FOS: A Modular FPGA Operating System for Dynamic Workloads

DOI:
https://doi.org/10.1145/3405794

Document Version
Accepted author manuscript

Citation for published version (APA):
Vaishnav, A., Pham, K., Powell, J., & Koch, D. (2020). FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM Transactions on Reconfigurable Technology and Systems*, *13*(4), [20]. https://doi.org/10.1145/3405794

Published in:
ACM Transactions on Reconfigurable Technology and Systems

FOS: A Modular FPGA Operating System for Dynamic Workloads

ANUJ VAISHNAV, KHOA DANG PHAM, JOSEPH POWELL, and DIRK KOCH,
Department of Computer Science, The University of Manchester, UK

With FPGAs now being deployed in the cloud and at the edge, there is a need for scalable design methods that can incorporate the heterogeneity present in the hardware and software components of FPGA systems. Moreover, these FPGA systems need to be maintainable and adaptable to changing workloads while improving accessibility for the application developers. However, current FPGA systems fail to achieve modularity and support for multi-tenancy due to dependencies between system components and lack of standardized abstraction layers. To solve this, we introduce a modular FPGA operating system – FOS, which adopts a modular FPGA development flow to allow each system component to be changed and be agnostic to the heterogeneity of EDA tool versions, hardware and software layers. Further, to dynamically maximize the utilization transparently from the users, FOS employs resource-elastic scheduling to arbitrate the FPGA resources in both time and spatial domain for any type of accelerators. Our evaluation on different FPGA boards shows that FOS can provide performance improvements in both single-tenant and multi-tenant environments while substantially reducing the development time and, at the same time, improving flexibility.

CCS Concepts: • **Software and its engineering → Operating systems**; **Multiprocessing / multiprogramming / multitasking**; *Layered systems*; *Scheduling*; • **Computer systems organization → Heterogeneous (hybrid) systems**; **Reconfigurable computing**; • **Hardware → Hardware-software codesign**; **Hardware accelerators**; *Reconfigurable logic applications.*

Additional Key Words and Phrases: FPGA, Operating System, Resource-elasticity, Modular Development, Dynamic Workloads, FPGA Shell, High-level Synthesis, Runtime systems

ACM Reference Format:
Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. 2020. FOS: A Modular FPGA Operating System for Dynamic Workloads. In *ACM Transactions on Reconfigurable Technology and Systems 2020*. ACM, New York, NY, USA, Article 1, 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

FPGA accelerators can provide high-performance computing at low energy cost for applications ranging from neural-networks to network processing. This has brought FPGAs into cloud datacenters as well as into embedded systems at the edge. However, to sustain this large scope of requirements present in terms of heterogeneity of devices, environments, EDA tools, users and developer needs, we require a standardized way to manage and integrate FPGA systems components, in other words: we need an FPGA operating system.

The idea of an FPGA operating system is old and has taken many forms over time. It has evolved from providing a) OS-level APIs such as hardware threads [4, 39] and UNIX interfaces [30] for allowing hardware accelerators to use existing software OS services, to b) light-weight FPGA shells with library APIs for hardware acceleration [5, 9, 15, 28, 40].

TRETS, June 03–05, 2020,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

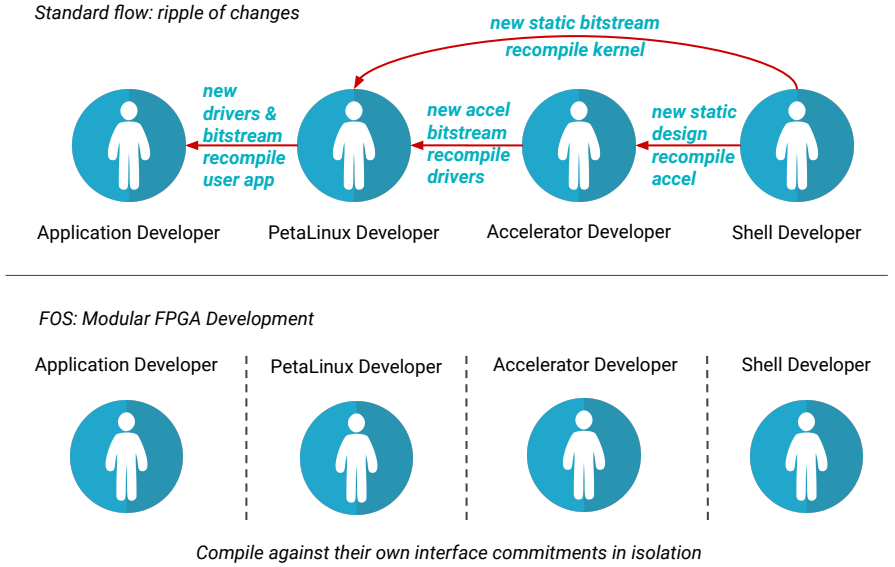


Fig. 1. If we update the shell IP or EDA tool version, the rest of the system components must be recompiled because of the dependencies in the standard tool flow (see Section 2.2). Ideally, we want each component to be compiled in isolation with given interfaces, as we do for FOS.

We believe this evolution continues for two primary reasons: *overhead and lack of portability*. The introduction of intermediate layers and communication cost must be low in terms of latency as well as in the number of resources required. However, to achieve this is difficult because hardware accelerators and boards require unique optimizations and implementations, leading to heterogeneity in FPGA shells, accelerators, EDA tools, and the low-level software required to interface with it. In particular, this reduces not only the portability of a system but also its *maintainability*, which is a crucial aspect of any operating system (OS). When using, for example, the current tool flows of the major FPGA vendors, a simple change in tool version or addition of system IP can lead to re-compilation of the whole FPGA stack [18, 32], as shown in Figure 1. Essentially, implying that an operating system update means re-compilation of all applications which run on top of it.

Moreover, with the need for multi-tenancy in dynamic environments like the cloud, we expect multiple types of workloads and accelerators to execute concurrently [33] (e.g., for enabling resource pooling on FPGA accelerator cards). However, most present FPGA systems operate and support either static accelerators or single accelerator execution [2, 10, 29, 41]. Systems which support hosting multiple accelerators, do not allow resources to be re-allocated during execution or employ only time-domain multiplexing [21, 31, 38]. This commonly leads to under-utilization and a lack of *adaptability* in the system.

At the same time, there is a need for better APIs to make FPGAs more *accessible* to software programmers or application domain experts that are commonly not FPGA experts. This should be achieved while ensuring that hardware developers can integrate their accelerators into software stacks with high productivity, i.e. without changing their hardware designs or writing complex software wrappers.

To solve these challenges of maintainability, adaptability, and accessibility, we are introducing a modular FPGA Operating System (FOS). FOS adopts a modular FPGA development flow to allow each type of OS component (hardware or software) to be replaced, reused, or ported to different

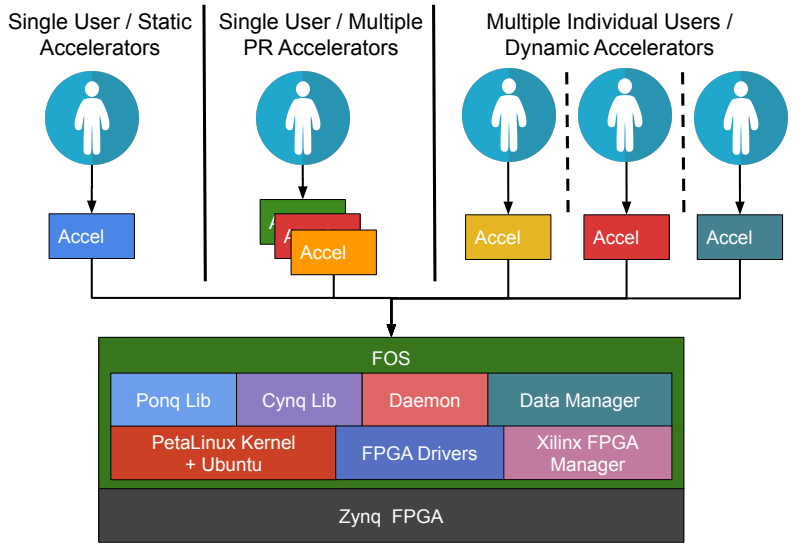


Fig. 2. System components and usage mode of the FPGA Operating System – FOS.

FPGA systems without extensive effort or compilation time. Moreover, the runtime system provides full support for multi-tenancy with the ability to concurrently execute accelerators written in various languages (C, C++, OpenCL, or RTL) and dynamically replicate or switch to a different version of an accelerator to improve utilization and system performance. To interact with these accelerators, application developers can use high-level APIs (available in multiple languages) to access the FPGA in three modes: 1) static acceleration for a single user, 2) dynamic acceleration for a single user, and 3) dynamic acceleration in multi-tenant environments, as shown in Figure 2. At the same time, hardware developers can write light-weight interface descriptions to integrate their hardware accelerators into the FOS platform and benefit from its high-level software APIs.

All the research artifacts proposed in this paper were presented at the FPL 2019 Demo Night [36] and are available online under an open-source license[1]. The key contributions of this paper are:

- A *modular FPGA development flow* to encapsulate the heterogeneity at each level of the development process (Section 2.2).
- An *open-source software platform* to improve programmability for static acceleration systems, dynamic systems for single-user as well as multi-user dynamic acceleration systems (Section 3 and Section 4).
- A *decoupled compilation flow for shell and modules* with support for module relocation and flexibility to combine PR regions to host accelerators of varying sizes (Section 4.1).
- A *resource-elastic runtime system for dynamic workloads* supporting virtually any type of accelerators (written in RTL, C, C++, and OpenCL) with a unified user interface and programming model (Section 4.4).
- A thorough evaluation of the FPGA Operating System (FOS) on resource overhead, compilation latency, memory throughput performance, application performance, as well as flexibility and maintainability of the system (Section 5).

[1]https://github.com/khoapham/fos

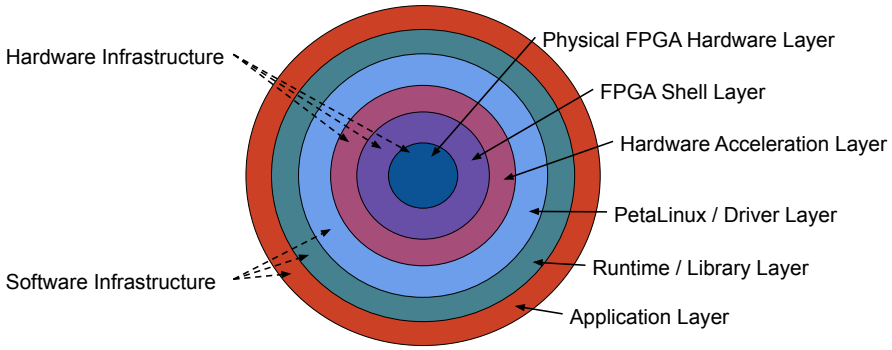


Fig. 3. The layered architecture of an FPGA operating system.

2 CONCEPTS

2.1 FPGA Operating System

Unlike standard CPU operating systems which consist only of a software infrastructure, an FPGA operating system requires an infrastructure at both the software and the FPGA hardware level (as shown in Figure 3). The following subsections describe the role of each level in detail.

