Accepted for IEEE CLOUD 2021

A Case for Function-as-a-Service with Disaggregated FPGAs

Burkhard Ringlein[*][†], François Abel[*], Dionysios Diamantopoulos[*], Beat Weiss[*], Christoph Hagleitner[*], Marc Reichenbach[†], and Dietmar Fey[†]
[*]IBM Research Europe, [†]Friedrich-Alexander University Erlangen-Nürnberg
{ngl, fab, did, wei, hle}@zurich.ibm.com, {burkhard.ringlein, marc.reichenbach, dietmar.fey}@fau.de

***Abstract*—The slowdown of Moore's law and the end of Dennard scaling created a demand for specialized accelerators, including Field Programmable Gate Arrays (FPGAs), in cloud data centers. At the same time, compute resources are increasingly consumed via public and private clouds and traditional applications are modernized using scalable microservices and Function-as-a-Service (FaaS) offerings. Nonetheless, true FaaS based on FPGAs or other accelerators is virtually absent from the offering catalogs of all major cloud providers. In addition, FPGA applications are typically coded in a monolithic fashion, due to device and vendor specific dependencies, which reduces the portability and usability of FPGA cloud offerings further.**

However, FPGA-based FaaS can improve execution efficiency and minimize (tail-) latencies while decreasing costs. We propose a novel system architecture, called `Mantle`, that uses disaggregated FPGAs to enable scalable, usable, portable and efficient FaaS offerings for FPGAs. Our experimental results demonstrate a significant reduction of end-to-end service provisioning time to below 7 seconds and an increase in execution efficiency by a factor of 4 with negligible overhead.

***Index Terms*—cloud computing, reconfigurable computing, disaggregated FPGAs, Shell Role architecture, partial reconfiguration, FaaS, forward compatibility, dynamic ISA extensions**

I. Introduction

Throughout the past few years, cloud computing has evolved from simple online storage and web services into the new way of providing any kind of IT service. While initially many companies used cloud Infrastructure-as-a-Service (IaaS) to deploy their applications, the emergence of microservices frameworks and Function-as-a-service (FaaS) or serverless computing has helped them to greatly simplify the development and operations (DevOps) of their cloud-based applications and services. Today, serverless computing is offered by all major cloud providers [1]–[4] and the underlying automation provided with frameworks like Knative [5], with its building, serving and eventing components, helps countless companies to efficiently develop and deploy their cloud services and apps [6]–[8].

At the same time, the hardware powering today's IaaS offerings has evolved, too. Due to the limited performance and efficiency gains provided by new generations of CPUs, new compute architectures with innovations beyond technology scaling were needed to sustain the performance and efficiency gains demanded from every new generation [9]. This has led to the emergence of accelerators ranging from GPUs and Field Programmable Gate Arrays (FPGAs) all the way to domain specific Application Specific Integrated Circuits

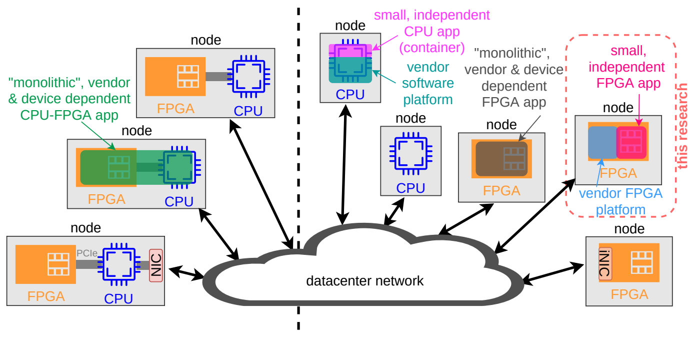

node
FPGA
CPU
"monolithic", vendor & device dependent CPU-FPGA app
node
FPGA
CPU
node
FPGA
PCIe
CPU
NIC
node
small, independent CPU app (container)
CPU
vendor software platform
node
CPU
"monolithic", vendor & device dependent FPGA app
node
FPGA
small, independent FPGA app
node
FPGA
vendor FPGA platform
this research
datacenter network
node
NIC
FPGA

Fig. 1. Closely-coupled vs. disaggregated FPGAs.

(ASICs) for Artificial Intelligence (AI) acceleration including Googles TPU [10], Inferentia and Trainium chips from AWS [11], [12], Intels[‡] Habana series [13], Graphcore [14], or Cerebras Wafer scale engine [15]. Consequently, in reaction to this demand, IaaS offerings have been expanded to cover accelerators [16]–[20]. One could assume that the two mega-trends "serverless computing" and "heterogeneous, accelerated IaaS" would seamlessly lead to new offerings for accelerated serverless computing but this has not — yet — happened.

In this work we explore the missing components of FaaS offerings for FPGA-based accelerators. FPGAs are increasing in popularity, because they provide their users with unparalleled flexibility, due to their reconfigurable logic. For example, FPGAs can accelerate a wide range of compute-intensive or latency-sensitive workloads, such as deep learning, blockchain, text processing, data analytic, or genetics [21]–[25]. In addition, FPGAs can beat GPUs in terms of performance for more and more domains [26], [27] and most of the time in energy efficiency [28]. We propose to use disaggregated FPGAs in combination with a new Shell Role architecture to build a cost and energy-efficient FaaS solution based on FPGAs.

II. Problem Statement

Figure 1 shows the spectrum of hardware platforms, which could be used to offer IaaS based on FPGAs. The first option is to offer bus-attached FPGAs, typically via Peripheral Component Interconnect Express (PCIe), in combination with a Virtual Machine (VM) in a closely-coupled fashion, as depicted on the left hand side. This means that for each FPGA, a VM must be booted and the network connection is mostly routed via a peripheral Network Interface Card (NIC) and managed by the CPU, as depicted in the lower left corner of

This is the accepted version of the article published by IEEE: DOI 10.1109/CLOUD53861.2021.00047

Figure 1. This kind of deployment is used by cloud offerings like AWS, Baidu or Xilinx Academic Cloud [18]–[20]. Using these services therefore requires a "monolithic" application that is dependent on the exact environment and FPGA device, as shown in **green** on the left hand side of the figure. The second way is to turn the FPGA as a standalone node and break the close-coupling, including integrating the network interface (iNIC) into the FPGA logic. This is depicted on the right hand side of Figure 1. These disaggregated FPGAs were shown to be beneficial in recent research [29]–[35]. In particular, disaggregated FPGAs are more flexible, energy-efficient and can be provisioned faster.

For FaaS based on FPGAs, we decided to focus on the second approach, *disaggregated FPGAs*, building on the following observations: First, the integration of the network and control path in the FPGA logic minimizes the latency and code-path, because it cuts out the "detours" through NICs, operating systems, or run-time environments. Second, as we described in [32], a cloud vendor maintains full control of a disaggregated platform if using partial reconfiguration technology. Consequently, the *eventing* component of FaaS frameworks will benefit from these low latencies. Third, the *serving* unit can leverage the short startup times of disaggregated FPGAs as well as the flexible combination of various different types of disaggregated nodes. All of the above leads to very short provisioning times and minimizes tail latencies, subsequently.

Ideally, FPGAs could be deployed with small and vendor-independent applications, similar to containers in the software world, as it is highlighted on the right hand side of Figure 1. However, today's disaggregated FPGA offerings still limit the flexibility of developers, since applications are bound to a specific type of device, or the device itself, as shown by the **grey** monolithic app in the middle of the right hand side of Figure 1. Also, the constraints that arise from the disaggregation limit developers flexibility further [32].

Following our observations, we aim to solve this issues and close the gap for "true" FPGA-based FaaS offerings. The goal is to provide FPGAs in the cloud with similar flexibility as container-based CPU platforms. In particular, the main contributions of this research are:

1) Enable forward and backward compatibility on binary level for FPGAs by presenting the `Mantle` architecture
2) Enhancing FPGA designs with a method to automatically adapt to user requirements while de-coupling the static security-critical parts of the design.
3) Enabling a cold-boot time for new FPGA instances below 7 seconds.
4) Provide APIs to integrate disaggregated FPGAs into an existing cloud software stack.

Overall, we aim i) to increase the scalability and efficiency of FPGA offerings in the cloud and ii) to reduce the platform adaption work for FPGA developers. Within the scope of this work, we focus on providing cloud FPGAs with an FaaS offering, but do not address the design or compilation of such FPGA-embedded functions.

