When Application-Specific ISA Meets FPGAs: A Multi-layer Virtualization Framework for Heterogeneous Cloud FPGAs

Yue Zha
University of Pennsylvania
Philadelphia, PA, USA
zhayue@seas.upenn.edu

Jing Li
University of Pennsylvania
Philadelphia, PA, USA
janeli@seas.upenn.edu

ABSTRACT

While field-programmable gate arrays (FPGAs) have been widely deployed into cloud platforms, the high programming complexity and the inability to manage FPGA resources in an elastic/scalable manner largely limits the adoption of FPGA acceleration. Existing FPGA virtualization mechanisms partially address these limitations. *Application-specific (AS)* ISA provides a nice abstraction to enable a simple software programming flow that makes FPGA acceleration accessible by the mainstream software application developers. Nevertheless, existing *AS* ISA-based approaches can only manage FPGA resources at a per-device granularity, leading to a low resource utilization. Alternatively, *hardware-specific (HS)* abstraction improves the resource utilization by spatially sharing one FPGA among multiple applications. But it cannot reduce the programming complexity due to the lack of a high-level programming model.

In this paper, we propose a virtualization mechanism for heterogeneous cloud FPGAs that combines *AS* ISA and *HS* abstraction to fully address aforementioned limitations. To efficiently combine these two abstractions, we provide a multi-layer virtualization framework with a new system abstraction as an indirection layer between them. This indirection layer hides the FPGA-specific resource constraints and leverages parallel pattern to effectively reduce the mapping complexity. It simplifies the mapping process into two steps, where the first step decomposes an *AS* ISA-based accelerator under no resource constraint to extract all fine-grained parallel patterns, and the second step leverages the extracted parallel patterns to simplify the process of mapping the decomposed accelerators onto the underlying *HS* abstraction. While system designers might be able to manually perform these steps for small accelerator designs, we develop a set of custom tools to automate this process and achieve a high mapping quality. By hiding FPGA-specific resource constraints, the proposed system abstraction provides a homogeneous view for the heterogeneous cloud FPGAs to simplify the runtime resource management. The extracted parallel patterns could also be leveraged by the runtime system to improve the performance of scale-out acceleration by maximally hiding the inter-FPGA communication latency.

We use an *AS* ISA similar to the one proposed in BrainWave project and a recently proposed *HS* abstraction as a case study to demonstrate the effectiveness of the proposed virtualization framework. The performance is evaluated on a custom-built FPGA cluster with heterogeneous FPGA resources. Compared with the baseline system that only uses *AS* ISA, the proposed framework effectively combines these two abstractions and improves the aggregated system throughput by 2.54× with a marginal virtualization overhead.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; • **Security and privacy** → **Virtualization and security**.

KEYWORDS

Heterogeneous cloud FPGAs, Virtualization, Application-specific ISA, Parallel patterns

ACM Reference Format:
Yue Zha and Jing Li. 2021. When Application-Specific ISA Meets FPGAs: A Multi-layer Virtualization Framework for Heterogeneous Cloud FPGAs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3445814.3446699

1 INTRODUCTION

In the last decade, field-programmable gate arrays (FPGAs) have been widely deployed into several private and public clouds to enhance their computing capability (e.g., Amazon F1 [1], Microsoft Azure [34], InAccel [22] and Alibaba Cloud F3 [13]). While delivering compelling performance for diverse applications [11, 17, 30, 54], the high programming complexity and the inability to manage cloud FPGA resources in an elastic and scalable manner largely limits the adoption of FPGA acceleration. The primary culprit is the lack of a complete virtualization support. *Application-specific (AS)* ISA and *hardware-specific (HS)* abstraction are two promising virtualization mechanisms that have been used in many prior works for cloud FPGA virtualization [9, 18, 21, 25, 27, 53]. However, these two mechanisms only partially address the aforementioned limitations.

AS ISA has received increasing attention in both industry [18, 38, 43, 44] and academia [21, 24, 32, 47] as it substantially reduces the programming complexity of cloud FPGAs. It inherits the advantages of general-purpose ISA (e.g., x86, PowerPC, RISC-V and ARM) and decouples the software development from the underlying FPGA hardware designs (Fig. 1a), thereby providing a simple software programming flow and making FPGA acceleration accessible by the mainstream application developers. Different from the general-purpose ISA, *AS* ISA fully exploits the customization opportunities

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446699

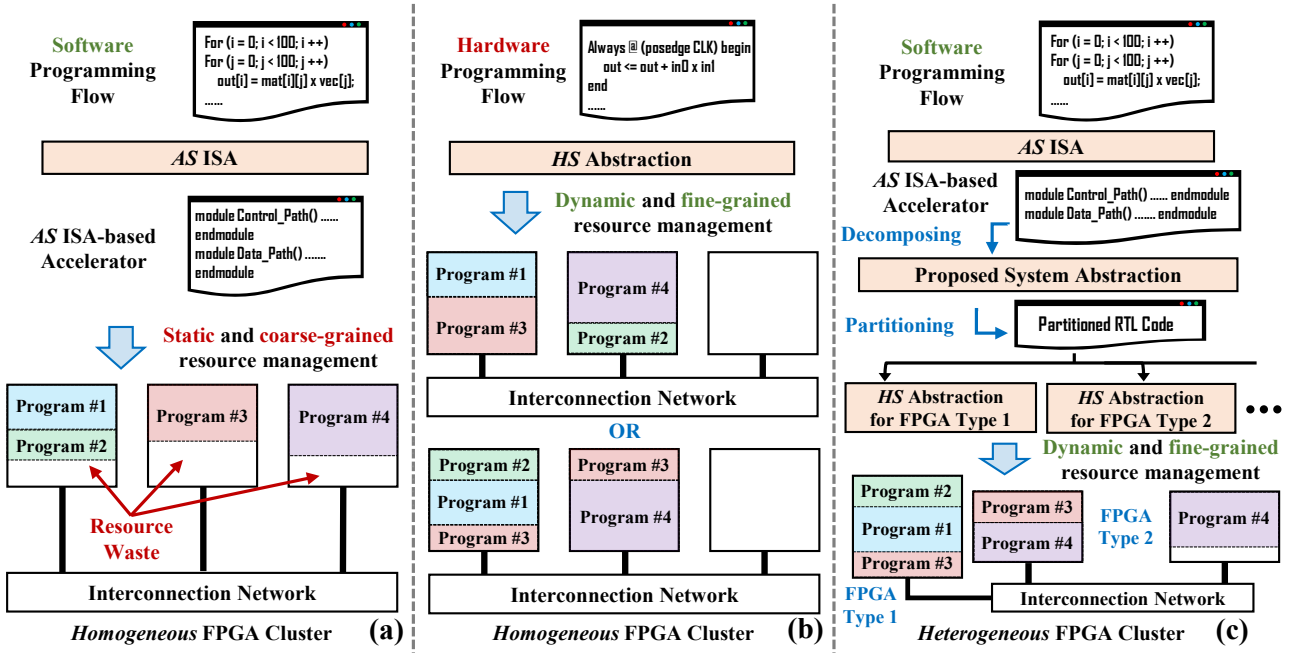

Figure 1: (a) *AS* ISA provides a software programming flow to reduce the programming complexity, but it cannot enable an efficient resource management and lacks scale-out acceleration support as it is not originally designed for the spatial hardware. (b) *HS* abstraction improves the management efficiency by enabling a dynamic and fine-grained FPGA sharing, but it has a high programming complexity due to the lack of a high-level programming model. (c) In this paper, we propose to virtualize heterogeneous cloud FPGAs by combining *AS* ISA and *HS* abstraction. We provide a multi-layer virtualization framework that contains a new system abstraction as an indirection layer to bridge the gap between *AS* ISA and *HS* abstraction.

from the application itself and provides a customized instruction set to reduce the storage/control overhead by generating more compact code. However, as ISA is not originally designed for the *spatial* hardware, existing *AS* ISA-based virtualization solutions only allow to manage the pool of FPGA resources at a per-device granularity by allocating one FPGA device to one or multiple *AS* ISA-based accelerators. This management method faces a *NP-hard* bin packing problem due to the diversity of both *AS* ISA-based accelerators and FPGAs. To avoid the time-consuming recompilation, the resource allocation is *statically* determined at the offline compilation time, resulting in a low elasticity. Moreover, *AS* ISA itself lacks native support for scale-out acceleration. Consequently, *AS* ISA-based accelerators need to be *manually* partitioned across the physical FPGA boundary to exploit scale-out acceleration, which could be a time-consuming and error-prone process.