2.1.1 Hardware Infrastructure. The hardware infrastructure is implemented directly on top of the physical FPGA resources and is commonly built using a partial reconfiguration (PR) flow (e.g., Amazon F1 FPGA instances [29]). A PR flow generates a system in two partitions: a static system and one or more reconfigurable regions. An FPGA shell is essentially the static system of the PR flow and can be considered hardware equivalent to a software OS kernel. It provides common functionalities required by the accelerators such as an interconnect, network, and memory access as well as (optional) other I/O and management IPs necessary for the target system. The counterpart of the shell is the partially reconfigurable region (also called a *slot*), which can host different hardware accelerators at runtime. A shell that supports multiple partial regions can support multi-tenancy in the spatial domain by allowing multiple accelerators to execute in parallel. Figure 4 shows an example of such a shell. However, the number of partial regions, their location and sizes depend on the system requirements and changes from system to system. An integral part of the shell is the compilation flow required to map the hardware modules onto the resources of a PR region with the required physical interface for the target system infrastructure.

Moreover, a shell often includes a CPU, either in the form of a soft-core (e.g., on datacenter FPGAs [42]) or a hardened CPU-core on MPSoC platforms used for embedded systems [42]. This CPU is used to host the software infrastructure, which is necessary for managing the FPGA resources (see Section 2.1.2 for details).

With this, a shell provides the basic OS functionality at the hardware level to host hardware applications (accelerators) on an FPGA.

2.1.2 Software Infrastructure. The software infrastructure of an FPGA operating system serves as a middle layer between the hardware accelerators and the user applications (software) while executing on a host CPU (soft or hardened). However, unlike software operating systems, the software infrastructure for an FPGA OS should be resilient to dynamic changes in the underlying hardware and be portable to different FPGA boards while hiding the heterogeneity from the software programmer. Consequently, an FPGA OS is responsible for managing the heterogeneity of the hardware resources available on the FPGA and providing the high-level APIs and software

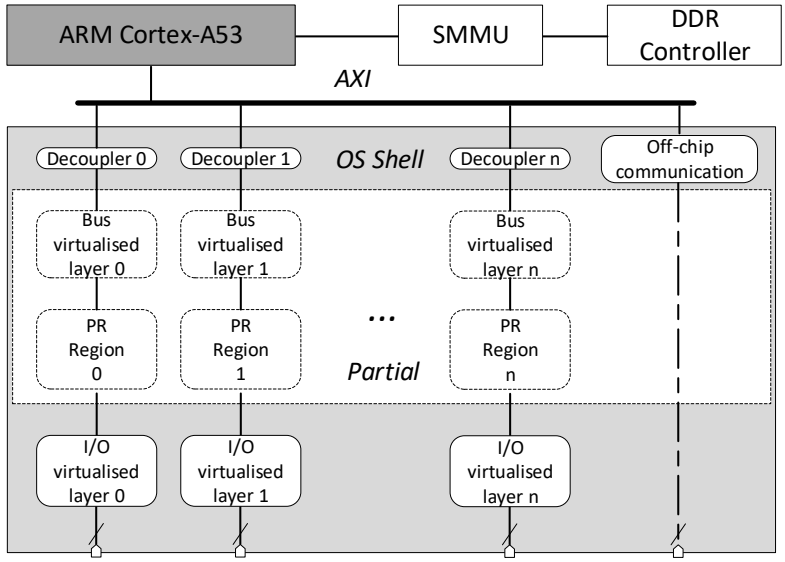


Fig. 4. The overall organisation of an FPGA shell.

integration for ease of development. This requires the software infrastructure to provide five important functionalities:

(1) A boot-loader and a kernel to bring up the FPGA system in an operational state.
(2) A set of drivers and hardware abstraction layer (HAL) to communicate with the hardware accelerators.
(3) A scheduler to dynamically allocate FPGA resources (i.e. partial regions) and hardware accelerators to software applications.
(4) High-level standard libraries to make hardware acceleration easily accessible.
(5) Integration with existing CPU software stacks to benefit from legacy code.

2.2 Modular Development Flow

There are five primary stages of development required for a multi-tenant/cloud FPGA system, as shown in Figure 5. The bottom two stages of this stack are hardware development stages which drive the application performance and the support for multi-tenancy (via PR).

In particular, 1) *shell development* requires designing and implementing the common system functionalities required by the user accelerators. An essential step at this stage is floorplanning which includes deciding the number of resources to be allocated to accelerators and a definition of the interface exposed to the accelerators. This stage also requires identifying the address mappings at which the driver layer will access the accelerators.

With the shell providing the infrastructure, we can continue with its counterpart: 2) *accelerator development* which involves designing RTL or HLS accelerators with application-specific optimizations (commonly for the highest performance achievable with the allocated resource budget). To compile the accelerators for the shell, the compiler must know the exact resources available in the partial regions and the physical locations of the interface pins. Without this information, we cannot implement partially reconfigurable accelerators. However, with the current FPGA vendor partial reconfiguration development flows, accelerator compilation directly depends on the shell and requires knowing the implementation (internal resource allocation and routing) of a shell [45].

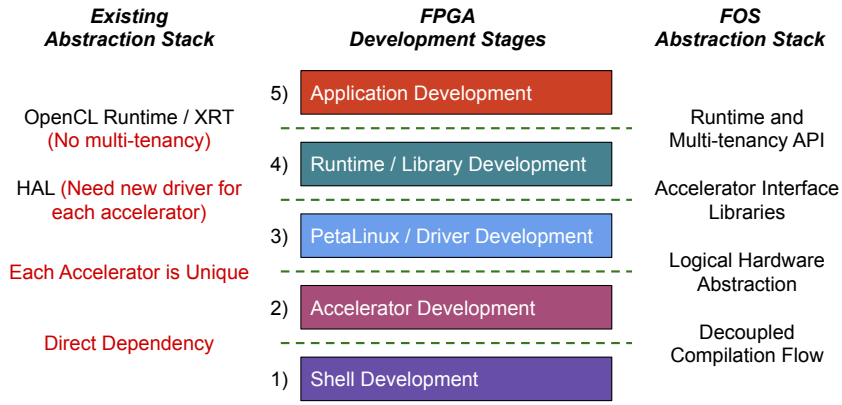


Fig. 5. The abstraction layers required between FPGA development stages.

This is because accelerator modules are implemented as an increment to a specific shell. Hence, any update made to the shell requires the recompiling of the accelerators. Note that this flow is used to build most state-of-the-art shells [5, 8, 10, 14, 25].

The remaining three stages of the stack are software development stages which aim at programming the accelerators, performing runtime optimizations, and improving the developer productivity.

The 3) *PetaLinux/Driver development* involves building the embedded software required to boot up the board with the necessary I/O and high-level functionalities as well as means to communicate with the accelerators in order to send and receive data. The current Xilinx tool flow expects the user to write a new driver for each accelerator or generate one automatically when using Vivado HLS [11]. This new driver has to be either built with the embedded Linux kernel [44] directly (for foreknown hardware) or with a device-tree overlay (for hardware known only at runtime). EDA tool vendors provide a hardware abstraction level (HAL) to make the development of the drivers easier via APIs to read and write to accelerators, to perform reconfiguration calls as well as basic C functionalities to enable debugging and fast development [43]. However, an accelerator developer or embedded Linux developer must take the responsibility to write and integrate the driver correctly into the rest of the Linux environment.

The layer above this fundamental driver and kernel layer forms 4) *a runtime system and/or libraries* for hardware acceleration. This provides a high-level API (such as for OpenCL) to the user for serving reconfiguration and acceleration requests to the lower hardware layers [18, 32, 43]. However, currently, no FPGA vendor tools provide support for multi-tenancy.

Finally, 5) in the *application development stage* a developer (commonly a domain expert) performs the required task in software while using hardware acceleration where appropriate. Note that a developer at this stage does not need to possess the skills required to implement the underlying FPGA development stack.

Overall, each step of this stack requires sophisticated knowledge, which makes designing systems challenging for individuals or small design teams. In the case of large teams, these dependencies require frequent synchronization and integration tests, which are slowing down the development process. Given the complexity and the effort required for the task involved, the final system is often application-specific and cannot be reused or ported for different needs. To avoid this, we need a standard set of APIs such that a component can be swapped at each stage without recompilation or redevelopment of the components above or below it as long as the APIs are maintained. Thus, allowing each stage to be a modular artifact in itself.

Such an abstraction stack would also allow 1) the software stack to be reusable across all types of FPGAs and FPGA boards, 2) not require the accelerator developers to write drivers, and 3) update system components in the shell with no need to propagate the changes to other stages in the stack.

We can achieve this by implementing 4 key layers of abstraction:

- **Decoupled Hardware Compilation Flow**: Ability to compile accelerators and shells in isolation from each other, i.e. the accelerators would compile against a fixed physical interface and a bounded resource region. This prevents changes in IPs or the shell from propagating to accelerators (given that we do not change PR regions or interfaces).
- **Logical Hardware Abstraction**: Exposing the accelerator and the shell in a high-level format with only a minimum set of parameters as required to build the drivers and perform the resource allocation. For the shell, includes information such as how many regions it supports and at what addresses regions can be accessed. Whereas for the accelerators, it would include the i) internal hardware address register mapping (ADR map) for programming the accelerators and ii) meta-data associated with the accelerators for scheduling and management purposes (e.g., the size or maximum execution latency). Note that HLS compilation (through Vivado HLS) provides this information without the need of any manual step.
- **Accelerator Interface Libraries**: Using a standardized register mapping format to provide generic driver support for accelerators with streaming or master-slave interfaces (which are the most common interfaces provided by shells). Thus, it relieves the accelerator developer from the responsibility of writing drivers. In the case that the adoption of the standardized format is not feasible, a developer can use HAL APIs to build drivers.
- **Runtime and Multi-tenancy API**: Execution API which can abstract the interface at the logical level and which can span across multiple languages to provide high-level integration to existing software stacks, while supporting the necessary primitives or programming models for dynamic resource allocation.

3 FOS OVERVIEW

The concepts for building an FPGA operating system (as introduced in Section 2.1) have been developed into FOS. FOS is a modular and lightweight FPGA Operating System which provides three primary modes of operation (as shown in Figure 2):

(1) Execution on static accelerators in a single-tenant mode.
(2) Using multiple partially reconfigurable accelerators in a single-tenant mode.
(3) Dynamically offloading acceleration request in a multi-tenant mode.

To provide the first two-modes of operation with ease of access, FOS provides two libraries: Cynq and Ponq. They provide high-level APIs to interact with hardware from C++ and Python applications, respectively. These APIs are platform-independent and can support the traditional Xilinx PR flow as well as the decoupled compilation flow introduced with FOS.

The multi-tenant mode is provided through a daemon which orchestrates the hardware acceleration requests from different users and schedules them in time *and* in the spatial domain. Moreover, the daemon can concurrently execute hardware acceleration requests on heterogeneous accelerators (i.e. accelerators written in different languages) while providing the same user interface.