The remaining of this paper is structured as follows: Next, we introduce an illustrative example application and give a brief overview of FPGAs in the cloud and their design flow, before we discuss current disadvantages. Afterwards, we propose the `Mantle` architecture design pattern to overcome the formerly elaborated shortcomings and present its implementation for FPGAs and cloud management stacks. Finally, we evaluate our framework in depth using real hardware and a commercial cloud, before we discuss related work and provide a conclusion.

III. Illustrative Example

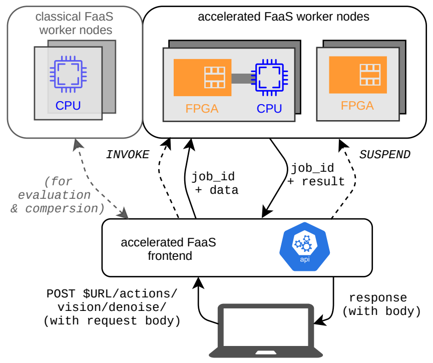

classical FaaS worker nodes
CPU
accelerated FaaS worker nodes
FPGA
CPU
FPGA
INVOKE
SUSPEND
(for evaluation & compersion)
job_id + data
job_id + result
accelerated FaaS frontend
api
POST $URL/actions/vision/denoise/ (with request body)
response (with body)

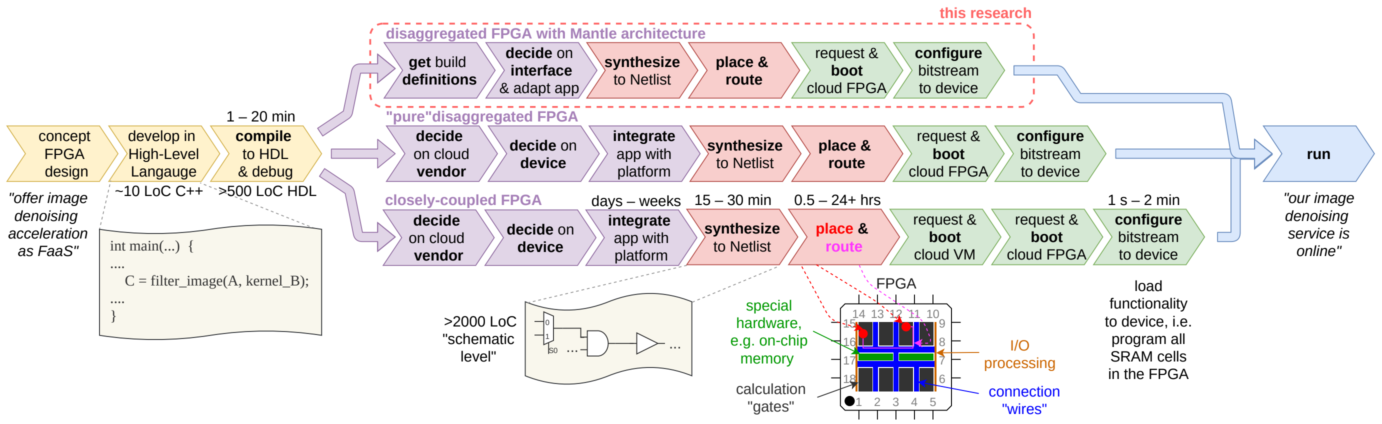
Fig. 2. Illustrative example: Image denoising using FPGA-based FaaS.

Figure 2 shows one illustrative use case for FPGA-based FaaS: The user wants to (pre-) process images with denoising filters. Therefore (s)he *posts* a request containing the image to the respective service URL pointing to an FaaS framework frontend, i.e. the eventing component of such a framework. Subsequently, this component validates the request and realizes that the corresponding service had scaled to zero before and is therefore asking the serving component to provision an instance of this service. Hence, the framework *invokes* an FPGA instance, e.g. a closely-coupled or disaggregated FPGA. Afterwards, the data of this request is send to the new FPGA instance, together with an identifying job-id using queues. Next, the FPGA processes the workload. In the case of closely-coupled FPGAs, the data must pass through the CPU and it's network stacks or run-time environments before reaching the FPGA. When the FPGA is done, it sends the result and it's job-id to the FaaS framework. Finally, the result is forwarded to the user and, depending on the current load and the service configuration, the FPGA instance could be powered-off again.

For this type of workload — image filtering — we would expect a speedup between 5 and 20 and energy savings of 60 – 80% due to the use of disaggregated FPGAs [35]. Some of the speedup can be accounted to the fast checking and routing of network packets in the FPGA logic. For example, the disaggregated FPGA platform used for our experiments in Section VII can validate and route a UDP packet in less than 0.1 µs to the corresponding kernel. Similarly, a TCP connection can be processed in less than 0.6 µs [31].

IV. FPGAs in Clouds and How to Use Them

The major types of today's cloud FPGA offerings were introduced in the previous sections and summarized in Figure


this research
disaggregated FPGA with Mantle architecture
get build definitions
decide on interface & adapt app
synthesize to Netlist
place & route
request & boot cloud FPGA
configure bitstream to device
"pure"disaggregated FPGA
decide on cloud vendor
decide on device
integrate app with platform
synthesize to Netlist
place & route
request & boot cloud FPGA
configure bitstream to device
run
concept FPGA design
develop in High-Level Langauge
1 – 20 min
compile to HDL & debug
~10 LoC C++
>500 LoC HDL
"offer image denoising acceleration as FaaS"
closely-coupled FPGA
decide on cloud vendor
decide on device
days – weeks
integrate app with platform
15 – 30 min
synthesize to Netlist
0.5 – 24+ hrs
place & route
request & boot cloud VM
request & boot cloud FPGA
1 s – 2 min
configure bitstream to device
"our image denoising service is online"
>2000 LoC "schematic level"
special hardware, e.g. on-chip memory
calculation "gates"
FPGA
I/O processing
connection "wires"
load functionality to device, i.e. program all SRAM cells in the FPGA

```
int main(...) {
 ....
   C = filter_image(A, kernel_B);
 ....
}
```

Fig. 3. Overview of the typical design flow for FPGAs in the cloud. The place and route phase tries to optimize the location of each operation and their connections while the logical, physical and timing constraints of the design must be satisfied. In general, place and route is shown to be NP-complete [36].

1. However, FPGAs in the cloud usually have some design patterns in common, due to the similar constraints in such platforms. We will illustrate this general design pattern in Section IV-C and why it must be adapted to the requirements of disaggregated FPGAs in Section IV-D. But first, we will give a short introduction of the general FPGA design flow and explain how these FPGAs are configured in a cloud setting.

A. Cloud FPGA Design Flow for Container Designers

An overview of the different steps of the FPGA design Flow is given in Figure 3. Based on the initial motivation and concept of an FPGA application, the application kernel is developed using some kind of high-level languages. Those "high-level" languages can range from C/C++-like (e.g. [37], [38]) to Haskell-based tools (e.g. [39]). Next, this High-Level Language is compiled to a Hardware-Description Language (HDL), typically VHDL or Verilog. Alternatively, the application can be coded directly in HDL. The abstraction level of HDLs can be compared to Assembler languages for CPUs and HDL descriptions can be vendor-agnostic, in general. Next, the developer must decide which device and cloud vendor (s)he targets, because the next steps are mostly *device- and/or vendor-specific*. At this step, the chosen deployment approach of the FPGA must also be considered. How the platform specific logic of the FPGA interacts with the application specific part is explained in the Sections IV-C and IV-D.

Let's compare the showed flow in Figure 3 with the development of cloud containers to highlight the differences that are important for this work. The two major changes between the classical closely-coupled approach, depicted in the last row in Figure 3, and the enabled shorter flow in the upper branch of Figure 3, are the possible adaption of the platform with respect to the FPGA app — not vice versa as classical — and the independence of a VM at run time.

In the classical case, adapting an FPGA app to the vendor can be a lengthy process and can be compared to writing a complete new `Dockerfile` for a complex application on a new CPU architecture, starting from a very tiny base image. In opposition to that, using our novel flow, the user decides which app interface and dependencies her/his app requires based on the interfaces the provider offers and informs the provider about the requirement upon deployment. This step can be compared to deciding on which prepared *parent image* the application builds its Dockerfile [40]. Consequently, the developer can deploy the app faster on the cloud service and is furthermore not vendor and/or device bound.