Alternatively, various *HS* abstractions [2, 6, 9, 25–27, 48, 53] have been proposed to spatially share one FPGA device among multiple accelerator designs at a sub-FPGA granularity (Fig. 1b), thereby dramatically improving the runtime management efficiency. Moreover, some *HS* abstractions [53] also provide essential system support so that one accelerator design can be dynamically deployed into multiple identical FPGA devices at runtime without time-consuming recompilation to exploit scale-out acceleration. However, these *HS* abstractions lack a high-level programming model and require a hardware programming flow (e.g., Verilog), leading to a high programming complexity. Additionally, existing *HS* abstractions typically target *homogeneous* FPGA clusters and it is not trivial to extend them to virtualizing *heterogeneous* FPGA clusters due to the high resource management complexity. Specifically, since FPGA hardware designs describe physical circuits that are directly mapped onto the FPGA architecture under tight spatial resource constraints, most *HS* abstractions are designed to expose these *FPGA-specific* spatial resource constraints to the compilation framework, so that the compiler can leverage this information to perform optimization and improve mapping quality. Consequently, these *HS* abstractions cannot provide a homogeneous view for the heterogeneous FPGA cluster, which substantially increases the resource management complexity.

In this paper, we propose to virtualize heterogeneous cloud FPGAs by combining *AS* ISA and *HS* abstraction to get the best of both worlds. To efficiently combine these two abstractions, we provide a multi-layer virtualization framework that comprises a new system abstraction, a set of custom tools for application mapping, and a runtime management system. The core of this framework is the new system abstraction that serves as the indirection layer between *AS* ISA and *HS* abstraction (Fig. 1c). It comprises a pool of soft blocks that adopt a multi-level tree structure to effectively represent the parallel patterns extracted from the ISA-based accelerators (Fig. 2b). While parallel pattern have been widely used in previous works to simplify parallel programming [3, 19, 36, 41, 42], it is leveraged in the proposed abstraction to expose the structural information in the *AS* ISA-based accelerators to the provided custom tools, thereby simplifying the mapping process. The spatial resource constraints (e.g., resource type/capacity) of one soft block can be arbitrarily chosen

to abstract away the FPGA-specific hardware details. Consequently, the proposed system abstraction provides a *homogeneous* view for the heterogeneous cloud FPGAs to substantially reduce the resource management complexity of the heterogeneous FPGA cluster. *HS* abstraction is still used for mapping *AS* ISA-based accelerators onto the physical FPGA architecture to ensure the compilation quality.

Without the proposed indirection layer, it is a non-trivial process to map *AS* ISA-based accelerators onto *HS* abstractions as it is hard to find the optimal mapping under the tight resource constraints. The proposed system abstraction simplifies the mapping process into two steps: decomposing step and partitioning step (Fig. 1c). In the first step, an *AS* ISA-based accelerator is decomposed into a group of soft blocks based on two primitive parallel patterns (data parallelism and pipeline parallelism). As soft blocks have no spatial resource constraint, the accelerator can be decomposed under no constraint to easily extract all fine-grained parallelism. In the second step, the decomposed accelerator is partitioned into several clusters of soft blocks to provide a multi-FPGA support. The extracted parallel patterns are leveraged to reduce the timing complexity of the partition process by pruning the search space. The partitioned accelerator is then mapped onto the underlying *HS* abstractions of all types of FPGA to enable a flexible runtime deployment (Fig. 1c). This mapping process reuses the compilation tool provided by the corresponding *HS* abstraction-based solution. While system designers might be able to manually perform these two steps for small accelerator designs, we provide a set of custom tools in the framework to automate this mapping process and reduce the burden on system designers.

The proposed virtualization framework includes a runtime management system that dynamically allocates physical FPGAs to deploy the soft blocks of decomposed *AS* ISA-based accelerators. It then send requests to the system controller provided by the specific *HS* abstraction-based solution to configure FPGA devices. We also incorporate an optimization technique into the runtime system to improve the performance when exploiting scale-out acceleration. This technique can maximally overlap computation and inter-FPGA communication by automatically 1) scaling down one large accelerator into multiple small accelerators based on the extracted parallel patterns, and 2) inserting necessary instructions into application code for inter-FPGA communication and synchronization. More details of this technique are presented in Section 2.3.

In particular, we made following major contributions:

- We identify the limitations of two representative FPGA virtualization methods: the *AS* ISA provides a simple software programming flow but cannot dynamically share one FPGA device in the spatial domain, while *HS* abstraction enables a fine-grained FPGA sharing but lacks a high-level programming model, leading to a high programming complexity. To address these limitations, we propose to combine these two abstractions to get the best of both worlds.
- We provide a multi-layer virtualization framework that includes a new system abstraction as an indirection layer to efficiently combine *AS* ISA and *HS* abstraction. This system abstraction hides *FPGA-specific* hardware details and leverages parallel patterns to effectively reduce the compilation and management complexity. We also provide a set of custom tools to automate the mapping process and a runtime system that can leverage the extracted parallel patterns to further improve the runtime system performance.
- We evaluate the effectiveness of the proposed framework on a custom-built heterogeneous FPGA cluster. We use an *AS* ISA similar to the one proposed in BrainWave project [18] and a recent *HS* abstraction [53] as a case study. Compared with the baseline system that only uses *AS* ISA, the proposed framework can improve the aggregated system throughput by 2.54× on average with a marginal virtualization overhead.

The rest of the paper is organized as follows. Section 2 presents the details of the proposed multi-layer virtualization framework, while Section 3 describes the case study used in the evaluation. Section 4 presents the evaluation results. Section 5 discusses the related work, followed by Section 6 to conclude the paper.

2 MULTI-LAYER VIRTUALIZATION FRAMEWORK

The core of the proposed virtualization framework is a new system abstraction (Section 2.1) that serves as an indirection layer between *AS* ISA and *HS* abstraction. *AS* ISA-based accelerators are first mapped onto this abstraction by decomposing them based on two primitive parallel patterns (Section 2.2.1) and are then partitioned and mapped onto the underlying *HS* abstraction (Section 2.2.2). The proposed system abstraction hides the FPGA-specific resource constraints to simplify the decomposing step, and the extracted parallel patterns are leveraged to simplify the partitioning step. This system abstraction provides a *homogeneous* view for *heterogeneous* cloud FPGAs to substantially reduce the runtime management complexity (Section 2.3). The extracted parallel patterns are also leveraged by the runtime system to improve the performance of scale-out acceleration using multiple FPGAs (Section 2.3).

