HETERO-VITAL: A Virtualization Stack for Heterogeneous FPGA Clusters

Yue Zha  
Department of Electrical and Systems Engineering  
University of Pennsylvania, Philadelphia, USA  
zyh@seas.upenn.edu

Jing Li  
Department of Electrical and Systems Engineering  
University of Pennsylvania, Philadelphia, USA  
janeli@seas.upenn.edu

Abstract—With field-programmable gate arrays (FPGAs) being widely deployed into data centers, an efficient virtualization support is required to fully unleash the potential of cloud FPGAs. Nevertheless, existing FPGA virtualization solutions only support homogeneous FPGA clusters comprising identical FPGA devices. Representative work such as ViTAL provides sufficient system support for scale-out acceleration and improves the overall resource utilization through a fine-grained spatial sharing. While these existing solutions (including ViTAL) can efficiently virtualize a homogeneous cluster, it is hard to extend them to virtualizing a heterogeneous cluster which comprises multiple types of FPGAs. We expect the future cloud FPGAs are likely to be more heterogeneous due to hardware rolling upgrade.

In this paper, we rethink FPGA virtualization from ground up and propose HETERO-VITAL to virtualize heterogeneous FPGA clusters. We identify the conflicting requirements of runtime management and offline compilation when designing the abstraction for a heterogeneous cluster, which is also the fundamental reason why the single-level abstraction as proposed in ViTAL (and other prior works) cannot be trivially extended to the heterogeneous case. To decouple these conflicting requirements, we provide a two-level system abstraction in HETERO-VITAL. Specifically, the high-level abstraction is FPGA-agnostic and provides a simple and homogeneous view of the FPGA resources to simplify the runtime management. On the contrary, the low-level abstraction is FPGA-specific and exposes sufficient spatial resource constraints to the compilation framework to ensure the mapping quality. Rather than simply adding a layer on top of the single-level abstraction as proposed in ViTAL and other prior work, we judiciously determine how much hardware details should be exposed at each level to balance the management complexity, mapping quality and compilation cost. We then develop a compilation framework to map applications onto this two-level abstraction with several optimization techniques to further improve the mapping quality. We also provide a runtime management policy to alleviate the fragmentation issue, which becomes more severe in a heterogeneous cluster due to the distinct resource capacities of diverse FPGAs.

We evaluate HETERO-VITAL on a custom-built FPGA cluster and demonstrate its effectiveness using machine learning and image processing applications. Results show that HETERO-VITAL reduces the average response time (a critical metric for QoS) by 79.2% for a heterogeneous cluster compared to the non-virtualized baseline. When virtualizing a homogeneous cluster, HETERO-VITAL also reduces the average response time by 42.0% compared with ViTAL due to a better system design.

I. INTRODUCTION

Field-programmable gate arrays (FPGAs) have been deployed in several commercial cloud platforms (Amazon F1 [5], Microsoft Azure [53], InAccel [35], Alibaba Cloud F3 [20], Nimbia [54], etc.) to support on-demand acceleration. To better manage FPGAs in the cloud, system mechanisms have been proposed to virtualize FPGAs from both academia [17][41][63][80][82] and industry [15][27][34][56]. According to the supported cloud service model (Fig. 1), we divide these virtualization methods into three broad categories: (1) the first class of methods [15][42][56][63][83] only provide virtualization support for the I/O interface through an application-independent shell, while reconfigurable resources (e.g., lookup tables) are not virtualized. Users control the runtime management and scheduling (if needed), which is similar to that in the IaaS model. (2) the second class of methods [9][17][41][45][80][82] provide a system abstraction to virtualize both the I/O interface and the reconfigurable resources. They also provide a runtime system for resource management to support the PaaS model. (3) The other methods [27][33][34][79] abstract a set of application-specific accelerators into pre-defined APIs to support the SaaS model. More details will be discussed in Section V-A.

Fig. 1. A conceptual diagram illustrates the service models for cloud FPGAs.

We choose to explore the FPGA virtualization in the context of the PaaS service model as it provides a simple and scalable platform for developers to build their own FPGA applications and has been widely used in several prior works [9][17][41][45]. Among these works, ViTAL [80] is a representative one that provides a virtualization stack for the PaaS model. The core of ViTAL is a single-level system abstraction that abstracts a homogeneous FPGA cluster into an array of identical virtual blocks. Applications are transparently partitioned and mapped onto these virtual blocks by the provided compilation framework. Therefore, ViTAL...
creates the illusion of an infinitely large virtual FPGA to users to reduce the programming complexity and enable scale-out acceleration. This abstraction also allows the compilation framework to generate position-independent mapping so that applications can be relocated to arbitrary positions at runtime without time-consuming recompilation. This enables a fine-grained spatial sharing that can dynamically respond to the actual load and resource availability. Despite of its promise, it is nontrivial to extend ViTAL to a heterogeneous FPGA cluster which is composed of different types of FPGAs.

In this paper, we rethink FPGA virtualization from ground up and generalize the method to the case of heterogeneous FPGA cluster. At first, we identify the conflicting requirements of exposing how much spatial hardware details to the abstractions in order to achieve both low runtime management complexity and high compilation quality for virtualizing heterogeneous FPGAs. Specifically, the runtime system requires the abstraction to hide the distinct spatial resource constraints (e.g., resource type, layout and capacity) of different types of FPGAs to reduce the management complexity. On the other hand, the compilation framework requires the abstraction to expose sufficient resource constraints to ensure the mapping quality. These conflicting requirements are the fundamental reason why it is hard to efficiently virtualize heterogeneous FPGA cluster using the previously proposed single-level abstraction [80]. With the growing popularity of FPGAs in data centers, we expect the heterogeneity in FPGA clusters will further grow due to hardware rolling upgrade. The limitation of a single-level abstraction can manifest itself as the major limitation in either runtime management efficiency or compilation quality.

To address this limitation, we propose the HETERO-ViTAL stack that provides a two-level system abstraction to decouple the aforementioned conflicting requirements (Architecture Layer, Section III-A). This system abstraction is inspired by ViTAL’s abstraction, but rather than simply adding an abstraction layer on top of ViTAL’s abstraction, we judiciously determine the amount of hardware details that should be exposed at each level to reap the benefits of having a two-level abstraction. As illustrated in Fig. 5, the high-level abstraction is designed to be FPGA-agnostic and hide as many hardware details as possible to maximally simplify the runtime management. It comprises a pool of high-level virtual blocks (HL virtual blocks) and the resource type/capacity of each HL virtual block can be arbitrarily chosen to abstract away the heterogeneity across different types of FPGA devices. On the contrary, the low-level abstraction is designed to be FPGA-specific and expose as many hardware details as possible to ensure the mapping quality. As shown in Fig. 5, it virtualizes FPGA devices into an array of identical low-level virtual blocks (LL virtual blocks). To capture the distinct spatial resource constraints of diverse FPGAs, the low-level abstraction comprises multiple types of LL virtual blocks, where one type of LL virtual block is a partition of a specific type of a physical FPGA. In comparison to ViTAL’s virtual blocks that provide a latency-insensitive interface for inter-block communication, the LL virtual blocks provide an interface with a deterministic latency for the inter-block communication to better capture the heterogeneity within one FPGA, i.e., the different inter-die and intra-die communication latency. This allows the compilation framework to fully utilize the resource provided by LL virtual blocks and minimize the internal fragmentation. Similar to ViTAL, LL virtual blocks also provide virtualization support for the peripherals, e.g., on-board DRAM.