To not exceed the scope of this paper, we refer the reader to standard references like [41] for more details on the FPGA design flow.

B. Commanding the Fleet: Configuration of FPGAs

To "tell" an FPGA what it should do, a device-specific binary must be loaded into the device, as shown in Figure 3. This step is usually called the *configuration* of FPGAs and the FPGA specific binary is commonly referred to as *bitstream*. In our illustrative example of Figure 2, the developer adds the bitstream of the image denoise application to the serving component of the FaaS framework and offers the service to users who may be clueless about FPGAs but just want to leverage their acceleration capabilities. Consequently, if the corresponding bitstream is configured, the FPGAs "know" what they have to do if invoked by the framework.

An additional feature of FPGAs is *partial reconfiguration*. When using partial reconfiguration, parts of the FPGA design are updated or exchanged, while the remaining part of the FPGA continues to operate [42]. For example, in Figure 4, the upper part can be exchanged during run time without interrupting the lower part. Partial reconfiguration offer various merits, e.g. a very short adaption time or additional security guarantees [32], [42].

Despite the differences in architecture and density, closely-coupled and disaggregated FPGAs differ also in booting and configuration. Disaggregated or "standalone" FPGAs run completely on their own without being controlled or configured by an attached CPU. This means that they require a boot-medium to load the deployed bitstream, or at least the initial version of it. This bitstream must then contain the logic that allows the FPGA to operate in its environment. Non-standalone FPGAs are tightly coupled with a CPU, usually via a bus interface like PCIe. Therefore, the CPU can control the FPGA during run time, provide the bitstream configuration at initialization, or the necessary updates during run time. However, an idle

CPU dissipates more energy than an FPGA under full load, so standalone FPGAs have advantages when it comes to energy efficiency (see e.g. [35]). On the other side, in the absence of a controlling CPU, it is the responsibility of the standalone FPGA itself to handle the updates of the logic during run time, if changing environment conditions demand so.

Hence, standalone FPGAs may offer an advantage in terms of energy efficiency, size or flexibility. This comes with potential disadvantages due to a more complex control logic within the FPGA design and more dependencies for applications. Should the advantages of disaggregated FPGAs be leveraged by a cloud service, this complex control logic for standalone FPGAs is a key requirement. Consequently, the next sections explain how to fulfill this requirement.

C. The Shell Role Architecture Design Pattern

To make an FPGA platform usable for various applications, its design is often split into a platform specific and an application specific part. We will refer to this pattern as *Shell Role architecture* and one example is depicted in Figure 4. The platform specific part — i.e. the *Shell* — contains all I/O processing, the PCIe or network cores, the memory access and some run time management. The application specific part — the *Role* — contains the user's application on this platform. The Shell and the Role are linked via the *Shell Role interface*. This interface is at "wire level" in the FPGA chip.

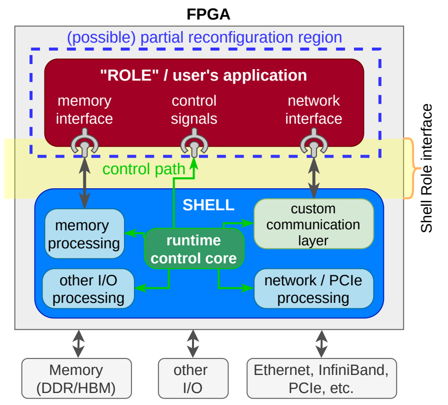


Fig. 4. The general principle of a Shell Role architecture and the corresponding interface.

The Shell Role architecture principle has several advantages: On one hand, it allows platform vendors to develop their platform for various applications. On the other hand, the application developer does not need to deal with the I/O details of a given platform and can rely on the abstractions that are provided by the vendor. This idea also follows established design patterns like *Separation of Concerns* or *Single responsibility principle*. Additionally, if the Shell Role interface is clearly defined it also allows the usage of partial reconfiguration. Ideally, when using partial reconfiguration, the user does not need to share her/his source code with the platform vendor, (s)he is only required to share the binary partial bitfile. Furthermore, the platform vendor can control the Shell at deployment and during run time and can consequently enforce all necessary security measures [32]. Due to these advantages, the Shell Role architecture design pattern — with and without partial reconfiguration — can be widely observed [18], [29], [32], [33], [35], [43].

Using our example of Figure 2, the denoising algorithm would be part of the Role and would consume the payloads from the network interface of the Shell. The lower-level network control including the job routing is handled entirely by the Shell.

D. Limitations of Shell Role Architecture Design Patterns

Despite their widespread dissemination, Shell Role architectures create a multitude of problems for FPGAs in the cloud. The seamless interaction between Shell and Role only works (and compiles) if both are using the same Shell Role interface. This leads to a strong dependency of the complete ecosystem around a platform on this Shell Role interface and consequently "freezes" the interface once it is released and deployed. After the release, application developers design their app using this interface and consequently expect that the platform will continue to offer this version of the Shell Role interface in the future. Therefore, if the platform vendor wants to improve the interface, e.g. with a faster memory connection, a new encryption capability or the next version of a third-party core, the vendor has to maintain the old and the new version of the Shell to not disrupt the users of the platform. This is the case for designs without and with partial reconfiguration. In the latter case, the constraint is even stronger, because the Shell Role interface needs to stay the same all the way down to the level of the physical partition pins in the FPGA.

On the other hand, the increasing usage of third-party dependencies, like e.g. Xilinx's Vitis library, [44] the OpenCores project [45], or different usable communications models like MPI [35] or OpenCL [46], weakens the clean separation between Shell and Role, because if an app relies on run-time environment or library support from the platform vendor, the Role is no longer the single responsibility of the user. In addition, the Shell is then no longer application independent. This mixture would lead to a large number of different Shell Role interfaces, if following the classical Shell Role architecture approach.

Hence, if the platform vendor wants to offer multiple run-time environments, i.e. different versions of the Shell, for his/her FPGAs, a large number of Shells must be kept available for the user. In addition, the new Shell must be distributed either to the developer or to the building component, in the case of no partial reconfiguration, or to the deployed systems in the case of partial reconfiguration.

This also impacts the provision time: FPGAs have a memory to boot from when they are powered on, which comes with some consequences. First, no matter how big this boot-memory is, the right Shell has to be selected or written to the memory before the FPGA is operational. In case of disaggregated FPGAs, this undermines the advantages of the independence. Second, at some point the number of available Shells becomes limited by the size of the available boot memory of FPGAs, standalone or not. Therefore, providing a significant number of different Shells would require frequent rebooting of the FPGAs to meet the specific Shells demanded by different workloads. Besides the huge management effort, this reduces the efficiency of the system by diminishing the up-time. In addition, no guarantees for execution times could be given for such a system.

As a consequence, the usage of Shell Role architectures limits the scalability and portability of FPGAs in the cloud. Furthermore, the simplifying motivation behind the *Separation of Concerns* principle would lead to a much more complex dependency management effort — the opposite of the intention.

V. MANTLE ARCHITECTURE TO THE RESCUE – OR HOW TO BUILD EFFICIENT AND SECURE CLOUD FPGA PLATFORMS

Our goal is to bring the flexibility found in container- and package-based classical CPU platforms to FPGAs in the cloud, while at the same time simplifying the deployment process for the cloud vendor. Therefore, we introduce the `Mantle` FPGA design architecture in the following. Afterwards, we describe the cloud stack integration.

A. FPGA Design Pattern

The main ambition of the Shell Role architecture is to simplify the platform and the development of applications for it. This ambition is undermined by the strong dependency of the interface on the platform logic (Shell) and the app logic (Role). To resolve this, we propose to add a logical layer in between the Shell and the Role: The `Mantle`, which acts as glue between the static platform logic and a user's application. In the software world, this notion is sometimes called *Middleware*.

While the Mantle sits in between the physical Shell and Role components, its logical function is actually part of both. Therefore, we will refer to the Shell+Mantle part as (dynamic and static) *platform logic* and the application specific part of the Role as *app logic*, as depicted in Figure 5. In order to separate the design specific part of the platform — e.g., memory management, encryption engines or data conversion cores — from the design independent part — e.g., physical layer communication, I/O processing, or physical memory — we split the platform logic into a static and dynamic part. From a logical point of view, the design is still split into Shell, the platform specific part, and Role, the application specific part. Technically, the app logic is the part of the Role that comes from the user. The static platform logic is the part of the platform that is independent of application and use case. The dynamic platform logic contains the Intellectual Property (IP) cores that are application or use case specific and for which the platform vendor is responsible. Hence, the dynamic platform logic — or Mantle — has parts that belong to the logical Shell and the logical Role.