To support the underlying hardware interaction in each mode, the libraries include generic drivers to program accelerators and the Xilinx FPGA manager [44] for partial reconfiguration. Moreover, FOS uses the PetaLinux Kernel [44] and Ubuntu rootfs, to adopt the standard functionalities provided by the Ubuntu Linux distribution, such as access to file systems, network access (via the host-CPU), standard libraries, debugging tools, and other forms of I/O.

The hardware infrastructure used in this paper is based on the open-source ZUCL 2.0 shell [6]. It currently supports three boards of varying FPGA capacity: ZCU102 (Xilinx MPSoC development kit), UltraZed, and Ultra-96 (suitable for IoT and edge deployment). The shell provides the ability to reallocate hardware accelerators across different partial regions as well as to combine multiple adjacent partial regions to host bigger accelerators.

With these additions to the software and hardware ecosystem for FPGAs, FOS achieves a similar level of support for rapid development, deployment, and ease of use, as known from standard operating systems for CPUs.

4 IMPLEMENTATION

4.1 Decoupled Compilation Flow for Shell and Modules

The primary requirement for abstracting accelerators from the shell is to decouple their compilation. However, this alone does not enable the flexible resource allocation required for the maximum utilization of the resources. This is because for flexibility, multiple partial regions should be combinable for hosting different sized modules and modules should be relocatable and duplicatable (at bitstream level). However, none of these requirements are met when using standard FPGA vendor EDA tool flows. To solve this, there are strict requirements in order to achieve isolation of the development with system support for relocation and PR region flexibility:

(1) *PR regions should be homogeneous* in terms of their resource foot-print (i.e. the relative layout of FPGA primitives) to allow accelerator relocation, and regions should be adjacent to each other to allow hosting bigger accelerators without interfering with other system components.
(2) *Communication interfaces between modules and the static system must be identical* in terms of both logical protocol and their physical implementation such that relative positions of connection wires are the same in all PR regions. This ensures that modules can receive operation commands from the host CPU, and it provides interfaces to transfer data back and forth to the main memory, irrespective of a module placement position.
(3) *Clock signals must follow the same regular pattern across each PR region.* This defines constrained clock routing paths, which will be used to provide clock signals to the modules regardless of the final target position on the FPGA.
(4) *Routing from the static part must be prohibited from passing through the reconfiguration part and vice versa* (except the interface routing) to ensure that module relocation does not interfere with other parts of the system[2].

The proposed design methodology depends on the standard FPGA development flow (Vivado toolchain for Xilinx FPGAs) in order to implement 1) the basic infrastructure which is acting as the OS shell, and 2) the hardware applications (i.e. accelerator modules). Our contributions, however, include all the additional design steps and their corresponding tools that customize and adapt the default flow to build the final system automatically. This integrates the entire shell and module development process into a unified design framework, as illustrated in Figure 6.

The main steps of the compilation process are as follows:

(1) *Planning*: In this step, a shell developer needs to make a series of system-level design decisions, including:
   (a) **Resource partitioning**: Based on the FPGA fabric layout and the number of resources available, a shell developer needs to split the FPGA fabric into two parts: 1) the static region for the FPGA shell and 2) one or more reconfigurable regions for hosting hardware

[2]This is not a strict requirement and the tool GoAhead [3] used to implement the shell can be used in a mode that allows static routing to cross reconfigurable regions.

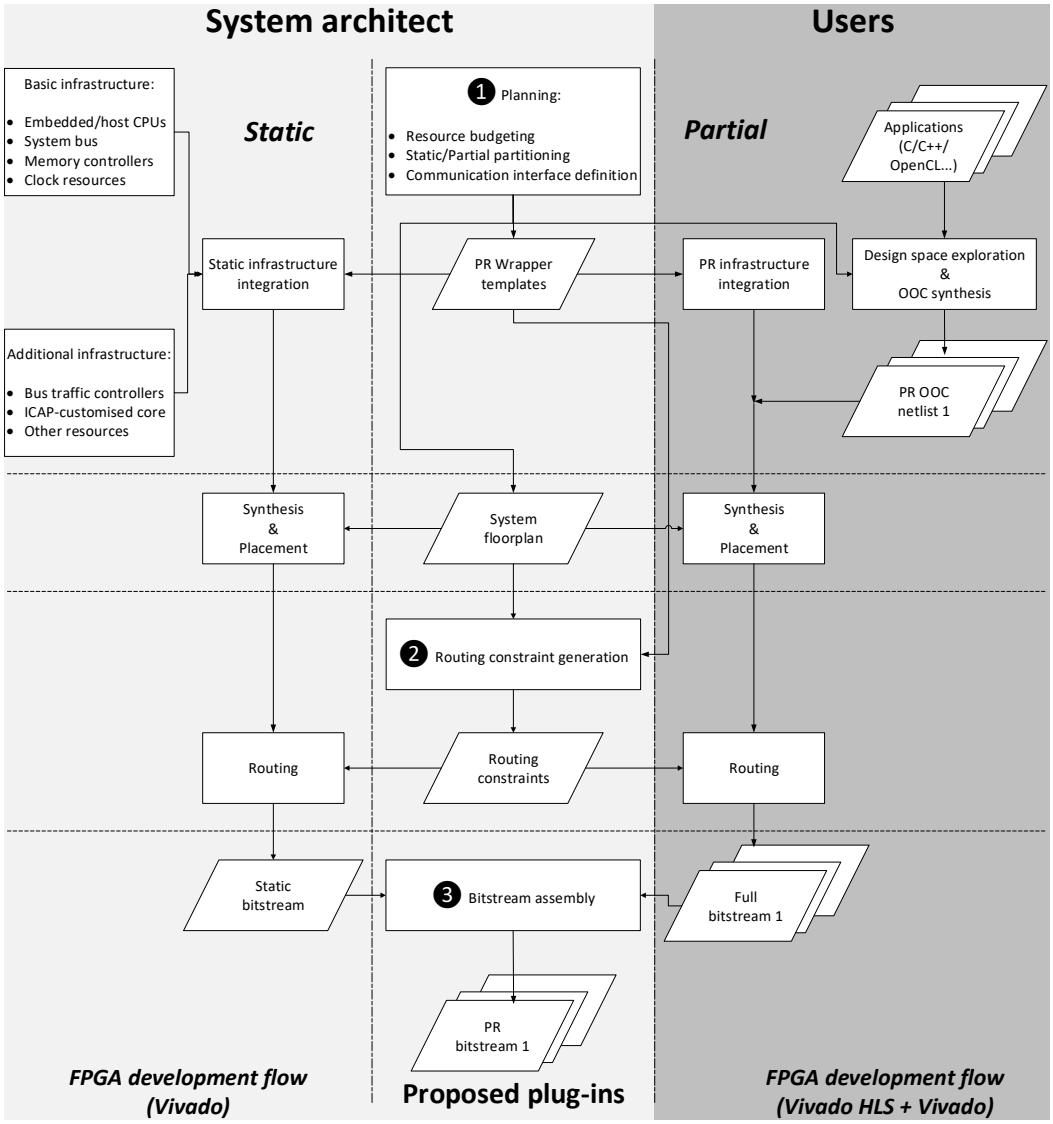


Fig. 6. The proposed design methodology. The left part is performed once for a specific version of a shell by the system architect while the right part (with the darker background) is performed once for each module. Modules do not have to be aware of the shell and are compiles in out of context (OOC) mode.

accelerators. This also defines the size of each PR region on which the High-Level Synthesis (HLS) tools [11] can perform the Design Space Exploration (DSE) for throughput optimization [24]. The trade-off to consider at this stage requires the developer to allocate the maximum number of resources to the reconfigurable part while leaving sufficient resources to the static part for shell functionalities and future upgrades. This trade-off is system-dependent and requires a thorough understanding of the target FPGA device and system requirements. However, this step is only performed once for a particular system.

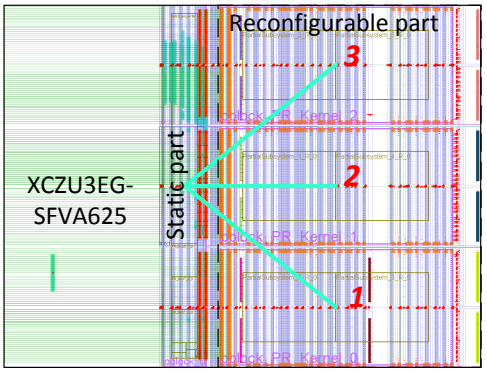


Fig. 7. The physical shell implementation on the UltraZed and Ultra96 boards. This version has three PR regions which can host up to three FPGA applications simultaneously [6].

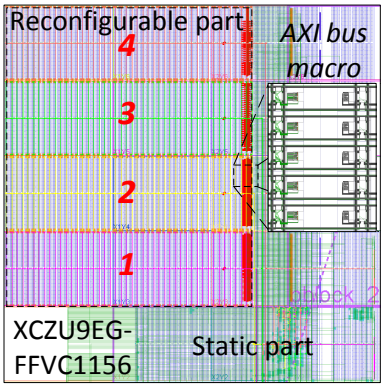


Fig. 8. The physical shell implementation on the ZCU102 board. This version has four PR slots which can host up to four FPGA applications simultaneously [16].

The result of this process is 1) the static system floorplan and 2) bounding boxes of the reconfigurable regions.

(b) **Communication interface definition**: Selection of the protocol and data-widths for the communication between the static system and the partial regions based on the system requirements.

(2) *Routing Constraint Generation*: The system floorplan and the PR interface template from the previous step are used to generate routing constraints. We can describe the implementation rules with the help of academic PR frameworks (GoAhead [3] and a TCL library [37]) in the form of TCL files for routing constraints automatically.

(3) *Configuration Bitstream Generation*: The proposed flow results in *full static bitstreams* for both shell and module designs. To compose *partial bitstreams* for accelerator modules from these bitstreams, we use the bitstream manipulation tool BitMan [26], as detailed in Section 4.1.3.

4.1.1 Shell Development. The static design starts with integrating basic infrastructure IP and additional infrastructure to a top-level unified design. In our designs, the basic infrastructure includes 64-bit ARM Cortex-A53 CPU cores, AXI4 interconnects, memory controllers, Xilinx PR Decouplers for disabling/enabling static and module communication, clock management tiles for tuning module frequencies, and PR Module Interfaces. It is possible to add other resources, such as memory management or network communication IPs, if required.

The PR Module Interface provides an AXI4-Lite slave for control register access via the CPU and an AXI4 master for memory access. This fixed interface between the hardware module and the static system is implemented using the available routing resources in the Zynq UltraScale+ FPGA devices. In particular, we keep this interface identical for all PR regions to serve relocatable hardware modules by pre-placing and pre-routing these communication signals in a constrained, predefined manner. This reassembles the bus macro approach, which was very popular for Xilinx Virtex-II devices [7]. However, our flow can implement this without logic cost (in terms of LUTs used for the bus macro).