2.1 System Abstraction

This system abstraction leverages parallel patterns to effectively close the gap between *AS* ISA and *HS* abstraction. As depicted in Fig. 2a, this abstraction comprises a pool of soft blocks and each soft block provides a latency-insensitive interface for the inter-block communication. In comparison to previous *HS* abstractions using a single-level structure [9, 27, 53], this system abstraction adopts a multi-level tree structure to efficiently represent the parallel patterns extracted from *AS* ISA-based accelerators. Specifically, a leaf soft block contains a basic module, where the basic module is defined as a Verilog module that does not instantiate other Verilog modules. A non-leaf soft block can have an arbitrary number of soft blocks as its child blocks, and these child blocks are connected following one of the two primitive parallel patterns, i.e., the data parallelism and pipeline parallelism (Fig. 2b). We choose these two parallel patterns since they are sufficient to construct other complex/nested parallel patterns [42] (e.g., the reduction pattern in Fig. 2c). Another difference between this abstraction and previous *HS* abstractions is that the spatial resource constraints of these soft blocks (e.g., resource type/capacity and interface to peripheral components) can be arbitrarily chosen so that 1) the decomposing step can be performed with no resource constraint to extract all

fine-grained parallelism (Section 2.2.1), and 2) it provides a *homogeneous* resource pool for the heterogeneous cloud FPGAs to simplify the runtime resource management (Section 2.3).

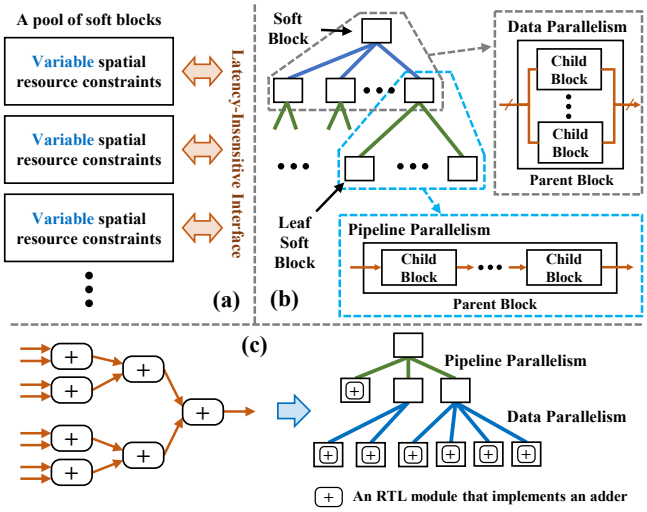

Figure 2: The proposed system abstraction (a) comprises a pool of soft blocks and (b) adopts a multi-level tree structure to effectively represent the two primitive parallel patterns, i.e., the data and pipeline parallelism. (c) These two primitive parallel patterns are sufficient to construct other complex patterns, such as the reduction pattern.

2.2 Mapping Process

A set of custom tools is developed to 1) decompose an *AS* ISA-based accelerator into a group of soft blocks based on two primitive parallel patterns (Section 2.2.1), and 2) partition the decomposed accelerator into several clusters of soft blocks based on extracted parallel patterns (Section 2.2.2). These clusters are separately mapped onto the underlying *HS* abstraction by reusing the compilation tool provided in the corresponding *HS* abstraction-based solution.

2.2.1 Decomposing Step. A given *AS* ISA-based accelerator is decomposed onto the proposed system abstraction by extracting all fine-grained parallel patterns. This decomposing process can be performed at different levels, such as the level of HLS or netlist. We choose to perform this decomposing process at the intermediate RTL level. This allows us to provide an extendable virtualization framework to support various high-level programming languages/frameworks [12, 14, 16, 33, 40, 49], as HLS designs can be converted into RTL designs. Moreover, RTL-designs are *FPGA-independent* while netlist could contain FPGA-specific primitives, such as BRAM/URAM in Xilinx FPGAs [52] and M20K in Intel FPGAs [23]. Since the extracted parallel patterns are also *FPGA-independent*, the decomposing results can be fully reused across different types of FPGAs. This minimizes the additional compilation time introduced by this step, which is negligible compared with the conventional FPGA compilation time (Section 4.3).

To decompose a monolithic *AS* ISA-based accelerator, we first split the control and data path at the top level of the design and map them into two separate soft blocks (Fig. 3a). This is feasible since *AS* ISA-based accelerators are FPGA-based soft processors with well-separated control and data path. Explicitly separating control and data path enables the optimization technique used for improving the runtime performance (Section 2.3). We then recursively decompose the soft block that contains the data path while keeping the soft block with control path unchanged. The soft block with data path can be decomposed either in a top-down flow or a bottom-up flow. In the top-down flow, one soft block is decomposed into multiple child blocks based on one of the two primitive parallel patterns (Fig. 3b). This decomposing process is recursively applied on the newly generated soft block until it contains a basic module (a Verilog module that does not instantiate other Verilog modules). Alternatively, in the bottom-up flow, we first extract all basic modules contained in the data path and assign each of them into a leaf soft block (Fig. 3c). We then identify a cluster of soft blocks that are connected in one of the two primitive patterns and create a parent soft block for them. This cluster is then replaced by the created parent block and this process is recursively performed until there is no soft block can be clustered.

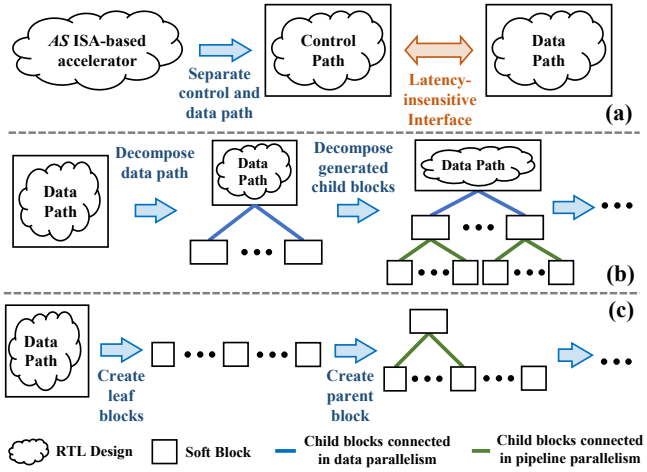

Figure 3: A conceptual diagram illustrates the decomposing flow, where (a) the control and data path in one accelerator design is first separated into two soft blocks, and the soft block that contains data path can be decomposed either in (b) a top-down flow or (c) a bottom-up flow.

While experienced system designs might be able to manually decompose small *AS* ISA-based accelerators based on aforementioned process by directly examining the source code, this decomposing process may become more difficult and time-consuming for large and complicated accelerators. Therefore, we develop a software tool to automate this decomposing process using the bottom-up flow due to the ease of implementation. Also, as it is hard to automatically identify the control path from the RTL source code, we need system designers' assistance to mark the control path by providing the corresponding RTL module name to the automation tool. We expect the required effort is relatively trivial as these modules can be easily identified at the top level. For HLS-generated RTL code that might not be human-readable, this marking process can be

performed at the level of HLS code. Specifically, system designers separate the HLS code for control and data path and synthesize them separately to obtain the RTL module name for control path. The decomposing tool has following five steps:

1. **Build block graph:** This step parses the input RTL design to extract all basic modules and then identifies the basic modules that belong to the data path. Each of these basic modules are assigned into one soft block. The inter-block connection is built based on the interconnection between the corresponding basic modules. If the input RTL design contains large basic modules, the primitives in these modules (e.g., logic gates and flip-flops) will be extracted and each of them will be assigned to one soft block.