We then develop a compilation framework to map applications onto HETERO-ViTAL’s system abstraction (Compilation Layer, Section III-B). The proposed two-level system abstraction not only decouples the conflicting requirements but also enables a high-quality and low-complexity two-stage mapping process. Specifically, the first stage maps applications onto the high-level abstraction by decomposing them into a number of HL virtual blocks with the objective of minimizing the required inter-block communication bandwidth. As HL virtual blocks have arbitrary resource capacity and type, no hardware constraint is introduced during this decomposing process, leading to a lower complexity compared with the partition process in ViTAL. The second stage maps one HL virtual block into an array of identical LL virtual blocks with the objective of minimizing the number of allocated LL virtual blocks. To support a flexible runtime deployment, one HL virtual block is mapped onto all types of LL virtual blocks. Compared with ViTAL’s latency insensitive interface, the interface with a deterministic latency in the LL virtual blocks allows the compilation framework to fully utilize the on-chip routing network, thereby improving the resource utilization. To deploy compiled LL virtual blocks onto physical FPGAs, we partition physical FPGAs into regions (Fig. 5). Certain regions are reserved by the system during runtime deployment. Compared with ViTAL’s abstraction, HETERO-ViTAL’s abstraction supports a more flexible runtime deployment as the HL virtual blocks can hide more hardware details. Consequently, we can reduce the amount of resources reserved by the system and thus maximize the resources that are available to users to improve the aggregated system performance.

A system controller is included to perform the runtime resource management and expose APIs for system integration (System Layer, Section III-C). Resource allocation is a challenging task for heterogeneous FPGAs due to the diverse characteristics of FPGA devices and cloud applications (Section II-B). HETERO-ViTAL provides a heuristic-based policy to optimize the resource allocation by considering several factors, including the performance variance of each application across different FPGAs, the potential performance interference across different applications, and the resource fragmentation issue. Moreover, this system controller applies appropriate scheduling strategies for different cloud instances (Section II-C) to improve the quality of service (QoS).

In particular, we made the following major contributions:

1. We identify the conflicting requirements of runtime management and offline compilation for virtualizing heterogeneous FPGAs. Specifically, the runtime system requires the abstraction to hide spatial resource constraints for low runtime
management complexity, whereas the compilation framework requires the abstraction to expose them for high mapping quality. Such contradictory requirements are the fundamental reason why it is hard to apply prior single-level abstraction to virtualize a heterogeneous FPGA cluster.

2. We rethink the FPGA virtualization and propose the HETERO-VITAL stack for virtualizing heterogeneous FPGA clusters. It provides a new two-level system abstraction to effectively decouple aforementioned conflicting requirements. This two-level system abstraction also enables a low-complexity two-stage mapping flow that is implemented by the provided compilation framework. We also provide a heuristic-based policy for resource management and appropriate scheduling strategies for different cloud instances to maximize the aggregated system performance.

3. We demonstrate the benefits of HETERO-VITAL on a system prototype — an FPGA cluster with two types of FPGAs (Xilinx XC7VU35P and XCKU115). We perform a comprehensive evaluation for each layer on representative benchmarks. The results show that HETERO-VITAL reduces the response time by 79.2% on average for virtualizing a heterogeneous FPGA cluster compared with the non-virtualized baseline. Moreover, HETERO-VITAL also reduces the average response time by 42.0% compared with ViTAL when virtualizing a homogeneous cluster (only deployment applications onto XC7U35P).

The rest of the paper is organized as follows. Section II provides the background information. Section III presents the details of each layer in the HETERO-VITAL stack. Section IV presents the evaluation results. Section V discusses the related work, followed by Section VI to conclude the paper.

II. BACKGROUND

A. FPGA Deployment

This subsection discusses the various deployment methods for integrating FPGAs into the cloud platform. We note that the methods such as using FPGA as network switch for software-defined network [77] [78] are not included.

Tightly-attached: FPGAs can be tightly integrated with CPUs either in the same package [22] [36] or on the same board using a low-latency and cache-coherent interconnection (Fig. 2a), such as Intel QuickPath Interconnect (QPI) [55] [57] [58]. Nevertheless, such tight integration is not expected to be widely adopted in cloud, since it breaks the homogeneity of computing modules and increases the complexity of design, deployment and maintenance [69].

PCIe-attached: FPGAs can be implemented on a daughter-card and connected to the host CPU through the high-speed point-to-point PCIe interconnection (Fig. 2b). This is a popular deployment option and has been used for other hardware accelerators such as GPUs.

Network-attached: FPGAs can also be directly connected to the datacenter network and communicate with CPU nodes through this network (Fig. 2c). This reduces the deployment and management complexity and has been used for deploying other hardware accelerators such as Google TPU [28].

Existing commercial clouds typically adopt a hybrid method to deploy FPGAs. For instance, Microsoft [27] and Amazon [5] attach one or multiple FPGAs to the host CPUs using PCIe and deploy a secondary network for the inter-FPGA communication (Fig. 2d). HETERO-VITAL is designed for this hybrid deployment method.

Fig. 2. Conceptual diagrams illustrates the popular deployment methods for FPGAs, which are (a) tightly integrated with CPU in the same package or on the same board, (b) connected to CPU through PCIe, or (c) directly attached to the datacenter network. (d) Commercial FPGA clouds [27][5] typically use a hybrid method, which is the target configuration of HETERO-VITAL.

B. Cloud Workflow Characterization

In order to develop an efficient runtime management policy for cloud FPGAs, it is necessary to understand the characteristics of cloud workloads. In general, workloads for cloud FPGAs can be broadly categorized into two types: batch processing and streaming processing [39]. Batch workloads process high volumes of data that have already been collected and stored in datacenter and the main performance metric is throughput. On the contrary, streaming workloads process streaming data and the main performance metric is latency. An application (e.g., AI inference task) could be either batch or streaming workload depending on users’ demands.