To keep the clocking resources of PR regions identical for relocatable hardware modules, we block all routing wires except for a defined subset of the `BUFCE_LEAF` primitives inside the PR regions. This forces the router to use only a defined subset of these clock driver primitives that each drive a specific vertical clock spline, which ultimately connects to the flops, BRAMs, and DSPs in

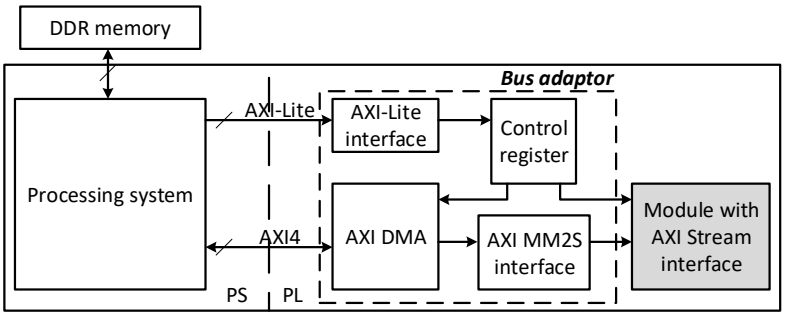


Fig. 9. An example for bus virtualisation: the module has a 32-bit AXI-Lite interface and a 32-bit AXI Stream interface without DMA engine. In this case, the *bus adaptor* with AXI DMA and AXI MM2S IPs are chosen to carry out the communication with the rest of system.

the region. However, we only route the clock for the PR regions this way. This means when routing the static system, we 1) route the PR module clocks with `prohibit` constraints on the `BUFCE_LEAF` primitives, then 2) remove these constraints, and 3) incrementally route the rest of the system. This allows routing additional clock nets as needed by the static system (e.g., for providing clocks to memory controllers or gigabit transceivers).

Finally, to prevent any static signal from violating PR regions, we insert a blocker macro. This blocker is non-functional but uses all local wire resources inside the reconfigurable regions before routing the static system. We generate this blocker macro using GoAhead [3] or the TedTCL library [37] according to the system floorplan.

The above steps result in the final static physical implementation, as shown in Figure 7 for the UltraZed/Ultra96 board and in Figure 8 for the ZCU102 board. We can see that the PR region interfaces have the same relative physical positions and the distribution of clock splines across all PR regions is identical. Both systems support combining multiple adjacent regions for hosting larger monolithic modules. In this case, we only use one PR module interface.

4.1.2 Bus Virtualisation. Operating a hardware accelerator needs communication with the host CPU to issue commands and access to DDR memory for the actual data processing. In the here implemented system, a module can use a wide range of bus widths (such as 32/64/128-bit width) and various bus protocols (such as AXI4 Master/Stream). In particular, HLS modules, often by default, include DMA engines for fetching data from memory and for writing back results. However, this is not always the case with hand-crafted RTL or customized netlist accelerators which may operate on data streams rather than data stored in DDR memory.

We tackle this issue by providing another level of abstraction for bus interfaces between the FPGA applications and shells. For this, we selected the interface to provide the 32-bit AXI-Lite protocol and the 128-bit AXI4 protocol at the shell (static) side. The decision for 128-bit is based on the size available in Zynq UltraScale+ FPGAs that provide this size between DDR memory controller and the FPGA fabric. Depending on the exact physical interface required by a module, we instantiate a module wrapper with a set of *bus adaptors* such that a module can communicate with the rest of the system as required by the individual FPGA modules. Figure 9 shows a *bus adaptor* being used to translate between different AXI bus standards. We can perform this wrapping at design time (where we instantiate the wrapper transparently when designing the module) or at runtime with partial reconfiguration (where the bus adaptor is a partially reconfigurable module located between the static system infrastructure and the accelerator module). The bus adaptor uses mostly IP components provided by the FPGA vendor Xilinx [11]. These components are then

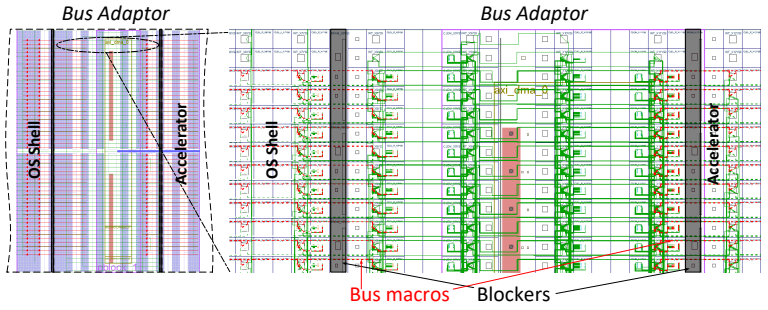


Fig. 10. Implementation of a bus abstraction layer on UltraZed/Ultra96 platforms. The bus adaptor is provided as an implemented module bitstream and stitched into the shell at run-time by using partial reconfiguration. The adaptor is a partial module that, in turn, interfaces to other partial modules. This technique avoids re-compiling of bus adaptors but comprises an area overhead for a partial region to host the bus adaptor.

automatically parameterized and integrated according to the specific interface requirements of the accelerator module. With this, shells can remain light-weight, operational, and unchanged while supporting a wide range of accelerator module AXI interfaces.

It is important to understand that a static acceleration system would also use such bus adaptors provided by the vendor. Hence, our bus adaptors do not necessarily cause additional overhead unless changing it at runtime, which requires pre-allocation of resources. The advantage of the here proposed *bus adaptor* concept is that an adaptor is only integrated into a module if needed and not speculatively provided by the shell.

We also use a bus adaptor to translate between AXI Master and AXI Stream protocols. FOS provides different versions of AXI Stream adaptors to be used depending on the AXI Stream channel width. A user can either re-compile their modules with a logical wrapper of the bus adaptor at design time or stitch their modules with a pre-built partial configuration bitstream binary of that bus adaptor at runtime. Figure 10 shows the implementation of the bus adaptor with its logic for interfacing an accelerator with the AXI protocol, AXI MM2S, and AXI DMA services for a module which has a 32-bit AXI-Lite and a 32-bit AXI Stream interface.

4.1.3 Module Compilation. The module design begins from either high-level language (HLL) source code (C, C++, OpenCL etc.) or a hardware description (RTL/netlist). In the case of HLL source code, we go through a High-Level Synthesis (HLS) step [11] to generate the RTL source codes. We then synthesize the resulting RTL source code in out-of-context (OOC) mode. In the Xilinx Vivado tools, OOC allows implementing our accelerator modules bottom-up meaning that no optimizations are performed across module boundaries, which is not applicable to partially reconfigurable modules.

The PR Wrapper templates are used to create a minimal top-level placeholder for the physical module implementation. This temporary placeholder acts as sink/source connection points and substitutes the surrounding static system. We then integrate the module OOC netlist into this placeholder for the synthesis and placement stages.

We generate blockers (as TCL routing constraints) to enforce that all partial module's primitives and routing resources are following the strict implementation rules mentioned in Section 4.1 (by using GoAhead [3] or TedTCL library [37]). In contrast to the static system where blockers are placed inside reconfigurable regions, here, the blockers are placed around the module as a fence for implementing hard module bounding box constraints. The blockers include routing tunnels for the communication to and from the temporary placeholder. The position of these tunnels matches exactly the tunnels used in the static design to implement the communication between static and partial areas. The placeholders and blockers stay the same for all modules requiring the same

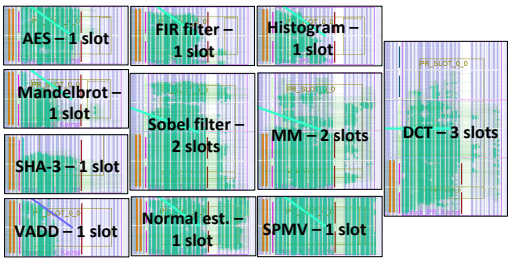


Fig. 11. Compiled modules for Spector benchmark suite [27] and our in-house accelerators for Ultra-96 board.

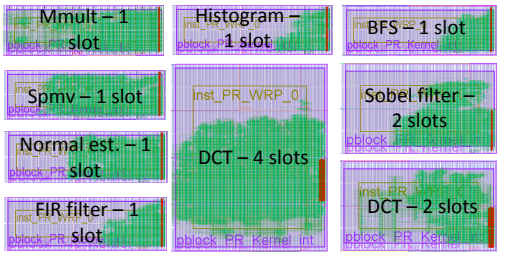


Fig. 12. Compiled modules for Spector benchmark suite [27] accelerators for ZCU102 board.

number of resource slots. For convenience, pre-implemented templates (one for each number of possible slots) are provided in our FOS distribution for rapid implementation of accelerator modules.

As we implement a module in separation from the static system, the result generated by Vivado is a full configuration bitstream. We pass this full bitstream to BitMan [26] to extract the configuration data that corresponds to the module only as a partial bitstream. At run-time, BitMan manipulates those partial bitstreams to relocate modules to the desired partial region of the static system. Figure 11 and 12 shows the resulting modules of the Spector benchmark suite [27] and additional accelerators for Ultra-96 and ZCU102 boards, respectively.

4.2 Logical Hardware Abstraction

This is the first layer between the hardware and software infrastructure. It is designed to hide differences or changes in the hardware from the software, in order to detach the software infrastructure from the underlying hardware layer as much as possible. To achieve this conveniently, we propose describing the shell in terms of logical functionalities using a JSON file description providing the following information (see listing 1 for an example):

(1) Name of the shell
(2) Bitstream name of the shell
(3) Partial region:
  (a) Name of the partial region
  (b) Blanking bitstream for the partial region
  (c) AXI bridge decoupler address
  (d) Base address of an accelerator placed in the region

Similarly, for the accelerator, we defined a JSON file description (see Listing 2) with the following details for accelerator programming and management:

(1) Name of the accelerator
(2) Bitstreams:
  (a) Name of the bitstream
  (b) Name of the shell it is compiled for
  (c) Type of AXI interface used (if not AXI4 master and slave)
  (d) Name and number of the PR regions it is compiled for
(3) Register mappings:
  (a) Name of the HW register
  (b) Address offset at which the HW register can be access

Listing 1. JSON description example of a shell.

```
{
  "name": "Ultra96_100MHz_2",
  "bitfile": "Ultra96_100MHz_2.bin",
  "regions": [
    {"name": "pr0", "blank": "Blanking_slot_0.bin", "bridge": "0xa0010000", "addr": "0xa0000000"},
    {"name": "pr1", "blank": "Blanking_slot_1.bin", "bridge": "0xa0020000", "addr": "0xa0001000"},
    {"name": "pr2", "blank": "Blanking_slot_2.bin", "bridge": "0xa0030000", "addr": "0xa0002000"}
  ]
}
```

Given that the control register map follows the standard Vivado HLS [11] interface (see Listing 3), an accelerator can have an arbitrary number of 32-bit registers. This allows building generic drivers for accelerators to relieve hardware developers from the responsibility of writing and integrating drivers. Hence, with this logical hardware abstraction, shells or accelerators can be arbitrarily updated with no need to recompile the Linux kernel or drivers. Section 4.3 will describe the driver and acceleration library interface.