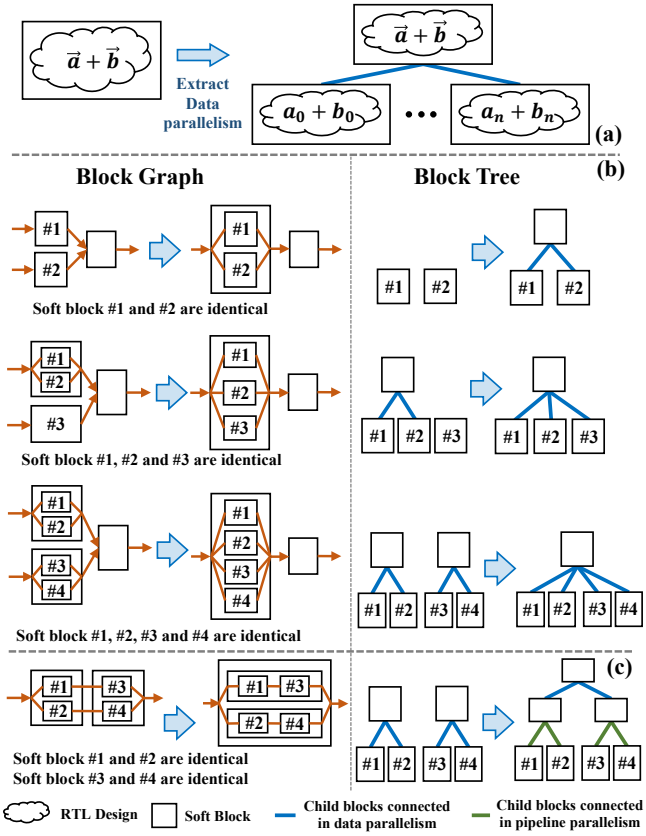

Figure 4: Conceptual diagrams illustrate (a) the step of extracting the data parallelism within a leaf soft block, (b) the step of identifying inter-block data parallelism, and (c) the step of identifying pipeline parallelism.

2. **Extract Intra-Block Data** Parallelism: This step is applied to extract the fine-grained data parallelism inside a soft block. The data parallelism can be identified by performing the equivalence checking on the logic within a soft block [20, 35, 46]. A group of child blocks will be created for one soft block if it has data parallelism (Fig. 4a).

3. **Identify Inter-Block Data Parallelism:** This step checks whether two input blocks of one soft block have data parallelism (Fig. 4b). Three cases are considered: 1) the two input blocks are identical, then a parent block will be created for these two soft blocks, 2) one input block has child blocks connected in data parallelism and the other input block is the same as the child block, then these soft blocks will be grouped into a single sub-tree, and 3) both input blocks have child blocks connected in data parallelism and these child blocks are identical, then these child blocks will be grouped into a single sub-tree. This step iterates through all soft blocks and terminates when no such pattern is identified.

4. **Identify Pipeline Parallelism:** The step checks whether the child blocks of two soft blocks are connected in pipeline parallelism. Specifically, if these two blocks both have child blocks that are connected in data parallelism and the number of child blocks are same, then these child blocks will be grouped into a two-level sub-tree, where the top level is data parallelism and the bottom level is pipeline parallelism (Fig. 4c). This step also iterates through all soft blocks and terminates when no such pattern is identified.

5. **Iteration:** Step 3 and 4 are repeated to identify all parallel patterns and terminate when no soft block can be merged.

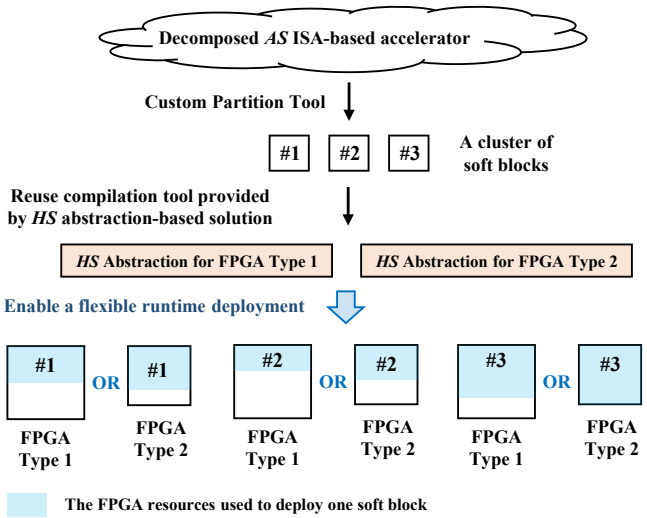

Figure 5: A conceptual diagram illustrates the process of mapping one decomposed accelerator onto underlying *HS* abstractions. In this example, the decomposed accelerator is partitioned into three soft blocks. Each of them is mapped onto the *HS* abstractions for two types of FPGAs.

2.2.2 Partitioning Step. This step enables an efficient multi-FPGA support for *heterogeneous* cloud FPGAs, which is not provided by existing *HS* abstractions [9, 25, 27, 53]. As illustrated in Fig 5, it uses a custom tool to partition a decomposed accelerator into several soft blocks. These soft blocks are then mapped onto the underlying *HS* abstraction by reusing the compilation tool provided by the corresponding *HS* abstraction-based solution. To support a flexible runtime deployment, one soft block is mapped onto the *HS* abstraction of all feasible FPGA devices that provide sufficient amount of resource and the required interfaces to peripherals.

This step generates different partitioning results to support a flexible runtime deployment (e.g., deploying one accelerator into different number of FPGAs). As illustrated in Fig. 6, it uses an iterative method to efficiently generate these results. The extracted parallel patterns are leveraged to reduce the timing complexity of

this partition process. Specifically, one soft block is partitioned into two clusters of soft blocks in one iteration. If its child blocks are connected in the pipeline parallelism, the custom tool will examine all inter-block connections and identify the one with the minimal communication bandwidth to divide these child blocks into two clusters. Alternatively, if the child blocks are connected in the data parallelism, these child blocks will be evenly grouped into two clusters. Two parent soft blocks are then created for these two clusters, which are the input for the next iteration. This iterative method partitions one accelerator into a cluster of soft blocks, which are the basic units for the runtime deployment (Section 2.3). For most accelerator designs that can fit into a single FPGA device, a small number of iterations (e.g., 1 or 2) is sufficient to achieve a flexible runtime management with a reasonable compilation cost.

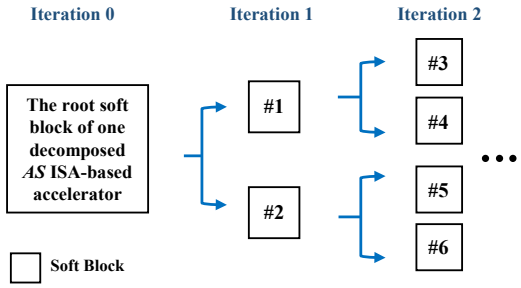

Figure 6: A conceptual diagram illustrates the iterative partition method. With a given number of iteration (N), the partition results can be used to deploy the accelerator into up to 2^N FPGA devices. For instance, soft block #2, #3 and #4 can be used to deploy the accelerator into 3 FPGA devices.

2.3 Runtime Management

The runtime management system performs resource allocation to deploy compiled *AS* ISA-based accelerators (Section 2.2) onto physical FPGAs. A runtime management policy is provided to maximize the aggregated system performance. It also provides an optimization technique to improve the performance of scale-out acceleration (Section 2.3).

As illustrated in Fig. 7, a system controller is provided to perform the resource management. It maintains a database to store the mapping results of *AS* ISA-based accelerators (a cluster of soft blocks). When the high-level system (e.g., Hypervisor) requests to deploy an accelerator, this system controller searches the database to find the appropriate mapping results based on the runtime policy and then sends requests to the low-level controller (provided by the underlying *HS* abstraction-based solution) to configure physical FPGAs. One soft block is deployed into one FPGA device (Fig. 5). The soft blocks of different *AS* ISA-based accelerators can be deployed into the same FPGA to enable an efficient FPGA sharing if this FPGA device has sufficient amount of resources. This system controller also provides APIs for communicating with the high-level system to enable an easy system integration.

