Performance Evaluation of CUBE: One-Dimensional 512 FPGA Cluster," in *ARC*, 2010.
[79] Y. Shan et al., "FPMR: MapReduce Framework on FPGA," in *18th FPGA*, 2010.
[80] Y. T. Chen et al., "When Spark Meets FPGAs: A Case Study for Next-Generation DNA Sequencing Acceleration," in *24th FCCM*, May 2016.
[81] Amazon Web Services, "AWS EC2 FPGA Hardware and Software Development Kit." 2009. [Online]. Available: https://github.com/aws/aws-fpga
[82] J. Ouyang et al., "SDA: Software-defined Accelerator for Large-scale DNN Systems," in *IEEE 26th HCS*, Aug 2014.
[83] E. El-Araby et al., "Virtualizing and Sharing Reconfigurable Resources in High-Performance Reconfigurable Computing Systems," in *HPRCTA*, Nov 2008.
[84] A. Verma et al., "Large-scale Cluster Management at Google with Borg," in *EuroSys*, 2015.
[85] C. Beckhoff et al., "Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration," in *FPL*, Aug 2010.
[86] D. R. E. Gnad et al., "Voltage drop-based fault attacks on FPGAs using valid bitstreams," in *27th FPL*, Sept 2017.
[87] I. Giechaskiel and K. Eguro, "Information Leakage Between FPGA Long Wires," *arXiv preprint arXiv:1611.08882*, 2016.
[88] P. Swierczynski et al., "Bitstream Fault Injections (BiFI)-Automated Fault Attacks Against SRAM-Based FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 348–360, March 2018.