The name of the PR region for each bitstream allows backward compatibility to the Xilinx PR flow, where we must compile an accelerator for each region (no relocation support). Moreover, the name of the accelerator is unique, but it may contain bitstreams of varying sizes corresponding to differently sized accelerator module implementations providing the same functionality. The scheduler can later use these implementation alternatives to perform resource-elastic scheduling, i.e. dynamically changing the resource allocation used by the accelerator based on the workload and available resources at runtime.

We then register these JSON descriptions for shell and accelerators into a JSON based registry to enable a centralized view of the available hardware to the upper software layers. This allows the application developers or the runtime system to request hardware based on just a logical name (a logical accelerator functionality) and corresponding input data, without needing any further information about the underlying hardware layer and accelerator implementation.

Note that we can automatically generate the JSON descriptor for accelerators generated by the Vivado HLS compilation flow [11]. Whereas, the shell developer must write the JSON description to allow the runtime system to manage the resource allocation and use generic drivers. However, this process is commonly required only once per system and can be omitted when using the pre-built shells provided with FOS.

Listing 2. JSON description example of a vector add accelerator.

```
{
  "name": "vadd",
  "bitfiles": [
    {"name": "vadd.bin", "shell": "Ultra96",
     "region": ["pr0", "pr1"]},
  ],
  "registers": [
    {"name": "control", "offset": "0"},
    {"name": "a_op", "offset": "0x10"},
    {"name": "b_op", "offset": "0x18"},
    {"name": "c_out", "offset": "0x20"},
  ]
}
```

Listing 3. Control bits for the accelerator.

```
// 0x00 : Control signals
//        bit 0 - ap_start (Read/Write/COH)
//        bit 1 - ap_done (Read/COR)
//        bit 2 - ap_idle (Read)
//        bit 3 - ap_ready (Read)
//        bit 7 - auto_restart (Read/Write)
//        others - reserved
```

4.3 Acceleration Interface Libraries

In general, an OS has to provide high-level APIs that can be used from existing software stacks to improve the accessibility of FPGAs to software developers (who are often non-FPGA experts). One such effort is the PYNQ [43] framework from Xilinx, which allows accessing FPGA accelerators through a high-level Python API. However, the current implementation of PYNQ relies on an existing vendor development flow and contains many direct dependencies on artifacts produced by the Vivado compilation flow, which restrict modular development [11, 43]. For instance, it requires the *entire block diagram* of the shell to enable programming some vendor IPs, and this creates a strong direct dependency between shell implementation and runtime environment. While it is possible to engineer around these shortcomings, the resulting system would suffer from code inflation and would be non-scalable due to its legacy support. Hence, we built new lightweight acceleration interface libraries called Ponq and Cynq for Python and C++ languages, respectively. These libraries are built based on a modular development flow and provide 1) access to static and dynamic FPGA accelerators via its generic drivers for programming accelerators, 2) the Xilinx FPGA manager for partial reconfiguration [44], 3) memory-mapped I/O (MMIO) modules for direct access to accelerators, and 4) a data manager for i) contiguous physical memory allocation and ii) virtual memory protection based on the system MMU which is available on-chip [6] and which provides memory encapsulation for the accelerators. For providing extra hardware security against side-channel or power hammering attacks, accelerator configuration bitstreams can be checked with FPGADefender [17, 19]. Figure 13 shows the integration of Ponq and Cynq libraries into the FOS modular development flow and existing high-level software libraries.

The libraries are backward compatible with the standard Xilinx development flow, i.e. both the PR flow and the static acceleration environment, as we built them on top of the logical hardware abstraction layer. The libraries provide the following basic HAL functionality and generic drivers for hardware acceleration:

- Load an FPGA shell
- Load a partially reconfigurable accelerator
- Load a static accelerator
- Load and program an accelerator based on a logical function name
- Program an accelerator for execution via generic drivers
- Read and write calls to HW registers of the accelerators
- Contiguous physical memory allocation

These functionalities can coexist with other legacy drivers for both the shell or some accelerator modules.

4.4 Runtime and Multi-tenancy API

To support multi-tenancy, a runtime system is necessary to arbitrate access to reconfigurable resources between multiple users *transparently*. The conventional approach for allocation is to employ time-domain multiplexing with a run-to-completion model, but there had also been spatial domain scheduling mechanisms proposed for FPGAs [20, 34]. These approaches base the spatial domain scheduler on pipeline replication, however, only supporting a specific type of accelerator, such as OpenCL or DSL based accelerators. In contrast, an operating system must support all types of accelerators and execute them concurrently while using space-time domain scheduling. To make this possible, we need three main components: an API to a daemon scheduler API, a programming model, and an actual scheduler, as discussed in the following paragraphs.

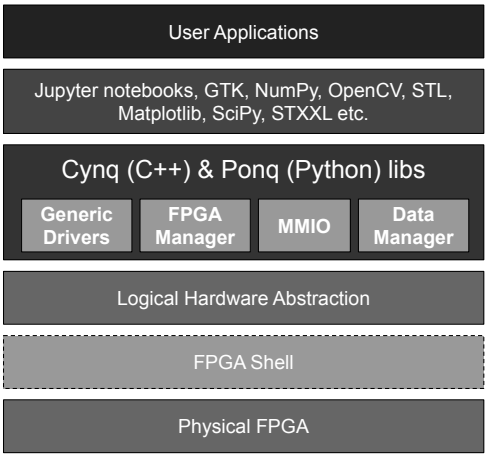


Fig. 13. Cynq and Ponq libraries as an acceleration interface layer for static and dynamic acceleration on FPGAs.

Listing 4. C++ daemon execution call example.

```
// Create a job
Job &job = jobs.emplace_back();
job.accname = "Partial_accel_vadd";

// Set accelerator parameters
job.params["a_op"] = a_op_buffer_ptr;
job.params["b_op"] = b_op_buffer_ptr;
job.params["c_out"] = c_out_buffer_ptr;

// Launch jobs
fpgaRpc.Run(job);
```

Listing 5. Python daemon execution call example.

```
# Create a job and set accelerator parameters
jobs = [{
    "name": "Partial_accel_vadd",
    "params": {
        "a_op": a_op_buffer_ptr,
        "b_op": b_op_buffer_ptr,
        "c_out": c_out_buffer_ptr,
    }]

# set accel parameters and run hardware unit
fpga_rpc.Run(jobs)
```

4.4.1 Daemon Scheduler API. To truly support multi-tenancy with portability, we need to design an API that can span across multiple languages and which is portable to different OS kernels or a base processor system with ease. The two efficient ways to perform this inter-process communication (IPC) are 1) message passing and 2) shared memory.

In our platform, we adopt the gRPC framework [12], which is a standard RPC framework with support for multi-languages. We use gRPC to send the acceleration requests from the client process to the daemon process, whereas, we pass the data via shared memory to avoid additional latency of copying data (i.e. zero-copy operation). The adoption of gRPC allows us to extend the runtime to accept acceleration requests from remote nodes in the future. The final interface exposed to the application developer, in C++ and Python, is shown in the examples in Listing 4 and Listing 5. Note that each user can offload multiple data-parallel acceleration requests in a single RPC call to the daemon.

4.4.2 Programming Model. Dynamic resource allocation is achieved in the spatial-domain by using resource-elasticity [34] in two forms: module replication and module replacement. However, to make this possible for many types of accelerators, we allow applications to expose data-parallelism to the scheduler, i.e. an application developer can choose to express an acceleration job into a varying degree of parallelism appropriate for the application. A typical example of this is partitioning an image into multiple parts for image-processing accelerators. Note that this is analogous to a software

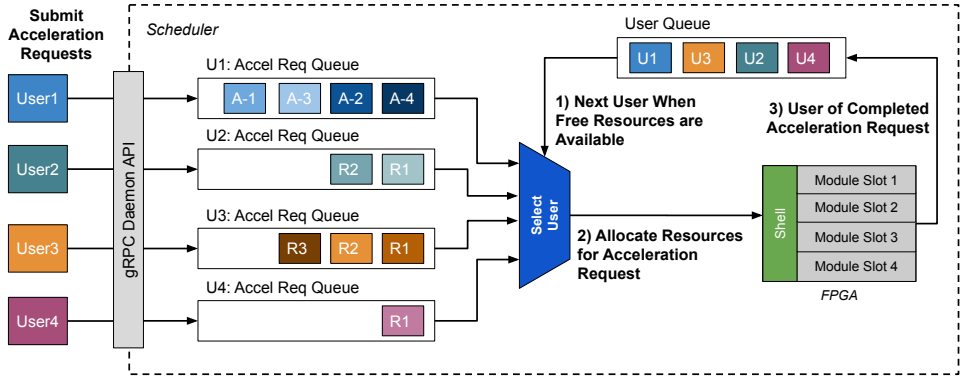


Fig. 14. Scheduler organization for resource allocation between different users and their data-parallel acceleration requests.

developer deciding for the number of threads for efficient execution on a multi-core system. The runtime is then responsible for executing those parts i) in parallel, ii) use a better implementation, or iii) in case of exceeding FPGA resource capacity, perform time-domain multiplexing of the resources.

4.4.3 Scheduling. The scheduler maintains a queue for each user and performs round-robin scheduling between users at a coarse granularity of data-parallel acceleration requests, as shown in Figure 14. Each user has an individual queue of acceleration requests. Each request in this queue is independent of other requests in the queue and can execute in parallel and in any order. At the end of each acceleration request, the scheduler relinquishes the accelerator and selects an acceleration request from the next user in the queue. This scheme implements a cooperative scheduling policy (by breaking down a job into fine-grained run-to-completion acceleration requests) where each request includes fetching operands and writing back results to main memory (DDR). This corresponds directly to the OpenCL programming model, where work-groups can execute in any order. Note that the *scheduling granularity* depends on the execution latency of fine-grained acceleration requests rather than fixed timing intervals like in preemptive scheduling.

In the case there is no other user, the scheduler executes requests from the same user in parallel and attempts to use the biggest module (assuming that the biggest module provides the highest performance per resources used, i.e. Pareto-optimal) to maximize the utilization and performance. Moreover, the scheduler avoids partial reconfiguration and reuses an accelerator if it is already available on-chip. This allows multiple different applications that require acceleration of the same functionality to share the same accelerator in time without paying an additional configuration penalty or user effort.

Consequently, a single task execution call may execute on multiple accelerators in parallel or use an implementation alternative or share an accelerator with other tasks in time, during its execution lifetime. All these modes can be used arbitrarily without an application being aware of other tasks and types of accelerators it executes on. Hence, the runtime system is responsible for dynamically arranging the loading and unloading of these heterogeneous accelerators (written in C, C++, OpenCL or RTL) transparently from the user. Figure 15 shows an example of how such resource allocation can allow maximizing the FPGA utilization and performance compared to standard fixed-module scheduling policies.