The presence of a Mantle repeats the engineering motion behind the original Shell Role architecture. It divides responsibilities by defining clear interfaces, but with a different interface to the platform and to the application. It consequently acts as an adapter between platform logic and app logic. Hence, the platform vendor can update the platform logic if necessary and then adapt to different requirements of the app logic by issuing a new version of the dynamic platform logic.

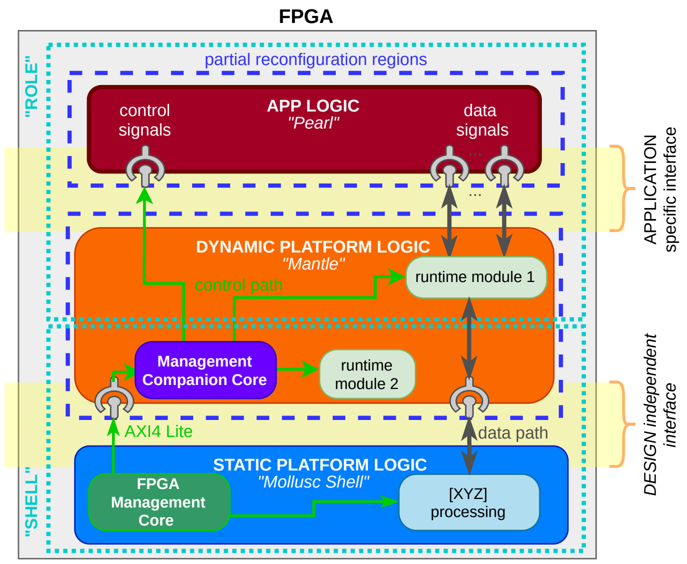


Fig. 5. Composable Mantle architecture. It allows an application specific interface for the user (app logic) and a design independent interface between static platform logic and Mantle (dynamic platform logic).

The opposite is possible too: If the application needs more features provided by the platform, the Mantle can be adapted without the necessity to replace all static platform logics in the deployed system. If the Mantle can also be deployed by partial reconfiguration, the FPGA can continue to run while adapting the platform to the needs of a user. As a consequence, the FPGA platform can now *offer backward and forward compatibility*, which are necessary features for user friendly and scalable FPGA platforms in the cloud. Furthermore, due to the split of platform and application logic, the platform vendor retains control of the platform control path. If combined with partial reconfiguration, this guaranteed control also holds during run time, because the deployed binaries of the platform logic are not “touched” or manipulated by the user.

Nonetheless, just adding another partial reconfiguration region is not sufficient, because the IP cores and their control path within the Mantle can still vary significantly. Moreover, it needs to be possible that the control path of the dynamic platform logic can be updated without changing an already deployed and running static platform logic. This requirement is fulfilled best when using a standardized bus between the static platform logic and Mantle, with endpoints on both sides that can handle the communication and necessary actions. Hence, we introduce a *Management Companion Core* (MMC) to the Mantle that is connected to the control path of the static platform logic — or *FPGA Management Core* (FMC) — via a standard AXI4 Lite bus [47], as depicted in Figure 5. We decided to use the Lite version of AXI4 because most communications in this context are single register reads and writes without a need for burst transfers. We will refer to this composable design pattern as *Mantle architecture*. With this approach, we can update the Mantle, also by using partial reconfiguration, and the interface between static platform logic and Mantle remains static even if there are new cores added to the Mantle. Consequently, static and dynamic platform logics

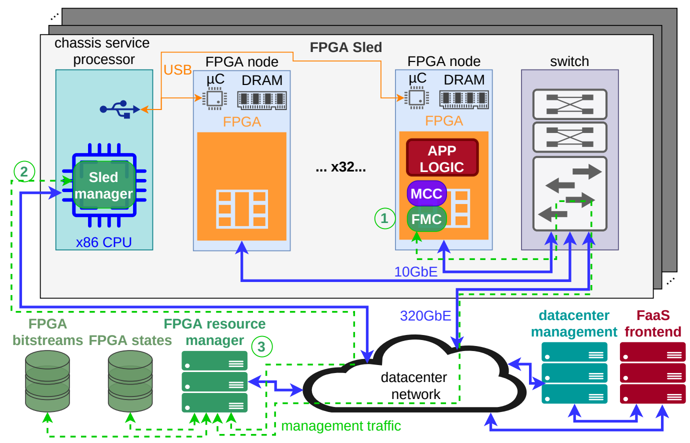


Fig. 6. The data center view of the `Mantle` architecture. The FPGA resource manager tracks the configuration states of each FPGA. Figure based on [32].

can be developed and deployed independently of each other.

However, the FMC inside the static platform logic still needs to know how to handle all versions of the dynamic platform logic. To avoid the need for future knowledge inside the FMC, the dynamic platform logic needs to contain all necessary information for the FMC, so that the FMC knows how to manage the Mantle. Therefore, the MCC provides a header table that can be read via the AXI4 Lite bus. Since the structure of the table can be designed in a way that is independent of its content, the FMC can still handle all versions of Mantles, including new versions that are designed after the deployment of the static platform logic.

The concept of adding a layer between the platform specific part of the design and the application specific part follows classical software design patterns, like the emergence of containers or the dynamic code linking, e.g., of the `glibc` [48]. The GNU C library maintains different versions of its Application Binary Interface (ABI) to stay compatible with old applications and to serve as a link between applications and the Linux[§] kernel. Similar mechanisms are present in modern cloud computing frameworks like Docker or Kubernetes. Those frameworks also allow version mismatches within a certain range [49]. Since our proposed Mantle concept also allows this kinds of forward and backward compatibility, it can be seen as a container stack or glibc for FPGAs. These properties can support a large scale of different third-party dependencies on the same cloud FPGA.

B. Cloud Stack Integration

The changed micro-architecture of the FPGA must be reflected in the data center architecture as well. In general, there exist two approaches to incorporate disaggregated FPGAs into a data center management environment: Either, to treat the FPGA as part of the infrastructure and *not* expose it to a user as available resource [29], or to expose the FPGA node as resource in the same way as CPU resources are handled, e.g. in combination with a cloud stack framework like *OpenStack* [32], [33]. Consequently, to realize an FaaS offering that leverages explicit acceleration, we decided to build on top of the second approach.

Disaggregated FPGAs have three levels of management dependencies [32], as depicted by the **green numbers** in Figure 6: First, controlling what is happening *inside* the FPGA. The details of this level are handled by the Mantle micro-architecture with its FMC and MCC, as previously discussed. Second, controlling the power state and initial setup of the FPGA, or a group of FPGAs. Third, managing and assigning FPGAs to users or functions at data center level.

While the Mantle architecture does not require changes on the second level, the third level must reflect the split between dynamic and static platform logic within the FPGA. Therefore, the FPGA resource manager tracks and stores which dynamic platform and app logic are configured on each device. In addition, to audit the status of the FPGAs, a HTTP `GET /status` request can be used, as e.g. described in [32], using REpresentational State Transfer (REST) concepts. With this approach, the resource manager can quickly determine if there are FPGAs with the right dynamic platform logic available when a request from the FaaS frontend arrives. In case a suitable FPGA resource exists, only the app logic must be updated by a partial reconfiguration. Otherwise, the dynamic platform logic must be changed first, also by using partial reconfiguration. Consequently, the execution efficiency is increased, because deployed logic can be reused.

VI. Implementation

A. FPGA Design

To leverage the full advantages of the composable Mantle architecture, the control path inside the static platform logic is designed in a way that makes it forward compatible. That way, once deployed and running, it can be updated at run time to handle a new version of a dynamic platform logic. To control the Mantle, the run time management needs to know which IP cores are in the Mantle region and what information these cores need at run time and how to deliver this information. When using the general interface shown in Figure 5, the information from the static control path — the FMC — must be forwarded via the MCC to the dynamic part.

To retrieve the necessary information at run time itself, the FMC implements a RESTful API frontend. This frontend is used to submit commands and configurations to the FPGA at run time. The FMC needs to parse and process the API requests and forward the resulting information via the control path to the IP cores. The FMC of our composable Mantle architecture can parse, process and respond to HTTP API requests, such as `POST /routing HTTP/1.1 \r\n` *`(payload)`* `\r\n`.