C. Cloud Instance Characterization

Commercial cloud platforms allow users to request different cloud instances to explore the trade-off between cost and performance. On-demand instances and spot instances (or preemptible instances in Google Cloud) are the two major types of instances in existing cloud [2][30]. The main difference is that on-demand instances cannot be interrupted and have a higher priority for scheduling, while the spot instances can be interrupted by the management system with a lower priority, thereby having a lower cost. These two types of instances are also available for hardware accelerators such as GPU [4] and Google TPU [29]. Although only on-demand instances are provided for FPGAs in existing commercial clouds [3], we expect both instances will be available when cloud FPGA resources are virtualized, following the same trend as in other hardware accelerators. Thus, we consider both instances when designing the runtime scheduler in HETERO-VITAL.

D. FPGA Virtualization Stack: VITAL

Most commercial clouds such as Amazon F1 [5] manage a pool of FPGA resources at a per-device granularity, i.e., allocating an entire FPGA device exhaustively to one application regardless of the amount of used resources. As being extensively discussed in prior works [41][80], such management method leads to an inefficient resource utilization.
ViTAL is a recent FPGA virtualization stack proposed by Zha et al. [80]. As illustrated in Fig. 3, it abstracts a homogeneous FPGA cluster into an array of virtual blocks with identical spatial resource constraints, i.e., standardized resource type, layout, capacity and interface. Applications are partitioned and mapped onto a certain number of virtual blocks based on their resource requirement. As the partition is transparently performed by the compilation framework, ViTAL creates the illusion of an infinitely large virtual FPGA to users to simplify the programming complexity and enable scale-out acceleration. One virtual block is deployed into one physical block at runtime. As all physical blocks are also identical, one compiled virtual block can be relocated into an arbitrary physical block without time-consuming recompilation to enable a dynamic and fine-grained FPGA sharing. Virtual blocks provide a latency-insensitive interface for the inter-block communication to hide the latency and bandwidth difference between on-chip and off-chip interconnection network. Consequently, a group of virtual blocks can be either deployed into one physical FPGA or multiple FPGAs to support different resource allocations (application #2 in Fig. 3). The inter-block communication and synchronization are realized in a similar way as the solution for producer-consumer problem [7][8]. Specifically, when the fixed-size buffer in the interface is full (empty), the corresponding virtual block that generates (consumes) the data goes to sleep by disabling its clock. The interface contains a control logic to disable or enable the clock based on the buffer’s status.

Despite its promise, it is hard to extend ViTAL to efficiently virtualizing a heterogeneous FPGA cluster due to three reasons. At first, it cannot decouple the conflicting requirements of runtime management and offline compilation. This is a common limitation of single-level system abstractions when virtualizing a cluster with different types of FPGAs, even not in the context of cloud computing. For instance, the overlay architecture is proposed to enable code portability across different types of FPGAs [11][70]. But it introduces a high degradation in the mapping quality due to the limited hardware details exposed to the compilation framework (e.g., the resource usage increases by ∼ 40× in [11]). Moreover, the unified latency-insensitive interface is another impediment to extending ViTAL to heterogeneous clusters, since it cannot capture the increasing heterogeneity of the interconnection network either within one FPGA or across different types of FPGAs. This could lead to a non-negligible degradation in the compilation quality. Finally, ViTAL’s scheduling policy only considers the inter-FPGA communication cost when allocating resources, which leads to a sub-optimal resource allocation due to the resource fragmentation issue (Fig. 4). As resource fragmentation becomes more severe in a heterogeneous FPGA cluster, ViTAL’s runtime policy might not be able to generate acceptable resource allocation.

III. THE HETERO-ViTAL STACK

The proposed HETERO-ViTAL stack comprises three layers to provide an efficient virtualization support for heterogeneous FPGA clusters. The following subsections presents a detailed description of each layer.

A. Architecture Layer

The architecture layer comprises a two-level system abstraction that (1) serves as an intermediate layer between physical FPGAs and the compilation layer (Section III-B) and (2) creates a manageable resource pool for the system layer (Section III-C). This two-level system abstraction effectively decouples the conflicting requirements of runtime management and offline compilation by providing separate abstractions for these two process. Specifically, the high-level abstraction is used for the runtime management, while the low-level abstraction is used for the offline compilation.

The high-level abstraction is designed to be FPGA-agnostic and hide as many hardware details as possible. As depicted in Fig. 5a, the high-level abstraction comprises a pool of high-level virtual blocks (HL virtual blocks) that are connected
by the latency-insensitive interface. Different from ViTAL’s virtual block that exposes spatial resource constraints, the HL virtual block in HETERO-ViTAL has no such constraint to abstract away the heterogeneity across FPGAs. The resource type, layout and capacity of one HL virtual block can be arbitrarily chosen by the provided compilation framework (Section III-B). This not only simplifies the mapping flow (Step 2 in Section III-B) but also provides a homogeneous resource pool to the runtime system (Section III-C). Another difference is that ViTAL’s virtual blocks are organized in an array (Fig. 3), while HL virtual blocks are organized in a pool. The array organization is used in ViTAL because it needs to expose this spatial information to its compilation framework to guarantee the compilation quality and correctness. Nevertheless, this also poses constraints on the runtime deployment. The HL abstraction can hide this spatial constraint as it is not used for compilation, thereby enabling a more flexible runtime deployment. This also enables a more efficient FPGA implementation as discussed in Section III-A1. HL virtual blocks also contain an interface for peripherals to provide the necessary virtualization support (Fig. 5a).

The low-level abstraction is designed to be FPGA-specific and expose as many hardware details as possible to the compilation framework. As illustrated in Fig. 5b, it uses an array of identical low-level virtual blocks (LL virtual blocks) to virtualize the resources of one FPGA and comprises multiple arrays to support different types of FPGAs. The number of LL virtual block arrays is equal to the number of FPGA types in the cluster. One LL virtual block is a partition of a specific type of FPGA with reconfigurable resources organized under the corresponding spatial constraints (Fig. 5b). Different from ViTAL’s virtual block that only contain one inter-block communication interface, the LL virtual blocks provide two such interfaces. A latency-insensitive interface is used for the communication between LL virtual blocks from different arrays and an interface with a deterministic latency is used for the communication between adjacent LL virtual blocks in the same array (Fig. 5b). This exposes additional hardware details to the compilation framework to improve the mapping quality. As we will show, the latency-insensitive interface is applied for inter-FPGA communication, while the other one is used for intra-FPGA communication. This low-level abstraction also allows the compilation framework to apply appropriate optimization goals for different interconnections, i.e., minimizing the required bandwidth for the inter-FPGA communication and maximizing the utilization of the on-chip interconnection network.