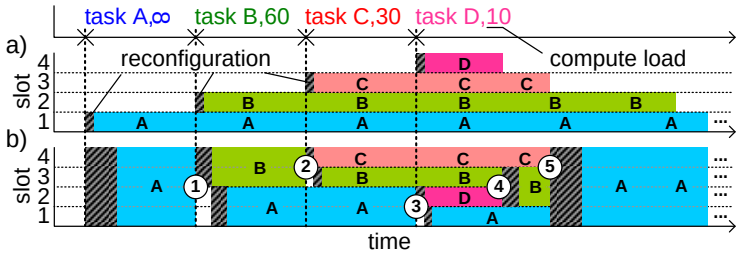


Fig. 15. Resource allocation for kernels (tasks) A, B, C and D in time when using a) Standard fixed module scheduling and b) Resource-elastic scheduling on a 4 PR region FOS FPGA shell. The circled events highlight cases where resources are needed to accommodate new arriving tasks (①, ②, ③) or cases where tasks complete (④, ⑤).

5 EVALUATION

There are five primary dimensions in which we can evaluate an FPGA operating system: 1) FPGA resource overhead, 2) software stack overhead, 3) available memory performance, 4) level of modularity and 5) application performance. We detail and discuss the performance of FOS on each dimension for Ultra-96 and ZCU102 boards in the following paragraphs.

5.1 FPGA Resource Overhead

5.1.1 FPGA shell. The resources used by an FPGA shell have a direct impact on the FPGA resources available for user hardware accelerators. Hence, it is crucial to minimize the overhead as much as possible. Table 1 shows the resources available for hardware acceleration when using FOS on ZCU102, Ultra-96, and UltraZed boards. For the ZCU102 board (an MPSoC development kit from the FPGA vendor Xilinx), around 50% of the resources are available for user acceleration, whereas on a small IoT category Ultra-96 board, it is about 80%. This is because the layout of the ZCU102's chip (XCZU9EG) is irregular, limiting the available resources when supporting relocatable modules (see Figure 8). With a regular resource layout like in the Ultra-96's chip (a XCZU3EG, see Figure 7), we can maximize the resource allocation for relocatable modules and considerably reduce the resource overhead of a FOS shell. However, it is important to note that the resources of the shell are not entirely wasted. They are available for future extensions as well as to implement other host system components, for example, static accelerators or I/O functionalities.

5.1.2 Bus Virtualisation. In order to achieve modularity at the hardware interface layer, bus virtualization (see 4.1.2) is vital. However, in order to be able to dynamically load the interconnect wrappers implies pre-allocation of a partial region. Table 2 shows the overhead of this pre-allocation of resources. We can identify that only about 18% (448 LUTs) of the pre-allocated partial region is unused when we change the interconnection interface and protocol considerable, such as from AXI Stream to AXI master and slave. In particular for large FPGAs, this overhead is negligible. However, when using small FPGAs such as on the Ultra-96 or performing minor changes to the interface (e.g., changing bus width), the overhead of dynamic bus virtualization is considerable (assuming the same partial region allocation). Hence, in such scenarios, we recommend using compile-time bus wrappers.

5.2 Software Stack Overhead

The software stack of an FPGA OS incurs two types of latencies: compile-time and runtime. Compile-time latency relates to the time taken to compile hardware accelerators with all the additional

Table 1. Available resources for acceleration on the ZCU102 (XCZU9EG) platform and the UltraZed & Ultra96 (XCZU3EG) platforms. The version on ZCU102 has 4 PR regions in total, while the other platforms provide 3 PR regions in total.

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

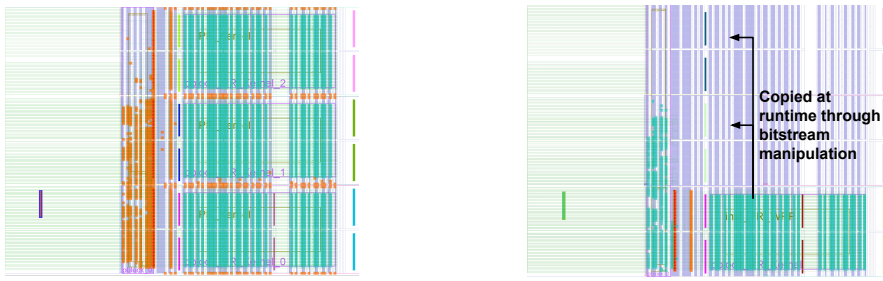
Table 2. Resource overheads for bus virtualization at the logical and physical levels.

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

constraints for partial reconfiguration in the implementation phase, while the other relates to the overhead caused by intermediate layers in the software stack during accelerator execution.

5.2.1 Compilation Latency. The standard Xilinx partial reconfiguration (PR) flow requires compiling both static and accelerator designs together and, more importantly, it needs to perform place and route (P&R) and the generation of a dedicated bitstream for each partial region. This leads to additional latency compared to our decoupled compilation flow, where we first generate a *full-static* bitstream with Vivado and then a *relocatable* partial bitstream with BitMan [26] for all regions. To quantify this, we used accelerators of three different types of size: sparse (AES), medium (Normal Est. [27]) and dense (Black Scholes [22]). The utilization for each is 33%, 63% and 81%, respectively. Figure 11 shows the AES and Normal Est. modules while the Black Scholes module is shown in Figure 16 with a comparison to a design generated by the Xilinx PR flow.

Table 3 shows the latency breakdown for place and route and bitstream generation for both Xilinx PR flow and FOS compilation flow on Ultra-96. The results show that P&R latency per region is higher for FOS as it adds additional constraints for supporting relocatability. However, when compiling for multiple regions (i.e. 3 for Ultra-96 platform), it outperforms the traditional compilation flow (by up to 2.34×) by duplicating a single slot accelerator. Overall, when increasing the number of partial regions, the compilation flow latency of FOS stays constant, whereas the latency of the Xilinx PR flow increases linearly. This relationship turns into an exponential increase



(a) Xilinx PR compilation flow result.

(b) FOS compilation flow result.

Fig. 16. Place and route result of Black Scholes accelerator [22] for Xilinx PR flow and FOS. Note, FOS uses BitMan [26] to generate a relocatable bitstream which is copied to the other PR regions.

Table 3. Total place and route (P&R) times, and bitstream generation latency for AES, Normal Est. [27], and Black Scholes accelerator [22] when compiling for all three partial regions on Ultra-96 shell. The evaluation is conducted using Vivado 2018.2.1 on an Intel core i7-4930K CPU running at 3.4 GHz with 64 GB of RAM.

Applications	Region Util.	Xilinx PR (3x one region)			FOS (one region)			Speed Up
		P&R (s)	Bitgen (s)	Total (s)	P&R (s)	Bitgen (s)	Total (s)	
AES	33%	429.40	176.19	605.59	284.18	64.06	348.24	1.74 ×
Normal Est.	63%	747.75	201.21	948.96	387.41	70.09	457.50	2.07 ×
Black Scholes	81%	1296.26	231.27	1527.53	574.56	77.11	651.67	2.34 ×

in compilation time when compiling several applications with standard Xilinx PR flow, as it needs to compile *each module for each partial region*. Hence, it is important to adopt a scalable and modular PR flow when targeting multiple applications and shell versions which are common in datacenter environments.

5.2.2 Runtime Execution Overhead. The runtime overhead occurs because of the four main steps performed when using the FOS software stack: i) initialization of the gRPC server, ii) parsing of JSON files for accelerators and shells, iii) gRPC calls to the daemon and iv) scheduling latency. Table 4 details the latency of each step. The first two steps (i and ii) have dependencies on I/O as they use the network and file system, respectively. This leads to latencies in the range of milliseconds. However, this overhead is amortized over time as we perform steps i) and ii) only once at system start. The gRPC call to the daemon goes through many levels of the Linux stack (processes) before reaching the FOS multi-tenancy daemon, leading to a latency of about one millisecond. We can speed up the gRPC call by reducing the Linux timer interrupt period to achieve a better response time from the Linux kernel for a quick turnaround between processes if required. The scheduling latency is in the range of microseconds and comparable to standard CPU schedulers. Moreover, the scheduler is event-driven and executed only when an accelerator finishes or a new acceleration request arrives (due to its cooperative nature) rather than at every timer interrupt like preemptive scheduling.

5.3 Memory Performance

One central characteristic of an FPGA operating system on the hardware acceleration performance is the memory bandwidth that it can provide given the shell implementation (interconnect to

Table 4. Execution overhead caused by various software layers.

Software Layer	Latency (ms)
Initialize gRPC (once)	12.20
JSON parsing (once)	2.27
gRPC Call to Daemon	0.71
Scheduler	0.02

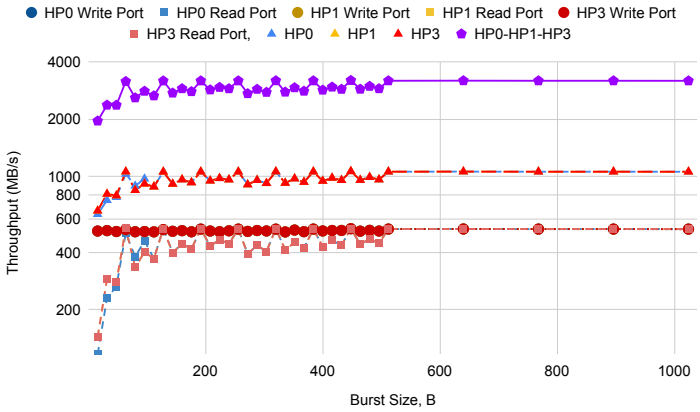


Fig. 17. Memory throughput for varying burst sizes on the 128-bit duplex High Performance AXI ports (HP0, HP1, and HP3) available to the accelerators in PR regions 0 to 2 of Ultra-96 FOS platform at 100 MHz.

the memory). Hence, we evaluated the available memory throughput of different 128-bit High-Performance AXI ports (HP ports) provided to partial regions on the FOS shells for the ZCU102 and Ultra-96 boards with accelerators running at 100 MHz using the memory evaluation kit proposed in [23]. In particular, we measured the performance of each port for 1) read only, 2) write only, and 3) aggregated read-write transactions of varying burst sizes from the PL (FPGA). Additionally, we examine the combined performance of all ports running in parallel.

Figure 17 shows the breakdown of the read and write throughput of each AXI port and the total accessible bandwidth for the Ultra-96 board, respectively. On average, there is an even split of read and write bandwidth with a throughput of 530 MB/s for each read and write operation. The aggregated read-write throughput of the individual AXI ports is about 1060 MB/s; and when activating all three ports at the same time, collectively they achieve up to 3187 MB/s. This translates to about 25% and 74% of the theoretical DDR peak throughput when using AXI ports individually and concurrently, respectively.

Figure 18 shows the breakdown of read and write throughput on the ZCU102 board. Each AXI port achieves an even throughput distribution of 1600 MB/s between read and write transactions. The total throughput is 3200 MB/s for individual AXI ports and 8804 MB/s when using all AXIs together, as shown in Figure 18. A possible explanation for this sub-linear improvement in the total throughput when using all AXI ports (HP0-3) concurrently is because of the row pollution and AXI interconnect multiplexing in the memory controller. This result also reveals that the shell is saturating the memory throughput that is available through the ARM SoC and that it is not worth to spend more resources in the shell to implement wider backplane buses.