Since the processing steps for multiple API requests are similar, we implemented small functional cores that can be used by different calls. In order to increase reusability even further, we implemented the request processing with a tiny Instruction Set Architecture (ISA). The structure of the ISA is depicted on the left hand side of Figure 7 and is described in the following. A global Finite State Machine (FSM) evaluates the incoming signals and distinguishes between situations like a new API command arrival, another data junk of partly

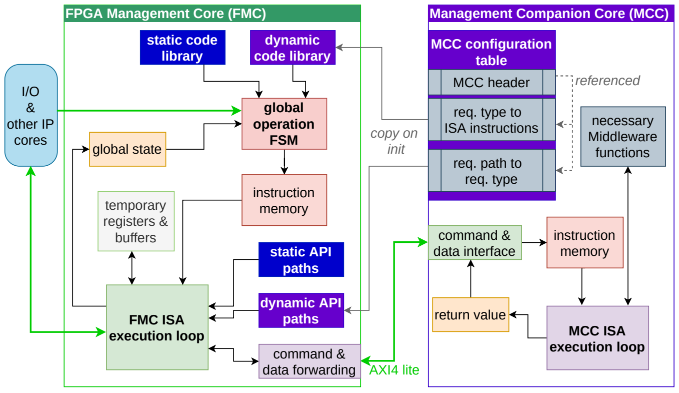


Fig. 7. Extendable management ISA. The bus between FMC and MCC is sketched from a logical point of view, physically all connections are mapped to one AXI4-Lite bus (peripherals are not depicted).

processed API command arrival, or detects the state changes of other peripherals. Then, the FSM issues instructions according to this global situation based on a static code library. Subsequently, these instructions are executed by the ISA execution loop. The valid API paths are stored in a table. Examples would be different RESTful HTTP paths like `GET /status`, `POST /mantle`, or `PUT /reset`. This table also contains the start addresses of the static code library for processing the request. Therefore, the ISA instructions are independent of the API calls. The instructions parse the request and then map the request to the content of the HTTP API paths table. If the API call is not part of the table, the FMC can immediately respond with the corresponding error codes.

In addition to the reusability of logic blocks, implementing the management core inside the FPGA by using a special ISA enables the dynamic extension of the code and architecture at run time. Hence, the MCC can implement a table that contains the dynamic extensions of the API request paths table and the code library, as depicted on the right hand side of Figure 7. The FMC can copy these tables via the AXI4 Lite bus every time when a new Mantle is partially configured. Therefore, the FMC "learns" how to handle new API calls for this specific Mantle. Since the sizes of the API paths and code tables vary between different Mantles, the configuration table of the MCC needs a header that declares the positions of the dynamic contents within the AXI4 register space. Furthermore, each API path and the corresponding code snippet also varies in length and may not always be a multiple of the AXI bus width. Therefore, an additional table providing the indexes of the begin of each API call and each corresponding code snippet within the data is required. One example of such an header table is given in Table I.

This header table can be seen as the equivalent of the header tables in the Executable and Linkable Format (ELF) [50]. Since the version number is always on address 0 of the AXI4 Lite address space, the FMC can also be developed in a way that supports multiple MCC header versions. Different IP cores in different Mantles may require completely new instructions to be introduced to the FMC. The most flexible way of updating the ISA architecture is to allow the execution of special instructions in the MCC as well, as a kind of an mantle-specific co-processor. Therefore, the FMC can forward instruction opcodes and parameters to the MCC via the AXI4 Lite bus. The number of parameters and the register addresses is different for each Mantle and therefore also provided by the header table of the MCC. Finally, the dynamic code library contains the instruction opcodes that have to be forwarded to the MCC in order to process a given API request. The dynamic code library can also enclose code to adapt the parameters of these instructions.

This implementation of the dynamic extensions does not modify the FMC itself at all and consequently can be changed every time a new Mantle is configured. One detailed example using the composable Mantle architecture design pattern, including an interface for Vitis Vision kernels [51], memory address translations and symmetric encryption, is depicted in Figure 8. This example would be one specific implementation for an image filtering service, e.g. denoising, as illustrated in Figure 2. In this example, the control path must be extended to adapt the memory layout, the encryption core and network configurations of the `Vitis2Network` adapter. Finally, the start and stop of the application must be controlled.

TABLE I
STRUCTURE OF THE MCC HEADER TABLE

Word position	**Data** (32-bit)
0	Header structure version
1	Pointer to Mantle version string
2	Length of Mantle version string
3	Number of additional API calls
4	Pointer to API call string index table
5	Pointer to API call strings
6	Pointer to API call code index table
7	Pointer to API call code table
8	Pointer to MCC configuration space
9	Pointer to MCC status registers
10	Pointer to status space
11	Pointer to initialization code
12	Length of initialization code
13	Address of the MCC instruction register
14	Maximum parameters per MCC instruction
15	*content based on pointers*
...	....

B. Cloud Stack Integration and Deployment

Through leveraging the composable Mantle architecture design concept, the user can develop her/his application using an app logic interface (s)he likes and choose the appropriate Mantle at run time. Consequently, the user must inform the cloud service about the requirements of the implemented app logic and does not need to adapt to the specific requirements of one vendor. Therefore, the user submits the synthesized, placed and routed partial bitstream along with the description of the required Mantle to the cloud FPGA platform. The platform management framework then knows which dynamic platform logic must be configured for this particular app. As an example to illustrate this, Listing 1 shows the requirement description for the implementation of Figure 8 in JSON syntax. In this example, the application specific interface for Vitis Vision is selected and the platform is asked to provide the necessary "network adapter". Upon upload of this description along with the corresponding partial bitfile by the user, the cloud FaaS service can infer which dynamic platform logic is required. Furthermore, in this example, the cloud provider decided to use a symmetric encryption for the network traffic in the data center, as depicted in the Mantle in Figure 8. This core is controlled by the provider and its presence or absence does not affect the user application or the specified interface. Finally,

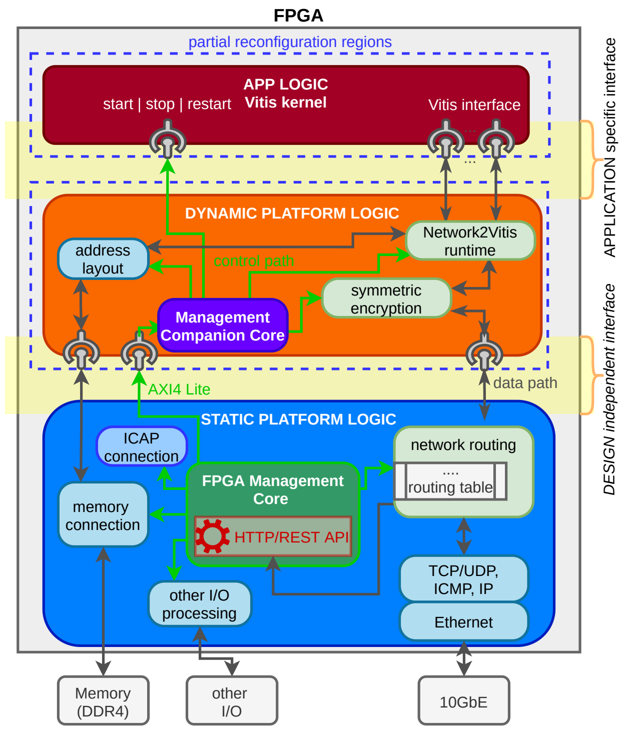


Fig. 8. Detailed example using the composable Mantle architecture.

```
{
  "name": "iamge_denoise-jacobi",
  "version": "0.9.42",
  "interface": {
    "app_vitis_vision"      : "1.1.0"
  },
  "dependencies": {
    "VitisVision2Network" : "~1.2.0"
  }
}
```

Listing 1. FPGA in cloud application requirement JSON description for the FPGA design in Figure 8 using the composable Mantle architecture.

after deciding which Mantle should be used, either an existing FPGA that fulfills the requirements is selected or a new one is powered-on and configured accordingly.

VII. Evaluation

A. *Experimental Setup*

To evaluate our proposed system architecture and design flow, we compare two FPGA-based systems and one CPU-based system, similarly to the setup of the example in Figure 2: First, we evaluate the Mantle architecture on the IBM** cloudFPGA (cF) platform [30], a disaggregated FPGA system which management framework includes RESTful HTTP APIs [32]. The cF system architecture is depicted in Figure 6 and it consists of standalone Xilinx KU60 FPGAs[1], grouped by 32 in so called "Sleds". Each FPGA boots from a flash chip that can contain only one Shell (or static platform logic).