1) Virtual-to-Physical Mapping: As illustrated in Fig. 6, one HL virtual block is offline mapped into an array of LL virtual blocks and then deployed into one physical FPGA at runtime. To support a flexible runtime deployment, one HL virtual block is mapped onto all feasible LL virtual block arrays. One LL virtual block array is feasible if it provides all the resources required by the HL virtual block. Consequently, one HL virtual block can have multiple mapping results, and the runtime system selects the appropriate mapping result to deploy one HL virtual block into the corresponding type of FPGA (Fig. 6). One HL virtual block is deployed into one FPGA device, thus, the latency-insensitive interface in both HL and LL virtual blocks is used for inter-FPGA communication. As the size of one HL virtual block could be smaller than the capacity of one FPGA, the runtime system can deploy HL virtual blocks from different applications onto the same FPGA device as long as that FPGA’s capacity is large enough to enable FPGA sharing.

Fig. 5. A conceptual diagram illustrates the proposed two-level system abstraction. (a) The high-level abstraction comprises a pool of high-level virtual block (HL virtual block) that are connected by the latency-insensitive interface. One HL virtual block has no spatial resource constraint to hide the heterogeneity across FPGAs. (b) The low-level abstraction comprises multiple arrays of low-level virtual blocks (LL virtual blocks), where one array abstracts one type of FPGA. One LL virtual block contains reconfigurable resources (e.g., BRAM) organized under specific spatial constraints. Two sets of interfaces are provided for inter-array and intra-array communication.

Fig. 6. A conceptual diagram illustrates the virtual-to-physical mapping, where one HL virtual block is offline mapped onto an array of LL virtual block and deployed into one FPGA at runtime. Multiple mapping results are generated for one HL virtual block to support a flexible runtime deployment. To deploy an array of LL virtual blocks into one physical FPGA, one FPGA is partitioned into Service Region and User Region.
To deploy $LL$ virtual blocks onto physical FPGAs, one FPGA device is partitioned into two regions, as illustrated in Fig. 6. The Service Region is reserved by the system and is not exposed to users. It contains dedicated modules to realize the virtualization support for the peripheral devices attached to the physical FPGAs, such as the on-board DRAM [80]. The User Region is further divided into a group of identical physical blocks. We adopt the method proposed in ViTAL to create these physical blocks. Different from ViTAL, HETERO-ViTAL does not create the Communication Region in physical FPGAs. In ViTAL, the implementation of its latency-insensitive interface depends on the actual runtime deployment. Therefore, it creates the Communication Region as a partial reconfigurable region to dynamically load the appropriate implementation at runtime. Nevertheless, due to the constraints in creating partial reconfigurable regions [75], the amount of resources provided by the communication region is more than required, which leads to a non-negligible resource waste ($\sim 10\%$ of the total FPGA resources). HETERO-ViTAL separates the interface for inter-FPGA and intra-FPGA communication, thus, the implementation can be determined at the offline compilation time. Consequently, it does not require the communication region and the implementation of the interface can be deployed into the physical blocks together with the user logic. The elimination of the communication region increases the amount of resource available to users (Section IV-B). Sharing the resources in one physical block between user logic and the communication interface also improves the resource utilization. Specifically, if one $LL$ virtual block does not use all communication interfaces (e.g., it has no inter-FPGA communication), then more resources can be provisioned for user logic. On the contrary, in ViTAL, the resources in Communication Region is wasted if one virtual block does not use the latency-insensitive interface.

2) Heterogeneity Within one FPGA: The state-of-the-art commercial FPGAs have more complex architectural features than the simplistic FPGA architecture that is frequently cited in textbooks or publicly available tutorials, such as the multi-die package [57]. The $LL$ virtual blocks need to expose these additional hardware details to the compilation framework to improve the mapping quality. As illustrated in Fig. 7a, the low-level abstraction comprises multiple $LL$ virtual block arrays for one type of FPGA to account for the difference between intra-die and inter-die communication. One $HL$ virtual block is mapped onto all these $LL$ virtual block arrays to support different deployment. The number of required $LL$ virtual block arrays is equal to the number of physical blocks in one die. As vendors typically adopt small dies to improve yield, the number of required $LL$ virtual block arrays and the added compilation cost is limited, e.g., 4 for XCVU37P FPGA. Moreover, one $LL$ virtual block array could be reused across a set of FPGAs to effectively amortize the compilation cost (Fig. 7b). This is because vendors reuse a large portion of one die design across a set of FPGAs\(^1\) to minimize the design cost [74]. The major difference is the number of dies and the provided I/O components (e.g., the high-speed transceivers), which does not change the low-level abstraction.

B. Compilation Layer

The compilation layer provides a generic compilation flow to map applications written in high-level programming languages onto the proposed system abstraction (Section III-A) and generate mapping results that can be managed by the runtime system (Section III-C). We develop a set of custom tools either from scratch or leveraging the APIs provided by RapidWright [48]. These tools allow us to maximally reuse the proprietary FPGA tools (Vivado in our implementation) in our compilation flow (Fig. 8) to achieve a compilation quality comparable to the conventional FPGA compilation flow.

As shown in Fig. 8, the proposed compilation flow comprises five steps: synthesis, mapping stage I, mapping stage II, relocation and global place&route (P&R).

Step 1: Synthesis. This step reuses existing high-level synthesis tools to convert applications written in high-level programming languages into Verilog RTL code.

Step 2: Mapping Stage I. This step has two sub-steps to map the input RTL code onto the high-level abstraction. The first sub-step uses a custom tool to partition the RTL code into a given number of $HL$ virtual blocks with the optimization goal of minimizing the inter-block communication cost (in terms of the number of inter-block connections). As the capacity of one

\(^{1}\)For instance, VU31P, VU33P, VU35P, VU37P, VU45P, VU47P and VU57P from Xilinx have a similar die design.
**HL** virtual block can be arbitrarily chosen, this partition step is performed with no hardware constraint to obtain the optimal result. This is different from the partition step in ViTAL, which is performed under a tight resource constraint, i.e., the fixed capacity of virtual blocks. This custom tool (the partition step in Fig. 8) builds the dataflow graph (DFG) of the input RTL and uses the min-cut algorithm [62] to partition one application. This partition is performed at the granularity of Verilog module, i.e., one node in the built DFG is a module. This prunes the search space with a negligible degradation in the partition quality since inter-module communication bandwidth is typically much lower than the intra-module communication bandwidth. This partition process uses a recursive method that is described in Section III-B1.

The second sub-step uses a custom tool to generate the latency-insensitive interface for each **HL** virtual block. Rather than transferring the output signals of user logic in a cycle-by-cycle manner, the generated interface only transfers the valid output data. This is achieved by leveraging the observation that most FPGA applications use standard interfaces (e.g., AXI interface [47]) to fetch input data from peripherals (e.g., DRAM) and these interfaces contain the data valid signal. The custom tool then generates necessary logic to propagate this valid signal into the latency-insensitive interface (DFFs are included for timing correctness), so the interface only buffers the valid input data, as illustrated in Fig. 9. The input data signals that share the same valid signal are combined and buffered by the same FIFO to minimize the number of required FIFOs. Users can provide the description of the custom interface used in the applications to utilize **HETERO-ViTAL**’s compilation flow.