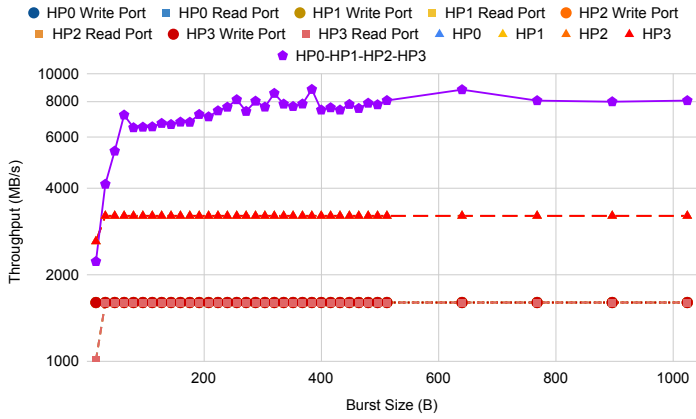


Fig. 18. Memory throughput for varying burst sizes on the 128-bit duplex High Performance AXI ports (HP0 to HP3) available to the accelerators in PR regions 0 to 3 of ZCU102 FOS platform at 100 MHz.

Table 5. Re-initialization latencies for component changes on FOS platforms. Note, in contrast to FOS, for any change in these components, the standard PetaLinux flow [44] requires a generation of a new Linux image, and hence, a complete reboot and re-initialization of all components even after excluding compilation latencies.

Component Updated	U-96 Latency (ms)	ZCU102 Latency (ms)
Accelerator	3.81	6.77
Shell	20.74	98.4
Runtime	15.2	15.2
Kernel	66000[3]	15760

5.4 Quantifying Modularity

Compared to the standard FPGA development flow, where a change in the shell means recompilation of all system components (both hardware and software) [32, 44], the modular FOS FPGA development flow provides the freedom to *update individual components* without recompilation of other components which may take hours [5, 11, 44]. This means that FOS only has to pay compilation and re-initialization latency (as shown in Table 5) for the changed component, given that it does not modify the component's defined interface.

From Table 5, we can identify that this avoidance of recompilation allows changing the shell at runtime with additional functionalities or bug fixes costs less than 21 ms on Ultra-96 (IoT device) and 99 ms on ZCU102 (Xilinx MPSoC development kit). Similarly, swapping an accelerator implementation is easy and includes only the partial reconfiguration latency, because FOS provides generic drivers. In contrast to FOS, the standard flow would require generating/writing new drivers and re-installing them separately. Changing the kernel involves the most significant re-initialization latency, as this includes a system reboot which takes 66 seconds, including I/O setup (for keyboard, mouse, Wi-Fi, monitor, and webcam) on Ultra-96[3]. However, this still avoids the need to compile

[3]The Ultra-96 BSP (board support package) and vendor drivers included for Wi-Fi and Bluetooth have a bug which causes a time-out during boot up which is inflating reboot time. After boot-up, our work-around in the Ubuntu setup fixes the bug in order to operate Wi-Fi correctly.

user software binaries for re-integration as required in the standard PetaLinux flow (causing latency in tens of minutes) [44].

Overall, the modularity of FOS removes the recompilation latencies and re-development steps (i.e. mandatory recompilation of all shell, accelerators, Linux and user applications caused by Vivado [11] and PetaLinux [44] for any changes in a single component like the shell) which cost in the range of hours [5]. This reduces the component change latency by two-orders of magnitude compared to the standard development flow while supporting all the features required from an FPGA OS.

5.5 Application Case Study

We evaluated a case-study in two different environments: 1) single-tenant but multiple partial regions and 2) multi-tenant with dynamic offloading. All the accelerators used in this case study operate at the default frequency of ZUCL 2.0 [6] shells of 100 MHz as a baseline environment.

5.5.1 Single-tenant with Multiple Partial Regions. To evaluate the benefits of multiple partial regions and the ability to replicate accelerators dynamically, we selected OpenCL accelerators from the Spector benchmark suite [27] for three reasons: 1) the benchmark comprises a wide range of application behaviors from multiple application domains, 2) it is annotated with HLS pragmas to generate implementation alternatives without modifying the source code, and 3) it demonstrates that third party accelerator modules can be compiled for FOS without further modifications. Figure 19 shows the results of the execution latencies with a varying number of resources available for acceleration on the ZCU102 platform. Most of the benchmark applications show an almost *linear* performance improvement when replicated across multiple partial regions. This is expected from our memory evaluation experiments (in Section 5.3) which show that multiple ports can be used in parallel without major performance drops on individual memory port performance. The effect of implementation alternatives can be seen for the DCT accelerator which benefits from the ability to switch to a bigger module implementation (by using larger data buffers and a larger unrolling factor in the bigger alternative) and which achieves a *super-linear* performance improvement of 3.55× for 2× the resources.

To understand the effects of exposing more parallelism than available on the FPGA platform, we conducted the experiments on Ultra-96 using compute-bound (Mandelbrot and Black Scholes [22]) and memory-bound (Sobel [46]) applications which are more sensitive to reconfiguration overhead due to their smaller execution latencies than Spector benchmarks [27]. Figure 20 shows the results in which the number of requests dictates the amount of parallelism exposed to the runtime system. The performance improves almost linearly until it hits the number of available partial regions (3 in the case of Ultra-96 platform), after which the performance tends to stagnate (see Figure 21). This is because the scheduler uses time multiplexing to provide the illusion of an unlimited number of regions, leading to a behavior similar to multi-threading on CPUs. In particular for cases where the number of requests is a multiple of the number of physical accelerators, we see better performance than in other cases as they avoid unused resources that are caused by the (leftover) pending requests at the end of the execution of a job queue.

Overall, this highlights that FOS can help to improve performance (by up to 2.54×) even when using a single application along with its other benefits of modularity and developer productivity.

5.5.2 Multi-tenant Dynamic Offload. To understand the performance changes when using multiple applications simultaneously, we execute the Mandelbrot and Sobel [46] applications concurrently on the Ultra-96 platform. Note that individual applications are not aware of other applications executing on the platform. In particular, the accelerators that are written and accessed (at runtime)

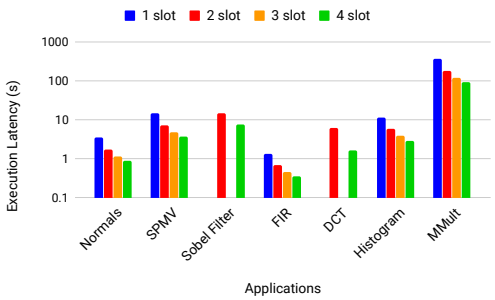


Fig. 19. Execution latencies of accelerators from the Spector benchmark suite running on the ZCU102 platform.

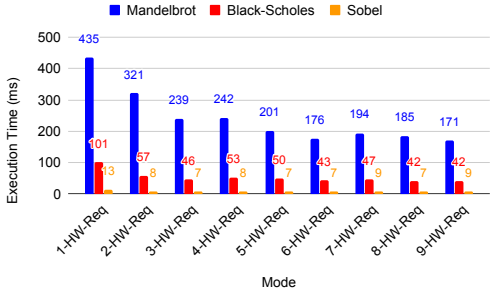


Fig. 20. Execution latencies of Mandelbrot, Black Scholes (European option) [22], and Sobel [46] when executing concurrently with varying the number of hardware requests on the Ultra-96 platform.

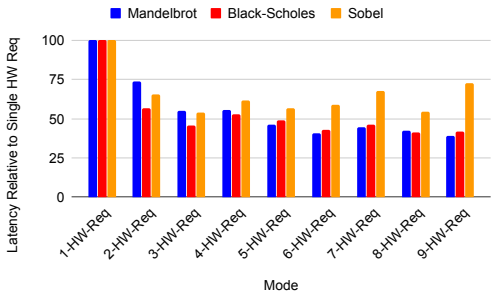


Fig. 21. Relative execution latencies of Mandelbrot, Black Scholes (European option) [22], and Sobel [46] applications when exposing varying amounts of parallelism to process a frame on the Ultra-96 platform.

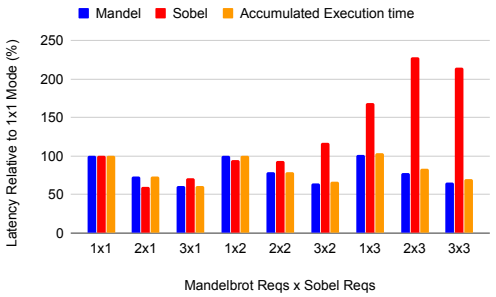


Fig. 22. Relative execution latencies of Mandelbrot and Sobel [46] when executing concurrently with a varying number of HW requests on the Ultra-96 platform.

in C (Mandelbrot) and OpenCL (Sobel), demonstrate the support for multiple different languages used *concurrently*.

Figure 22 shows the execution latencies relative to 1-Mandel×1-Sobel scenario with a varying number of acceleration requests. As we can see, the execution latencies tend to decrease with an increase in the number of requests as in the standalone case. However, here the optimal performance is achieved at 3-Mandel×1-Sobel rather than 3-Mandel×3-Sobel. This is because of two reasons: 1) adding more Sobel units reduces memory performance due to row-bank pollution and 2) multiple requests from different applications (over capacity) induces higher reconfiguration overhead for swapping them in and out because of FOS scheduler's fairness policy. Regardless of this behavior, it is essential to note that if each application takes a greedy decision to request the highest amount of parallelism suited for the application based on the standalone results, the system can still achieve a near-optimal performance resulting in 46% improvement over 1-Mandel×1-Sobel by dynamically reallocating resources using the same accelerators.

6 RELATED WORK

There have been many proposals for building an FPGA operating system or providing similar features as surveyed in [33]. Some of the very early contributions had been ReconOS [4] and

BORPH [30], where hardware accelerators were treated in the same manner as software processes and integrated into software operating systems. However, this level of abstraction introduces high overhead due to the communication synchronization and management hardware required to attain the sophisticated APIs of software processes at the hardware level.

To address this, recent proposals aim at function level abstraction and focus on hardware acceleration for software processes instead of making them their equal [2, 8–10, 14]. A good example of this is Chen et al. [9] which addressed this issue for cloud workloads by integrating an FPGA shell with virtual machine monitor and support for priority-based time-domain scheduling for accelerators. However, [9] does not explore the spatial domain scheduling required to achieve higher FPGA utilization and the approach is bound to static acceleration and single user with dynamic PR modes. FOS is taking the fact that FPGAs are spatial compute devices into its core by firstly allocating resources in the spatial domain and only falling back to time-domain multiplexing in case of over-utilizing the FPGA capacity.

Other related works include Microsoft's Catapult [2], ViTAL [47], Optimus Prime [1], and NICA [13]. Catapult takes a more low-level customization approach for high performance at the cost of design productivity and hence does not provide all OS functionalities of FOS (including transparent multi-tasking), while ViTAL [47] focuses on multi-tenancy and spanning across FPGAs by splitting an application into sub-parts, which can be connected by latency insensitive channels. However, ViTAL [47] does not explore runtime scheduling to arbitrate the resources efficiently and software level abstractions required from an FPGA OS. While Optimus Prime [1] and NICA [13] explore time domain sharing and multiple pipelines for spatial sharing in multi-tenant systems, they focus on bump-in-the-wire execution (transparent in-network acceleration) rather than general workload as it is supported by FOS and these approaches do not support modular development of system components.