Second, to compare our proposed architecture with commercial available technologies, we also tested the possibilities using one AWS Elastic Compute Cloud (EC2) F1 instance [18]. An F1 instance is a CPU-FPGA heterogeneous platform where Xilinx VU9P FPGA[2] devices are tightly attached to

[1] `xcku060-ffva1156-2-i`
[2] `xcvu9p-flgb2104-2-i`

TABLE II
Configuration Times

operation	**file size** in MiB	**total time** in $seconds$	**effective speed** in $\frac{kiB}{s}$
JTAG config. of the compl. design	24.5	55.09	455.39
JTAG partial reconfig. of Mantle	1.8	11.07	166.43
JTAG partial reconfig. of app logic	12.8	30.85	424.82
`POST /configure` of partial Mantle logic via TCP	1.8	0.17	10,788.41
`POST /configure` of partial app logic via TCP	12.8	1.07	12,215.09

TABLE III
FPGA Design Build Times

build of	**time** in $minutes$
complete design	168
app logic + Mantle	37
app logic	32

TABLE IV
Provisioning Times

cold boot of	**time** in $seconds$
CPU	271.10
AWS EC2 F1	82.26
cloudFPGA (cF)	6.20

Intel Xeon E5-2686 v4 (Broadwell) CPUs through a high-bandwidth PCIe x16 interface. For our testbed we used the `f1.2xlarge` type of EC2, featuring a single FPGA, one virtual CPU with 8 cores and 122GB of memory.

Lastly, to compare with CPUs, we used one bare-metal server containing two Intel Xeon E5-2630 v3 @ 2.4GHz CPU with 8 cores each and 126GB memory running Ubuntu 20.04.

In the remainder of this section, we evaluate the main goals of the proposed Mantle architecture: The improvement of the building (see VII-B) and provisioning times (VII-C & VII-E), including cold-boot, of FPGA nodes without significant resource (VII-D) or latency overheads (VII-F). Consequently, we did *not* evaluate the deployment of the FaaS framework. Neither did we evaluate specific applications, e.g. image filtering or denoising as used in the illustrative example of Figure 2, to not exceed the scope of this work. Because a fair comparison of applications on different hardware platforms would require an in depth analysis of the resulting performance and would be biased by the used or not-used optimizations of each workload on each platform, which is not the focus of this paper. Nevertheless, we provide an application agnostic analysis of the performance impact of our architecture in Section VII-F using a roofline model.

B. *Evaluation of Configuration and Build Times*

Table II shows the configuration times of the different components of the design using the bus standard Joint Test Action Group (JTAG) or TCP/IP protocol via the network. As expected, deploying Mantle and app logic separately using partial reconfiguration, while using the static platform logic that is present after boot, leads to shorter configuration times, which also reduces the down-time or the switching costs of a service. The JTAG speed for the experiments in Table II was set to 5 $\frac{Mbit}{s}$, the TCP is based on a 10Gb/s Ethernet. This experiment highlights the advantage of the proposed approach with disaggregated network-attached FPGAs, because using an FPGA-based TCP stack to configure bitstreams outperforms the usual JTAG procedure.

Additionally, if the user only needs to build her/his FPGA application without the static platform logic or the third-party

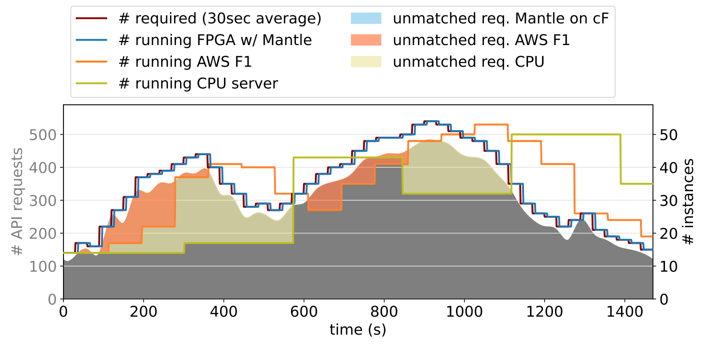


Fig. 9. Model of the impact of provisioning times on the gap between demanded and running instances for a dynamic, random request profile.

modules, the build time for the user is reduced as well, as shown in Table III. These builds were measured on an Intel Core i7-6700K CPU @ 4.00GHz running RHEL 7.5 and always include synthesis, place, route, and generation of the (partial) bitstreams.

C. Evaluation of Provisioning Time

After building and configuration, we evaluated the provisioning time of new instances. Our experiments assume that the FaaS frontend states what the current number of instances should be and their corresponding configuration (as it would be, e.g., with Kubernetes/Knative). Therefore, we measured the time of a cold boot until the FPGA or CPU application began to execute. The averaged result of all three platforms are shown in Table IV. As can be seen, the cF platform using the Mantle architecture is up and running more than 40 times faster than the CPU server and more than 10 faster than the AWS F1 instance. For AWS, we could not request FPGAs attached to bare-metal servers. Consequently, the measured results in Table IV are not completely fair (to our disadvantage), since we compare a bare-metal CPU with a VM and a bare-metal FPGA. However, we think the comparison is “fair enough” to exhibit the impact of our proposed architecture, in combination with disaggregation, on the provision time of FPGA instances.

Based on these results, we modeled the behavior of a large FaaS offering that has a dynamic request load in Figure 9. In this model, one instance can handle up to 10 requests in parallel and the autoscaler targets a utilization of 90%. The result exhibits the advantage of the faster response time of our composable and disaggregated Mantle architecture. By contrast, the “classical” FPGA with a closely-coupled CPU for the platform management, instead of the proposed static platform logic withing the FPGA, can not scale as fast or meet the requirements, as represented by the AWS F1 experiment. This is also emphasized by the results in Section VII-E.

D. Evaluation of Resource Overhead

Third, we wanted to evaluate the overhead of the composable Mantle architecture concept in terms of FPGA resources. The results for the FMC and MCC with an example design similar to the one in Figure 8 are shown in Table V. The resource usage of the composable Mantle architecture

TABLE V
RESOURCE USAGE IN A XCKU060

Resource	Available	Used FMC		Used MCC	
LUT	331680	14711	(4.44%)	523	(0.16%)
LUTRAM	146880	162	(0.11%)	0	(0.00%)
FF	663360	14314	(2.16%)	332	(0.05%)
BRAM	1080	15	(1.39%)	6	(0.56%)
DSP	2760	12	(0.43%)	0	(0.00%)

management framework is below five percent of all available resources. This very limited overhead supports the idea of offering user-friendly dynamic Mantle extensions.

E. Evaluation of Switching Time

Fourth, to come back to our goal of increasing execution efficiency of the FaaS platform, we also evaluated the switching time of multiple workloads under resource constrains. In particular, we measured the total time to execute three different apps with only one FPGA or CPU. In the case of the FPGA with Mantle architecture, two apps have the same app interface. To avoid confusing application-agnostic boot and switching times with application-specific acceleration, we replaced the real execution times for all platforms with the same placeholders (10, 25, and 45 seconds). The results are given in Figure 10. Our Mantle architecture finishes executing all apps when the closely-coupled FPGA starts the first app and while the CPU is still booting. In addition, the time actually spent on execution is close to 90% for our architecture. This result underlines the general advantage of stand-alone FPGAs.