This latency-insensitive interface also needs to halt the execution of user logic when the corresponding input FIFO is empty or the output FIFO is full. The key is to keep the internal states of user logic unmodified when the execution is halted, such as the on-chip memory, result registers in an accumulator and the state registers in FSMs. The custom tool identifies the logic primitives that stores the internal states, e.g., DFFs with a feedback loop, and routes the corresponding valid signal to their clock enable port (Fig. 9). When execution is halted, the states of these elements are not modified because of the disabled clock. The custom tool also routes the valid signal to the write enable port of on-chip memories to guarantee that the content of these memories is not modified.

**Step 3: Mapping Stage II.** This step has two sub-steps to map the user logic and the latency-insensitive interface in one **HL** virtual block into an array of **LL** virtual blocks. Instead of first partitioning one **HL** virtual block and then mapping each partition into one **LL** virtual block, we choose to reuse the commercial P&R tool to monolithically map one **HL** virtual block onto the physical FPGA and then split the mapping results to generate the mapping of each **LL** virtual block. The size of the pre-defined region is determined by the size of the **HL** virtual block. This flow has two benefits compared with the alternative flow: (1) the mapping of all **LL** virtual blocks are jointly optimized in the monolithic mapping process, and (2) the highly-optimized commercial FPGA P&R tool ensures the mapping quality of each **LL** virtual block. This step is performed for all types of FPGAs and generates multiple mapping results for one application. The mapping process for all FPGAs can be fully paralleled to minimize the compilation time. Moreover, since the monolithic mapping process is similar as the P&R process in conventional FPGA compilation flow, the techniques proposed in prior works [31][72][73] that improves the mapping quality and reduces the compilation time could also be applied on this step.

The first sub-step uses a custom tool to estimate the number of **LL** virtual blocks required by one **HL** virtual block. It then generates the Vivado constraint file to allocate a specific region in the physical FPGA based on the estimation result. The commercial FPGA P&R tool is used to map the input **HL** virtual block into the defined region using the out-of-context mapping flow [76]. The second sub-step uses a custom tool that leverages the APIs from RapidWright project to split the monolithic mapping result, such as Cell.getSite().getName() and Design.createAndPlaceCell(). This process has a low timing complexity as it only needs to read the location of each placed logic primitive (e.g., LUTs) and assign it into the corresponding **LL** virtual block. Compared with the monolithic mapping process, the runtime of this step is negligible (< 60s).

**Step 4: Relocation.** This step uses a custom tool to relocate one mapped **LL** virtual block into other feasible physical
blocks without recompilation. This tool leverages the APIs from the RapidWright project (e.g., `Module.setAnchor()` and `ModuleInst.place()`) and can be finished in just few minutes.

**Step 5: Global P&R.** This step reuses the commercial FPGA tools to integrate the individually mapped components into a complete design and generate the partial reconfigurable bitstreams to support dynamic runtime management. This process is not supported by the Vivado GUI and we develop a Tcl script to automate it.

![Diagram](image)

**Recursive Partition Process:** We apply a recursive partition method to map one application into HL virtual blocks. As illustrated in Fig. 10, one application is first mapped into a single HL virtual block, which is then partitioned into two HL virtual blocks using the min-cut algorithm. This process is recursively performed $N$ times and totally generates $\sum_{i=1}^{N} 2^i = 2^{N+1} - 1$ HL virtual blocks. All these HL virtual blocks are mapped onto LL virtual blocks to support various runtime deployments, such as deploying the application into a single FPGA or up to $2^N$ FPGAs. Each round can also generate $K$ different partition results (Fig. 10) to further increase the runtime deployment flexibility. Overall, the number of HL virtual blocks generated by this step is

\[ \#\text{Blocks} = 1 + K \sum_{i=1}^{N} 2^i = 2K(2^N - 1) + 1 = O(K2^N) \]

We need to judiciously determine the value of parameters $N$ and $K$ to balance the compilation cost and the runtime deployment flexibility. For the parameter $K$, one application can have various mapping results with a large $K$, so that the runtime system can always find the appropriate one to deploy this application no matter what the actual resource availability is. From a system view, if the system has a large number of applications that need to be deployed, even each application only has one mapping result, the runtime system can always find the application that can be deployed no matter what the actual resource availability is. Thus, we choose $K = 1$ in the context of the cloud environment, which already achieves a high aggregated system performance (Section IV-D). The impact on the performance of one application can also be effectively mitigated by the scheduling policy. For parameter $N$, the mapping result that can deploy one application into multiple FPGAs ($N \geq 1$) is needed to alleviate the external resource fragmentation issue caused by the boundary of physical FPGAs. Moreover, the value of parameter $N$ is also related to the size of the application, where a large application needs a large $N$ to generate HL virtual blocks that are small enough. Based on our design space exploration (Section IV-D), for applications that can fit into one FPGA device (the majority of existing FPGA applications), $N = 1$ is sufficient to achieve a high aggregated system performance.

**C. System Layer**

The system layer performs task scheduling and runtime resource allocation to deploy compiled applications (Section III-B) onto physical FPGAs. It exploits the characteristics of various cloud instances and workloads to improve the aggregated system performance through an efficient FPGA sharing. It also provides a heuristic-based resource allocation policy to alleviate the resource fragmentation issue.

![Diagram](image)

As shown in Fig. 11, the system controller maintains two task queues to schedule on-demand and spot instances separately. Specifically, on-demand instances are scheduled in a first-come first-served (FCFS) manner to guarantee fairness, while spot instances are scheduled whenever the FPGA cluster have sufficient resources to improve the aggregated system performance by exploring the opportunity of task backfilling [25]. When the FPGA cluster does not have sufficient resources for a newly arrived on-demand instance, deployed spot instances will be interrupted and evacuated from the cluster one by one based on the deployment sequence until the cluster has sufficient resources. The evacuated spot instances are placed at the end of the corresponding task queue, thus, spot instances are backfilled in a round-robin manner to ensure fairness. It is possible that the newly arrived on-demand instance cannot be deployed after evacuating all running spot instances. Then the system controller will try to deploy it again when one running on-demand instance is terminated. This scheduling policy is effective in the cloud environment, which has insufficient runtime information to support a more sophisticated policy. For instance, it is impossible to obtain/estimate the completion time of tasks, as tasks can be terminated by users at anytime under the pay-as-you-go pricing mechanism.