As discussed in Section 2.2.2, the decomposed *AS* ISA-based accelerator is partitioned so that it can be deployed onto multiple physical FPGAs to alleviate the fragmentation issue caused by the physical FPGA boundary. Nevertheless, this also incurs an inter-FPGA communication overhead. To tackle this issue, the proposed runtime system adopts a greedy runtime management policy that minimizes the number of allocated FPGAs. Specifically, it sorts the mapping results based on the number of soft blocks in ascending order. It then tries to find a feasible allocation starting from the first mapping result. This policy effectively minimize the inter-FPGA communication overhead and introduces a negligible runtime overhead. Further exploration on more comprehensive runtime policy will be our future work.

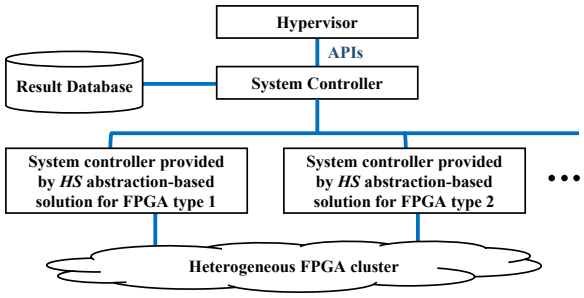

Figure 7: A conceptual diagram illustrates the runtime system. The proposed framework provides a system controller to perform resource allocation and sends requests to the low-level controller (provided by corresponding *HS* abstraction-based solution) for the FPGA configuration.

Optimization for Scale-out Acceleration: While scaling out one *AS* ISA-based accelerator onto multiple physical FPGAs can provide more computing capability, the limited inter-FPGA communication bandwidth and the long communication latency might outweigh the benefits obtained from the additional computing capability. Leveraging the extracted parallel patterns (Section 2.2.1), we propose a technique to improve the performance of scale-out acceleration by maximally overlapping the inter-FPGA communication and computation. This technique is applied for the *AS* ISA-based accelerators that have data parallelism in the root soft block of the data path. We expect this is a common case for *AS* ISA-based accelerators as most of them implement data processors to fully exploit the abundant spatial parallelism in FPGAs.

Instead of splitting an *AS* ISA-based accelerator onto multiple FPGAs, we propose to scale down one accelerator into multiple smaller accelerators and deploy these accelerators into FPGAs. As illustrated in Fig. 8a, scaling down one accelerator can be easily realized by reducing the number of data processing units. As the control path (e.g., instruction decoder) is not modified, the original software programs can still run on these small accelerators. We provide a parameterized template module and reuse the instructions for reading/writing on-board DRAM to perform inter-FPGA communication and synchronization. As illustrated in Fig. 8b, this template module monitors the DRAM interface for reading/writing data. If a data entry is wrote into a pre-defined address (e.g., an out-of-range address), then this module will send this data entry to the corresponding accelerator through the inter-FPGA network. If the accelerator reads a pre-defined address, this module will send a response to the accelerator only when it receives data from another accelerator to realize a barrier synchronization (assuming the accelerator implements an in-order processor). This module sets a flag when identifying this special read request. When this

flag is set, this module will combine the received data entry and the data entry read from the DRAM based on the index register for the next read request (Fig. 8b). This module invalidates these special read/write requests to ensure functional correctness. The parameters of this template module (e.g., buffer width, the value of pre-defined address and the content in the index register) are configured during the offline compilation time. A custom tool is developed for a specific *AS* ISA to automatically insert the corresponding DRAM read/write instructions for a given software program. We also provide another custom tool for a specific *AS* ISA to perform instruction reordering under the dependency constraint to maximally overlap the communication and computation.

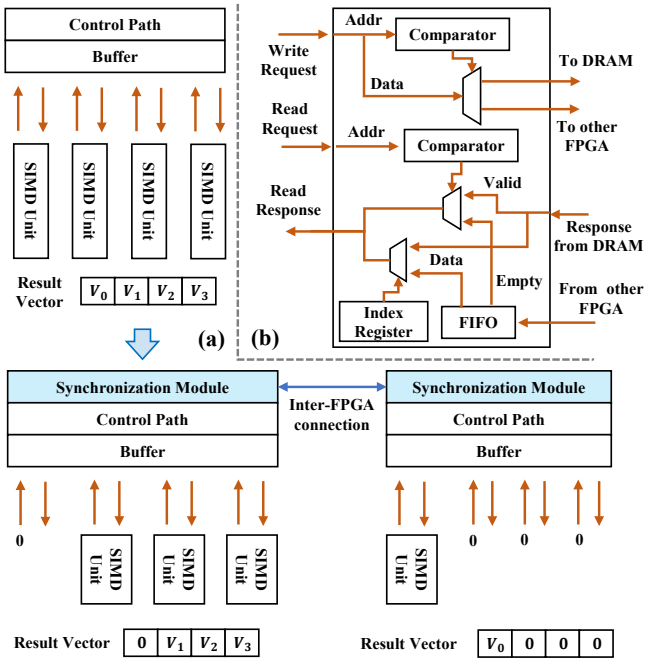

Figure 8: (a) A conceptual diagram illustrates the technique of scaling down one *AS* ISA-based accelerator. We provide a template module for inter-FPGA synchronization (highlighted in blue). (b) The key building blocks of this synchronization module are drawn in the figure.

3 CASE STUDY

In this case study, we use an *AS* ISA similar to the one proposed in Microsoft BrainWave project [18]. It is a representative use case of *AS* ISA and has been deployed in the commercial FPGA cloud to build a product-scale system for the low-latency DNN inference. We then use the recently proposed ViTAL [53] as the underlying *HS* abstraction as it outperforms other *HS* abstractions by providing a fine-grained resource management.

We first provide a parameterized accelerator design for this *AS* ISA as the design of the BrainWave project is not publicly available. The organization of this accelerator design is similar to that described in [18], e.g., tile engines and multi-function units (Fig. 9). It uses the block floating point format (BFP) for the matrix-vector multiplication to increase the computing capability and half-precision floating point format (float16) for other secondary operations to avoid quantization noise (e.g., point-wise vector multiplication and activation). The number of tile engines in the design can be adjusted to generate accelerator instances with different computing capabilities to account for the varying performance/cost demands. We provide a parameterized memory module in the accelerator design so that it can leverage the unique hardware resources (e.g., URAM) when being deployed onto heterogeneous FPGAs. The parameter of this module will be configured when mapping it onto the *HS* abstraction of a specific type of FPGA. While this solution provides a unified memory interface to simplify the design of the computing units, it leads to a under-utilization of the URAM resources as URAM provides a larger capacity (4096 72-bit words) than the BRAM does (512 72-bit words). Further exploration on the accelerator design will be our future work. We also include an instruction buffer to minimize the memory access. We do not provide a custom tool in the proposed framework for this buffer insertion because this is not an indispensable function for virtualization and it is not required by most *AS* ISA-based accelerators [47] that already comprise an instruction buffer.

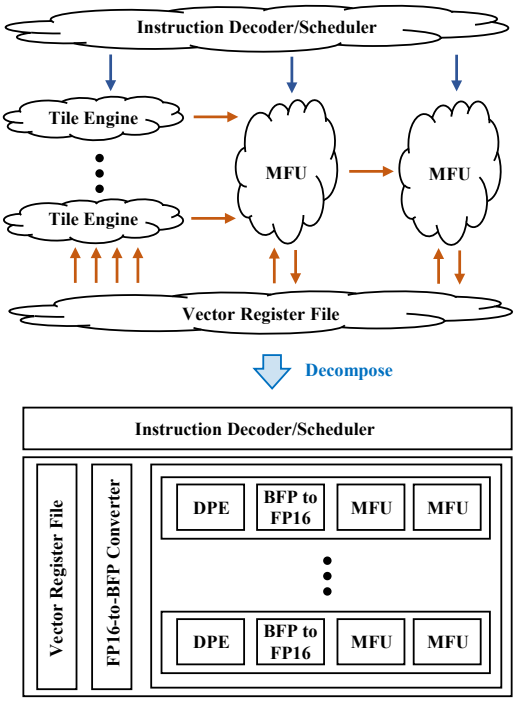

Figure 9: A conceptual diagram illustrates the organization of our accelerator design and the decomposing results.