F. Evaluation of Performance Penalty

Finally, we calculated the potential performance penalty for the cF FPGAs using the Mantle architecture. We wanted to ensure that we do not achieve significant efficiency gains for execution and deployment at the cost of decreased performance, because this could render the acceleration useless. To avoid the bias of a few selected applications, we decided to calculate the roofline model [52] of the cF platform instead, which is given in Figure 11 and is adjusted for the clock frequency of 156 MHz, as used by the cF platform. In this figure, the attainable performance of one cF FPGA before and after the introduction of the Mantle architecture is shown. The roofline indicates that the theoretically attainable performance for relevant kernel domains [53], [54] is affected only minimally by the introduction of the Mantle architecture. For example, the image denoising of Figure 2 would have an operational intensity around one. As a methodological remark: Calculating the peak performance for FPGAs is always very application specific, since the application defines the microarchitecture and therefore the operations. However, summing the performances of the available Digital Signal Processors (DSPs) of an FPGA is a good upper estimate. In addition, the usable share of the bandwidth provided by LUTRAM and BRAM also depends on the application-specific FPGA design. For this analysis, we assumed (over-optimistic) that all those building blocks would be used in the theoretically maximal performative way. Consequently, the impact on realistic FaaS

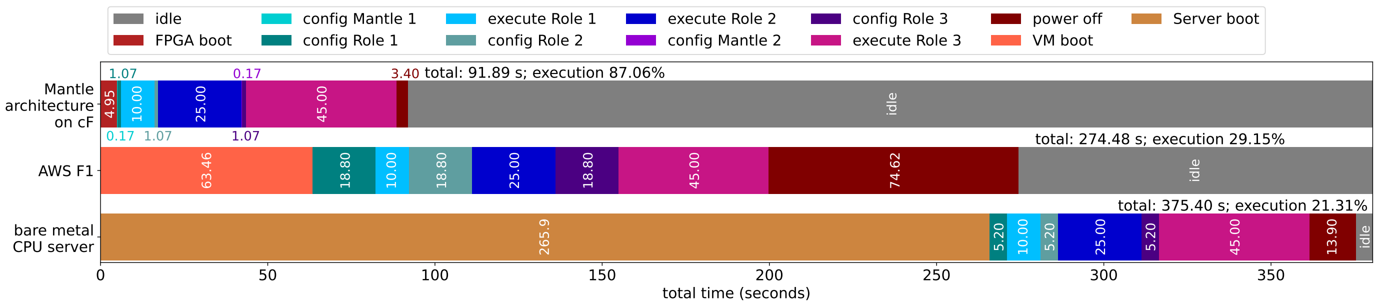


Fig. 10. Meassurements of provisioning and execution times under resource constraints. All platforms executed three different apps with only one FPGA/CPU available. For application agnostic evaluation, execution times were replaced by representative constants (10, 25, 45 seconds) for all platforms.

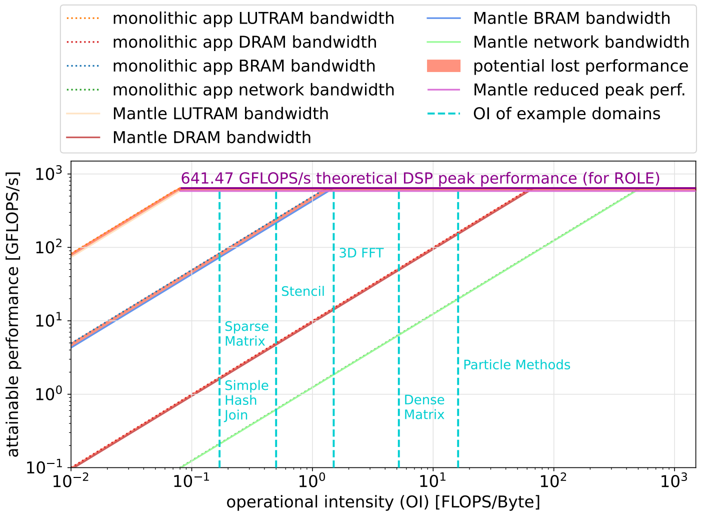


Fig. 11. The Roofline model with relevant kernels for the cF platform.

functions is even smaller, since we expect those workloads to be dominated by the 10 GbE network traffic and not by, for example, the reduced BRAM bandwidth of 429 GB/s.

VIII. Related Work

Two recent research activities focuses on decoupling app and platform logic within the FPGA (cf. right-hand side of Figure 1): First, in 2015, *Kirchgessner et al.* proposed a project named "RC Middleware" to increase the productivity of FPGA application developers by providing application independent wrappers and compiler-like tool chains [55], [56]. However, their approach is completely static and does not consider partial reconfiguration, a separate control path, or forward compatibility. A second middleware layer for network-attached FPGAs was proposed by *Tarafdar et al.* in 2019 to provide run-time specific communication functionality [33], [57]. Their middleware can also be used to hide the actual physical nodes behind logical kernels by providing a common address space. Eventually, the authors do not consider different run-time environments or middlewares executed at the same time on an FPGA platform. Also, their middleware is a static part of the platform and cannot be updated during run time.

Beyond related research within the FPGA and our own work, there are complementary efforts to establish FPGAs in cloud environments by improving the resource management of closely-coupled FPGAs for monolithic FPGA designs [43], [58]–[61]. For example, *InAccel*'s FPGA management framework enables the usage of bus-attached FPGAs with Kubeless applications, by providing the necessary resource detection, FPGA configuration and deployment functionalities [17], [62]. These efforts are valuable to increase the availability of FPGAs to VMs or containers and to enhance the flexibility of cloud FPGA deployments, but target different hardware platforms and deployment models (i.e. IaaS/PaaS such as the left-hand side of Figure 1) than this work. Nonetheless, we expect these frameworks could gain further efficiency and flexibility if they were to adopt the Mantle architecture.

IX. Conclusion

To saturate the world's ever increasing need for energy- and cost-efficient compute power, data center providers rely on acceleration more and more. FPGAs offer energy-efficiency and the flexibility to adopt to a broad range of accelerated applications but lack integration with modern cloud offerings like Function-as-a-Service. Deploying them as standalone, disaggregated compute nodes enables a seamless integration with scalable cloud services. Furthermore, the proposed dynamic three-layer architecture offers forward and backward compatibility while simplifying the design and deployment environment for FPGA-based Function-as-a-Service offerings. The evaluation of this proposed architecture design pattern — the `Mantle` architecture — shows an increase of the execution efficiency by factors of 2.9 – 4.1 to approximately 90%, even if the FPGA is used for just 80 seconds. Additionally, using partial reconfiguration and the self-management capability of the proposed FPGA management cores, the cold-boot and provisioning time of new FPGA instances is reduced by factors of 13.2 – 43.7 to below 7 seconds. We hope that our research highlights the promise of disaggregated FPGAs for cloud offerings and helps FPGA-based Function-as-a-Service offerings becoming a reality.

Notices

‡ Intel, Intel logo, and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

§ The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on worldwide basis.

** IBM and the IBM logo are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on ibm.com/trademark .

REFERENCES

[1] Amazon Web Services, Inc. (2021). "AWS Lambda," [Online]. Available: https://aws.amazon.com/lambda/.

[2] IBM. (2021). "IBM Cloud Code Engine," [Online]. Available: https://www.ibm.com/cloud/code-engine.

[3] Google, Inc. (2021). "Cloud Functions," [Online]. Available: https://cloud.google.com/functions.

[4] Microsoft, Inc. (2021). "Azure Functions," [Online]. Available: https://azure.microsoft.com/en-us/services/functions/.

[5] The Knative Authors. (2020). "Kubernetes version and version skew support policy," [Online]. Available: https://kubernetes.io/docs/setup/release/version-skew-policy/.

[6] P. Jamshidi *et al.*, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018. DOI: 10.1109/MS.2018.2141039.

[7] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18. DOI: 10.1145/3297858.3304013.

[8] S. Fouladi *et al.*, "Outsourcing everyday jobs to thousands of cloud functions with gg," *Usenix Login*, 2020.

[9] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communication of the ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019. DOI: 10.1145/3282307.

[10] Google, Inc. (2021). "Cloud TPU," [Online]. Available: https://cloud.google.com/tpu.

[11] Amazon Web Services, Inc. (2021). "AWS Inferentia," [Online]. Available: https://aws.amazon.com/machine-learning/inferentia/.

[12] ——, (2021). "AWS Trainium," [Online]. Available: https://aws.amazon.com/machine-learning/trainium/.

[13] Habana Labs Ltd., An Intel Company. (2021). "habana.ai," [Online]. Available: https://habana.ai.

[14] Graphcore. (2021). "Graphcore," [Online]. Available: https://www.graphcore.ai.

[15] Cerebas Systems. (2021). "The Cerebras Wafer Scale Engine," [Online]. Available: https://cerebras.net/product/#chip.

[16] K. Keahey *et al.*, "Managing allocatable resources," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, Jul. 2019, pp. 41–49. DOI: 10.1109/CLOUD.2019.00019.

[17] I. InAccel. (2021). "InAccel FPGA orchestrator," InAccel, Inc., [Online]. Available: https://inaccel.com/fpga-manager/.