The system controller then allocates FPGA resources for one scheduled instance. It searches the resource database to collect all possible resource allocations, i.e., different combinations of HL virtual blocks and different mapping results for each HL virtual block. For streaming workloads that are sensitive to the inter-FPGA communication latency, the resource allocations are filtered by comparing the number of allocated FPGAs with a given threshold ($T_{stream}$). For batch workloads that are sensitive to the inter-FPGA communication bandwidth, the resource allocations are filtered by comparing its bandwidth...
score (Equation 1) with a given threshold \((T_{\text{batch}})\). As one HL virtual block (an array of LL virtual blocks) needs to be deployed into contiguous physical blocks, it is necessary to minimize the resource fragmentation. Thus, a fragmentation score (Fig. 12) is calculated for the remaining resource allocations and the one with the highest score is selected for deploy the application. The threshold \(T_{\text{stream}}\) and \(T_{\text{batch}}\) are used to control the degree of the potential performance interference between deployed applications. For instance, for small streaming workloads that can fit into one FPGA, \(T_{\text{stream}}\) can be set to 1 to avoid performance degradation due to the long inter-FPGA latency. \(T_{\text{batch}}\) can be set to 1 to avoid throughput degradation for batch workloads. Users can adjust these thresholds to explore the tradeoff between performance and cost.

\[
\text{Fragmentation Score} = \text{Geomean} \left( \frac{\text{#contiguous physical blocks}}{\text{FPGA capacity}} \right)
\]

![Fig. 12. A conceptual diagram illustrates the calculation of fragmentation score.](image)

\[
\text{Bandwidth Score} = \frac{\text{Required Bandwidth}_{ij}}{\text{Provided Bandwidth}_{ij}}
\]

**IV. EVALUATION**

In this section, we evaluate HETERO-ViTAL on a custom FPGA cluster with two types of FPGAs, Xilinx XCVU37P and XCKU115. We use three sets of benchmarks to comprehensively evaluate each layer in the HETERO-ViTAL stack.

A. Experimental Setup

1) Benchmark Selection

We apply three benchmark sets with varying size and complexity to evaluate the three HETERO-ViTAL layers.

The first set comprises a small benchmark that is synthetically generated to evaluate the implementation of the two-level system abstraction (Architecture Layer, Section III-C). It generates random data traffic to identify the maximum bandwidth provided by the inter-FPGA communication interface, which is used to calculate the bandwidth score (Equation 1) for the runtime resource allocation.

The second set contains benchmarks from the Rosetta benchmark suite [84] to evaluate the offline compilation performance (Compilation Layer, Section III-B). These benchmarks are highly optimized HLS-based FPGA designs from machine learning and image/video processing domains. We adjust the design parameters to provide three variants of accelerator designs (small, medium and large) for each benchmark to better account for the varying performance/cost demands in the dynamic cloud environment (Table 1).

The third set is the most complex one among the three. It comprises multiple applications that can concurrently run on the FPGA cluster and is used to evaluate the quality of the runtime scheduler and resource allocation policy. As there is no publicly available real-world cloud workloads using FPGAs, we follow the widely used approach [55] to synthetically generate several workload sets. Each workload set contains a sequence of workloads from the second benchmark set. The requests for deploying these workloads are issued with a random time interval to emulate the dynamic cloud environment. The ratio between the number of on-demand and spot instances, the ratio between the number of batch and streaming workloads, and the termination time of each workload are also randomly generated.

2) Platform Configuration

We implement HETERO-ViTAL on a custom-built FPGA cluster that has three Xilinx Virtex UltraScale+ FPGAs (XCVU37P) and one Xilinx Kintex UltraScale FPGA (XCKU115). More benefits can obtained from HETERO-ViTAL when the cluster comprises more types of FPGAs. More comprehensive evaluation on this will be our future work. These four FPGAs are attached to the host machine through PCIe, and a secondary bidirectional ring network is deployed to connect these FPGAs. Specifically, Xilinx XCVU37P is a large and latest FPGA device fabricated in the 14/16nm technology node. One FPGA board provides four \(1 \times 4\) ganged 28Gb/s QSFP+ cages for 100Gb Ethernet connection. Two DIMM sites are provided and each support up to 128GB DDR4×72 with ECC. XCKU115 is a relatively small and old FPGA device fabricated in the 20nm technology node. One FPGA board provides two QSFP28 cages for 40Gb/s Ethernet connection. It also provides 12GB DDR4 memory with ECC and 4GB DDR4 memory without ECC.

3) Baseline

The resource management method that allocates FPGA resource at a per-device granularity is used as the non-virtualized

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LUTs</th>
<th>Resource Usage</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering</td>
<td>187k</td>
<td>12.4k</td>
<td>91</td>
</tr>
<tr>
<td>Digit</td>
<td>25.5k</td>
<td>11.3k</td>
<td>0</td>
</tr>
<tr>
<td>Recognition</td>
<td>75k</td>
<td>36.2k</td>
<td>0</td>
</tr>
<tr>
<td>Spam</td>
<td>14.5k</td>
<td>10.7k</td>
<td>896</td>
</tr>
<tr>
<td>Filtering</td>
<td>36.2k</td>
<td>26.9k</td>
<td>2240</td>
</tr>
<tr>
<td>Optical</td>
<td>57.9k</td>
<td>42.7k</td>
<td>3584</td>
</tr>
<tr>
<td>Flow</td>
<td>84.2k</td>
<td>91.0k</td>
<td>372</td>
</tr>
<tr>
<td>Face</td>
<td>141k</td>
<td>153k</td>
<td>620</td>
</tr>
<tr>
<td>Detection</td>
<td>253k</td>
<td>276k</td>
<td>1116</td>
</tr>
</tbody>
</table>

**TABLE I**

THE RESOURCE USAGES OF EVALUATED BENCHMARKS.
baseline. This management method has been widely used in existing FPGA clouds. The recent work, ViTAL [80], is also included as one baseline to compare the performance of virtualizing a homogeneous FPGA cluster (only deploy applications onto XCVU37P). Vivado HLS 2019.2 is used to convert the benchmarks written in C into Verilog RTL. Vivado 2019.2 is then applied to map the Verilog RTL codes for the non-virtualized baseline. The corresponding compilation flows are applied for ViTAL and HETERO-VITAL, respectively.

![Diagram of two commercial FPGAs](image)

**Fig. 13.** Two commercial FPGAs are partitioned into regions to support the proposed system abstraction. The mapping results are obtained from Vivado 2019.2.