We then use the provided tool to decompose this accelerator design into a soft block tree (Fig. 9) and map it onto the *HS* abstraction provided by ViTAL (an array of virtual blocks). The root soft block of the data path has child blocks connected in the pipeline parallelism. Nevertheless, we found that the FP16-to-BFP converter and the vector register file are much smaller than the remaining components. Therefore, we move these two components into the soft block that has control path. With this modification, the soft block of the data path has data parallelism and the optimization

technique described in Section 2.3 can be applied on this accelerator design. We also develop a set of custom tools to perform the instruction insertion/reordering required by the optimization technique.

4 EVALUATION

In this section, we evaluate the proposed multi-layer virtualization framework on a custom-built heterogeneous FPGA cluster. We provide an extensive evaluation on the performance impact of the proposed framework on both a single application as well as the entire system.

4.1 Benchmark Selection

Two sets of benchmarks are used to evaluate the proposed multi-layer virtualization framework at both the application level and the system level.

We use DeepBench [37] as the first benchmark set to evaluate the framework at the application level, which contains representative layers from various DNN models. At this application level, we measure the latency of the GRU/LSTM inferences tasks with a batch size of one to evaluate the virtualization overhead introduced by the proposed framework.

Table 1: Several representative compositions are considered when synthetically generating the second benchmark set.

Set Index	Composition
1	100% **S**
2	100% **M**
3	100% **L**
4	50% **S** + 50% **M**
5	50% **S** + 50% **L**
6	50% **M** + 50% **L**
7	33% **S** + 33% **M** + 34% **L**
8	10% **S** + 30% **M** + 60% **L**
9	30% **S** + 60% **M** + 10% **L**
10	60% **S** + 10% **M** + 30% **L**

S: Tasks using small LSTM/GRU model, #hidden units <= 1024
M: Tasks using medium LSTM/GRU model, 1024 < #hidden units <=2048
L: Tasks using large LSTM/GRU model, 2048 < #hidden units

The second set of benchmarks are used to evaluate the proposed framework at the system level. One benchmark contains multiple applications that can concurrently run on the FPGA cluster. As no real-world cloud workload set using FPGAs is publicly available, we choose to synthetically generate several workload sets with different compositions (Table 1). This is a common approach that has been widely used in prior works for cloud evaluation [39]. Specifically, each workload set comprises a sequence of GRU/LSTM inference tasks (from the first benchmark set) that arrives at a random time interval to emulate the dynamic runtime environment.

4.2 Experimental Setup

We evaluate the proposed virtualization framework on a real system — a custom-built FPGA cluster that contains three Xilinx Virtex UltraScale+ FPGAs (XCVU37P) and one Xilinx Kintex UltraScale FPGA (XCKU115). These four FPGAs are attached to the host machine through PCIe, and a secondary bidirectional ring network is deployed to connect them.

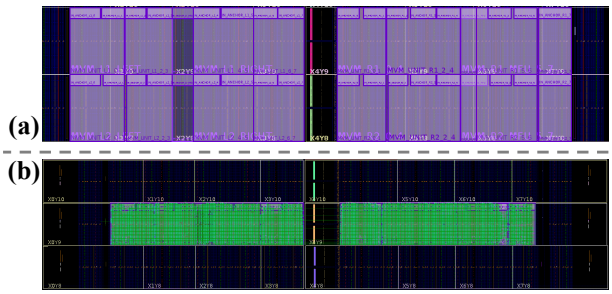

Figure 10: The floorplanning is leveraged to improve the mapping quality of the baseline accelerator. Part of the floorplanning used for XCVU37P FPGA is shown in (a). This function is also leveraged to improve the mapping quality of one virtual block in ViTAL to ensure a fair comparison. The optimized implementation result is shown in (b).

We fit our BrainWave-like accelerator design (Section 3) into these two types of FPGAs by adjusting the number of tile engines and create two accelerator instances as the baseline. In order to provide a high-quality baseline and ensure a fair comparison, we use the floorplanning function provided by Vivado [51] to improve the implementation quality by manually optimizing the placement (Fig. 10a). The resource usage and the performance of these two baseline accelerators are reported in Table 2. By levering floorplanning, the peak computing capability of this baseline is comparable to that reported in [18]. We then map the decomposed BrainWave-like accelerator onto the *HS* abstraction provided in ViTAL. The floorplanning function is also used to optimize the mapping results of virtual blocks (Fig. 10b) for a fair comparison. The implementation results of one virtual block for both FPGAs are reported in Table 3. To match the varying computation requirements of different LSTM/GRU inference tasks, multiple accelerator instances with different number of MVM Tiles (the SIMD units) are compiled.

4.3 Application Evaluation

In this subsection, we measure the latency of LSTM/GRU inference tasks and the compilation time (the first benchmark set) to show the proposed virtualization framework only introduces a marginal virtualization overhead. Two different scenarios are considered when evaluating the inference latency: 1) the *AS* ISA-based accelerator used to process one inference task is deployed onto a single FPGA device (no inter-FPGA communication overhead), and 2) the accelerator is deployed onto two FPGA devices. The inference latency in the first scenario is reported in Table 4. We observe that there is only a marginal increase in the inference latency (3%~8%), which is mainly caused by the latency-insensitive interface in ViTAL. This negligible overhead is achieved by leveraging the extracted parallel patterns. Specifically, instead of using the partition tool provided in ViTAL, we use the partition tool provided by this framework (Section 2.2.2) to partition decomposed accelerators into virtual blocks. Since this BrainWave-like accelerator has data parallelism in the top-level soft block (Section 3), the proposed partition tool can avoid placing the pipelined data path within a SIMD unit across virtual blocks. Consequently, it effectively minimizes the additional latency introduced by the latency-insensitive interface.

Table 2: Hardware implementation results of the two baseline accelerators.

	Device	#MVM Tiles	LUTs	DFFs	BRAMs	URAMs	DSPs	Freq. (MHz)	Peak TFLOPS
BW-V37	XCVU37P	21	610k (46.8%)	659K (25.3%)	51.5Mb (72.6%)	22.5Mb (8.3%)	7517 (83.3%)	400	36
BW-K115	XCKU115	13	367k (55.3%)	386k (29.1%)	45.4Mb (59.8%)	-	5073 (91.9%)	300	16.7

Table 3: Hardware implementation results of one virtual block when mapping the decomposed accelerator onto ViTAL.

Device	LUTs	DFFs	BRAMs	URAMs	DSPs	Freq. (MHz)	Peak TFLOPS
XCVU37P	44.9k (56.8%)	48.8k (30.8%)	3.9Mb (92.4%)	2.1Mb (9.5%)	576 (99.4%)	400	3.69
XCKU115	39.9k (78.8%)	34.9K (41.8%)	4.5Mb (87.5%)	-	552 (100%)	300	2.07

We then evaluate the impact of the inter-FPGA communication latency when one *AS* ISA-based accelerator is deployed onto two FPGA devices. We implement a programmable module that includes a counter and a FIFO on FPGAs to intentionally add a certain amount of latency into the inter-FPGA communication. This allows us to comprehensively evaluate the effectiveness of the proposed optimization technique (Section 2.3) under various conditions. As shown in Fig. 11, the proposed technique can effectively hide the inter-FPGA communication latency for LSTM inference tasks by overlapping the data transfer of vector h_t and the matrix multiplication related to x_t. For GRU inference tasks, this technique can overlap the data transfer and computation for small GRU model ($h = 1024$) when the added communication latency is less than $0.6\mu s$. Nevertheless, the inter-FPGA communication latency cannot be hided for a large GRU model ($h = 2560$). This is because a large GRU model needs a large *AS* ISA-based accelerator that provides sufficient on-chip storage for weight. Such large accelerator also provides more computation capability than the one used for small model, leading to a shorter computation time. On the other hand, the data transfer time increases in a larger model as it has a longer vector. Therefore, compared with small GRU models, it is harder to hide the inter-FPGA communication latency for large GRU models.