Further approaches that relate to FOS include AmorphOS [5] and Resource elastic virtualization (REV) by Vaishnav et al. [34, 35]. AmorphOS provides system calls for hardware acceleration, high-level accelerator descriptions, and spatial resource sharing by changing the implementation of accelerators at runtime, which are features also available in FOS. However, AmorphOS does only consider accelerator replication and spatial sharing by merging netlists together (not bitstreams as in FOS) which, in turn, requires time-consuming recompilation and bitstream generation steps (as shown in Section 5), and consecutively, storage for each permutation. This is the opposite philosophy as available with the modular design flow in FOS where accelerators for FOS are physically implemented unaware of any other accelerator in the system. The AmorphOS approach suggests keeping a repository of existing physical implementations [5] and the corresponding bitstreams. Nevertheless, because each selection of accelerators (including their size options) requires generating a full bitstream, this is making AmorphOS unfeasible for more than just a few different accelerators, and even worse, an update of any accelerator would invalidate all permutations in the repository using the updated accelerator. AmorphOS [5] is aiming at datacenter FPGAs which, in the case of AWS F1 instances, means that each permutation is taking about 100MB of configuration bitstream storage. To illustrate the problem, let us consider 4 different accelerators each with 3 differently sized versions (e.g., using 1/3, 2/3 and 3/3 of the resources), then AmorphOS would require $1 + 3 + 6 + 10 = 20$ permutations (and corresponding number of times this configuration bitstream storage) if any load and accelerator combination exists. In the case of FOS when using module replication only, 4 bitstreams each being 1/3 the size of a single AmorphOS bitstream would be needed to provide the same flexibility. The reason for this unfavorable scaling in AmorphOS is that the approach does not foresee *partial reconfiguration of individual accelerator modules*, which further implies more substantial configuration time overheads. Another issue with

AmorphOS is that for implementing the configuration bitstreams, all IP has to be provided as netlists or even source code which has implications on security and IP protection management.

The FOS approach will pay, however, a penalty in fragmentation which would be less of an issue in AmorphOS. The fragmentation is a result of fitting modules into pre-defined module bounding boxes in order to support partial reconfiguration. Thanks to the significant advancements in HLS compilation tools, it is now possible to easily implement modules optimized for a given resource target (given by the number PR slots). This means that the fragmentation problem is shifted from defining tight bounding boxes to an HLS optimization problem such that accelerator modules in FOS do not have to under-utilize the FPGA.

REV [34, 35], in contrast to AmorphOS [5], is built on a version of the ZUCL shell [16] used in this paper and supports spatial sharing with both replications and implementation alternatives. However, REV focuses solely on OpenCL workloads, whereas FOS supports all commonly used types of accelerators (written in RTL, C, C++, and OpenCL) as well as running such accelerators together in any combination.

A common issue with all of the mentioned related works in this section (except for REV) is that they rely on vendor tool flows and suffer from dependencies between all OS components (software and hardware alike) as well as the vendor tool versions. This restricts changing the component functionalities or reusing them on other platforms and leads to higher compilation times w.r.t number of applications. Hence, one core innovation in FOS is the modular development flow, which supports multi-tenancy with existing accelerators (i.e. without modifications) and accessible user interfaces for both software and hardware developers.

7 CONCLUSION

In this paper, we described the underlying concepts and abstraction layers involved in building a modular FPGA operating system that can adapt to dynamic workloads. The resulting FPGA operating system – FOS, provides support for traditional as well as multi-tenancy environments with low overhead and easy to use software interfaces. The dynamic resource allocation capabilities of FOS, allows it to share multiple FPGA accelerators transparently between multiple users in both the time and the spatial domain. Moreover, FOS can switch between different accelerator implementations on the fly in order to maximize resource allocation for the best performance at any load scenario.

Our evaluation shows that FOS can speed up the compilation time by up to 2.34x and avoids standard recompilation requirements, which can reduce the time to perform updates in the system by over 100x due to its modular FPGA development flow. The overheads caused by the hardware and software layers are minor and recovered by the ability to dynamically schedule resources in both single and multi-tenant environments. Overall, FOS directly caters to the needs of upcoming FPGA systems that are being deployed at scale (cloud and edge) and allows systems to be more maintainable, adaptable, and accessible. With this, FOS is benefiting both FPGA and application domain experts. As a distinct feature, FOS provides an application-centric view to the developers by hiding most of the complexity encountered when using a heterogeneous CPU-FPGA acceleration system with a Linux backend.

With this, FOS is empowering a larger FPGA user community to implement complex and scalable heterogeneous systems. Current work includes porting the FOS principles to datacenter FPGAs usable in cloud settings.

ACKNOWLEDGMENTS

This work is supported by 1) the European Commission under the H2020 Program and the EuroEXA project (grant agreement 754337) and 2) the UK Research Institute in Secure Hardware and

Embedded Systems (RISE) through the project *rFAS - reconfigurable FPGA Accelerator Sandboxing* (grant agreement 4212204/RFA 15971). We also thank the Xilinx University Program for providing tools and boards.

REFERENCES

[1] A. Pourhabibi et al. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[2] A. Putnam et al. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*.
[3] C. Beckhoff, D. Koch, and J. Torresen. 2012. GoAhead: A Partial Reconfiguration Framework. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
[4] E. Lübers and M. Platzner. 2007. ReconOS: An RTOS Supporting Hard-and Software Threads. In *International Conference on Field Programmable Logic and Applications (FPL)*.
[5] A. Khawaja et al. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *Symposium on Operating Systems Design and Implementation (OSDI 18)*.
[6] K. D. Pham et al. 2019. ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs. In *International Workshop on FPGAs for Software Programmers (FSP)*.
[7] P. Lysaght et al. 2006. Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*.
[8] Q. Zhao et al. 2018. Enabling FPGA-as-a-Service in the Cloud with hCODE Platform. *IEICE Transactions on Information and Systems* (2018).
[9] F. Chen et. al. 2014. Enabling FPGAs in the Cloud. In *ACM Conference on Computing Frontiers*.
[10] S. A. Fahmy, K. Vipin, and S. Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *International Conference on Cloud Computing Technology and Science (CloudCom)*.
[11] T. Feist. 2012. Vivado design suite. *White Paper* 5 (2012), 30.
[12] Google. 2019. gRPC Framework. https://grpc.io/ Accessed: 2019-12-19.
[13] H. Eran et al. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.
[14] J. Weerasinghe et al. 2015. Enabling FPGAs in Hyperscale Data Centers. In *12th Intl Conf on Ubiquitous Intelligence and Computing and 12th Intl Conf on Autonomic and Trusted Computing and 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*.
[15] J. Weerasinghe et al. 2016. Network-Attached FPGAs for Data Center Applications. In *International Conference on Field-Programmable Technology (FPT)*.
[16] K. D. Pham et al. 2018. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *International Workshop on FPGAs for Software Programmers (FSP)*.
[17] K. Matas et al. 2020. Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.
[18] L. Wirbel. 2014. Xilinx SDAccel: A Unified Development Environment for Tomorrows Data Center. *The Linley Group Inc* (2014).
[19] T. La et al. 2020. FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2020).
[20] M. Asiatici et al. 2017. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *IEEE Access* (2017).
[21] M. Happe et al. 2015. Preemptive Hardware Multitasking in ReconOS. In *Applied Reconfigurable Computing*.
[22] L. Ma, F. B. Muslim, and L. Lavagno. 2016. High Performance and Low Power Monte Carlo Methods to Option Pricing Models via High Level Design and Synthesis. In *European Modelling Symposium (EMS)*. 157–162.
[23] K. Manev, A. Vaishnav, and D. Koch. 2019. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems. In *International Conference on Field-Programmable Technology (FPT)*.
[24] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. 2017. Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis. *IEEE Access* 5 (2017), 2747–2762.
[25] O. Knodel et al. 2017. Virtualizing Reconfigurable Hardware to Provide Scalability in Cloud Architectures. *Reconfigurable Architectures, Tools and Applications (RECATA)* 2 (2017).
[26] K. D. Pham, E. Horta, and D. Koch. 2017. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *Design, Automation Test in Europe Conference Exhibition (DATE)*.
[27] Q. Gautier et al. 2016. Spector: An OpenCL FPGA Benchmark Suite. In *International Conference on Field-Programmable Technology (FPT)*.

[28] S. Byma et al. 2014. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
[29] Amazon Web Services. 2009. AWS EC2 FPGA Hardware and Software Development Kit. https://github.com/aws/aws-fpga Accessed: 2017-12-04.
[30] H. So and R. Brodersen. 2007. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.
[31] T. Xia et al. 2016. Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators. In *International Conference on Field-Programmable Technology (FPT)*.
[32] V. Kathail et al. 2016. SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*.
[33] A. Vaishnav et al. 2018. A Survey on FPGA Virtualization. In *International Conference on Field Programmable Logic and Applications (FPL)*.
[34] A. Vaishnav et al. 2018. Resource Elastic Virtualization for FPGAs using OpenCL. In *International Conference on Field Programmable Logic and Applications (FPL)*.
[35] A. Vaishnav et al. 2019. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*.
[36] A. Vaishnav, K. D. Pham, K. Manev, and D. Koch. 2019. The FOS (FPGA Operating System) Demo. In *International Conference on Field Programmable Logic and Applications (FPL)*.
[37] M. Vesper. 2018. *Dynamic Stream Processing Pipelines on FPGAs Examplified on the PostGreSQL DBMS*. Ph.D. Dissertation. The University of Manchester.
[38] M. Vesper, D. Koch, and K. D. Pham. 2017. PCIeHLS: an OpenCL HLS Framework. In *International Workshop on FPGAs for Software Programmers (FSP)*.
[39] W. Peck et al. 2006. Hthreads: A Computational Model for Reconfigurable Devices. In *International Conference on Field Programmable Logic and Applications (FPL)*.
[40] W. Wang et al. 2013. pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.
[41] Xilinx. 2017. Reconfigurable Acceleration in the Cloud. https://www.xilinx.com/products/design-tools/cloud-based-acceleration.html#alibaba
[42] Xilinx. 2019. Platform Overview — Xilinx Runtime 2019.1 documentation. https://xilinx.github.io/XRT/master/html/platforms.html
[43] Xilinx. 2019. PYNQ. https://github.com/xilinx/pynq
[44] Xilinx. 2019. UG1144 - PetaLinux Tools Documentation Reference Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1144-petalinux-tools-reference-guide.pdf
[45] Xilinx. 2019. UG909 - Vivado Design Suite User Guide: Partial Reconfiguration. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug909-vivado-partial-reconfiguration.pdf
[46] Xilinx. 2019. Xilinx SDAccel Examples. https://github.com/Xilinx/SDAccel_Examples/
[47] Y. Zha and J. Li. 2020. Virtualizing FPGAs in the Cloud. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.