### B. Architecture Layer Evaluation

As discussed in Section III-A, we partition one physical FPGA into two regions to support the proposed system abstraction. We use the method proposed in ViTAL [80] to identify the optimal partition the two FPGAs, as shown in Fig. 13. Region 2 is the service region that contains the standard IP cores to share the interface of DRAM (Region 2b) and Ethernet (Region 2a) among the physical blocks (Region 1). These interfaces are time-multiplexed among physical blocks in a round-robin manner to ensure fairness. The amount of resources provided by one physical block is presented in Table II, which is 20% more than that provided by the physical block in ViTAL due to the elimination of communication region. Removing the communication region also increases the number of physical blocks by 20 ~ 25%. We then apply the first benchmark set to evaluate the inter-FPGA communication interface. The maximum bandwidth provided by this interface is 90Gb/s. This is slightly lower than that in ViTAL, mainly because all physical blocks in HETERO-VITAL can access the inter-FPGA interconnection network (support more flexible runtime management) and additional bits in the packet are required to index the physical blocks. The maximum inter-die communication bandwidth is 3.7Tb/s on XCVU37P FPGA and 2.1Tb/s on XCKU115 FPGA.

<table>
<thead>
<tr>
<th>LUTs</th>
<th>DFFs</th>
<th>DSPs</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCVU37P</td>
<td>93.6k</td>
<td>18.7k</td>
<td>696</td>
</tr>
<tr>
<td>XCKU115</td>
<td>53.3k</td>
<td>106.6k</td>
<td>528</td>
</tr>
</tbody>
</table>

**Table II** RESOURCES PROVIDED BY ONE PHYSICAL BLOCK.

### C. Compilation Layer Evaluation

The second benchmark set is used to evaluate the compilation layer. We first report the breakdown of the compilation time, which is obtained on a machine with Intel Xeon Gold 6244 CPU@3.6GHz (using 16 cores). As shown in Fig. 14, the compilation time is dominated by the place&route stage (94%), while the runtime of custom tools only takes < 0.5% of the total time. The main reason is that the custom tools and synthesis stage only run once for one application, while the P&R stage needs to run for all generated HL virtual blocks. Compared with the non-virtualized baseline, the compilation time of HETERO-VITAL increases about 5.9×. When virtualizing a homogeneous cluster, the compilation time of HETERO-VITAL is 3.3× longer than that of ViTAL. This is mainly because HETERO-VITAL generates multiple mapping results for one application to provide the optimal mapping for different resource allocations, while ViTAL and the baseline only generate one mapping result that is used for all possible resource allocations. In addition, HETERO-VITAL also supports the heterogeneous FPGA cluster. Thus, it might not be fair to compare the runtime of these different compilation flows. Moreover, all these compilation flows are performed offline (not on the critical path) and do not degrade the runtime system performance.

We then evaluate the quality of HETERO-VITAL’s compilation results. We first observe that by fully utilizing the intra-FPGA interconnection network (both intra-die and inter-die networks), HETERO-VITAL generates much more compact mappings and reduces the number of allocated physical blocks by up to 2.0× compared with ViTAL (Table III). The resource utilization is also improved by 10% and 12% on average for XCVU37P and XCKU115, respectively. We then confirm the effectiveness of the new latency-insensitive communication interface, which can reduce the required bandwidth by more than 12× due to the elimination of redundant data traffic.

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>XCVU37P</th>
<th>XCKU115</th>
<th>XCVU37P</th>
<th>XCKU115</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering</td>
<td>3</td>
<td>2</td>
<td>2 (1.50×)</td>
<td>2 (1.00×)</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>6</td>
<td>9 (1.42×)</td>
<td>7 (1.14×)</td>
</tr>
<tr>
<td>Digit</td>
<td>2</td>
<td>2</td>
<td>2 (1.00×)</td>
<td>1 (1.00×)</td>
</tr>
<tr>
<td>Recognition</td>
<td>5</td>
<td>4</td>
<td>4 (1.25×)</td>
<td>3 (1.33×)</td>
</tr>
<tr>
<td>Spam</td>
<td>2</td>
<td>3</td>
<td>2 (1.00×)</td>
<td>2 (1.50×)</td>
</tr>
<tr>
<td>Filtering</td>
<td>5</td>
<td>6</td>
<td>4 (1.25×)</td>
<td>5 (1.20×)</td>
</tr>
<tr>
<td>Optical</td>
<td>3</td>
<td>3</td>
<td>2 (1.50×)</td>
<td>2 (1.50×)</td>
</tr>
<tr>
<td>Flow</td>
<td>5</td>
<td>5</td>
<td>3 (1.67×)</td>
<td>4 (1.25×)</td>
</tr>
<tr>
<td>Face</td>
<td>2</td>
<td>3</td>
<td>2 (1.00×)</td>
<td>3 (1.00×)</td>
</tr>
<tr>
<td>Detection</td>
<td>7</td>
<td>12</td>
<td>7 (1.00×)</td>
<td>10 (1.20×)</td>
</tr>
</tbody>
</table>

**Table III** THE NUMBER OF ALLOCATED PHYSICAL BLOCKS. APPLICATIONS ARE MAPPED INTO ONE HL VIRTUAL BLOCK IN HETERO-VITAL (N = 0, K = 1).
D. System Layer Evaluation

The third benchmark set is applied to evaluate the runtime performance. The response time (wait time and execution time) is used as the performance metric, which is normalized to that of the baseline. We first perform a design space exploration to identify the optimal value for parameter \( N \) and \( K \) used in the compilation flow (Section III-B). As shown in Fig. 15a, with a fixed parameter \( K \), increasing parameter \( N \) from 0 (applications can only be mapped into a single FPGA) to 1 (applications can be mapped into up to two FPGAs) effectively reduces the response time by 38% at a high busy degree\(^2\), while further increasing it to 2 (applications can be mapped into up to four FPGAs) leads to a smaller reduction (18%). This is because \( N = 0 \) cannot enable FPGA sharing across physical FPGA boundary, degrading the runtime performance due to the resource fragmentation. Since \( N = 1 \) already enables a fine-grained FPGA sharing for the evaluated benchmarks, larger \( N \) only leads to a marginal improvement. We also find that the parameter \( K \) has a marginal impact < 4% on the response time. Thus, we use parameter \( N = 1 \) and \( K = 1 \) in the following evaluation.

We then compare the runtime performance of HETERO-VITAL with the non-virtualized baseline and ViTAL. Overall, HETERO-VITAL reduces the average response time by 79.2% compared with the non-virtualized baseline on a heterogeneous FPGA cluster (Fig. 16a). It also reduces the response time by 42.0% on average compared with ViTAL on a homogeneous FPGA cluster (only deploy applications onto XCVU37P FPGAs), as shown in Fig. 16b. This improvement mainly comes from the better runtime management policy (Section III-C) and the better mapping quality (Section IV-C). The elimination of the redundant traffic contributes to about 8% of the response time reduction. In particular, HETERO-VITAL reduces the average response time by up to 98.8% for the performance-driven on-demand instances even under a heavy load. This is achieved by applying different scheduling strategies for on-demand and spot instances. The elimination of the redundant data traffic has a negligible impact on the demand instance, but it contributes to about 12% of the response time reduction for the spot instance. We observe that by considering the characteristics of applications (batch or stream processing), the proposed heuristic-based runtime policy only introduces a negligible overhead in applications’ execution time (< 0.6%). We also confirm that the runtime policy can support the dynamic cloud environment and provide a stable performance under different composition of cloud instances and workloads (Fig. 16c,d). The resource utilization of HETERO-VITAL is higher than 96% under all scenarios.