[18] Amazon Web Services, Inc. (2021). "Amazon EC2 F1 Instances," [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[19] Baidu, Inc. (2020). "FPGA Cloud Server — Example project description," [Online]. Available: https://cloud.baidu.com/doc/FPGA/s/Rjwvyh0cn.

[20] Xilinx, Inc. (2021). "Xilinx Adaptive Compute Cluster (XACC) Program," [Online]. Available: https://www.xilinx.com/support/university/XUP-XACC.html.

[21] Y. Umuroglu *et al.*, "LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications," in *Proceedings of the 30th IEEE International S Conference on Field-Programmable Logic and Applications (FPL)*, Gothenburg, Sweden: IEEE, 2020, pp. 291–297. DOI: 10.1109/FPL50879.2020.00055.

[22] Z. István *et al.*, "Consensus in a box: Inexpensive coordination in hardware," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA: USENIX Association, 2016, pp. 425–438.

[23] R. Polig *et al.*, "A hardware compilation framework for text analytics queries," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 260–272, 2018. DOI: 10.1016/j.jpdc.2017.05.015.

[24] K. Kara *et al.*, "High Bandwidth Memory on FPGAs: A Data Analytics Perspective," in *Proceedings of the 30th IEEE International S Conference on Field-Programmable Logic and Applications (FPL)*, Gothenburg, Sweden: IEEE, 2020, pp. 1–8. DOI: 10.1109/FPL50879.2020.00013.

[25] M. Purandare, R. Polig, and C. Hagleitner, "Accelerated analysis of boolean gene regulatory networks," in *Proc. 27th Int. Conf. Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–6. DOI: 10.23919/FPL.2017.8056778.

[26] J. Cong *et al.*, "Understanding performance differences of fpgas and gpus," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18, Monterey, CA, USA: ACM, 2018, pp. 288–288. DOI: 10.1145/3174243.3174970.

[27] E. Nurvitadhi *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 5–14. DOI: 10.1145/3020078.3021740.

[28] M. Qasaimeh *et al.*, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, Jun. 2019, pp. 1–8. DOI: 10.1109/ICESS.2019.8782524.

[29] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49, Taipei, Taiwan: IEEE Press, 2016, 7:1–7:13.

[30] F. Abel *et al.*, "An FPGA platform for hyperscalers," *Proceedings - 2017 IEEE 25th Annual Symposium on High-Performance Interconnects, HOTI 2017*, pp. 29–32, 2017. DOI: 10.1109/HOTI.2017.13.

[31] J. Weerasinghe *et al.*, "Disaggregated FPGAs: network performance comparison against bare-metal servers, virtual machines and linux containers," *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, no. i, pp. 9–17, 2017. DOI: 10.1109/CloudCom.2016.0018.

[32] B. Ringlein *et al.*, "System Architecture for Network-Attached FPGAs in the Cloud using Partial Reconfiguration," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Barcelona, Spain: IEEE, 2019, pp. 293–300. DOI: 10.1109/FPL.2019.00054.

[33] N. Tarafdar *et al.*, "Building the infrastructure for deploying fpgas in the cloud," in *Hardware Accelerators in Data Centers*, C. Kachris, B. Falsafi, and D. Soudris, Eds. Cham: Springer International Publishing, 2019, pp. 9–33. DOI: 10.1007/978-3-319-92792-3_2.

[34] V. Krishnan, O. Serres, and M. Blocksome, "Configurable network protocol accelerator (copa) † : An integrated networking/accelerator hardware/software framework," in *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, Aug. 2020, pp. 17–24. DOI: 10.1109/HOTI51249.2020.00018.

[35] B. Ringlein *et al.*, "Programming Reconfigurable Heterogeneous Computing Clusters Using MPI With Transpilation," in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, IEEE, Nov. 2020, pp. 1–9. DOI: 10.1109/H2RC51942.2020.00006.

[36] Yu-Liang Wu, Shuji Tsukiyama, and Malgorzata Marek-Sadowska, "On computational complexity of a detailed routing problem in two dimensional fpgas," in *Proceedings of 4th*

Great Lakes Symposium on VLSI, Mar. 1994, pp. 70–75. DOI: 10.1109/GLSV.1994.289993.

[37] Xilinx Inc., "Vivado Design Suite User Guide: High-Level Synthesis (UG902) [2019.2]," Tech. Rep., 2019.

[38] Intel Corporation, "Intel® oneAPI Programming Guide (Beta)," Intel Corporation, Tech. Rep., 2019.

[39] C. P. Baaij *et al.*, "CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, Sep. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[40] Docker Inc. (2021). "Dockerfile reference," [Online]. Available: https://docs.docker.com/engine/reference/builder/.

[41] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx MicroBlaze MCS SoC, 2nd Edition*. John Wiley & Sons, 2017.

[42] Xilinx Inc., "Vivado Design Suite User Guide: Partial Reconfiguration (UG909) [2019.2]," Tech. Rep., 2019.

[43] K. Pham *et al.*, "Moving compute towards data in heterogeneous multi-fpga clusters using partial reconfiguration and i/o virtualisation," 2020.

[44] Xilinx Inc. (2019). "Vitis Accelerated Libraries," [Online]. Available: https://github.com/Xilinx/Vitis_Libraries.

[45] Oliscience. (2020). "OpenCores," [Online]. Available: https://opencores.org.

[46] A. Vaishnav *et al.*, "Resource elastic virtualization for fpgas using opencl," in *Proc. 28th Int. Conf. Field Programmable Logic and Applications (FPL)*, Aug. 2018, pp. 111–1117. DOI: 10.1109/FPL.2018.00028.

[47] ARM, "AMBA AXI and ACE Protocol Specification," ARM, Tech. Rep., 2011, pp. 1–306.

[48] The GNU C Library Community. (2020). "The GNU C Library (glibc)," [Online]. Available: https://www.gnu.org/software/libc/.

[49] The Kubernetes Authors. (2021). "Welcome, Knative! — Kubernetes-based platform to deploy and manage modern serverless workloads," [Online]. Available: https://knative.dev.

[50] TIS Committee. (1995). "Executable and Linking Format (ELF) Specification — Version 1.2," [Online]. Available: https://refspecs.linuxbase.org/elf/elf.pdf.

[51] Xilinx Inc. (2021). "Vitis Vision," [Online]. Available: https://github.com/Xilinx/Vitis_Libraries/tree/master/vision.

[52] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, pp. 65–76, 2009. DOI: 10.1145/1498765.1498785.

[53] Lawrence Berkeley National Laboratory – Computing Sciences. (2021). "Roofline — Introduction," [Online]. Available: https://crd.lbl.gov/departments/computer-science/par/research/roofline/introduction/.

[54] X. Chen *et al.*, "Is fpga useful for hash joins?" In *CIDR*, 2020.

[55] R. Kirchgessner, A. D. George, and G. Stitt, "Low-overhead fpga middleware for application portability and productivity," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, Sep. 2015. DOI: 10.1145/2746404.

[56] R. Kirchgessner, A. D. George, and H. Lam, "Reconfigurable computing middleware for application portability and productivity," Washington, DC: IEEE, 2013, pp. 211–218. DOI: 10.1109/ASAP.2013.6567577.

[57] N. Eskandari *et al.*, "A modular heterogeneous stack for deploying fpgas and cpus in the data center," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: ACM, 2019, pp. 262–271. DOI: 10.1145/3289602.3293909.

[58] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov. 2015, pp. 430–435. DOI: 10.1109/CloudCom.2015.60.

[59] A. Iordache *et al.*, "High performance in the cloud with fpga groups," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, ser. UCC '16, Shanghai, China: Association for Computing Machinery, 2016, pp. 1–10. DOI: 10.1145/2996890.2996895.

[60] J. Mbongue *et al.*, "Fpgavirt: A novel virtualization framework for fpgas in the cloud," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 862–865. DOI: 10.1109/CLOUD.2018.00122.

[61] J. Lallet, A. Enrici, and A. Saffar, "FPGA-Based System for the Acceleration of Cloud Microservices," Valencia: IEEE, 2018, pp. 1–5. DOI: 10.1109/BMSB.2018.8436912.

[62] I. InAccel. (2021). "FPGAs goes serverless on kubernetes," InAccel, Inc., [Online]. Available: https://inaccel.com/fpgas-goes-serverless-on-kubernetes/.