Table 4: The latency of LSTM/GRU inference tasks.

Benchmark	Device	Latency (ms)		
		Baseline	This work	Overhead
GRU	VU37P	0.0131	0.0136	3.8%
h=512 t=1	KU115	0.0227	0.0236	3.9%
GRU	VU37P	5.01	5.4	7.8%
h=1024 t=1500	KU115	18.5	19.9	7.8%
GRU	VU37P	1.83	1.96	7.5%
h=1536 t=375	KU115	6.91	7.43	7.5%
LSTM	VU37P	0.726	0.767	5.7%
h=256 t=150	KU115	1.31	1.38	5.6%
LSTM	VU37P	0.129	0.136	5.3%
h=512 t=25	KCU115	0.232	0.245	5.3%
LSTM	VU37P	0.146	0.157	7.0%
h=1024 t=25	KCU115	0.263	0.282	7.1%
LSTM	VU37P	0.238	0.258	8.4%
h=1536 t=50	KCU115	-	-	-

-: Cannot fit into the FPGA.

We finally evaluate the compilation overhead introduced by the proposed virtualization framework. Compared with the baseline compilation flow, the proposed framework introduces three additional compilation steps: decomposing (Section 2.2.1), partitioning (Section 2.2.2) and mapping the scaled-down accelerators (Section 2.3). By leveraging parallel patterns, the first two steps have a low timing complexity and their runtime is negligible compared with that of the baseline compilation flow (< 1%). The third step needs to generate multiple combinations (2~5 in our evaluation) to enable a flexible runtime management, which could substantially increase the compilation time for one *AS* ISA-based accelerator. Nevertheless, we note that multiple accelerator instances with different number of SIMD computing units are required to account for the varying performance/cost demand. Most scaled-down accelerators can be reused across these accelerator instances to effectively amortized the compilation cost. For the evaluated benchmarks, 10 different accelerator instances are provided for the two types of FPGAs. By amortizing the compilation cost of scaled-down accelerators among these instances, the proposed framework only introduces a 24.6% compilation overhead compared with the baseline. Since the FPGA compilation is performed offline, this compilation overhead has no impact on the runtime performance.

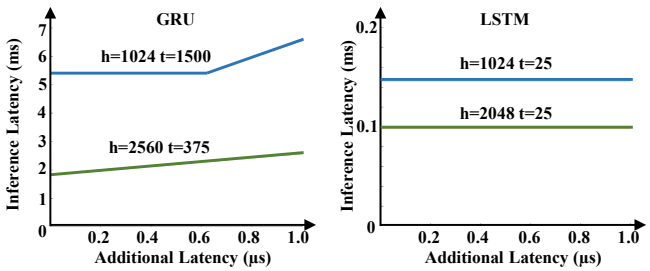

Figure 11: The impact of the inter-FPGA communication latency on the inference latency when the *AS* ISA-based accelerator is deployed onto two FPGA devices.

4.4 System Evaluation

In this subsection, we evaluate the system performance using the second benchmark set (Table 1). We first measure the average system throughput in terms of processed inference tasks, i.e., task per second. As shown in Fig. 12, the proposed virtualization framework efficiently combines the *AS* ISA and *HS* abstraction, thereby improving the aggregated system throughput by 2.54× on average when compared with the baseline system. One unique feature of the proposed framework is the support for deploying one accelerator onto *heterogeneous* FPGAs, while existing *HS* abstractions only provide multi-FPGA support for *homogeneous* FPGAs. Thus, we include an additional system with a restricted runtime policy in the evaluation (Fig. 12). In this added system, one *AS* ISA-based accelerator can only be deployed into FPGAs of the same type to emulate the multi-FPGA support provided by existing *HS* abstractions. The results show that the flexible runtime deployment in

the proposed multi-layer virtualization framework improves the aggregated system throughput by 16%.

This evaluation result also confirms the effectiveness of the added instruction buffer (Section 3). For evaluated LSTM/GRU benchmarks, the entire machine codes can be stored in this buffer to largely minimize the number of DRAM accesses, thereby avoiding contention on the shared DRAM interface. This enables a sufficient performance isolation and the inference latency in this resource-sharing environment is comparable to that in a non-sharing environment.

5 RELATED WORK

5.1 Parallel Patterns

Parallel patterns have been widely leveraged in domain-specific languages (DSLs) to reduce the programming complexity of various computing devices, including FPGAs [3, 19, 36, 41, 42], multi-core CPUs [5, 45] and GPUs [8, 29]. Different from previous works, we leverage parallel pattern to develop a new system abstraction (Section 2.1) that enables an efficient virtualization support for *heterogeneous* cloud FPGAs by bridging *AS* ISA and *HS* abstraction. Specifically, this new system abstraction adopts a multi-level tree structure to effectively represent the extracted parallel patterns. This structural information is then exploited to reduce the mapping complexity (Section 2.2) and improve the runtime system performance (Section 2.3). As parallel patterns are *FPGA-independent*, the proposed system abstraction can effectively hide *FPGA-specific* hardware details to substantially reduce the runtime resource management complexity.

5.2 Overlay Architecture

Various overlay architecture [4, 7, 15, 28, 31, 50] have been proposed to enable code portability across heterogeneous FPGAs for virtualization purpose. This method is functionally analogous to Java Virtual Machine (JVM), i.e., applications are first mapped onto the overlay architecture and then mapped onto physical FPGAs. However, it is challenging to apply this type of method in the cloud environment as they are developed for a *single* FPGA in a non-sharing environment. Moreover, in order to virtualize heterogeneous FPGAs, existing overlay architecture need to hide *FPGA-specific* hardware details. Without exposing sufficient spatial resource constraints, the overlay architecture could lead to a non-negligible degradation in the compilation quality (e.g., resource waste and frequency drop). In this work, we propose a multi-layer abstraction, where our new system abstraction enables code portability across different types of FPGAs and the underlying *HS* abstraction ensures the compilation quality by exposing sufficient spatial resource constraints.

5.3 Other Related Work

Cambricon-F [55] is an interesting work that builds computing devices with a fractal structure, i.e., the processing component of one Cambricon-F machine is a smaller Cambricon-F machine. Cambircon-F shares some similarity with this paper, i.e., 1) the fractal structure is similar to the multi-level tree structure used in our system abstraction, and 2) both works are proposed for virtualization purpose. The major difference is that we use the tree structure to decompose a given accelerator design to simplify the compilation and resource management, while Cambricon-F uses the fractal structure to build new computing machines at different scales that share the same software stack.

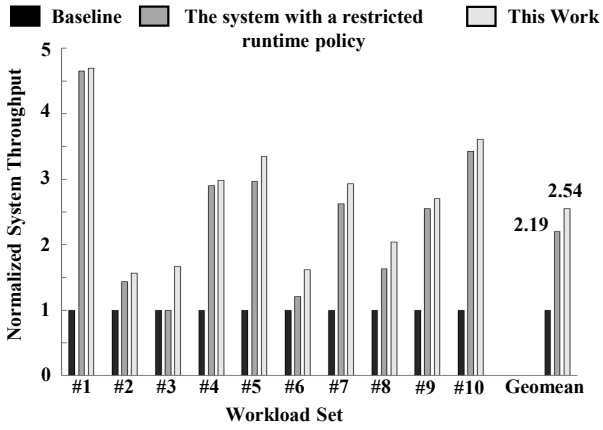

Figure 12: The aggregated system throughput when running different workload sets. The system with a restricted runtime policy also applies the proposed multi-layer virtualization framework, but one *AS* ISA-based accelerator can only be deployed into FPGAs of the same type in this system.