![Fig. 15. The average response time for on-demand and spot instances under different \( N \) and \( K \), which is normalized to that of the baseline. The percentage of on-demand instances and batch workloads is 50%.](image)

![Fig. 16. The comparison of the average response time under various conditions, which is normalized to that of the non-virtualized baseline. (a), (c), (d) are the results on a heterogeneous FPGA cluster, while (b) is the result on a homogeneous FPGA cluster. The percentage of on-demand instances is 50% in (a–c). The percentage of batch workloads is 50% in (a), (c) and (d). Left column is the result of on-demand instance, while the right column is that of the spot instance.](image)

V. RELATED WORK

A. FPGA Virtualization

1) Virtualization for IaaS Model

Several works [56][15][42][63][6][67] propose to abstract the I/O interface through an application-independent shell. This shell contains essential building blocks for data marshaling, host-to-FPGA communication and inter-FPGA communication (if available). Several other works [19][1][7][52] provide a more sophisticated virtualization support for memory. Besides the off-chip DRAM, these works also provide an abstraction for the on-chip memory to reduce the programming complexity. Compared with HETERO-VITAL, the

---

\(^2\)Busy degree is defined as the average ratio between execution time and the time interval between two requests.
virtualization support provided by this type of works is limited as other reconfiguration resources such as logic cells are not virtualized. These works also require users to perform resource management and runtime scheduling (if needed).

2) Virtualization for PaaS Model

We broadly category virtualization methods for the PaaS model into two groups: time-multiplexing and space-multiplexing. Time-multiplexing methods [23][16][65][10][40][64][49] share FPGA resources in the temporal domain. These works typically use multi-context FPGAs to reduce the configuration overhead. Different from the commercial FPGAs that is a single-context architecture and can only store one context of configuration, multi-context FPGAs contain multiple sets of configuration memories to store several contexts of configurations. When one context of configuration is used for computation, a new context can be loaded to hide the configuration overhead. However, the additional configuration memories significantly increases the implementation cost, thus, multi-context FPGAs have not been commercialized.

Space-multiplexing methods partition FGAs into fixed-size blocks that are connected by a network and can be further categorized into two types based on the implementation of the network. The first type of methods [17][14][82][45][9][12][24][44] (including the low-latency mode of AmorphOS [41]) use an all-to-all network to connect all blocks so that one application can be deployed into discontiguous blocks to enable a flexible runtime deployment. This type of methods faces a dilemma when determining the block size, i.e., a large block size leads to internal resource fragmentation, while a small block size (thus more blocks) exponentially increases the network complexity. The other type of method [80][43][66] use a simple network to only connect adjacent blocks so that one application can only be deployed onto contiguous blocks. Although this slightly increases the complexity of the resource allocation, it largely reduces the overhead of the network. Compared with the first type of methods, this type of methods can have a much smaller block size to alleviate the resource fragmentation issue and improves the aggregated system performance.

Different from previous space-multiplexing methods, HETERO-VITAL is a hybrid method, where the top-level abstraction uses an all-to-all network to improve the runtime management flexibility, while the bottom-level abstraction only connects the adjacent blocks to alleviate the resource fragmentation issue. With this hybrid method, HETERO-VITAL can efficiently virtualize both homogeneous and heterogeneous FPGA cluster, while most previous works only support a homogeneous FPGA cluster.

3) Virtualization for SaaS Model

Several works [27][34][79] abstracts a set of application-specific accelerators into pre-defined APIs to decouple software application from FPGA design. Thus, users can construct their own applications using a software programming flow with a substantially reduced programming complexity, and FPGA experts can focus on optimizing the performance of FPGA designs. A multi-layer virtualization framework is recently proposed [81] for the SaaS model that provides an efficient method to combine the pre-defined API based method and the above space-multiplexing method, thereby getting the best of both worlds. This multi-layer framework is orthogonal to this work, as it can utilize HETERO-VITAL as the underlying abstraction.

4) Overlay Architecture

FPGA overlay architectures [11][46][70][21][13][50] enable code portability across different types of FGAs by providing a single-level abstraction, which cannot decouple the conflicting requirements in virtualizing a heterogeneous FPGA cluster, leading to a non-negligible degradation in the mapping quality.

B. OS Support for FGAs

Many prior works [59][51][6][38][60][82][61][68][18] have explored the OS support for FGAs and provide valuable experiences on FPGA virtualization. These works typically abstract the tasks running on the FPGA devices as hardware threads. The interface provided for these hardware threads is the same as that for the software threads running on CPUs. Thus, the OS can manage/schedule these hardware threads in a similar way as that for the software threads. These hardware threads can also communicate with OS to access OS-managed resources in a similar way as other software threads. These works typically share the FPGA resources among hardware threads in the temporal domain [32][51] and/or in the spatial domain using the first type of methods with an all-to-all network [82][61], thereby having the limitations discussed in Section V-A2.

LEAP [26] is somewhat similar to HETERO-VITAL as it also leverages the latency-insensitive communication interface to enable scale-out acceleration. Nevertheless, different from HETERO-VITAL, LEAP is mainly developed for reducing the programming complexity of FGAs. Its system abstraction does not support a dynamic resource sharing in the multi-user cloud environment.

VI. Conclusion

We present HETERO-VITAL in this paper that provides a new two-level system abstraction, a compilation framework with a two-stage mapping process and a runtime management system with heuristic-based policy to efficiently virtualize both homogeneous and heterogeneous FPGA clusters. HETERO-VITAL effectively decouples the conflicting requirements of runtime management and offline compilation. We evaluate HETERO-VITAL on a custom-built FPGA cluster using a set of representative benchmarks. The results show that HETERO-VITAL reduces the average response time (a critical QoS metric) by 79.2% compared with non-virtualized baseline for a heterogeneous cluster and reduces it by 42.0% over ViTAL on a homogeneous cluster due to a better system design.

Acknowledgements

We thank anonymous reviewers for their insightful feedback to help improve the quality of the paper. We also would like to thank Jialiang Zhang for the system prototyping.
REFERENCES


[68] G. Wasi, M. E. A. Benkhillifa, G. Lawday, F. Verdier, and S. Garcia, “Multi-Shape Tasks Scheduling for Online Multitasking on FP-