6 CONCLUSION

In this paper, we propose to combine *AS* ISA and *HS* abstraction for virtualizing heterogeneous cloud FPGAs. To bridge the gap between these two abstractions, we provide a multi-layer virtualization framework. The key enabler of this framework is a new system abstraction that serves as an indirection layer between *AS* ISA and *HS* abstraction. This system abstraction adopts a multi-level tree structure to effectively represent the fine-grained parallel patterns extracted from the accelerator design. This extracted structural information is then utilized to simplify the compilation process and is also leveraged to enable an optimization technique that can maximally hide the inter-FPGA communication latency when exploiting scale-out acceleration. While experienced system designers might be able to manually perform the mapping process for small accelerator designs, we provide a set of tools to automate the entire mapping process and reduce the burden on system designers.

We combine an *AS* ISA similar to the one proposed in Microsoft BrainWave [10, 18] with a recent *HS* abstraction [53] and use this as a case study to demonstrate the effectiveness of the proposed virtualization framework. The performance is rigorously evaluated on a custom-built heterogeneous FPGA cluster. Compared with a system that only uses *AS* ISA, the proposed system can improve aggregated system throughput by 2.54× on average with a marginal virtualization overhead (latency and compilation time increase).

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and feedback to help improve the quality of the paper.

REFERENCES

[1] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/, 2016.

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

[3] Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 89–108, 2010.

[4] Alexander Brant and Guy GF Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *FCCM*, pages 93–96. IEEE, 2012.

[5] Kevin J Brown, Arvind K Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89–100. IEEE, 2011.

[6] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

[7] Davor Capalija and Tarek S Abdelrahman. A High-Performance Overlay Architecture for Pipelined Execution of Data Flow Graphs. In *FPL*, pages 1–8. IEEE, 2013.

[8] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 47–56, 2011.

[9] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014.

[10] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Accelerating Persistent Neural Networks at Datacenter Scale. In *Hot Chips*, volume 29, 2017.

[11] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad EI Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Grace Rapsang, Stenen K. Reinhardt, Bita Darvish Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[12] Eric S Chung, John D Davis, and Jaewon Lee. Linqits: Big data on little clients. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 261–272. ACM, 2013.

[13] Alibaba Cloud. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057, 2018.

[14] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[15] James Coole and Greg Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 13–22. ACM, 2010.

[16] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From OpenCL to High-Performance Hardware on FPGAs. In *FPL*, pages 531–534. IEEE, 2012.

[17] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*, pages 105–110. ACM, 2016.

[18] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weize, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrain M. Caulfield, Eric S. Chung, and Doug Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[19] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware System Synthesis from Domain-Specific Languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[20] Evguenii I Goldberg, Mukul R Prasad, and Robert K Brayton. Using SAT for Combinational Equivalence Checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 114–121. IEEE, 2001.

[21] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2017.

[22] InAccel. InAccel - Application Acceleration made simple. https://inaccel.com/, 2018.

[23] Intel. Intel® Stratix® 10 Embedded Memory User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-memory.pdf, 2021.

[24] Nachiket Kapre. Custom FPGA-based Soft-Processors for Sparse Graph Acceleration. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 9–16. IEEE, 2015.

[25] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 107–127, 2018.

[26] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures. In *International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*, 2017.

[27] Oliver Knodel and Rainer G Spallek. RC3E: Provision and Management of Reconfigurable Hardware Accelerators in a Cloud Environment. *arXiv preprint arXiv:1508.06843*, 2015.

[28] Dirk Koch, Christian Beckhoff, and Guy GF Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL*, pages 1–8. IEEE, 2013.

[29] HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 63–74. IEEE, 2014.

[30] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 51–57, 2020.

[31] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. QuickDough: a rapid FPGA loop accelerator design framework using soft CGRA overlay. In *FPT*, pages 56–63. IEEE, 2015.

[32] Rui Ma, Jia-Ching Hsu, Tian Tan, Eriko Nurvitadhi, David Sheffield, Rob Pelt, Martin Langhammer, Jaewoong Sim, Aravind Dasu, and Derek Chiou. Specializing FGPU for Persistent Deep Learning. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 326–333. IEEE, 2019.

[33] Ricardo Menotti, Joao MP Cardoso, Marcio M Fernandes, and Eduardo Marques. Automatic Generation of FPGA Hardware Accelerators Using A Domain Specific Language. In *FPL*, pages 457–461. IEEE, 2009.

[34] Microsoft. Real-time AI: Microsoft announces preview of Project Brainwave. https://blogs.microsoft.com/ai/build-2018-project-brainwave/, 2018.

[35] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to Combinational Equivalence Checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 836–843. IEEE, 2006.

[36] Karthik Nagarajan, Brian Holland, Alan D George, K Clint Slatton, and Herman Lam. Accelerating Machine-Learning Algorithms on FPGAs Using Pattern-Based Decomposition. *Journal of Signal Processing Systems*, 62(1):43–63, 2011.

[37] S Narang and G Diamos. Baidu DeepBench. https://github.com/baidu-research/DeepBench, 2017.

[38] Jian Ouyang, Ephrem Wu, Jing Wang, Yupeng Li, and Hanlin Xie. XPU: A Programmable FPGA Accelerator for Diverse Workloads. In *2017 IEEE Hot Chips 29 Symposium*, 2017.

[39] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. Energy Efficient Architecture for Graph Analytics Accelerators. In *ISCA*, pages 166–177. IEEE, 2016.

[40] M Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. FPGA-based Accelerator Design From A Domain-Specific Language. In *FPL*, pages 1–9. IEEE, 2016.

[41] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating Configurable Hardware from Parallel Patterns. *Acm Sigplan Notices*, 51(4):651–665, 2016.

[42] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A Reconfigurable Architecture for Parallel Patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE, 2017.

[43] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *2014 ACM/IEEE 41st*

International Symposium on Computer Architecture (ISCA), pages 13–24. IEEE, 2014.

[44] Aaron Severance and Guy GF Lemieux. Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[45] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–25, 2014.

[46] CAJ Van Eijk. Sequential Equivalence Checking based on Structural Similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2000.

[47] Chao Wang, Lei Gong, Fahui Jia, and Zhou Xuehai. An FPGA based Accelerator for Ubiquitous Clustering Applications with Custom Instructions. *IEEE Transactions on Computers*, 2020.

[48] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086. IEEE, 2015.

[49] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *DAC*, page 29. ACM, 2017.

[50] Tobias Wiersema, Ame Bockhorn, and Marco Platzner. Embedding FPGA overlays into configurable systems-on-chip: ReconOS meets ZUMA. In *ReConFigurable Computing and FPGAs, 2014 International Conference on*, pages 1–6. IEEE, 2014.

[51] Xilinx. Xilinx Floorplanning Methodology Guide (UG633). https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/Floorplanning_Methodology_Guide.pdf, 2013.

[52] Xilinx. UltraScale Architecture Memory Resources (UG 573). https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf, 2020.

[53] Yue Zha and Jing Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–858, 2020.

[54] Jialiang Zhang and Jing Li. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *FPGA*, pages 25–34. ACM, 2017.

[55] Yongwei Zhao, Zidong Du, Qi Guo, Shaoli Liu, Ling Li, Zhiwei Xu, Tianshi Chen, and Yunji Chen. Cambricon-F: Machine Learning Computers with Fractal von Neumann Architecture. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 788–801. IEEE, 2019.